Faculteit Toegepaste Wetenschappen

Vakgroep Informatietechnologie

Voorzitter: Prof. Dr. Ir. P. LAGASSE

Aspectgenerator voor declaratieve definitie van persistentie

door

Bram ADAMS

Promotor: Prof. Dr. Ir. H. TROMP

Scriptiebegeleider: Ir. K. VANDENBORRE

Scriptie ingediend tot het behalen van de academische graad van burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2003–2004

Woord vooraf

Een nieuwe technologie ontdekken omvat verschillende fasen:

- concepten bestuderen en documentatie lezen;
- tutorials bekijken en kleine tests uitvoeren;
- een concreet probleem aanpakken steunend op de nieuwe kennis.

Niet enkel de theoretische zaken leert men zo beter kennen, ook praktische problemen en op het eerste zicht onbelangrijke details komen dan aan de oppervlakte.

Van alle onderwerpen over Aspect-Oriented Programming (AOP) leek dit me dan ook de meest uitdagende. Er was niet alleen een stevig theoretisch luik (AspectJ, PDLF, XML, Tiger, ...), dit alles moest ook aangewend worden voor een concreet probleem. Ik dank dan ook professor Herman Tromp voor deze mooie kans en ook voor zijn vele hulp (en tijd) dit academiejaar.

Koenraad Vandenborre heeft zijn passie voor dit onderwerp nooit onder stoelen of banken gestoken. Hij heeft me goed op pad gezet en onderweg veel interessante ideeën aangereikt. Kris De Schutter, Geert Premereur en Fred Spiessens beantwoordden dan weer boeiende vragen gaande van compositie tot gedistribueerde werking.

Speciale dank (en lof) gaan uit naar mijn ouders, zus en huisdieren voor hun steun in deze tijden van terreur en voor het gezwind (en hardnekkig) speuren naar typfouten in dit werk. Ten slotte mag ik de goeie contacten met collega-(thesis)studenten David Ooms (generics!), David Tas, Stijn Van Wonterghem, Jan Van Besien, Steven Stappaerts en Jan Willem niet vergeten.

Bram Adams, mei 2004

Toelating tot bruikleen

"De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.
Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie."

Bram Adams, mei 2004

Aspectgenerator voor declaratieve definitie van persistentie

door

Bram ADAMS

Scriptie ingediend tot het behalen van de academische graad van burgerlijk ingenieur in de computerwetenschappen

Academiejaar 2003–2004

Promotor: Prof. Dr. Ir. H. TROMP
Scriptiebegeleider: Ir. K. VANDENBORRE

Faculteit Toegepaste Wetenschappen
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. P. LAGASSE

Samenvatting

In hoofdstukken 1 en 2 bekijken we hoe bestaande persistentieframeworks business model en persistentie trachten te scheiden. We bespreken kort AOP (in het bijzonder AspectJ) en wat het mogelijk nut ervan kan zijn in deze context. Ten slotte wordt uitgelegd waarom PDL-tags verder gebruikt zullen worden.

Hoofdstukken 3, 4 en 5 beschrijven de nodige stappen om aspecten te kunnen genereren die persistentie voorzien: de persistence engineer annoteert broncode, een doclet transformeert die naar persistentiebestanden en deze worden via XSLT naar aspecten getransformeerd.

Deze laatste worden in hoofdstukken 6 en 7 nader besproken. We zullen namelijk twee groepen aspecten onderscheiden: respectievelijk aspecten die voor de eigenlijke persistentie zorgen en andere aspecten die voor samenhang en diepgang zorgen.

Hoofdstuk 8 zal nog de invloed van Java Generics en de Metadata Facility (allebei deel uitmakend van de nieuwe J2SE 1.5) onderzoeken en het laatste hoofdstuk bevat alle besluiten.

Trefwoorden

software-ontwikkeling, Java, AOP, AspectJ, persistentie, impedance mismatch, PDL

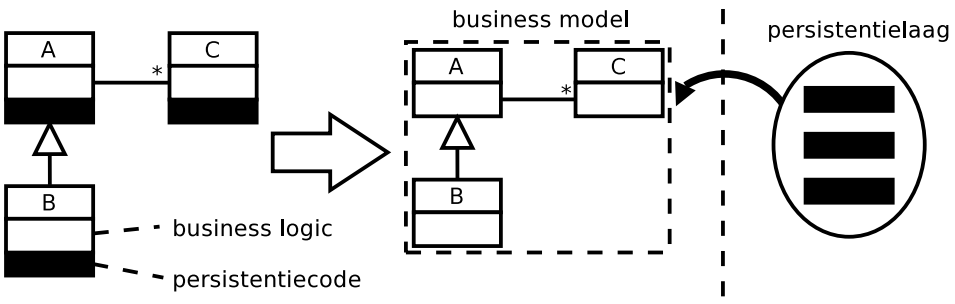
Inhoudsopgave

Hoofdstuk 1

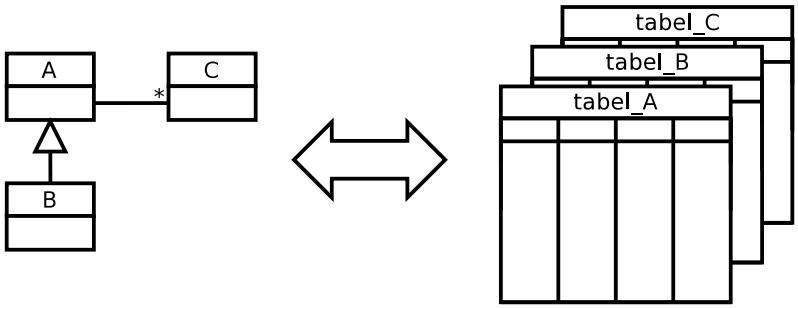
Inleiding

> IBM plans to bring a software development technique [AOP] that has been the subject of theoretical work for years to commercial products this year and next.
> ...
> IBM believes the technology is ready for use in business development, rather than academic scenarios.
> *(AOSD Conference 2004, Lancaster)*

De doorbraak van Aspect-Oriented Programming (AOP) lijkt niet meer te stuiten, althans in de OO-wereld. In wezen is het een verderzetting van het universeel adagium "separating of concerns". Men heeft (zie [16]) enerzijds "module-level" concerns, de hoofdconcerns die netjes in modules gestopt worden. Anderzijds zijn er de "system-level" concerns die een volledig systeem overspannen en dus op talloze modules ingrijpen.
AOP zorgt er nu voor dat de implementatie van deze crosscutting concerns niet langer tussen de hoofdmodules gemengd wordt, maar dat ook zij hun eigen modules, aspecten, krijgen. Die bevatten ook de voorwaarden die bepalen wanneer de aspectcode uitgevoerd mag worden. Dit wordt in Fig. 1.1, die we verderop meer in detail zullen bekijken, geïllustreerd.
Wij zullen AspectJ, de bekendste Java-implementatie van AOP, verder gebruiken in dit werk.



Figuur 1.1: Problemen in hedendaagse applicaties zonder AOP (links) en wat we uiteindelijk wensen te bereiken (rechts).



Figuur 1.2: Impedance mismatch.

PDLF Het probleem waarop we AOP gaan loslaten, is persistentie. In OO-talen houdt dit in dat men objecten langdurig wil bewaren door ze in een niet-vluchtig opslagmedium op te bergen. Zo zijn ze zelfs in staat om langer te leven dan de applicatie die hen creëerde en kunnen applicaties die nu nog niet bestaan later een beroep doen op hen.
De populairste opslagmedia in Java zijn het bestandssysteem, objectdatabanken, maar vooral relationele databanken. Daarin stuit men op een fundamenteel probleem dat "impedance mismatch" heet (Fig. 1.2). De structuur en opbouw van objecten en tabellen zijn niet zomaar compatibel. Men moet expliciet een mapping invoeren tussen de twee. Verschillende frameworks behandelen dit voor Java, en het PDLF-framework [20] gaan we nader bekijken.

Aspectgenerator Op basis van enkele bestaande publicaties, zullen we AOP aanwenden om een persistentielaag te bekomen. Persistentie is immers een crosscutting concern en leent zich dus uitstekend tot een behandeling met aspecten. Bovendien willen we die automatisch genereren, wat vereist dat er genoeg informatie voor handen is. De PDL-tags uit het PDLF-framework zullen hiervoor zorgen. Wat dit zijn en hoe ze tot stand komen zullen we in hoofdstukken 2 en 3 zien. Latere hoofdstukken leiden tot de aspectgenerator zelf en de aspecten die we nodig hebben.

Doel We willen het business model van applicaties zo onwetend mogelijk houden over eventuele persistentiemechanismen die gebruikt zullen worden, en dit om hergebruik te stimuleren. We proberen dat door te werken met de declaratieve PDL-tags. Bij ontwerp van een nieuwe applicatie dient men dan enkel een bepaald persistentiesysteem te kiezen, de nodige aspecten te genereren en die op het business model in te doen werken. We zullen onderzoeken of dit systeem haalbaar is en waar de eventuele knelpunten liggen.

Hoofdstuk 2

Probleembeschrijving

2.1 Inleiding

In de begindagen van het computertijdperk was het ontwikkelen van programma's een zaak voor enkelingen. Slechts weinigen konden overweg met assembler- of machinecode, laat staan dat men geavanceerde applicaties kon bouwen. Met de komst van hoge niveau-talen kwam er meer abstractie, hetgeen de handelbare complexiteit van programma's indrukwekkend deed toenemen. Het individu kon echter niet voldoende volgen, zodat men steeds meer met anderen begon samen te werken. Tegenwoordig werkt men samen in heus teamverband, te vergelijken met (enigszins overbevolkte) voetbalteams. Elk heeft immers zijn rol:

analyst: analyseert het probleem en tracht het zoveel mogelijk te formaliseren, zodat het werk in een latere fase beter omschreven wordt.

ontwerper: op basis van de analyse tracht men in een bepaald formalisme een ontwerp te maken dat aan bepaalde kwaliteitsnormen voldoet (onderhoudbaarheid, stabiliteit, ...).

programmeur: implementeert het ontwerp.

tester: test uitvoerig de implementatie in een poging alle (belangrijke) fouten te ontdekken.

documentatie: het schrijven van handleidingen, ...

We zien dat elke rol informatie van een ander gebruikt en er iets nieuws van maakt: communicatie is dus onontbeerlijk. Daarnaast bestaan er verschillende mogelijkheden omtrent de volgorde van het ontwikkelingsproces. Vroeger volgde men eerder het watervalmodel, waar men sequentieel de fasen analyse-ontwerp-implementatie-testen doorliep waarna de applicatie "af" was. Vrijwel altijd was het nodig om terug te keren wanneer fouten in het ontwerp gevonden waren, bugs in de implementatie, ...
Nu volgt men meestal het spiraalmodel. Men zal na het testen opnieuw naar de analysefase gaan om in een volgende iteratie na te gaan wat er fout ging en wat men kan verbeteren om een betere versie te bekomen. Mengvormen tussen de twee bestaan ook.

2.2 Persistence engineer

In [20] ijvert de auteur voor een nieuwe rol, de persistence engineer. Deze persoon wordt actief betrokken bij het ontwerpproces en houdt zich bezig met het persisteren van het business model. Hij zorgt er in het algemeen voor dat informatie op een efficiënte manier gerepresenteerd wordt in een databank (of desnoods een ander persistentiemiddel, zoals het bestandssysteem) en opvraagbaar is.

Zijn voornaamste taak is echter om het verschil in voorstelling van informatie in de te ontwikkelen applicatie en in het persistentiemiddel op te vangen. De in hoofdstuk 1 besproken "impedance mismatch" is hier een goed voorbeeld van: de structuur van objecten in OO-talen staat haaks op de tabellen die de relationele databanken gebruiken. Waarom blijft men dan deze databanken gebruiken i.p.v. de modernere objectdatabanken? Ten eerste is deze laatste technologie nog lang niet ten volle doorgrond, daar waar relationele databanken door en door bestudeerd en theoretisch onderbouwd zijn. Daarom is ze een pak minder efficiënt. Ten tweede is er de economische factor: enerzijds is er de erfenis uit het verleden, waardoor vele bedrijven gekluisterd zijn aan hun legacy-databanken (netwerk, hiërarchisch, relationeel, ...). Anderzijds investeert men liever in een technologie die zijn nut al bewezen heeft en die door velen gekend is, zodat voldoende personeel ermee vertrouwd is.

Om zijn moeilijke opdracht te vervullen, beschikt de persistence engineer over tools die de beschreven mapping-problemen zoveel mogelijk trachten te automatiseren. Zijn taak is dan om het juiste middel aan te wenden en de nodige informatie te verschaffen zodat het ding zijn werk kan doen. In deze publicatie wordt voornamelijk gekeken naar Java en Java-gerelateerde tools voor object-relational mapping (ORM). Verder ga ik me vooral concentreren op het PDLF-framework (zie 2.2.1) en Java Data Objects (JDO; zie 2.2.2). Hier en daar wordt verwezen naar Container Managed Persistence (CMP; zie 2.2.3). Er bestaan echter nog tal van andere (open source) tools voor Java, zoals Hibernate[1], XORM[2], Prevayler[3], ...
Voor C++ bestaan er enkele open source mogelijkheden, zoals Eternity[4], CommonC++[5], ... De meeste bedrijven gebruiken echter in-house producten, die niet publiek gemaakt worden.

2.2.1 Persistency Definition Language Framework (PDLF)

Het Persistency Definition Language Framework wordt beschreven in [20]. Het is een framework dat a.h.v. een declaratieve taal (PDL) en een bepaalde methodologie het ORM-probleem voor Java aanpakt.

2.2.1.1 Werkwijze

Eerst zal men m.b.v. PDL, een taal bestaande uit speciale javadoc-tags in de programmacode, enkele zaken moeten verduidelijken. Dit is nodig omdat Java, althans vóór J2SE 1.5, niet ex-

[1] http://www.hibernate.org/
[2] http://xorm.sourceforge.net/
[3] http://www.prevayler.org/wiki.jsp
[4] http://www.winghands.it/prodotti/eternity/
[5] http://sourceforge.net/projects/cplusplus/

pressief genoeg was om bijvoorbeeld aan te geven welk type de elementen van een Collection hebben, of het verschil tussen compositie en associatie, ... Ook het aangeven van versienummers, namen van tabellen en databank, ... kan men opgeven. Meer hierover in hoofdstuk 3.
Daarnaast dient elke klasse die men wenst te persisteren over te erven van een speciale klasse PObject. Dit bevat een object identifier (OID) die elk object een eigen unieke identiteit geeft. Verder worden de nodige persistentiemethoden voorzien, zoals write(), update(), delete(), read(), queryForOID(), ... (zie verder).

Via een speciale doclet worden de PDL-tags nu verwerkt tot een XML-bestand, één per Java-klasse. Samen met de code beschrijft deze alles wat nodig is om de betrokken klasse te kunnen persisteren.

Men presenteert vervolgens de broncode en het XML-bestand van elke relevante klasse aan het PDLF-framework voor registratie. Dit houdt in dat interne structuren opgebouwd worden die de meta-informatie i.v.m. persistentie opslaan in een toegankelijker vorm. Tevens worden abstracte representaties van tabellen uit een relationele databank opgebouwd. Ten slotte creëert men daadwerkelijk de benodigde tabellen in de databank. De layout daarvan ligt vast in het framework. Er is tevens een systeem voorzien om verschillende versies van klassen naast elkaar te gebruiken. Hierna zit de rol van de persistence engineer er op.

Eens dit alles achter de rug is, kunnen de programmeurs met eenvoudige static-methoden van de klasse PObject objecten in de databank opslaan, updaten, opvragen, ... Er werd ook een heel eenvoudige query-taal gecreëerd.

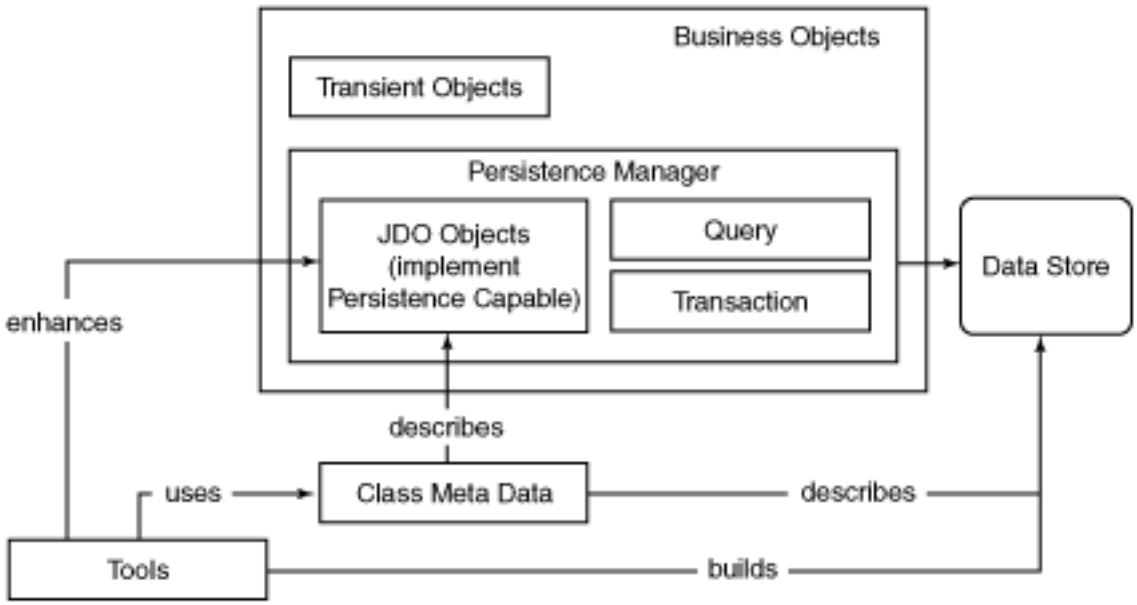
2.2.1.2 Opmerkingen

Hoe werken de persistentiemethoden precies? De bij registratie opgebouwde interne representatie wordt at runtime geraadpleegd om de juiste tabellen te localiseren (afhankelijk van de versie, ...). De informatie hieruit wordt dan aangewend om de SQL-code te genereren. Queries werken op dezelfde manier.
Er zijn voorzieningen getroffen voor een cache om de performantie te verhogen, en tevens zorgt een transactie-mechanisme voor concurrency control en crash recovery.

2.2.1.3 Nadelen

De belangrijkste nadelen hebben te maken met de mate van inmenging van persistentiebelangen in de ontwikkeling van de applicatie zelf. Zo moeten alle klassen uit het business model die wensen gebruik te maken van het PDLF-framework overerven van de klasse PObject. Naast het feit dat programmeurs goed moeten weten wat het gebruikte persistentiemechanisme zal zijn en dat meteen vastleggen, neemt dit ook de enige overerving in beslag. Persistentie- en modelleringsbelangen worden dus nog altijd duidelijk vermengd. Dit toont zich ook in het attribuut

```
public static ClassVersion classVersion=...;
```



Figuur 2.1: Overzicht van JDO.

dat PObject-instanties verplicht moeten krijgen. De scheiding tussen de declaratieve persistentie-eigenschappen en de implementatie van het business model werd niet volledig doorgetrokken.

Daarnaast is het framework volledig gericht op relationele databanken, hoewel de eerste fasen van de werking herbruikbaar zouden zijn voor andere opslagmethoden. We zullen dit in hoofdstukken 3 en 4 bekijken.

2.2.2 Java Data Objects (JDO)

JDO is een objectgeoriënteerd persistentiemechanisme, dat sedert maart 2002 gestandaardiseerd is (versie 1.0). Doel was het zo goed mogelijk afschermen van de gebruikte data sources van de gebruiker, zonder daarbij hoge eisen te stellen aan de code voor het business model. Dit heeft als gevolg dat er metadata nodig is naast de broncode zelf, omwille van dezelfde reden als we in 2.2.1.1 gezien hebben. Fig. 2.1 (uit [19]) toont een schematisch overzicht van JDO.

2.2.2.1 Werkwijze

In principe kan men business code ontwerpen zonder rekening te moeten houden met persistentie-eisen. Enige verplichtingen zijn het gebruik van de JavaBeans-conventie qua naamgeving van setters en getters en de aanwezigheid van een no-arg constructor. Alle attributen die niet als "transient" aangeduid werden, kunnen opgeslagen worden. In een persistence descriptor kan men echter nog extra metadata opgeven. De inhoud van dit xml-bestand komt overeen met informatie uit PDL-tags en *niet* met die van een persistentiebestand uit hoofdstuk 4.
Nadat alle code af is, zal een enhancer haar op basis van de persistence descriptor aanpassen, zodat ze aan de JDO-specificatie voldoet. Dit mag zowel op broncode- als op bytecode-niveau gebeuren, maar voornamelijk de laatste methode wordt toegepast dankzij de populariteit van

bijvoorbeeld de Byte Code Engineering Library (BCEL), een Jakarta-project[6].

De persistence descriptor is een XML-beschrijving die onder meer volgende zaken bevat:

- welke klassen en attributen persistent moeten zijn;
- relaties tussen persistente klassen (associatie, ...);
- types van de elementen van collecties.

De applicatie die het business model gebruikt, bevat specifieke JDO-methoden om objecten op te slaan, te vernietigen, op te vragen, ... Al deze acties staan onder supervisie van een soort container, de persistence manager, die ook transacties verzorgt. Een belangrijk principe dat hier gebruikt wordt, is "persistence by reachability". Bij het opslaan van een object bijvoorbeeld, worden ook alle objecten die ermee in verband staan, opgeslagen. Dit is heel makkelijk, maar niet altijd gewenst. In versie 2.0 van de specificatie zal een fijner verantwoordelijkheidsmodel opgesteld worden, PDL-tags kunnen dit echter al. Verder wordt er wel een onderscheid gemaakt tussen "first class"- en "second class"-objecten. De eerstgenoemde hebben een eigen persistentie-infrastructuur (tabel in databank bijvoorbeeld) en de andere worden bij hen geherbergd (cf. RADT's in het PDLF-framework).

Een eigen query-taal, Java Data Objects Query Language (JDOQL), werd ontworpen. Deze is eerder een objectversie van SQL dan een bewerking van de ODMG-standaard. In 6.3.5 zullen we meer ingaan op versie 1.0 hiervan.

2.2.2.2 Beoordeling

Er zijn grote gelijkenissen met de in dit werk gebruikte strategie (en het PDLF-framework tot op zekere hoogte):

- scheiding van modelling en persistentie-concerns;
- metadata (hetzij in een XML-bestand, hetzij in javadoc-tags);
- bytecode wordt aangepast om persistentie toe te laten.

De metadata is echter enkel nodig als de corresponderende klasse gebruikt wordt als het type van een persistent attribuut, en wordt dan in een apart bestand ondergebracht. De business klassen moeten dus niet stelselmatig getagd worden, wat meer code reuse toelaat en minder werk vereist dan de PDL-tags.

Dankzij een goeie scheiding van de modelling en persistentie-concerns, een eenvoudige API, een licht framework en een goeie respons uit de industrie verdringt JDO steeds meer CMP (zie 2.2.3). Deze laatste is immers een veel complexere, zwaardere technologie met een steile leercurve, die ook een sterke invloed (hypotheek?) heeft op het ontwerp van het business model. Sterker nog: veel applicatieservers gebruiken als onderliggende implementatie voor CMP juist

[6] http://jakarta.apache.org/bcel/

JDO. Een eenvoudige technologie die een complexere in leven houdt, is een op de lange duur onwerkbare situatie. Bij relationele databanken zal de strijd in de nabije toekomst meer dan waarschijnlijk gaan tussen Hibernate en JDO. Beide kunnen in BMP-Beans gebruikt worden, zodat andere middleware-diensten van applicatieservers bruikbaar blijven. De EJB 3.0-specificatie zal echter een veel lichter framework beschrijven dan het huidige en krijgt de steun van enkele grote applicatieserver-fabrikanten (IBM, BEA en Oracle).

Eind april 2004 is men begonnen aan versie 2.0 van de specificatie, met het doel enkele vragen vanuit de industrie te beantwoorden:

- tekortkomingen in JDOQL wegwerken, zoals het beperkt aantal return types bij queries;
- mapping naar databanken standaardiseren, zodat dit niet vendor-afhankelijk blijft;
- het persistent kunnen maken van interfaces;
- object-eigenaarschap beter specificeren;
- metadata voor een stuk met generics en de Metadata Facility uit J2SE 1.5 opgeven (net als in dit werk gedaan werd in hoofdstuk 8).

2.2.3 Container Managed Persistence (CMP)

Het Java 2 Enterprise Edition-platform (J2EE) is één van de drie Java-platforms van Sun Microsystems. Het richt zich op enterprise-niveau toepassingen, en bevat een waaier aan technologieën zoals JSP/servlet, EJB, ... Deze moeten de ontwikkeling van veilige, platform-onafhankelijke en gedistribueerde applicaties ondersteunen en vergemakkelijken.
De Enterprise JavaBeans-specificatie (EJB) is een heel uitgebreide technologie met een hele reeks aan nieuwe begrippen en technieken. Ze draait vooral rond componenten, de EJB's, en een container. Deze laatste levert een heel scala aan middleware-diensten, zoals gedistribueerde werking, caching, resource pooling, clustering, transacties en natuurlijk ook persistentie. Het maken van een applicatie komt er ruwweg op neer dat men componenten assembleert en "deployed" in een applicatie-server[7]. Deze laatsten zijn vendor-afhankelijke implementaties van containers die voldoen aan de geldende specificatie.

2.2.3.1 Werkwijze

Er bestaan drie soorten EJB's, namelijk Session Beans, Message Beans en Entity Beans. Elk heeft zijn eigen specifieke levenscyclus, maar ze hebben allemaal een hoop interfaces nodig (de local/remote (home) interfaces), een Bean klasse (met de eigenlijke business logic in) en een deployment descriptor (zie verder).

De container zorgt ervoor dat de EJB's, zonder dat ze het expliciet vragen in hun broncode, gebruik kunnen maken van de middleware-services. Hoe kan de container weten wat een Bean nodig heeft? Opnieuw gebruikt men hiervoor een declaratieve werkwijze. De deployment descriptor van de Beans zal namelijk al de nodige declaratieve info bevatten om de gewenste

[7] Voorbeelden hiervan zijn JBoss, IBM WebSphere Application Server, BEA WebLogic, ...

middleware-diensten te doen werken. Dit is een bijna identieke werkwijze als bij JDO, PDLF en in dit werk bekeken wordt (zie 2.3.2).

Van de drie geziene Beans stellen de Entity Beans business objecten voor die gepersisteerd kunnen worden. Nu kan men dit op twee manieren doen:

Bean Managed Persistence (BMP): de Beans bevatten zelf oproepen uit bijvoorbeeld de JDBC- of SQL/J-API die alle nodige persistentiemethoden implementeren. Dit is heel omslachtig werk, sterk gericht op één persistentie-mechanisme (meestal relationele databanken), maar door het handwerk wél sterk geoptimaliseerd.

Container Managed Persistence (CMP): hier bevat de Bean geen persistentiecode, maar moet hij aan een hele reeks voorschriften voldoen (abstracte getters en setters, ...). De deployment descriptor moet dan de nodige metadata bevatten, zoals de namen van de persistente attributen, speciale relaties tussen de huidige Bean en andere, vereiste query-methoden, het gekozen persistentiemechanisme, ... De container zal dan zelf klassen genereren met persistentiecode in. Deze manier van werken leidt veel vlugger tot resultaat, is robuuster (meer geautomatiseerd), maar niet zo op maat gemaakt als BMP-Beans. De manier van implementeren is vrij omslachtig en sterk verschillend van die van gewone Java-klassen.

Wat men precies kiest als strategie volgt uit een trade-off tussen enerzijds de tijdswinst die men bekomt door persistentie te automatiseren en anderzijds de snelheidswinst die handmatig coderen met zich mee brengt.

2.2.3.2 Opmerkingen

Ook hier zien we het belang van declaratieve informatie. Het laat toe om code-generatoren en andere geautomatiseerde tools te bouwen, zodat de business klassen voldoende generiek kunnen zijn en zich voornamelijk op hun taak kunnen richten.

Bij de ontwikkeling van applicaties steunend op EJB's wordt de rol van de persistence engineer meestal overgenomen door de component-ontwikkelaars van de Entity Beans. Het meeste werk bij de implementatie van een CMP-Bean kruipt immers in het creëren van de deployment descriptors en vervangt er het traditionele programmeren. Bij BMP-Beans geldt dit vanzelfsprekend niet.

Helaas hangt de EJB-specificatie vooral aaneen door afspraken en conventies, die vaak niet echt praktijkgericht zijn. Het gebruik van metadata in de deployment descriptors op zich is goed, maar praktische uitwerking ervan levert veel kommer en kwel op. De enorme hoeveelheid aan nieuwe termen en technologieën, die soms conflicten opleveren met gewone Java-gebruiken[8], remt het massaal gebruik van EJB's af[9].
JDO lijkt een veel beter alternatief, maar EJB 3.0 lijkt heel wat ballast overboord te zullen

[8] De bean klasse implementeert bijvoorbeeld NIET zijn local of remote interface.
[9] Dit in tegenstelling tot andere componentgebaseerde technologieën zoals Visual Basic, Delphi of gewone Java, die wél een florerende componentenmarkt hebben.

gooien. Home interfaces zullen verdwijnen, evenals deployment descriptors en SessionBean-interfaces. In plaats daarvan zal de Metadata Facility van J2SE 1.5 gebruikt worden (zie ook hoofdstuk 8).

2.3 Aspectgenerator

Dit werk bouwt voort op dat van M. Matar in [20], maar benadert de problematiek op een andere manier. In principe is persistentie een crosscutting concern, iets dat op zich niet het hoofddoel van modellering, de "core concern", is, maar dat belang heeft over het hele systeem heen.

Een boekhoudkundige applicatie vereist bijvoorbeeld dat men facturen opslaat om later te kunnen bekijken en als bewijskracht te dienen, maar de factuur zelf heeft dergelijke functionaliteit niet nodig om factuur te kunnen zijn. Sterker nog, bij modellering van een factuur in de ontwerpfase zou men zelfs geen rekening mogen houden met eventuele persistentiemechanismen.

Een ander voorbeeld is logging. Ofschoon het belangrijk en nuttig is voor eventuele foutenanalyse om gedetailleerde informatie te hebben van wat allemaal gebeurd is, toch heeft het loggen van gebeurtenissen niets te maken met het opgestelde model van een factuur, een persoon, ... Bovendien wordt logging verspreid over heel de applicatie gebruikt.

Deze voorbeelden tonen aan dat het business model zelf eigenlijk niets te maken heeft met persistentie, noch met logging, authenticatie, ... Zodoende leent het AspectOriented Programming-paradigma (AOP) zich uitermate om persistentie te beschrijven. AOP structureert namelijk crosscutting concerns, in die zin dat zij, net als klassieke (Java-)klassen, in aparte modules terechtkomen. Deze centralisatie vermijdt de twee grote kwalen die anders opduiken, nl. code scattering en code tangling.

code scattering: het fenomeen dat principieel dezelfde code overal verspreid optreedt. Voorbeelden bij uitstek zijn persistentie en logging. Dit heeft als groot nadeel het gigantisch werk dat optreedt bij refactoring. Men moet immers alle plaatsen opzoeken en aanpassen doorheen de volledige code in plaats van alles op 1 plaats te kunnen doen.

code tangling: het mengen van code horend bij de main concern en neven-concerns. Een voorbeeld hiervan kan men in [17] vinden. Dit compliceert de dingen dermate dat de onderhoudbaarheid eronder lijdt, aangezien de verschillende concerns elkaar kunnen beïnvloeden en ze ook de programmeurs afleiden van wat echt het belangrijkste is. Daarnaast wordt code reuse gehypothekeerd, omdat componenten niet één specifieke taak vervullen, maar verschillende door elkaar heen.

2.3.1 AOP en AspectJ

AOP kan op verschillende manieren gedefinieerd worden, de een al wat duidelijker dan de andere. Een mogelijke definitie voor AOP-statements is de volgende [10]:

> In programs P, whenever condition C arises, perform action A.

Een aspect is dan een module met een verzameling van dergelijke statements die samenhoren. We zien dat de programma's P oblivious zijn t.o.v. aspecten, d.w.z. dat ze niet gebouwd werden met het besef dat aspecten erop zouden toegepast worden. Bij ons komt P overeen met de business code, C met persistentie-acties en A met implementatie hiervan.

Volgens [10] verschillen specifieke AOP-implementaties op drie punten van elkaar:

quantificatie: welke condities C laat men toe en hoe worden ze gespecificeerd? Worden zowel statische als dynamische condities ondersteund?

interface: hoe verhouden de acties A zich t.o.v. elkaar en t.o.v. programma's P?

weaving: hoe en wanneer worden de acties A en programma's P gemengd?

AspectJ is een open source AOP-implementatie voor Java die sterk ondersteund wordt door IBM (zie ook hoofdstuk 1). De programma's P zijn dus geschreven in Java en zowel C als A zijn opgetrokken uit een uitbreiding van de Java-syntax. Deze drie componenten hoeven echter niet noodzakelijk in dezelfde taal geschreven te worden. We zullen de drie criteria nu verder bekijken in het licht van AspectJ (zie [12]).

2.3.1.1 Interface

Aspecten worden gegroepeerd in hun eigen modules, vergelijkbaar met gewone klassen in bestanden met extensie .aj of .java. Deze kunnen een no-arg constructor bevatten, maar instantiatie gebeurt automatisch. Men kan enkel aangeven wanneer dit moet gebeuren (één singleton, één instantie per this-object van een pointcut, ...).

Naast gewone methoden en attributen kan een aspect pointcuts en advice bevatten. Pointcuts zorgen voor quantificatie, daar waar advice instaat voor de acties A. Men kan aangeven dat advice vóór, na of rond de opgegeven plaatsen in P uitgevoerd wordt (resp. before-, after- en around-advice). In around-advice kan men de oorspronkelijke code uit P uitvoeren door proceed() op te roepen, maar men kan dit evengoed niet doen (advice vervangt dan volledig de oorspronkelijke code). In feite is advice gewoon een methode die niet opgeroepen wordt bij naam, maar automatisch uitgevoerd wordt als bepaalde voorwaarden voldaan zijn.

2.3.1.2 Quantificatie

Pointcuts zijn constructies die verzamelingen van join points voorstellen. Join points zijn dan weer aangrijpingspunten in gewone programma's P. De condities C worden dus uit pointcuts opgebouwd. De voornaamste primitieve pointcuts zijn:

call(Method-/ConstructorPattern): oproep van een methode/constructor aan de oproepende kant, d.w.z. dat het this-object het object is waarbinnen de methode/constructor-oproeper gedaan wordt en het target-object het object waarop de methode/constructor uitgevoerd wordt

execution(Method/ConstructorPattern): analoog, maar aan de kant van de opgeroepene. Het this-object is het object waarop de methode/constructor uitgevoerd wordt en er is geen target-object.

adviceexecution(): uitvoer van om het even welk advice

within(TypePattern): alle join points waarvan de code voorkomt in packages waarvan de namen op hun beurt voldoen aan het opgegeven patroon

cflow[below](Pointcut): alle join points die optreden in de control flow van de opgegeven pointcut [zonder het join point horende bij de pointcut zelf]

get(FieldPattern) en set(FieldPattern): het lezen/schrijven van attributen die voldoen aan het opgegeven patroon

if(Expression): alle join points waar de test geldt

this(Type|Variable): alle join points waarbij het this-object van het opgegeven type is of van het type van de variabele

target(Type|Variable): alle join points waarbij het target-object van het opgegeven type is of van het type van de variabele

args(Type|Variable,...): alle join points horend bij methode-oproepen waarbij het eerste argument van het opgegeven type is of van het type van de variabele

Deze kunnen alle door “and”, “or” of “not” gecombineerd worden tot krachtiger uitdrukkingen. De patronen zelf kunnen ook wildcards bevatten. De laatste drie opgesomde dienen vaak om context te vergaren over pointcuts, d.w.z. objecten die als argument opgegeven worden, ...
Pointcuts “if”, “this”, “target”, “args”, “cflow” en “cflowbelow” zijn dynamische condities, de rest is statisch.

Men heeft tevens ook de mogelijkheid om dingen te introduceren in bestaande code. Zo kan men klassen nieuwe attributen en methoden geven, hen als implementors van bepaalde interfaces of zelfs als children van een bepaalde klasse opgeven, ... De officiële naam hiervoor is Inter-Type Declaration (ITD)[10]. Dit opent vele perspectieven en wordt in dit werk dan ook uitvoerig gebruikt.

2.3.1.3 Weaving

Vóór versie 1.1 was AspectJ’s weaver gewoon een source code preprocessor. Vanaf 1.1 wordt bytecode-weaving gebruikt. I.p.v. javac gebruikt men ajc, dat eerst broncode gewoon compileert en dan de aspecten erdoorheen weeft. Het resultaat voldoet aan de Java Language Specification (JLS) voor class-bestanden.

Er bestaat ook de mogelijkheid om aspecten at load-time te weaven, maar puur at run-time is (nog) niet mogelijk.

[10] Dit is de nieuwe benaming. Vroeger sprak men over het introduceren van bijvoorbeeld attributen.

2.3.2 Aanpak

A. Rashid en R. ChitChyan hebben in [21] onderzocht of het haalbaar zou zijn om in Java persistentie m.b.v. aspecten te beschrijven zodat de originele applicatie niets moet afweten hiervan ("oblivious"). Dit bleek niet volledig te kunnen. Queries kan men niet verbergen en ook het verwijderen van data uit de databank moet expliciet door de applicatie zelf geseind worden, daar Java een delete-operator mankeert. Het inserten en updaten van data kan wel oblivious gebeuren, resp. bij constructie van een object en bij gebruik van de setters (die net als getters verplicht aanwezig dienen te zijn). Bij dit onderzoek werd een framework gebouwd dat voldoende generiek is om herbruikt te worden. Dit neemt niet weg dat het business model zelf wél oblivious is t.o.v. persistentie-concerns. Enkel het gebruik ervan in een applicatie is dit niet, omdat men dan een keuze gemaakt heeft over het gehanteerde persistentiemechanisme.

Een andere aanpak m.b.v. aspecten is die van K. Vandenborre et al. uit [25]. Hier gebruikt men ITD om alle te persisteren klassen een lege interface te doen implementeren. Daarna geeft men, opnieuw via ITD, die interface verschillende implementatieloze methoden zoals write(), read(), ... Specifieke aspecten horend bij een bepaalde klasse zullen dan oproepen naar de genoemde persistentiefuncties zinvol adviseren met persistentiecode.

Ik heb me vooral op deze laatste aanpak gebaseerd. Enerzijds is deze door de expliciete aanroep van functies als write() minder oblivious dan de eerste, maar ze is wel generieker. Men kan met een extra aspect (zie 7.3.5) de publieke interface naar de persistentielaag toe wat transparanter maken om toch een meer oblivious effect te bekomen.

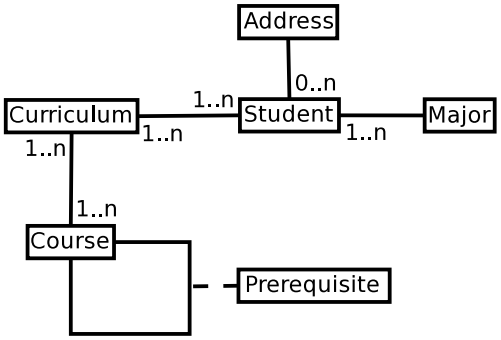
Daarnaast is de tweede aanpak makkelijk te koppelen aan de PDL-tags, omdat het declareren van persistentie (PDL en XML-bestand) en het daadwerkelijk persisteren met aspecten twee duidelijk gescheiden concerns zijn. Samenwerking tussen beide zal hier onderzocht worden. We willen immers uit de PDL-tags in de broncode de nodige aspecten genereren (via een tussenstap: generieke XML-beschrijving). [21] gebruikt applicatie-afhankelijke aspecten die de ORM aangeven. Ik vermoed dat enkel daar de te persisteren attributen aangeduid worden. Wellicht zijn dergelijke aspecten ook te genereren uit declaratieve info.

Het grootste nadeel van [21] is echter het ontnemen van de enige mogelijke overerving, net als in [20], omdat via ITD een bepaalde klasse als parent opgegeven wordt. Java heeft echter maar één overervingshiërarchie, zodat dit de modellering sterk beïnvloedt.

2.3.3 Roadmap

In de volgende hoofdstukken zullen alle stappen beschreven worden die nodig zijn om het business model van een applicatie te persisteren. Daarnaast zullen de implicaties hiervan op het ontwikkelproces onder de loep genomen worden. Hoewel de besproken implementatie niet alle mogelijke functionaliteit bevat (single-user, ...), zal aangetoond worden hoe dit in te bouwen is. Ten slotte zal de invloed van J2SE 1.5 kort onderzocht worden, voor zover dit al mogelijk is.

Inkapseling van dit werk in EJB's werd niet bekeken, maar BMP-Beans zouden er misschien wel kunnen van profiteren.



Figuur 2.2: UML-schema van voorbeeldapplicatie.

2.4 Voorbeeld

De volgende hoofdstukken zullen kort geïllustreerd worden met de voorbeeldapplicatie van M. Matar uit [20]. Deze wordt kort in Fig. 2.2 geschetst. Het is een ruw ontwerp van een studentenadministratie. Net als alle broncode horend bij dit werk, is de volledige code te vinden op http://faramir.ugent.be/thesissen/aspectgenerator/source.zip. Wij gaan enkel representatieve stukken bekijken.

Hoofdstuk 3

Stap 1: Code tagging

3.1 Inleiding

Zoals in 2.2.1 al werd aangegeven, mist Java de nodige expressiviteit om alle (meta)data te bevatten die nodig is om objecten te persisteren. Enkele voorbeelden van ontbrekende informatie:

- Het verschil tussen compositie en associatie kan niet aangegeven worden.
- Wil men efficiënte gegevensopslag en queries toelaten, dan dient men te weten wat het type is van in Collections opgeslagen objecten.
- Niet alle attributen zullen bij queries gebruikt mogen worden.

Uitbreiding van Java ware een oplossing geweest, maar dit stuit op een reeks bezwaren. Praktisch gezien had men de Java-compiler moeten aanpassen, het class-formaat wellicht ook en alle bestaande applicaties zouden herschreven dienen te worden. Nog belangrijker is de opmerking dat implementatie- en persistentiebelangen door elkaar gemengd zouden worden, terwijl we deze juist moeten ontkoppelen van elkaar. Een aparte taal, onafhankelijk van Java, is dus aangewezen, en daarom voerde M. Matar in [20] PDL in. Dit laat toe om de metadata ook voor andere talen te gebruiken, mits enkele kleine aanpassingen desnoods.

3.2 Persistency Definition Language (PDL)

Technisch gezien is PDL een Domain Specific Language (DSL), dit is een taal die slechts voor heel specifieke doeleinden, hier persistentie, bruikbaar is. Dit staat in scherp contrast met zogenaamde “all purpose”-talen zoals C++ of Java. DSL’s kunnen makkelijk ingebed worden in andere talen, en men schrijft er meestal ofwel een interpreter ofwel een codegenerator voor[1].

[1] http://www.martinfowler.com/bliki/DomainSpecificLanguage.html

In praktijk bestaat PDL uit javadoc-tags, ingebed in broncode. Ze is dus makkelijk uitbreidbaar en ook omvormbaar naar andere representaties zoals bijvoorbeeld een XML-bestand (zie hoofdstuk 4). De tags bevatten nodige aanvullingen bij de broncode om het gevraagde persistentiegedrag te bekomen. Ze worden alle beschreven in [20], maar in deze sectie bekijken we ze nog eens allemaal. Bovendien hebben we het een en ander aangepast, grotendeels enkel de syntax, maar soms ook meer inhoudelijk.

We dienen op te merken dat de declaratieve info niets afdwingt, maar enkel metadata bevat. Men zou zelf echter aspecten (wellicht genest in de klasse zelf) kunnen gebruiken om af te dwingen dat een Collection objecten van een bepaald type bevat, een String een bepaalde limietlengte heeft, ...

3.2.1 @db

```
@db database="<naam databank>" table="<naam tabel>"
```

Deze tag is een samenvoeging van @database en @table. In tegenstelling tot PDLF worden deze twee waarden nu wél in de XML-representatie opgenomen (zie hoofdstuk 4). Onze proof-of-concept implementatie houdt echter enkel rekening met de tabelnaam, omdat eenzelfde databank verondersteld werd voor het volledige systeem.

PDL moet zoveel mogelijk onafhankelijk blijven van welk persistentiemechanisme ook. Het begrip "table" kan men dan ook ruimer zien dan voor relationele databanken alleen, bijvoorbeeld als de naam van een bestand in een file-based systeem. Desnoods wordt het gewoon niet gebruikt, bijvoorbeeld bij een objectgeoriënteerde databank.

3.2.2 @pversion

```
@pversion major=<hoofdversienummer>[ minor=<nevenversienummer;
    default 0>]
```

Opnieuw een samenraapsel van twee aparte tags, nl. @major en @minor. In Java's standaard-doclet zit al een @version-tag die de versienummers van Java-klassen zélf documenteert. Hoewel implementatie van een Java-klasse nauw samenhangt met diens persistentiekarakteristieken, willen we beide toch zoveel mogelijk scheiden. Aanpassingen aan de persistentie-annotaties hebben niet noodzakelijk gevolgen voor de broncode en omgekeerd. Ze worden door resp. persistence engineers en programmeurs gemaakt, elk met een eigen versiesysteem.

De opsplitsing in tag-attributen "major" en "minor" werd gemaakt om de impact van de aangebrachte wijzigingen aan te geven. In principe kunnen klassen met hetzelfde major-nummer dezelfde persistentiefaciliteiten gebruiken, aangezien slechts een minder belangrijk ("minor") verschil bestaat tussen hen. In het andere geval geldt dit niet en dient een belangrijk onderscheid gemaakt te worden.
Dit herleidt zich tot de problematiek van versieconflicten, een uitgebreid onderwerp op zich dat ons te ver zou brengen. Daarom wordt er verder niet zoveel aandacht meer aan geschonken en wordt ook het volgende verplicht attribuut achterwege gelaten:

```
public static ClassVersion classVersion=new ClassVersion(<major>,<
    minor>);
```

We dienen wel op te merken dat een persistentietool steunend op aspecten, zoals bijvoorbeeld in hoofdstukken 6 en 7 beschreven, dit attribuut via ITD zou kunnen invoegen.

3.2.3 @persistent

Dit is een marker tag, die aangeeft of een klasse of attribuut persistent moet gemaakt worden. Dit geeft min of meer het omgekeerde aan van het keyword "transient". Gebruik daarvan zou expliciete wijzigingen vereisen IN de broncode zelf (zoals bij JDO), terwijl PDL enkel annotaties maakt BIJ de code. Bovendien kan men klassen niet transiënt verklaren.
Klassen, interfaces en attributen die persistent kunnen gemaakt worden, zullen we voortaan @persistent-klassen, -interfaces en -attributen noemen. Een @persistent-type is een verzamelnaam voor @persistent-klassen en -interfaces.

3.2.3.1 Bij een klasse of interface

```
@persistent
```

Zowel zuivere klassen, desnoods abstract, als interfaces kunnen @persistent gemaakt worden. Waarom dit laatste ook kan/moet, leggen we uit in 5.2.1.2. Eerder zagen we al dat JDO 2.0 dit ook zal ondersteunen in de toekomst.
Anderzijds zijn inner en nested classes ook @persistent-baar. Lokale inner classes (gedefinieerd in functies), kunnen niet zinvol geannoteerd worden.

3.2.3.2 Bij een attribuut

```
@persistent * met * maximaal 1 van volgende mogelijkheden:
a)size="<aantal karakters; default 256>"
b)contained="<false(default)/uni/bi>"
c)impClass="<klassenaam; default java.util.ArrayList of java.util.
    HashMap>"
```

In dit geval zijn er optionele tag-attributen. Bespreking hier zou verwarring teweeg kunnen brengen, omdat de semantiek nauw samenhangt met die van de @content-tag in 3.2.7. Daarom laten we dit voorlopig open. Belangrijk om nu reeds te vermelden, is dat extra type-informatie voor het @persistent-attribuut zélf, hier gespecificeerd wordt. De @content-tag bevat dergelijke info voor de elementen van het @persistent-attribuut indien dit een Collection, Map of array is.

3.2.4 @accessor

@accessor

Deze tag markeert @persistent-attributen die in queries gebruikt mogen worden om toegang te verkrijgen tot @persistent-objecten (zie 6.3.5).

3.2.5 @index

@index

Dit geeft aan dat een @persistent-attribuut best in een index voorkomt om queries te versnellen. In principe kan @accessor dit al aangeven, maar om dit fijner te kunnen regelen, komt @index van pas.

3.2.6 @get en @set

@get <naam getter>
@set <naam setter>

Standaard wordt de JavaBeans-conventie gevolgd om de namen van getters en setters te bepalen [24]. Kort geformuleerd worden de namen van attributen hier als de namen van de properties beschouwd (hoewel ze volgens de specificatie niet altijd hetzelfde zijn), wordt de eerste letter een hoofdletter en wordt er "get", "set" of "is" (bij booleans) voorgeplaatst.

Indien men dit niet wenst, kan men met @get en/of @set een andere naam opgeven. De argumenttypes (als die er al zijn) moeten wel dezelfde zijn als bij de standaardmethoden. Private methoden zijn wel toegelaten. Latere uitbreidingen zouden andere functiesignaturen kunnen toelaten.

3.2.7 @content

Deze tag is een samenraapsel van @compType, @size en @contained. Ook @persistent bevatte enkele elementen hiervan (zie 3.2.3.2 en 3.2.7.1). De invoering van HomogenousCollection en ByteField in [20] hebben we niet gehandhaafd, omdat persistentiebelangen dan opnieuw de implementatie beïnvloeden. Bovendien is hun enige functie het opleggen van constraints (elementtype van Collection of maximale lengte voor String). Zoals in 3.2 reeds gezegd, kunnen geneste aspecten dit beter oplossen.

HomogenousCollection: legt in feite op dat een Collection moet homogeen blijven tijdens het gebruik. Dit kan makkelijker en beter gecontroleerd worden met aspecten of in de toekomst m.b.v. generics in Java (zie hoofdstuk 8).

ByteField: werd ter vervanging van Strings ingevoerd, omdat de Java-codering van Strings 16 bit-karakters gebruikt en relationele databanken 8 bit-karakters. Dit is niet meer zo'n

probleem, aangezien databanksystemen character sets ondersteunen voor UNICODE, zoals UTF8 en UCS-2 bijvoorbeeld. Indien een databank dit niet ondersteunt, kan men er achter de schermen nog een mouw aan passen in de implementor-aspecten (zie 6.3). In beide gevallen is het niet nodig om ByteFields te gebruiken i.p.v. Strings.

3.2.7.1 Attribuut met primitief datatype

Eerst en vooral verduidelijken we wat we precies verstaan onder primitieve datatypes:

- int, long, float, double, boolean, byte, short, char en hun wrappers
- java.lang.String
- java.util.Date

Enkel bij Strings is extra informatie nodig in PDL-tags. ByteFields hielpen immers niet enkel om de overgang tussen 16 en 8 bit-encodering te maken, ze bevatten ook de precieze lengte van de String die ze emuleerden. Bij relationele databanken moeten de datatypes CHAR, VARCHAR en LONGVARCHAR, waarnaar Strings traditiegetrouw gemapped worden, een limietlengte opgegeven krijgen. Het is beter dat men in PDL ook die limiet opgeeft dan dat voor elke String eenzelfde overdreven hoeveelheid plaats gereserveerd wordt in de databank. Bij andere persistentiemethoden is dit niet per se noodzakelijk. Dit is dan ook één van de weinige tags die vooral op relationele databanken gericht is.
Om de tags wat overzichtelijker te houden, werd de @size-tag uit [20] weggelaten en moet deze informatie bij de @persistent-tag gegeven worden (zie 3.2.3.2 geval a)). Dit geldt ook voor arrays van Strings, maar niet voor String-elementen van Collections, Maps of geneste arrays (zie verder).

3.2.7.2 Attribuut met als type een @persistent-type

Dit soort attributen modelleert een relatie tussen objecten van de huidige klasse en van een andere klasse. De precieze aard hiervan kan één van de volgende zijn:

associatie: beide gerelateerde objecten kunnen onafhankelijk van elkaar leven en worden voor bepaalde doeleinden plots geassocieerd met elkaar. Het is duidelijk dat de één op gebied van persistentie niets te zeggen mag hebben over de ander. Dit betekent dat het opslaan, updaten of verwijderen van de ene geen invloed heeft op de andere.

aggregatie: hierbij is het ene object een deel (part) van het andere, maar beide zijn niet onlosmakelijk met elkaar verbonden. De container kan perfect leven zonder het part en omgekeerd. Het part is op zich een zuiver object dat een zinvolle entiteit modelleert, dus niet zomaar een ad hoc groepering van attributen. Op persistentiegebied herleidt dit zich tot hetzelfde gedrag als bij associatie.

compositie: opnieuw een "is deel van"-relatie, maar hier zijn beide delen wél gebonden aan elkaar. Het part heeft geen echte identiteit, d.w.z. dat het slechts een opeenstapeling van attributen is die toevallig samenhoren. Diepere methoden dan getters of setters bestaan meestal niet. Op persistentievlak is duidelijk dat elke actie die men onderneemt op de container ook moet doorgegeven worden op het part. Op persistentieniveau kunnen beide constituenten dus niet onafhankelijk functioneren. De vraag stelt zich dan of persistentie-acties op het part ook consequenties mogen hebben voor de container, of überhaupt toegelaten worden. Het antwoord is niet eenduidig en hangt af van het concrete probleem dat men tracht te modelleren.

Samenvattend kunnen we dus stellen dat er bij relationships op het vlak van persistentie drie aanpakken zijn:

not contained (associatie en aggregatie): persistentie-acties worden niet doorgegeven van de ene geassocieerde aan de andere;

unidirectional (compositie): persistentie-acties worden wel doorgegeven van de container naar het part, maar acties op het part alleen zijn niet toegelaten;

bidirectional (compositie): persistentie-acties worden wel doorgegeven van de container naar het part en omgekeerd.

Om de notaties niet te overladen, werd geopteerd om, analoog aan @size voor Strings, de @contained-tag voor @persistent-attributen bij de @persistent-tag te voegen (zie 3.2.3.2 geval b)). Dit maakt bovendien het in 3.2.3.2 vermelde onderscheid tussen @persistent en @content mogelijk. @persistent-interfaces hebben *geen* implementatieklasse nodig!

Een opmerking is nog nodig i.v.m. de kant van de relatie waar de "contained"-markering moet staan: bij het attribuut van de container of bij de volledige klasse van het part? Intuïtief en qua implementatie lijkt de tweede keuze het best: een klasse wordt dan altijd als "contained" aanzien of niet.
Desondanks werd net als in [20] gekozen voor het eerste. De grootste reden hiervoor is het belang van code reuse. Uit discussies i.v.m. het onderscheid tussen aggregatie en compositie bleek duidelijk dat de modellering van business processen uit de realiteit vaak onderhevig zijn aan de context en tijdsgeest. Men laat sommige dingen weg en belicht andere meer, maar in een ander project blijkt plotseling dat het andersom moet. Dit is menselijk en voor een groot stuk in de hand gewerkt door hiaten in hedendaagse programmeertalen. Daarom is het onnodig om het hergebruik van een klasse (op vlak van persistentie) nog meer te bemoeilijken met het "contained"-stigma. Beter is om bij het gebruik zelf aan te geven of objecten van een specifieke klasse als een composite object gebruikt zullen worden of niet.

In [21] benaderen de auteurs deze problematiek enigszins anders. Ze gebruiken aspecten die de verschillende soorten relaties vertolken, enigszins vergelijkbaar met de aspectversies van de GoF-patterns uit [13]. Een dergelijk systeem vermijdt grotendeels de behoefte aan een declaratieve taal als PDL.

3.2.7.3 Collection-attribuut

Hier en in de volgende twee puntjes gaat het niet meer over aparte attributen, maar over verzamelingen van attributen. Doordat vóór J2SE 1.5 in het Collections-framework niet gespecificeerd kan worden wat de elementtypes zijn, is een taal zoals PDL noodzakelijk om deze gegevens toch te kunnen registreren. Anders zou men geen specifieke veronderstellingen kunnen maken over de types, zodat een mapping naar bijvoorbeeld een relationele databank vrij algemeen moet blijven. Alle elementen zouden als Object behandeld moeten worden, wat nuttige queries fel zou bemoeilijken. Het efficiënt en gericht opvragen van gepersisteerde data is echter één van de pijlers van elke degelijke persistentielaag.
Invoering van generics in Java vanaf J2SE 1.5 verandert de zaak enigszins, zie hoofdstuk 8.

In [20] worden geneste Collections, arrays of Maps in andere Collections, arrays of Maps, niet ondersteund door het PDLF-framework. Maps worden in het geheel niet behandeld. Hoewel dit waarschijnlijk uit implementatie-overwegingen gedaan werd, zou PDL zo algemeen mogelijk moeten zijn en dus niet beïnvloed mogen worden daardoor. Door de syntax en het gebruik van @compType wat aan te passen, kan dit opgelost worden:

```
@content compType="<typenaam>" * met * maximaal één van volgende
    mogelijkheden:
a) size="<aantal karakters; default 256>"
b) contained="<false(default)/uni/bi>"
c) impClass="<klassenaam; default java.util.ArrayList of java.util.
    HashMap>"
```

<typenaam> is het type van de elementen van de Collection, de componenten. Als het één van de volgende zaken voorstelt, dienen nog extra zaken opgegeven te worden:

@persistent-type: men kan opnieuw opgeven welke persistentiestrategie gekozen dient te worden via het contained-tag attribuut (zie 3.2.7.2), nu echter bij de @content-tag (geval b)). In complexe gevallen zoals een Collection van arrays van @persistent-objecten, hebben we natuurlijk niet meer te maken met zuivere compositie, aggregatie of associatie, maar de corresponderende persistentie-acties blijven dezelfde semantiek hebben.

String: de maximale size kan gespecificeerd worden, opnieuw in de @content-tag (geval a)) in plaats van in de @persistent-tag.

Collection- of Map-interface: het impClass-tag attribuut van deze @content-tag (geval c)) kan een specifieke implementatieklasse specificeren. Standaard worden opgeslagen Collections of Maps immers als java.util.ArrayList of java.util.HashMap gereconstrueerd. Dit is niet altijd gewenst, zodat de persistence engineer hier de mogelijkheid krijgt zelf een type op te leggen.

Collection, array of Map: een nieuwe @content-tag moet opgenomen worden NA de huidige om details omtrent de objecten van <typenaam> op te geven. De huidige @content-tag speelt dan de rol van @persistent-tag. Dit systeem laat in principe een oneindige diepte toe van verzamelingen van objecten. Het nut hiervan is natuurlijk afhankelijk van de specifieke situatie: hoe complexer de modellering, hoe complexer de PDL-annotering. In elk geval mag PDL geen onnodige beperkingen opleggen, in tegenstelling tot de persistentietools die

erop zullen steunen. Zij dienen dergelijke constraints duidelijk aan hun gebruikers op te geven, zodat die, naargelang hun applicatie, de meest geschikte tool kunnen selecteren. Belangrijk is hier dat verandering van tool GEEN veranderingen aan broncode vereist: ze steunen allen op dezelfde PDL-annotaties.

We herhalen nog eens dat @content-tags in het algemeen enkel slaan op de elementen van Collections, arrays of Maps. Als de Collection bijvoorbeeld zélf als een interface-type opgegeven werd, kan men ook het concreet type van de gereconstrueerde Collection opgeven. Dit gebeurt echter bij het tag-attribuut impClass van de @persistent-tag (geval c)), NIET in een @content-tag.

Een voorbeeld kan alles wat verduidelijken:

```
/**
 * @persistent impClass="java.util.HashSet"
 * @content compType="java.util.Collection"
 * @content compType="testApp.PersistentClass" contained="false"
 */
private Collection coll;
```

Collection coll wordt best gereconstrueerd als een java.util.HashSet van Collections. Deze laatste dan weer best als java.util.ArrayList's (default-waarde) van testApp.PersistentClass-objecten. De 'contained="false"' is in feite overbodig (het is immers de default-waarde), en slaat vanzelfsprekend op de testApp.PersistentClass-objecten.

3.2.7.4 Map-attribuut

Zoals reeds eerder vermeld, werden deze niet ondersteund in [20]. Hoewel Collection en Map in het Collections-framework totaal niet verwant zijn (het zijn twee aparte hiërarchieën), kan men een Map zien als een speciale Collection met een aparte index (de "key").

```
@content compType="<typenaam_a>" * compTypeKey="<typenaam_b>" ** met
    * maximaal één van volgende mogelijkheden (** analoog, maar enkel
    bij tag attributen met [Key] achteraan):
a) size [Key]="<aantal karakters; default 256>"
b) contained [Key]="<false(default)/uni/bi>"
c) impClass="<klassenaam; default java.util.ArrayList of java.util.
   HashMap>"
```

In principe is het type van de keys vrij, maar Collections, arrays of Maps lijken echter niet aangewezen. Ze bevatten immers maar een bont allegaartje van objecten, die toevallig samenhoren. Deze verzamelingen hebben dus niet echt een eigen identiteit, anders waren ze wel geëncapsuleerd in een apart object. Bovendien is er bij arrays het praktisch probleem dat geen twee rijen dezelfde zijn. I.p.v. een nieuwe representatie in PDL te vinden en alles zo generiek mogelijk te houden, werd beslist om er niet verder op in te gaan. Indien dit echt nodig zou blijken, dient deze situatie later verder onderzocht te worden en de (an)notatie uitgebreid.
Collections, Maps of arrays zijn dus niet mogelijk als keys. Extra @content-tags kunnen dus

enkel slaan op het type van de values. Merk ook op dat het "impClass"-tag attribuut van de @persistent-tag opnieuw aangewend kan worden (zie 3.2.3.2 geval c)) voor de Map zelf.
Een voorbeeld kan alles wat duidelijker maken:

```
/**
 * @persistent
 * @content compType="java.util.Collection" impClass="mypackage.
     MyCollectionImplementation" compTypeKey="Double"
 * @content compType="int[][]"
 */
private Map map;
```

Hier mapt men Doubles op Collections van 2-dimensionale arrays van int's. Merk op dat de implementatieklasse van de Collection hier een eigen concrete implementatie is, en van map zelf de standaardimplementatie (java.util.HashMap).

3.2.7.5 Array

Aangezien een M-dimensionale rij met N-dimensionale rijen als elementen gewoon een (M+N)-dimensionale rij is, kan een rij niet als element van een andere rij optreden. Zij komt dus enkel in de volgende gevallen voor:

- rechtstreeks als attribuut
- als element van Collection of Map

Een rij is een soort statische Map (vaste lengte) die het type van zijn elementen specificeert. Annotaties zijn enkel nodig voor de elementen, omdat alle informatie over de rij zelf gekend is. We slaan als het ware een lijn tags over. Dit wordt duidelijk als we het voorbeeld uit 3.2.7.3 hernemen, nu met een array:

```
/**
 * @persistent impClass="java.util.HashSet"
 * @content compType="testApp.PersistentClass" contained="false"
 */
private Collection[][] coll2;
```

Aangezien de informatie dat coll2 een 2-dimensionale rij is, vervat zit in het Java-type, kunnen we doen alsof we enkel het elementtype annoteren, hier Collection. Indien de array voorkomt als een element van een Collection of Map, zoals in het voorbeeld van 3.2.7.4, dient men wel "[]"'s te gebruiken.

3.2.7.6 Heterogene Collections en Maps

Er werd overal ondersteld dat Collections en Maps homogeen zijn, dus dat alle componenten hetzelfde type hebben. Men kan desnoods, als dit al nuttig is, desnoods werken met een speciale

waarde ”unknown” voor de compType- en compTypeKey-tag attributen van de @content-tags. Dit bemoeilijkt echter deftige query-ondersteuning, omdat de PDL-tags niets bijbrengen over het mogelijke type van de componenten. Later zullen we dan ook geen rekening meer houden met heterogene Collections en Maps.

3.2.8 Weggelaten tags

@unique

Het opgeven van unieke attributen lijkt enkel van pas te komen bij mapping op relationele databanken. Bij ORM wordt echter de voorkeur gegeven aan kunstmatige sleutels, de object identifiers (OID's). Deze hangen niet af van specifieke klassen en kunnen dus voor alle @persistent-objecten gebruikt worden, hetgeen alles sterk vereenvoudigt. Zo kunnen query-methoden, los van welke object men opvraagt, alle hetzelfde type OID gebruiken.
Dit neemt niet weg dat andere attributen inderdaad uniek dienen te zijn, bijvoorbeeld studentennummers. Het is echter niet de taak van de persistentielaag om dit te controleren of op te leggen, dit moet deel uitmaken van de business logica. Het gebruik van AOP bij dergelijke kwesties kan zeker zijn vruchten afwerpen.

@state

Dit vervangt @persistent bij Restricted Abstract Data Types (RADT's). Dit zijn ADT's waarvan de attributen primitieve Java-types (of hun wrappers) hebben of zelf RADT's zijn. Zij hebben een interne en een externe representatie, waarvan enkel de laatste naar buiten toe zichtbaar is. In 2.2.1 worden bij het opslaan en inladen van een RADT-object resp. de volgende functies uitgevoerd:

```
public void fromExternal(...)
public void fromInternal(...)
```

Indien deze aanwezig zijn, passen ze de geëncapsuleerde interne representatie of de “zichtbare” externe vorm aan met de data van de andere. In het voorbeeld dat M. Matar geeft in [20], toont een RADT Name zowel voor- en achternaam naar buiten toe (extern). Intern wordt echter enkel de volledige naam bijgehouden. Bij het opslaan past men de volledige naam aan met de info uit de twee afzonderlijke delen en bij het inladen voltrekt zich het omgekeerde.

Ofschoon deze handelswijze waarheden bevat, denken we dat deze functionaliteit in de business logic zelf vervat moet zijn en niet zozeer in de persistentielaag. In principe komt het erop neer dat men bij het ontwerp al in de setters voor de volledige naam ook de andere twee attributen aanpast en vice versa, anders heeft men sowieso inconsistente datastructuren die naast elkaar bestaan. Het is niet aan de persistentielaag om dergelijke ontwerpsbeslissingen te controleren en te handhaven.

3.3 Kanttekening

Het toepassen van javadoc als mechanisme om persistentie-informatie te declareren, kan vragen oproepen. De PDL-tags zitten dan immers verweven in de broncode. Moet de programmeur zich dan toch bezighouden met persistentie? Nee, ten eerste is het de persistence engineer die de PDL-tags aanbrengt. Bij wijzigingen in de broncode moet hij controleren of de PDL-tags aangepast dienen te worden. Ten tweede is javadoc gewoon het meest handige vehikel om de PDL-tags uit te drukken. Enkel omwille van technische redenen zijn javadoc-tags in de broncode zelf opgenomen. Er kon evengoed een soort AOP-javadoc bestaan hebben, waar de tags in een afzonderlijk bestand zitten en met een predicaat op de goeie attributen gemapped worden. Ten derde is het zo dat javadoc-informatie NIET in de class-bestanden opgenomen wordt, daar het enkel een speciaal soort commentaar is. Implementatie en tags zijn dus onafhankelijk: als men het één verandert (bijv. de code), dan hoeft men het ander niet noodzakelijk te bewerken (transformeren naar bijv. een XML-bestand), en omgekeerd.

3.4 Voorbeeld

Fig. 3.1 toont de geannoteerde code van één van de casestudy-klassen, namelijk Course. We zien enkele String-attributen, waarvan één een maximale lengte krijgt. Attribuut “prerequisites” bevat andere Course-objecten en stelt een associatie voor met meervoudige multipliciteit.

Bij JDO is geen annotatie met tags nodig, maar gebruikt men een persistence descriptor (Fig. 3.2). Ook hier zien we dat het type van de elementen van een collectie opgegeven kan worden. Dit is niet verplicht, en desnoods kunnen alle elementen als Object beschouwd worden. JDO-implementaties zijn niet verplicht dit vangnet te ondersteunen, zodat dit gedrag portabiliteitsproblemen oplevert. Het is dus toch aangeraden expliciet de nodige informatie op te geven.

```
/**
 * @persistent
 * @pversion major="01" minor="00"
 */
public class Course {
  /**
   * @persistent size="6"
   * @accessor
   */
  private String courseID;
  /**
   * @persistent
   * @accessor
   */
  private String courseName;
  /**
   * @persistent
   * @accessor
   */
  private int courseWeight;
  /**
   * @persistent
   * @accessor
   * @content compType="casestudy.Course" contained="false"
   */
  private Collection prerequisites;
  ...
}
```

Figuur 3.1: Broncode van de klasse Course.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="casestudy">
    <class name="Course">
      <field name="prerequisites">
        <collection element-type="casestudy.Course"/>
      </field>
    </class>
  </package>
</jdo>
```

Figuur 3.2: Persistence descriptor voor de klasse Course.

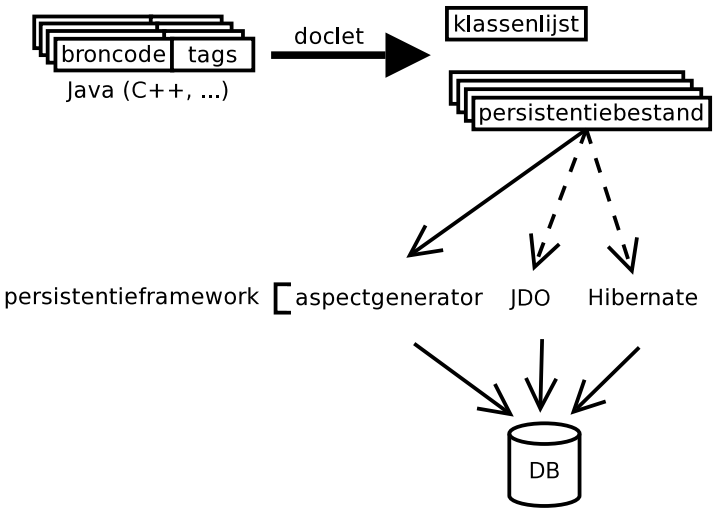
Hoofdstuk 4

Stap 2: Persistentiebestand

4.1 Doel

Nadat alle @persistent-types voldoende geannoteerd zijn met PDL-tags, kan men met een speciale doclet de informatie die vervat zit in zowel de code zelf als de tags, transformeren naar een XML-bestand, het persistentiebestand genaamd, één per @persistent-type. Als een bijproduct wordt ook een zogenaamde klassenlijst gegenereerd die de namen bevat van alle @persistent-types.
De persistentiebestanden hebben als doel een generieke beschrijving te geven van het persistentiegedrag van @persistent-types, los van welk persistentiemechanisme ook. Of men nu werkt met file-based opslag, relationele databanken of serialisatie, de persistentiebestanden dienen alle nodige informatie te bevatten. Het ultieme doel is het koppelen van bijvoorbeeld JDO of Hibernate aan dit systeem. Dit zou bewijzen dat persistentiebestanden voldoende generiek zijn, maar werd niet nader bekeken.
Omgekeerd zou ook onderzocht kunnen worden in hoeverre men code geschreven in C++, C#, Smalltalk, ... kan annoteren met PDL-tags, met als achterliggend idee eenzelfde structuur voor het persistentiebestand te bekomen. Dit is wellicht niet onvoorwaardelijk mogelijk, maar zou nieuwe perspectieven kunnen openen, zoals generieke persistentielagen voor alle talen. We hebben immers al opgemerkt in 2.3.1 dat bij AOP de basistaal en de taal van het advice en de condities niet noodzakelijk dezelfde hoeven te zijn.

In tegenstelling tot [20], waar de broncode samen met het XML-bestand al het nodige beschreef voor persistentie, volstaat hier het persistentiebestand alleen. Dit laat toe dat men tools kan bouwen die als input slechts het persistentiebestand nodig hebben, hetgeen iets minder prohibitief is dan de eis dat ook de bytecode nodig is.
Wat heeft dit voor nut? De interface tussen het business model enerzijds en de persistentietools anderzijds wordt heel duidelijk afgelijnd. Door enkel XML-bestanden als input nodig te hebben, vermijdt men problemen zoals het synchroon houden van class- en persistentiebestand, verlenen van rechten om reflectie toe te mogen passen, ... Aan de kant van de class-bestanden kan men ongehinderd obfuscators gebruiken (na generatie van XML), ... Er is dus een veel zwakkere koppeling nodig tussen beide kanten, wat elk een grotere flexibiliteit en onafhankelijkheid toelaat.



Figuur 4.1: Overzicht van stappen 1 en 2.

Daar goed klassenontwerp dit vereist, zijn de meeste (alle?) attributen, inner classes, ... geëncapsuleerd. D.w.z. dat ze de facto private of protected zullen zijn. Hier kan men niet onder uit. Persistentie-tools zullen tijdens hun werking dus toegang moeten zien te verkrijgen tot deze normaliter verborgen data.
Het persistentiebestand verbreekt nu deze encapsulatie, want men kan in XML immers geen modifiers gebruiken. Dit is echter niet zo erg, aangezien het enkel de ontwikkelaars, in feite enkel de persistence engineer, zijn die over de persistentiebestanden zullen beschikken en ze zullen gebruiken. Ze worden dus niet opgenomen in het afgewerkte product. Aangezien javadoc-tags niet weerhouden worden in class-bestanden, zijn de persistentiebestanden ook niet regenereerbaar. Er hoeft dus niet gevreesd te worden voor misbruiken van PDL voor reverse engineering-doeleinden. Bovendien is de functionaliteit van het business model zelf nog altijd verborgen.

4.2 Beschrijving van tags

De semantiek van deze XML-tags is sterk gelijkend op die van de PDL-tags, zodat weinig extra uitleg nodig is. In wezen weerspiegelt een persistentiebestand dan ook de in PDL opgegeven metadata aangevuld met type-informatie van de @persistent-types zelf. Enkele voorbeelden in 4.2.3 en 4.5 zullen alles nog verduidelijken. De onderverdeling in subsecties weerspiegelt de hiërarchie in het XML-bestand.

4.2.1 classUnit

```
<classUnit name="<klassenaam van container>"> ... </classUnit>
```

Deze tag is de root van elk persistentiebestand. Een onderliggende veronderstelling is dat elk

bronbestand één klasse beschrijft (de container) die zélf inner en nested classes kan bevatten. De persistentie-eigenschappen van die inner en nested classes komen in het persistentiebestand van de container terecht, tenminste als ze @persistent zijn. Dit is logisch, aangezien er bij het ontwerp bewust gekozen werd om ze sterk afhankelijk te maken van elkaar. Het geheel van @persistent-types uit één bronbestand noemen we de "class unit".

4.2.2 persistentClass en persistentInterface

```
<persistentClass name="<klassenaam>" extends="<klassenaam; default
    java.lang.Object>" role="<container/inner/nested>"> ... </
    persistentClass>
```

Voor elke @persistent-klasse horend bij de huidige class unit volgt de volledige persistentiebeschrijving in deze block tag. De 'role' geeft aan wat de precieze rol is van <klassenaam> in de huidige class unit. Voor @persistent-interfaces geldt iets analoogs:

```
<persistentInterface name="<interfacenaam>"[ extends="<interfacenaam
    >"] role="<container/inner/nested>"> ... </persistentInterface>
```

4.2.2.1 db

```
<db database="<naam databank>" table="<naam tabel>"/>
```

Dit volgt triviaal uit de PDL-tags.

4.2.2.2 pversion

```
<pversion major="<hoofdversie>" minor="<nevenversie>"/>
```

Analoog aan het vorige.

4.2.2.3 implements

```
<implements><interfacenaam></implements>
```

Dit is een block tag die op het eerste zicht overbodig lijkt. Interfaces kunnen in Java geen toestand bevatten, zodat een @persistent-interface dan enkel een speciaal gemarkeerde interface is. Met aspecten kan men echter attributen introduceren in interfaces, zodat die plots wel toestand kunnen bevatten. Helaas kan men geen PDL-tags injecteren met javadoc, omdat die enkel in broncode bewaard blijft. De Metadata Facility uit J2SE 1.5 zou dit dan weer wél kunnen, mits aanpassingen in AspectJ (die er aan komen). Conclusie is dus dat deze tag voor de hedendaagse technologie net te vroeg komt, maar niet voor lang meer.

4.2.2.4 persistentAttribute

```
<persistentAttribute name="<attribuutnaam>" accessor="<true/false
    >"> ... </persistentAttribute>
```

Dit is opnieuw duidelijk af te leiden uit de PDL-tags.

get Opnieuw vanzelfsprekend.

```
<get><naam getter></get>
```

set Analoog.

```
<set><naam setter></set>
```

content Dit is in feite een één-op-één transformatie van de @content-tags uit de PDL-beschrijving. Het enige verschil is dat een <content>-tag met de info over het @persistent-attribuut zelf ervoor werd gezet. Deze metadata wordt bekomen uit de @persistent-tag en de Java-code zelf. De daaropvolgende <content>-tags behandelen de (eventuele) componenten van dit @persistent-attribuut. Enkele voorbeelden in 4.2.3 zullen dit verduidelijken. Merk ook op dat het type van arrays hier gescheiden wordt van hun dimensie, wat te zien is aan het array-tag attribuut.

```
<content[ array="<dimensie rij>"] compType="<type van @persistent-
    attribuut>"[ impClass="<klassenaam>"][ size="<max. lengte van
    String>"]/>
<content[ array="<dimensie rij>"] compType="<component-type>"[
    impClass="<klassenaam>"][ size="<max. lengte van String>"][
    compTypeKey="<type van Map-key>"][ sizeKey="<max. lengte van
    String>"]/>
...
```

4.2.3 Enkele voorbeelden

De voorbeelden uit 3.2.7.3, 3.2.7.4 en 3.2.7.5 leiden tot volgende fragmenten:

```
<persistentAttribute name="coll" accessor="false" index="false">
  <get>getColl</get>
  <set>setColl</set>
  <content compType="java.util.Collection" impClass="java.util.
      HashSet"/>
  <content compType="java.util.Collection" impClass="java.util.
      ArrayList"/>
  <content compType="testApp.PersistentClass" contained="false"/>
</persistentAttribute>
```

```
<persistentAttribute name="map" accessor="false" index="false">
  <get>getMap</get>
  <set>setMap</set>
  <content compType="java.util.Map" impClass="java.util.HashMap"/>
  <content compType="java.util.Collection" impClass="mypackage.
      MyCollectionImplementation" compTypeKey="Double"/>
  <content array="2" compType="int"/>
</persistentAttribute>
<persistentAttribute name="coll2" accessor="false" index="false">
  <get>getColl2</get>
  <set>setColl2</set>
  <content array="2" compType="java.util.Collection" impClass="java.
      util.HashSet"/>
  <content compType="testApp.PersistentClass" contained="false"/>
</persistentAttribute>
```

Het derde voorbeeld is inderdaad identiek aan het eerste op de tweede content-tag en het array-tag attribuut na.

4.3 Klassenlijst

Dit is een XML-bestand dat eenvoudigweg de namen van alle @persistent-types, zowel containers als inner en nested classes, op een geordende manier bevat. In principe is dit niet iets noodzakelijks, daar het scannen van alle aanwezige persistentiebestanden dezelfde informatie kan opleveren. Bij generatie van de persistentiebestanden kan het echter vrij makkelijk als een residu bekomen worden.

Een eenvoudig voorbeeld:

```
<root>
  <classUnit name="mypackage.ClassOne">
    <class role="container">mypackage.ClassOne</class>
    <class role="inner">mypackage.ClassOne.InnerClass</class>
    <class role="nested">mypackage.ClassOne.NestedClass</class>
  </classUnit>
  <classUnit name="mypackage.ClassTwo">
    <class role="container">mypackage.ClassTwo</class>
    <interface role="nested">mypackage.InterfaceOne</interface>
  </classUnit>
</root>
```

4.4 Implementatie

Bij het maken van de doclet en van de aspecten in volgend hoofdstuk, werd gebruik gemaakt van Java Emitter Templates (JET). Dit is een generieke template engine die deel uitmaakt van het Eclipse Modelling Framework (EMF)[1]. Werking is vrij eenvoudig:

- Men kan volgens een subset van de JSP-syntax opgeven hoe een XML-bestand, aspect, ... er moet uitzien.
- JET maakt dan a.h.v. een skeletbestand een Java-klasse dat als enige taak het genereren van de gewenste output heeft.
- Elders, in de doclet bijvoorbeeld, kan men dan een object van de gegenereerde klasse creëren, desnoods extra informatie verschaffen (ClassDoc-objecten, XML-nodes, ...) en een generate-methode geeft de volledige inhoud terug van bijvoorbeeld een XML-bestand. Deze kan men dan zelf in een bestand opslaan, analyseren, ...

In wezen heeft dit dezelfde voordelen en nut als JSP's t.o.v. servlets.

4.5 Voorbeeld

Fig. 4.2 toont het persistentiebestand van de klasse Course. Dit is inderdaad makkelijk af te leiden uit Fig. 3.1. JDO heeft niets vergelijkbaars.

[1] http://www.eclipse.org/emf/

```
<?xml version="1.0" encoding="UTF-8"?>
<classUnit name="casestudy.Course">
<persistentClass name="casestudy.Course" extends="java.lang.Object"
    role="container">
  <db database="" table=""/>
  <pversion major="01" minor="00"/>
  <persistentAttribute name="courseID" accessor="true" index="false">
    <get>getCourseID</get>
    <set>setCourseID</set>
    <content compType="java.lang.String" size="6"/>
  </persistentAttribute>
  <persistentAttribute name="courseName" accessor="true" index="false
    ">
    <get>getCourseName</get>
    <set>setCourseName</set>
    <content compType="java.lang.String" size="256"/>
  </persistentAttribute>
  <persistentAttribute name="courseWeight" accessor="true" index="
    false">
    <get>getCourseWeight</get>
    <set>setCourseWeight</set>
    <content compType="int"/>
  </persistentAttribute>
  <persistentAttribute name="prerequisites" accessor="true" index="
    false">
    <get>getPrerequisites</get>
    <set>setPrerequisites</set>
    <content compType="java.util.Collection"  impClass="java.util.
      ArrayList"/>
    <content compType="casestudy.Course"  contained="false"/>
  </persistentAttribute>
</persistentClass>
</classUnit>
```

Figuur 4.2: Persistentiebestand van de klasse Course.

Hoofdstuk 5

Stap 3: Aspectgenerator

5.1 Inleiding

Nu alle persistentiegerelateerde informatie voorhanden is, kunnen we ons richten op een specifieke implementatie voor de persistentielaag, gebaseerd op aspecten en relationele databanken. Zoals in 2.3.2 vermeld, bouwen we verder op het voorstel van K. Vandenborre, M. Matar en G. Hoffman uit [25].

5.2 Onderdelen

5.2.1 PersistenceIntroducer-aspect

Dit aspect introduceert de geëxtraheerde persistentiecode in het business model, hetgeen correspondeert met de zwarte pijl uit Fig. 1.1.

5.2.1.1 Werkwijze

Eerst en vooral genereren we het introducer-aspect. Dit doet het door alle klassen vermeld in de klassenlijst, via ITD de interface Persistent te laten implementeren en default-implementaties te voorzien voor abstracte methoden. De bewuste klassen weten immers niets af van Persistent en diens methoden. Die omvatten alle mogelijke persistentie-acties, namelijk opslaan, opvragen, verwijderen en wijzigen.
Ontwikkelteams kunnen dus al werken met @persistent-enablede klassen en interfaces voordat implementor-aspecten beschikbaar zijn. Merk op dat er in tegenstelling tot het oorspronkelijke voorstel uit [25], gekozen werd om Persistent niet leeg te maken, maar er effectief alle persistentiemethoden in te zetten. Dit maakt de interface kenbaar en bruikbaar in parallelle ontwikkelteams, die niet noodzakelijk zélf met aspecten moeten werken.

Dit laatste lijkt eerder een technische zaak, maar AOP is een relatief nieuwe technologie, wiens implementaties (AspectJ, AspectWerkz[1], ...) nog niet zo lang bestaan. Tool support is dus nog niet zo uitgebreid en ondersteuning door de gewone Java-tools is nog niet optimaal. Het zo laat mogelijk in het ontwikkelproces inschakelen van automatisch gegenereerde aspectcode (die op zich correct werkt) kan een grote ontlasting zijn. Bovendien zijn gewone compilers (nog?) altijd sneller dan weavers, zodat onnodig gebruik hiervan best vermeden wordt.

5.2.1.2 Wat met interfaces?

Het gebruik van een volwaardige Persistent-interface heeft ook nog een ander gevolg. In 3.2.3.1 hebben we reeds vermeld dat interfaces ook als @persistent kunnen aangeduid worden. Dit lijkt onlogisch, omdat interfaces geen toestand bevatten en enkel een bepaald gedrag opleggen. Het taggen van een interface zou dan niet duiden op het zuiver geven van aanwijzingen over de interface zelf, maar eerder op het opleggen van bepaalde metadata aan klassen die de interface implementeren. Als een laatste argument kunnen we aanhalen dat er soortgelijke problemen kunnen aantreden als bij meerdere overerving in C++. Een klasse die zelf niet @persistent is (anders kan men het makkelijk oplossen), kan twee interfaces implementeren die elk wél @persistent zijn. Als die allebei verschillende waarden bevatten voor de @db- en @pversion-tags, dient men prioriteiten toe te kennen. Dit vermijden we best.

Daarom werd ervoor geopteerd om enkel de @persistent-tag zonder meer toe te laten bij interfaces. Het nut daarvan kan men inzien a.h.v. de volgende drie invalshoeken:

pragmatisch standpunt: het gebruik van interfaces voor attributen laat meer code reuse toe. Men is dan immers niet gebonden aan specifieke implementatieklassen. Door nu zo'n interface te taggen, is men later in staat voor *alle* implementatieklassen een default persistent gedrag te specificeren. Enkel zij die zelf ook PDL-tags bevatten, zullen dan op een nuttige manier gepersisteerd worden. Net zoals bij gewone klassen, bevatten getaggede interfaces hun core concern (methoden) en declaratieve aanwijzingen voor persistentie (PDL-tags).

praktische problemen: inner of nested classes van interfaces moeten ook getagged kunnen worden. Dit vereist het taggen van de omringende interface. Dit is gerechtvaardigd door de grote verbondenheid tussen deze modules[2].

ITD: via ITD kan men nieuwe attributen aan klassen en interfaces toekennen. Soms kan het nuttig zijn deze ook @persistent te maken. Dit heeft geen zin als interfaces zelf niet @persistent kunnen zijn. In 8.3 gaan we zien welke problemen er nog optreden in dit verband.

Het blijkt dat interfaces in JDO 1.0 automatisch enhanced worden. Hier moet men geen metadata voor opnemen in de persistence descriptor. In JDO 2.0 zal dit wel nodig zijn.

[1] http://aspectwerkz.codehaus.org/

[2] Bij gewone klassen is dit ook zo. Indien men absoluut niet wenst dat de allesomvattende klasse zelf gepersisteerd kan worden, hoeft men enkel de klasse zelf @persistent te maken. De attributen zelf blijven dan transiënt.

5.2.2 Implementor-aspecten

Eens de implementatie van het business model voldoende stabiel is en er een databank voor handen is, kan men uit de persistentiebestanden zogenaamde implementor-aspecten genereren. Deze bevatten specifieke implementaties voor alle persistentiemethoden, dus met SQL-code (zie de zwarte rechthoeken op Fig. 1.1). Alle persistentie-basisfunctionaliteit is nu beschikbaar. Ook wordt automatisch een script gemaakt om tabellen aan te maken of te verwijderen in de databank.

Om geavanceerdere features te kunnen gebruiken, kan men de basis-aspecten zoals ConnectionAspect uitbreiden of zelf nieuwe maken. Zo kan men complexere caches, fijner transactiemanagement, crash recovery, concurrency control, ... toevoegen. Elk bouwt verder op de reeds aanwezige aspecten.

Beperkingen Vooraleer uit te leggen wat wel kan en hoe dit gebeurt, kan het nuttig zijn om de beperkingen op te geven waaraan we ons gehouden hebben:

Java-types: enkel primitieve types (en wrappers), String, Collection, Map en @persistent-types.

no-arg constructor: dit is verplicht.

getters en setters: de opgegeven getters en setters moeten dezelfde signatuur hebben als de JavaBean-getters en -setters. Naamgeving zelf mag anders zijn, maar dit was al geëist bij de PDL-tags (zie 3.2.6).

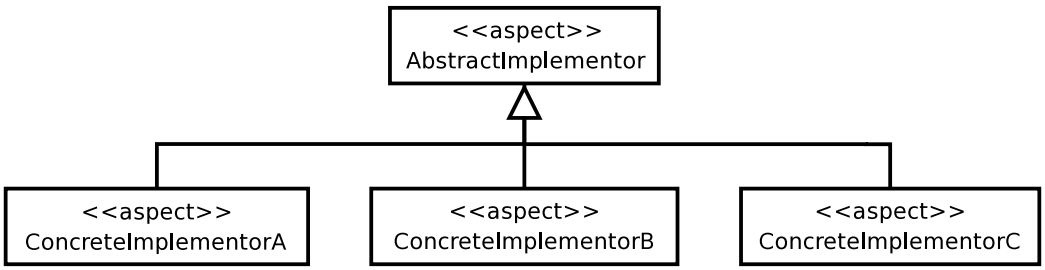
single-user: hoewel sommige basis-aspecten meerdere gebruikers aan zouden kunnen, hebben we dit absoluut niet getest. Ik vermoed dat een doordachte keuze en constructie van enkele extra aspecten gedistribueerde en multi-user werking mogelijk kan maken (zie ook hoofdstuk 7). Dit is echter een uitgebreid onderwerp op zich en ligt dus buiten de scope van dit werk.

compositie: in 3.2.7.2 hebben we drie mogelijke gedragingen gezien op vlak van persistentie, nl. “not contained”, “unidirectional” en “bidirectional”. Enkel de eerste twee zijn geïmplementeerd, omdat dit de belangrijkste zijn en ze doorgaan voor “best practice”. Gebruik van ”bidirectional” is in feite afgeraden, maar kan in sommige situaties toch nodig zijn, afhankelijk van het specifieke probleem.

versie-ondersteuning: zoals al vermeld in 3.2.2, werd dit ook genegeerd. In principe zou men met een custom class loader kunnen werken die de goeie versies van de geweven business model-code laadt. Run-time of load-time weaving kan misschien ook uitkomst bieden.

static attributen: hiermee werd geen rekening gehouden. Hier speelt immers de vraag wie wanneer wat mag opslaan, laden of verwijderen. De momenten waarop klassen ingeladen worden (door de class loader) en uit de JVM verdwijnen lijken geschikte momenten voor resp. laden en opslaan, maar hierbij komt nog veel meer kijken.

heterogene collecties: aangezien het nut hiervan niet volledig duidelijk is, werd dit niet geïmplementeerd. Een mogelijke oplossing is misschien het mappen van de elementen naar



Figuur 5.1: Ideale structuur voor implementor-aspecten.

Binary Large Objects (BLOB's). Dit kan nadelige gevolgen hebben voor queries (veel overhead), en bij gewone persistentie-acties. Alle mogelijke objecten die verbonden zijn met de collectie-elementen worden immers meegeblobbed.

performantie: deze implementatie zal niet ogenblikkelijk door IBM opgekocht worden om in productie te brengen, vandaar dat vooral conceptueel gewerkt werd. De ORM-mapping verschilt lichtjes van die uit [20] en is volgens het relationele model opgebouwd, wat niet per definitie leidt tot de grootste efficiëntie. In 5.4 wordt wel uitgelegd hoe men een mapping kan aanpassen of zelfs vervangen.

SQL-92: ondanks het feit dat [20] en [21] at run-time SQL genereren, maken ook zij geen gebruik van geavanceerde JDBC- en SQL-features. Het doel is immers om zo generiek mogelijk te zijn voor zoveel mogelijk databanken. Onze implementor-aspecten zullen ook SQL-92 bevatten en dus onafhankelijk zijn van vendor-specifieke uitbreidingen.

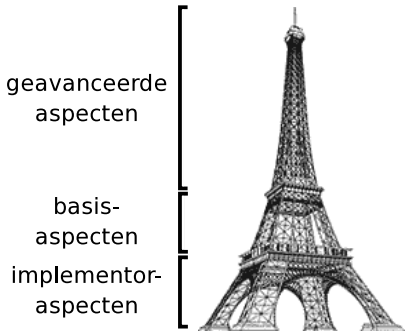
5.2.3 Basis- en geavanceerde aspecten

De implementor-aspecten zijn onderling volledig onafhankelijk. Fig. 5.1 toont een veel elegantere structuur die echter niet gecreëerd kan worden. AspectJ is immers niet geavanceerd genoeg op vlak van overerving:

- aspecten kunnen enkel overerven van abstracte aspecten;
- een abstract aspect kan enkel pointcuts als abstract bestempelen, advice niet;
- abstracte pointcuts moeten precieze types voor hun argumenten hebben. Men kan in Fig. 5.1 niet zeggen dat het abstracte aspect een abstract pointcut heeft met een argument van het type Persistent om dan in een concreet implementor-aspect het type te preciseren tot een bepaalde @persistent-klasse. Bij het opslaan, verwijderen en updaten van een object moet men echter wel het precieze type kennen.

We gooien het dus over een andere boeg, en genereren losstaande aspecten die met glue-aspecten geconnecteerd worden. Nog andere aspecten voorzien geavanceerdere faciliteiten. We onderscheiden dus twee soorten uitbreidingsaspecten:

basis-aspecten: deze zorgen dat doorheen de volledige afhandeling van elke persistentiemethode eenzelfde databankconnectie gebruikt wordt (ConnectionAspect van 7.2.1) en dit



Figuur 5.2: Conceptueel beeld van de volledige persistentielaag die we bekomen.

alles in een transactie opgenomen wordt (TransactionAspect uit 7.2.2). Queries worden verzorgd door QueryAspect (zie 7.2.3). Dit zijn de glue-aspecten.

geavanceerde aspecten: dit zijn uitbreidingen van de functionaliteit, zoals het groeperen van verschillende methoden in een supertransactie (SuperTransactionAspect van 7.3.1), implementatie van een cache (CacheAspect in 7.3.3), gedistribueerde werking (DistributedAspect uit 7.3.4), ...

Alles tesamen bekomen we een structuur zoals op Fig. 5.2 te zien is[3].

5.3 O/R-mapping (ORM)

In tegenstelling tot bijvoorbeeld Hibernate bevat PDL standaard geen tags die specificeren hoe klassen precies op tabellen afgebeeld kunnen worden. Dit is een gevolg van PDL's claim om generiek te zijn, dus los van één specifiek persistentiemechanisme. PDL is echter uitbreidbaar, dus niets belet de persistence engineer om nieuwe tags toe te voegen om toch wat meer controle te hebben.
De verschillende persistentietools die op basis van persistentiebestanden hun werk doen, zijn in principe vrij om hun eigen mapping in te voeren en te gebruiken (zie 5.4). Wenst men de structuur van een legacy-databank te hergebruiken, dan kan dit. Bovendien heeft men ook de vrijheid om voor eenzelfde mapping verschillende aspecten te genereren (zie wederom 5.4).

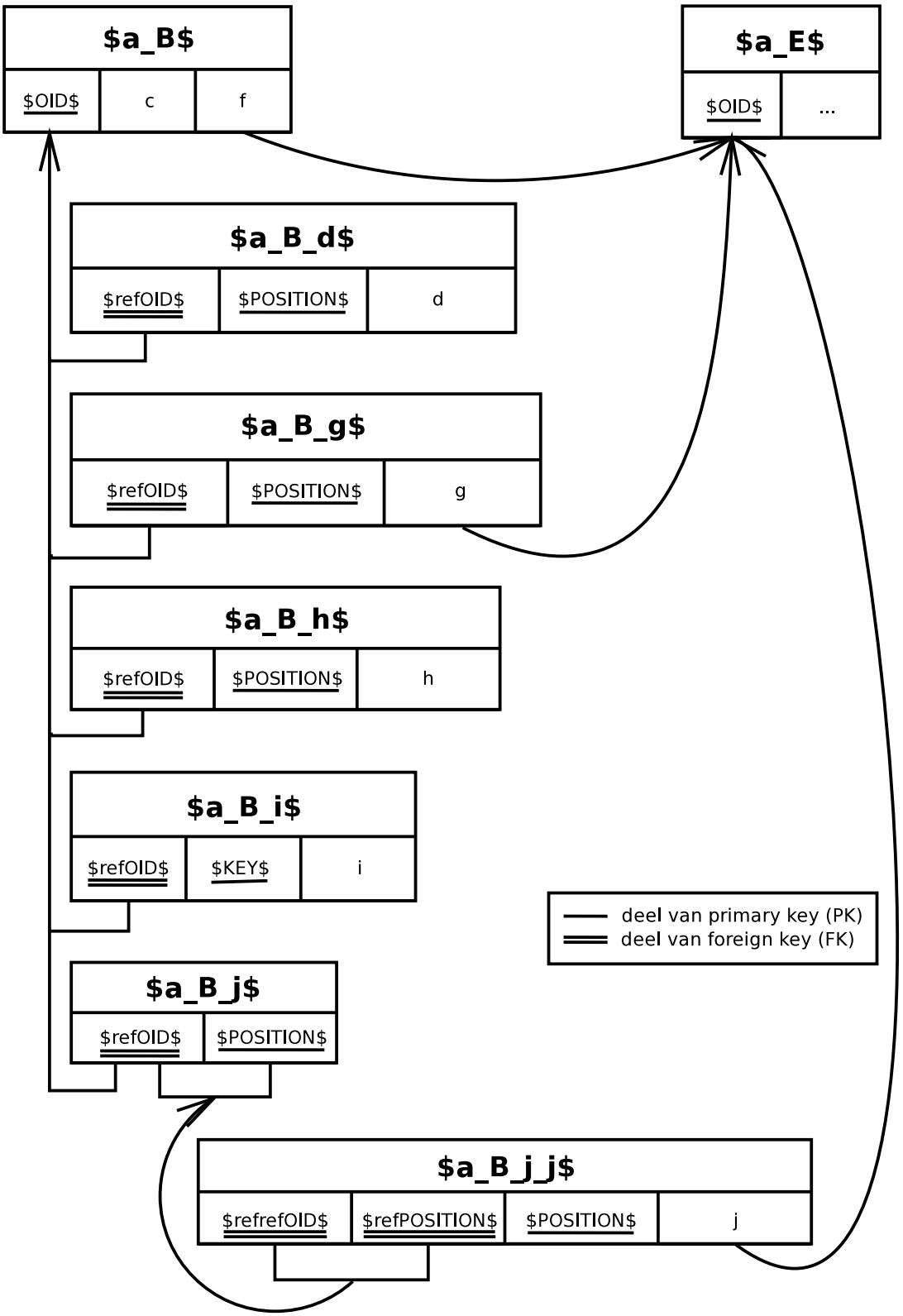
Principieel heb ik dezelfde mapping-strategie gebruikt als M. Matar in [20], met enkele kleine wijzigingen. Dit kan daarom beter met een uitgebreid voorbeeld geïllustreerd worden. De voorbeeldklasse uit Fig. 5.3 wordt zo gemapped op Fig. 5.4.

Deze mapping is eerder conceptueel dan echt efficiënt, want er werd zoveel mogelijk geprobeerd de relationele theorie te volgen. Men kan dit bijvoorbeeld zien aan de manier waarop array-, Collection- en Map-tabellen verwijzen naar de tabel van hun eigenaar: hoe dieper in de tabellenhiërarchie, hoe meer kolommen nodig zijn voor de foreign keys. Is men bereid het gebruik van joins op te offeren en zo hergebruik van de databank in het gedrang te brengen, dan kan men al die kolommen telkens samennemen in één enkele sleutel om plaats te besparen. Dit zijn echter keuzes die men moet afwegen als er commerciële belangen in het spel zijn.

[3]De figuur is afkomstig van http://www.tour-eiffel.fr/teiffel/uk/documentation/structure/page/structure.html.

```
package a;
public class B{
  /**@persistent*/
  private int c;
  /**@persistent*/
  private int[] d;
  /**@persistent*/
  private E f;
  /**@persistent*/
  private E[] g;
  /**@persistent
   * @content compType="java.lang.Double"*/
  private Collection h;
  /**@persistent
   * @content compType="java.lang.Integer" compTypeKey="java.lang.
       String"*/
  private Map i;
  /**@persistent
   * @content compType="a.E"*/
  private Collection[] j;
  ...
}
```

Figuur 5.3: Voorbeeldklasse met PDL-tags.



Figuur 5.4: Mapping van de klasse a.B naar een relationele databank.

Implementatie van deze mapping gebeurt via een TableComponent-hiërarchie.

5.4 TableComponent-hiërarchie

De tabelstructuur van Fig. 5.4 bevat sterke gelijkenissen met een composite-structuur [11]. Bovendien wensen we ook makkelijk voor eenzelfde mapping andere aspect-implementaties te kunnen maken. Dit leidde tot het idee om te werken met een composite-hiërarchie met visitors [11]. Generatie van aspecten gebeurt dan in twee fasen (zie ook Fig. 5.5):

1. Opbouwen van een composite bestaande uit TableComponent's die de O/R-mapping naar de relationele databank voorstelt.
2. Verschillende visitors overlopen de boomstructuur, elk met een ander doel: het genereren van de code voor 6.3.3, 6.3.4.1 en 6.3.5.2, creatie van een script om de nodige tabellen in de databank te creëren en te verwijderen, ...

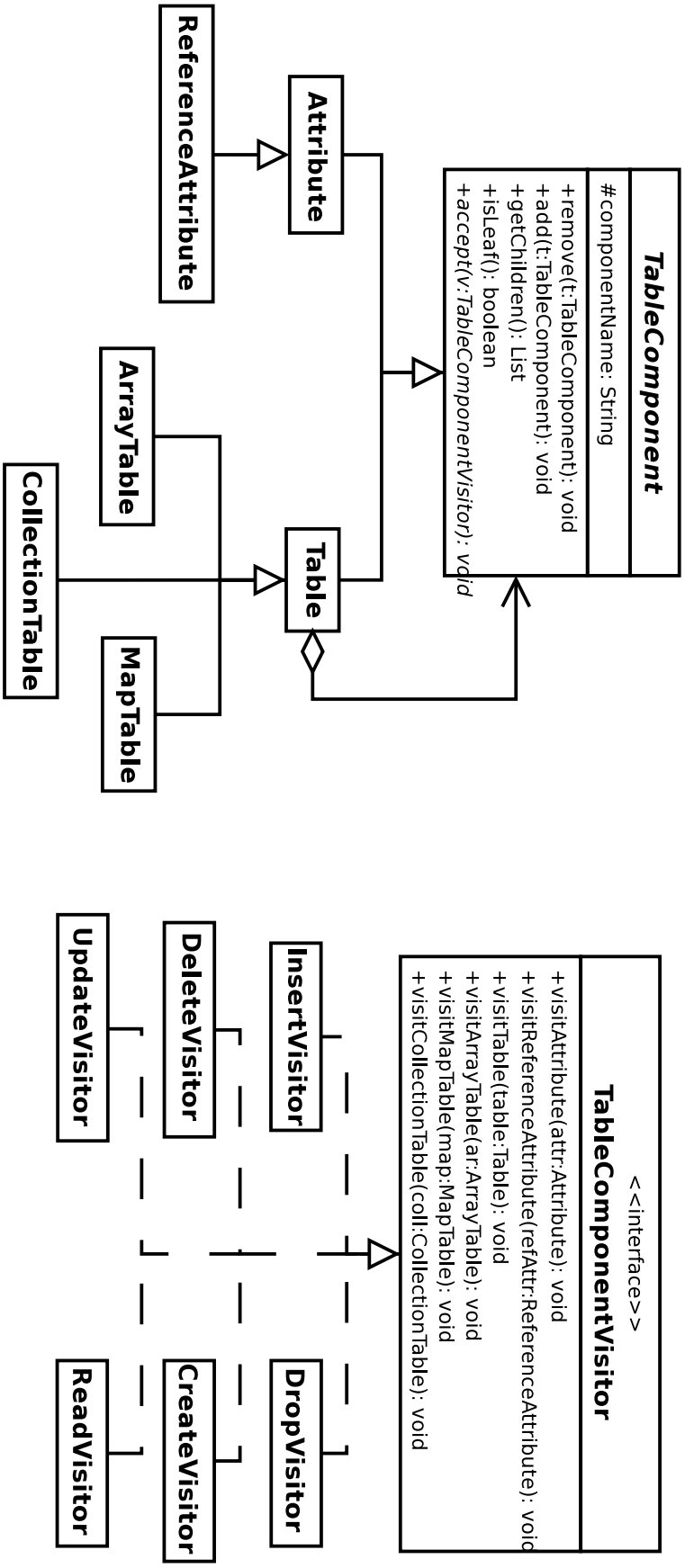
In Fig. 5.5 zien we een gedetailleerdere kijk op de klassen die in de proefimplementatie meespelen. De namen van de TableComponent-derivaten weerspiegelen hun betekenis en de visitors verraden ook in hun naam welke code ze genereren (zie hiervoor 6.3.3, 6.3.4.1 en 6.3.5.2). De canonieke tabelnamen van Fig. 5.4 kunnen heel lang worden. Daarom krijgen tabellen in praktijk een alias, tenzij indien expliciet een naam opgegeven werd in de PDL-tags.

Het spreekt voor zich dat de TableComponent-hiërarchie alle details van de gekozen mapping in zich draagt en dat de bijhorende visitors niet meteen bruikbaar zijn voor andere mappings. Aanpassing of creatie van een composite-structuur levert echter een nieuwe mapping op. Bij eenzelfde mapping kan men echter wel verschillende visitors maken, die bovendien niet per se aspecten moeten genereren. Dit alles leidt tot een flexibele structuur, opgebouwd volgens goedgekende principes.
Merk op dat de TableComponent-structuur gecreëerd wordt tijdens de XSLT-transformatie van de persistentiebestanden en enkel dan bestaat. Men zou ze evenwel kunnen opslaan door de broncode te annoteren. SQL-generatie at run-time of at load-time is dan mogelijk door op het gepaste moment visitors te gebruiken. Een dergelijk mechanisme lijkt flexibeler, maar neemt veel meer geheugen in (de TableComponent-composite) en zal trager werken.

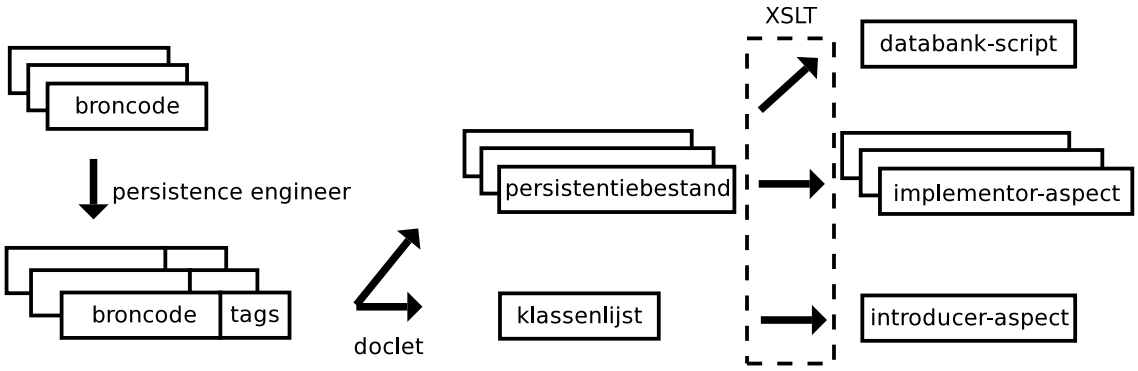
5.5 Overzicht

Dit lijkt het gepaste moment om het volledige proces te schetsen, daar we alle nodige onderdelen nu gezien hebben. Fig. 5.6 toont alle fasen die we doorlopen hebben:

1. De persistence engineer annoteert de broncode met PDL-tags (hoofdstuk 3).
2. Een doclet genereert uit de broncode en de PDL-tags een reeks persistentiebestanden en een klassenlijst. Een persistentiebestand is een generieke representatie van alle interessante info voor persistentiedoeleinden (hoofdstuk 4).



Figuur 5.5: TableComponent-hiërarchie met visitors.



Figuur 5.6: Het volledig proces.

3. Via een XSLT-transformatie wordt uit de klassenlijst rechtstreeks het introducer-aspect gegenereerd. De persistentiebestanden worden ook via XSLT omgevormd, maar nu tot een TableComponent-hiërarchie. Deze wordt door talloze visitors doorlopen, die elk code creëren voor specifieke persistentiefunctionaliteit (opslaan, verwijderen, updaten, ...). Per @persistent-type worden de verschillende bijdragen dan tot een nieuw implementor-aspect gesmeed. Ten slotte wordt ook een script gemaakt dat tabellen kan creëren, vernietigen en dat ook indices en FK-constraints aanmaakt.

Nu het gebruik van PDL-tags voor persistentietools en het generatieproces bekeken werden, is het tijd om te zien wat de gegenereerde aspecten precies doen en hoe we ze voldoende generiek kunnen laten om uitbreiding mogelijk te maken. Dit alles zullen we in het volgende hoofdstuk zien.

Hoofdstuk 6

Gegenereerde aspecten

6.1 Inleiding

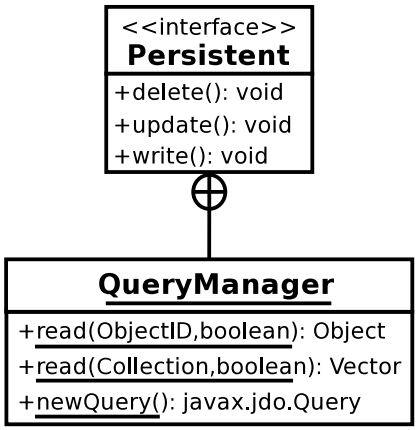
Dit hoofdstuk beschrijft alle noodzakelijke aspecten voor een basiswerking van een persistentielaag. Dit omvat aspecten voor het @persistent-enablen (zie 6.2) en voor implementatie van persistentiemethoden (zie 6.3). Andere basisfunctionaliteit zoals databankconnecties of rudimentaire inkapseling van elke persistentiemethode in een transactie, en meer geavanceerde dingen zoals caches of supertransacties zien we in volgend hoofdstuk.

6.2 PersistenceIntroducer

Dit aspect zorgt ervoor dat de @persistent-types van het business model @persistent-enabled worden. Dit betekent dat persistentiemethoden opgeroepen kunnen worden. Deze leveren vooralsnog geen nuttig werk, omdat er enkel een no-op implementatie voorzien werd. In 6.3 zullen we zien hoe er voor elk @persistent-type een specifiek aspect, de implementor, genereerd zal worden dat alle nodige functionaliteit voorziet.

6.2.1 Persistent

Deze interface (zie Fig. 6.1) bevat alle methoden die we verwachten van een persistentielaag, nl. opslaan, aanpassen, verwijderen en lezen. Zij zullen allen een eigen exceptie opwerpen indienen iets fout gaat. Het tweede argument van de read-methode duidt aan of men een deep of een shallow read ("ondiep") wil doen. Bij een deep read laadt men het volledige object in, met alle @persistent-objecten die bereikbaar zijn vanuit het ingelezen object. Een shallow read daarentegen leest van de bereikbare @persistent-objecten enkel een lege versie met enkel hun ObjectID, in.
In de nested class QueryManager zien we verscheidene factory methoden om een javax.jdo.Query



Figuur 6.1: Interface Persistent.

te creëren. Voor de query-kant hebben we immers getracht Java Data Objects Query Language (JDOQL) te implementeren i.p.v. zelf een querytaal uit te vinden. Meer details in 6.3.5.

Zoals al in 2.3.2 besproken werd, is deze aanpak niet volledig "oblivious". Het business model zelf is weliswaar volledig onafhankelijk van enig persistentie-concern, de applicatie is dit niet. Dit betekent dat de applicatie-programmeurs er zich moeten van bewust zijn dat de klassen waarmee ze werken @persistent-enabled zijn en ook moeten weten of die voorzieningen gebruikt worden of niet. Merk op dat deze situatie te vergelijken is met JDO (zie 2.2.2). Daar zijn de "persistence capable" klassen ook niet in staat om zichzelf op te slaan. Men moet zelf beslissen om expliciet de bytecode te enhancen (men moet ook weten dat dit kan) en de JDO-API te gebruiken. Zoals uit [21] echter blijkt, kan men de persistentielaag in Java nooit volledig maskeren.
We zien hier dus een duidelijke parallel, nl. het feit dat de broncode op zich het gebruik van implementor-aspecten of JDO niet afdwingt of vereist, maar de gebruiker de keuze laat. Dit leidt enerzijds tot grotere code reuse, want het persistentiemechanisme ligt niet vast. Wie geïnteresseerd is, kan echter altijd de declaratieve annotaties gebruiken die gemaakt werden door de persistence engineer (PDL, JDO, CMP, ...). Anderzijds berust alles op afspraken, wat een status quo oplevert met de huidige toestand inzake persistentiemechanismen.

6.2.2 PersistenceIntroducer

De code op de eerste lijn van Fig. 6.2 is het enige deel van het introducer-aspect dat echt gegenereerd moet worden uit de klassenlijst. De default-implementatie van de persistentiemethoden blijft vast. We zien ook dat via ITD een ObjectID als attribuut van elk @persistent-type wordt gedeclareerd. Dit maakt elk @persistent-object uniek in de databank: men kan het beschouwen als een persistente referentie naar het object in zijn toestand op het moment van persisteren. Een databank kan men immers zien als een niet-vluchtig RAM-geheugen. We zien dat objecten al een ObjectID krijgen als ze transiënt zijn en het voor de rest van hun leven bij zich hebben. Dit is aangeraden volgens [15].
Zoals bij JDO (zie [22]), bestaat het leven van een @persistent-object ook uit verschillende fasen. We onderscheiden er twee:

transiënt: net na de constructie of als het object uit de databank verwijderd werd;

```
public privileged aspect PersistenceIntroducer{
  declare parents: ... implements Persistent;
  ...
  private static ObjectIDFactory fab=new ObjectIDFactory();
  private ObjectID Persistent.OID=fab.getObjectID(this);
  public ObjectID Persistent.getOID(){
    return OID;
  }
  public void Persistent.setOID(ObjectID ob){
    OID=ob;
  }
  public void Persistent.write() throws WriteException{}
  public void Persistent.update() throws UpdateException{}
  public void Persistent.delete() throws DeleteException{}
  private String ObjectID.cClass="";
  public String ObjectID.getCClass(){
    return cClass;
  }
  public void ObjectID.setCClass(String ccClass){
    cClass=ccClass;
  }
  ...
}
```

Figuur 6.2: PersistenceIntroducerAspect.

persistent: als het object opgeslagen zit in de databank.

Dit is heel simplistisch in vergelijking met JDO, maar additionele aspecten die bijvoorbeeld een cache implementeren, kunnen extra fasen toevoegen.

Verder krijgt het ObjectID zelf een extra attribuut cClass, met zijn eigen getter en setter. Gebruik ervan buiten het aspects-package, waarin alle aspecten die we gaan tegenkomen opgenomen zijn, wordt verhinderd. Dit laatste gebeurt door de volgende lijn code:

```
declare error: call(public * ObjectID+.*etCClass(..))&&!within(
    aspects..*): ...;
```

De reden voor dit illuster attribuut zien we verderop in 6.3.4.1. Niet afgebeeld in de code listings zijn enkele definities van veelgebruikte pointcuts.

Praktisch gebruik In de praktijk is het zo dat een klassenlijst vroeger af is en vastligt dan de persistentiebestanden. Dit brengt met zich mee dat het introducer-aspect ook eerder te genereren is. Terwijl men nog druk bezig is met het implementeren van het business model, kunnen prototypes ervan al als @persistent-enabled gebruikt worden. Natuurlijk zijn er dan enkel no-op methoden, maar niets belet de ontwikkelaars om mock-aspecten te schrijven. Deze gaan iets

verder dan de lege implementaties en laten zinvoller tests toe.
De implementor-aspecten kan men ook genereren indien de PDL-annotaties al voorzien zijn. Zolang het business model echter niet stabiel is, kunnen de attributen van @persistent-klassen en relaties met andere @persistent-types wijzigen. Dit brengt natuurlijk veranderingen in de onderliggende databankstructuur met zich mee. Hoewel er automatisch een script gegenereerd wordt om die tabellen te creëren of te vernietigen, zou gebruik hiervan van in het begin leiden tot veel nutteloos opslaan en verwijderen van testdata in de databank en lange weave-tijden. Aangezien de persistentielaag (met de databank erbij) foutloos verondersteld wordt (want automatisch gegenereerd), wacht men beter tot het business model wat stabieler geworden is. Fouten in de annotaties kunnen echter wél gemaakt worden!

6.3 Implementor-aspecten

6.3.1 Wat?

Op basis van de persistentiebestanden voor de verschillende @persistent-types, willen we nu implementor-aspecten genereren. Deze zullen een serieuze invulling moeten bevatten voor de default-implementaties van het aspect PersistenceIntroducer. Daarnaast zijn ook voorzieningen nodig voor queries. Omdat private toestand bereikbaar moet zijn, zijn de implementor-aspecten priviliged. Dit veroorzaakt geen noemenswaardige (veiligheids)problemen. Merk op dat we voor elke class unit (zie 4.2.1) één aspect genereren met de nodige code voor alle daarin voorkomende @persistent-types. Dit sluit aan bij de ontwerpsfilosofie dat de aanwezige klassen nauw verbonden zijn met elkaar. Implementor-aspecten voor @persistent-interfaces zijn maar nuttig eens de interfaces attributen gekregen hebben via een aspect en Tiger's Metadata Facility gebruikt wordt.

6.3.2 Statische vs. dynamische SQL

In het PDLF-framework en [21] wordt de SQL-code at run-time gegenereerd, bij Hibernate at load-time. Allen maken gebruik van reflectie i.p.v. vaste SQL-statements, omwille van een aantal redenen:

- SQL is een apart concern
- grotere genericiteit en reusability
- Hibernate gebruikt de "CGLIB run-time bytecode generator"-bibliotheek. Deze voert reflectie-calls even snel uit als gewone code. Zo heeft men dezelfde snelheid als statische SQL-code zonder extra overhead bij compilatie.

Onze implementor-aspecten bevatten echter letterlijke SQL-code. Zij schermen echter nog altijd het SQL-concern af. Bovendien zijn at run-time geen geheugenstructuren nodig zoals de geziene TableComponent-hiërarchie. Nadeel is wel het trage weaving-proces.
Wat de herbruikbaarheid betreft: enkel de SQL-code in het gegenereerde databank-script is

```
pointcut <action>_<klassenaam'>(<klassenaam> obj): this(obj) &&
    execution(public void persistence.type.Persistent.<action>()
    throws persistence.exceptions.<action>Exception);
after(<klassenaam> obj) throws persistence.exceptions.<action>
    Exception: <action>_<klassenaam'>(obj){
  ConnectionFactory fac=new ConnectionFactory();
  Connection connection = fac.getConnection();
  try{
    ... (SQL-code)
  }catch(Exception e){
    throw new persistence.exceptions.<action>Exception(obj.getOID().
        toString(),e);
  }finally{
    fac.closeConnection(connection);
  }
}
```

Figuur 6.3: Blauwdruk voor advice op een <action> write(), update() of delete(). <klassenaam'> is de volledig gequalificeerde naam van een @persistent-klasse, waarbij de '.'-en in de naam vervangen werden door '_'-s.

databankafhankelijk (speciale datatypes). De implementor-aspecten bevatten SQL-92 code die op de meeste relationele databanken bruikbaar is (net als PDLF en [21] overigens).

Wenst men specifieke features van databanken te gebruiken, dan nog is het systeem generiek. De business code blijft immers onaangeroerd en men moet het enkel herweaven met nieuwe implementor-aspecten. Dit is zelfs mogelijk at load-time, d.w.z. bij het inladen van klassen door een eigen classloader. Snelheid hiervan werd niet getest.

6.3.3 write(), update() en delete()

Het voorstel uit [25] kan hier gevolgd worden: we onderscheppen oproepen naar write(), update() en delete() in de Persistent-interface en adviseren ze met een specifieke implementatie voor het bewuste @persistent-type. Dit houdt in dat we voor elk van deze drie methoden code zullen moeten genereren die er uitziet zoals in Fig. 6.3.

6.3.3.1 Werkwijze

Eerst leggen we met een uniek pointcut (unieke naam) vast welke methode van Persistent we met het huidige advice willen adviseren. Ook het lijdend object willen we ter beschikking hebben. Een niet onbelangrijk detail: we kozen hier bewust voor een execution-pointcut i.p.v. een call-pointcut. Alle weaving kan dan immers gebeuren binnen de Persistent-interface en de business model-klassen, d.w.z. bij integratie van het business model en de persistentielaag. Men kan die vervolgens volledig in een jar-archief opbergen. Gebruik hiervan in een applicatie vereist wel

het gebruik van ajc i.p.v. javac[1], maar extra weaving met de applicatie zelf is in principe *niet* nodig. Dit ontlast grotendeels het ontwikkelingsproces voor de client code en legt alle last bij het weaven van de business code.

Nu kunnen we de pointcut adviseren. Door gebruik van een Connection, verkregen via een factory, kunnen we alle nodige SQL-code uitvoeren. Indien iets misloopt, wordt een speciale exceptie opgeroepen. We merken op dat hier geen sprake is van transacties. Elk implementor-aspect leeft als het ware in zijn eigen wereld en weet zelfs niet dat er andere aspecten zijn die @persistent-types doorweven. Om bijvoorbeeld een ander @persistent-object op te slaan, wordt gewoon de write-methode op dat object uitgevoerd, ongeacht het feit of een implementor-aspect dat object zal bijstaan.
Het aspect vraagt in een bepaald advice gewoon een connectie op, gebruikt die en geeft ze terug, zonder rekening te houden met lopende transacties of connection pooling. Dit zijn in feite andere concerns die, dankzij het modulair karakter van aspecten, te abstraheren zijn naar een eigen aspect (zie 7.2.1, 7.2.2, 7.3.1 en 7.3.2).

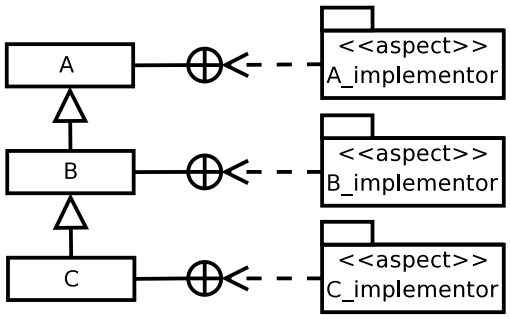
6.3.3.2 Overerving

Wat gebeurt er bij overerving? Indien de parent-klasse van een @persistent-klasse zelf niet @persistent is, kan niets gedaan worden met de overgeërfde attributen. Zelfs een soort blindemansmethode m.b.v. reflectie zou niet veel uithalen. We moeten immers in ons achterhoofd houden dat Java niet expressief genoeg is en dat PDL hiervoor nodig was. Zonder persistentiebestanden staan we dus nergens, zodat externe libraries enkel bruikbaar zijn indien ze geannoteerd werden. Bruikbaar betekent hier dat de attributen gepersisteerd kunnen worden.
In het andere geval kan men wel overgeërfde attributen persisteren. Uit Fig. 6.3 weten we dat afgegaan wordt op het type van het this-object bij uitvoer van een persistentiemethode om het juiste advice te kiezen. Het sterke aan het pointcut-mechanisme is dat we zo niets speciaals hoeven te doen om toch alle gedeelten van @persistent-objecten op te slaan, te wijzigen of te verwijderen. Het advice van alle @peristent-types boven het type van het this-object in diens overervingshiërarchie zal immers ook uitgevoerd worden en van nature uit worden eerst de join points horend bij de execution-pointcut van de root van de hiërarchie bereikt. Die hebben namelijk de hoogste precedentie. Vervolgens wordt afgedaald naar het specifieke type van het huidige @persistent-object, precies wat we nodig hebben. Enkele voorbeelden kunnen dit illustreren[2]:

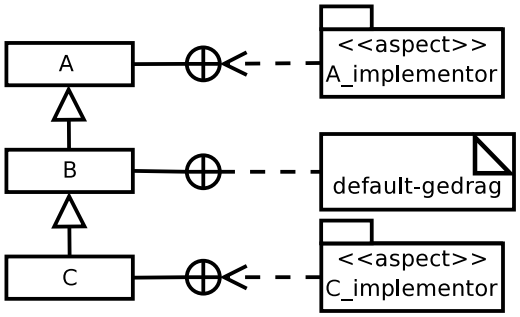
- In Fig. 6.4 zal van een object van type C eerst het implementor-aspect van klasse A een actie ondernemen, vervolgens dat van klasse B en ten slotte dat van A.
- Een speciaal geval is Fig. 6.5. Ofschoon B niet @persistent is, zal toch het A-gedeelte persisteerbaar zijn. Hoewel de aspecten voor A, B en C volledig geïsoleerd zullen zijn, wordt hun advice in de correcte volgorde in werking gesteld!

[1] javac kan blijkbaar bytecode met ITD's erin niet goed aan, want bij @persistent-enablede klassen weigert hij de ingevoegde methoden write(), ... te (h)erkennen. ajc is geen update van javac, maar produceert net zo goed gewone Java-code.

[2] De UML-standaard bevat nog geen specifieke richtlijnen voor AOP. We hebben ons daarom zoveel mogelijk gebaseerd op een voorstel van M. Basch en A. Sanchez [1]. Een $\oplus$ stelt bij hen specifiek pointcut voor, wij geven het echter de betekenis van een onbepaald pointcut. Door de aanwezigheid van aspecten is deze notatie te onderscheiden van die voor geneste klassen.



Figuur 6.4: A->B->C



Figuur 6.5: A->nietPers->C

6.3.4 read(ObjectID,boolean)

6.3.4.1 Eerste methode

De read-methode geeft de opgeslagen versie van het object met het opgegeven ObjectID terug, indien die bestaat. Er kan gekozen worden voor een “deep” of een “shallow” read (zie 6.2.1). Dit wordt bepaald door het tweede argument.

Deze methode van Persistent.QueryManager verschilt van die uit 6.3.3 op een belangrijk punt: ze is static. Bijgevolg kan men met de pointcuts die we daar gezien hebben, *niet* het juiste implementor-aspect selecteren, omdat er geen this-object is. Een oplossing waarbij in één gigantisch aspect via een switch-case constructie op de naam van de @persistent-klassen het juiste advice gekozen wordt, is absoluut niet gewenst. Gelukkig kan men in AspectJ ook dynamische primitieve pointcuts gebruiken (zie 2.3.1.2), zoals bijv. “if”. Het is nu de bedoeling om op basis van de @persistent-klassenaam het juiste advice te laten uitvoeren. Hoe dit gebeurt, kan men zien op Fig. 6.6.

Het is logisch om in het ObjectID informatie omtrent de klasse van het object op te nemen, en daarom eisen we dit ook van mogelijke ObjectID-implementaties. Dit brengt met zich mee dat men bij het opvragen van een object meteen ook het type kent en men weet welk soort object men moet reconstrueren.
Helaas zit er een addertje onder het gras. Fig. 6.4 maakt ons duidelijk dat verschillende aspecten opnieuw samen zullen moeten werken om alle delen van een object in te lezen. Het ObjectID is echter vast voor elk @persistent object, d.w.z. dat elk deel van het object in principe hetzelfde ObjectID bezit met het vaste reconstructietype. In Fig. 6.4 zal de restrictie van het C-object

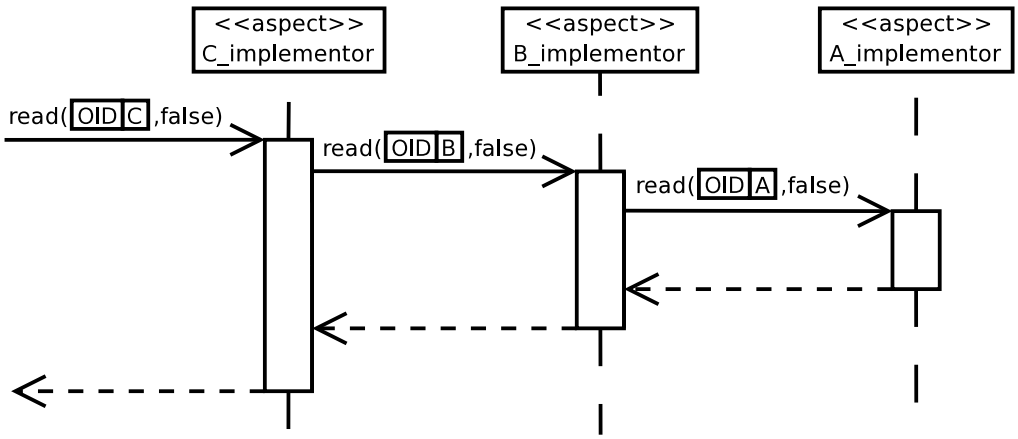
```
pointcut reader_<klassenaam'>(ObjectID OID, boolean deep): args(OID,
    deep) && execution(public Object persistence.type.Persistent.
    QueryManager.read(ObjectID,boolean)) && if(OID.getCClass().equals
    ("<klassenaam>"));
Object around(ObjectID OID, boolean deep): reader_<klassenaam'>(OID,
    deep){
  ConnectionFactory fac=new ConnectionFactory();
  Connection connection=fac.getConnection();
  try{
    ...
    /*volgend stuk wordt enkel uitgevoerd als er een
     *@persistent (grand)parent bestaat in de overervingshiërarchie
     */
    [ObjectID tmp=OID;
    tmp.setCClass("<klassenaam van dichtste @persistent-parent>");
    Object obj=(<klassenaam van dichtste      @persistent-parent>)
       Persistent.QueryManager.read(tmp,deep);]
    ...
    return obj;
  }catch(Exception e){
    //doe voorlopig niets en laat transactie-aspect dit opknappen
  }finally{
    fac.closeConnection(connection);
  }
  return null;//een Exception werd opgeworpen
}
```

Figuur 6.6: Blauwdruk van read-advice. Dezelfde opmerking geldt als bij Fig. 6.3.

tot zijn parent-klasse A hetzelfde ObjectID hebben als de beperking tot B of het oorspronkelijk object zelf. Men beschouwt immers hetzelfde object in een andere gedaante, wat volgt uit het begrip polymorfisme. Selectie van implementor-aspect op basis van het ObjectID alleen levert dus niets op, want alles ligt vast.
Nu wordt duidelijk waarvoor we het attribuut cClass, dat we in 6.2.2 tegenkwamen, kunnen gebruiken. Voegen we immers een variabel attribuut toe, cClass[3] in dit geval, dan kunnen we hierop de selectie uitvoeren. Dit zien we in het optioneel stuk van Fig. 6.6. We willen eerst de restrictie tot het meest algemene type creëren, dus passen we de cClass aan en roepen opnieuw de read-methode op. Dit procédé wordt herhaald tot men het aspect van de root van de overervingshiërarchie bereikt. Vanaf dan gebeurt het eigenlijk reconstrueren van objecten. Fig. illustreert het principe nog eens.

Dit alles betekent dat men over het algemeen niet in één keer een object zal inlezen, maar in meerdere etappen. Indien onze mapping in 5.3 anders geweest was, meer bepaald indien elke tabel alle overgeërfde attributen zou bevatten, dan zou dit alles overbodig geweest zijn. De huidige mapping laat echter meer toe op het vlak van querying (zie 6.3.5).

[3]De eerste 'c' staat voor 'current'. Naargelang het implementor-aspect dat men wenst te kiezen verandert het immers, terwijl de klasse van het eigenlijk object dat men wil lezen (de "actual class") altijd vast blijft.



Figuur 6.7: Inlezen van een object door mutatie van het ObjectID.

We kunnen ook zien dat in tegenstelling tot 6.3.3 hier around-advice gebruikt wordt en dat proceed(OID,deep) niet opgeroepen wordt. Dit betekent dat de originele no-op implementatie volledig vervangen wordt. Dit is hier noodzakelijk, omdat enkel around-advice een nieuw object kan creëren als terugkeerwaarde van een functie. Andere soorten advice kunnen enkel wijzigingen aanbrengen op bestaande objecten (cf. passing by value vs. by reference). Het advice is hier dus een brute vervanging van de echte functie. Deze praktijk kan in het algemeen gevaarlijk zijn, omdat toch op een heimelijke manier stukken code overgeslagen kunnen worden. Bovendien geven de ontwikkelaars van AspectJ in een email [7] ook toe dat around-advice meer overhead (gegenereerde code) met zich mee brengt dan before- of after-advice. Daarom werd in 6.3.3 geopteerd om het bij after-advice te houden.

6.3.4.2 Tweede methode

We kunnen het ook anders aanpassen:

1. Uit het ObjectID wordt opnieuw het doeltype gehaald, maar daarvan wordt in de static read-methode meteen een instantie gemaakt via de no-arg constructor (en reflectie).
2. Op die instantie wordt een private, niet-static read-methode uitgevoerd, vergelijkbaar met de (publieke) write-, delete- en update-methoden.
3. Het read-advice ziet er dan ook uit zoals dat van bijvoorbeeld de write-methode (zie Fig. 6.3).

Deze werkwijze lijkt iets sneller dan de vorige, omdat at weave-time al optimalisaties te doen zijn op het type van het this-object. Dit blijkt in praktijk niet zo doorslaggevend, omdat de toegang naar de databank via de JDBC-driver en de werking van de databank zelf enkele grootteordes trager zijn dan de aspectcode.

6.3.5 Querying

In [20] werd een beperkte query-taal bedacht voor het PDLF-framework. Het leek echter interessant om eens te proberen een bestaande query-taal te incorporeren. Het bedenken van een eigen taal, het schrijven van een parser ervoor en het interpreteren van de Abstract Syntax Tree (AST), bevat immers veel (onnodig) werk. De gelijkenissen van ons werk met JDO dreven ons naar JDOQL. Bovendien vonden we de referentie-implementatie van JDO, FOStore[4] genaamd, die gebouwd is voor een file-based opslagmedium. Deze bevatte al een parser voor JDOQL, zodat we konden verderbouwen op de relevantste stukken uit die code.

FOStore gebruikt geen relationele databank en bij querying leest men alle objecten uit een opgegeven extent (dit zijn alle opgeslagen objecten van een bepaald type) of collectie, in. Diegene die niet aan de JDOQL-query voldoen, filtert men eruit. Dit systeem zou heel snel werken in ons geval, al hebben we dan wel een methode nodig die de extent van een bepaald type teruggeeft. Toch is dit een gemakzuchtige oplossing en zeker niet efficiënt. Het neemt immers veel geheugen in en bovendien moeten diepe versies van alle te filteren objecten eerst uit onze databank gelezen worden.
Het lijkt beter om de expertise en voor queries geoptimaliseerde werking van de databank zelf aan te wenden, want daar zijn ze net goed in. Dit brengt met zich mee dat we uit de AST van een JDOQL-query een SQL-query moeten kunnen halen. De gemakkelijkste manier om dit te doen, is het construeren van een gigantische join-operatie waarop naar SQL vertaalde JDOQL-voorwaarden inwerken. Dit is een recht-voor-de-raap methode die niet echt efficiënt lijkt. Dat kan gedeeltelijk kloppen, maar in tegenstelling tot FOStore's werkwijze laat men de relationele databank de taak uitvoeren waarin hij juist gespecialiseerd is. Hij zal zelf de beste manier van uitvoeren kiezen, zodat deze strategie nog niet zo slecht is.

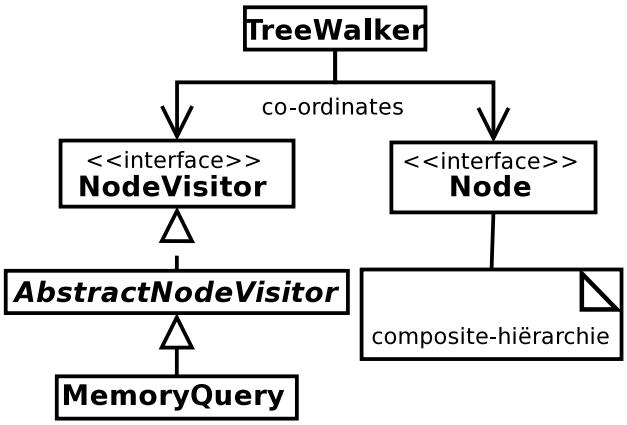
FOStore's query-component werkt grotendeels zoals in Fig. 6.8 aangegeven werd. Om de code te doen werken bij ons, werden enkel alle verwijzingen naar een PersistenceManager weggelaten, hetgeen vooral een invloed had op een klasse genaamd QueryImpl. Vervolgens moesten we voor de Node-composite een eigen visitor schrijven die sterk lijkt op de klasse MemoryQuery. Deze aangepaste klassen kregen de namen MyQueryImpl en MyMemoryQuery. Enkel in deze laatste is code nodig die verschillend is per @persistent-klasse. Daarom creëerden we er drie lege marker-methodes in, waarop we advice laten aangrijpen. Omdat de betekenis van deze methoden aanleunt bij de semantiek van JDOQL, gaan we die eerst kort bekijken.

6.3.5.1 JDOQL

In [22] neemt R. Roos JDO en JDOQL nauwkeurig onder de loep. Een goeie tutorial voor JDOQL zelf is te vinden in [14]. Aan dit laatste heb ik een goed voorbeeld ontleend, er de relevantste stukken uitgelicht en enkele stukken verduidelijkt (zie Fig. 6.9).

Het opzet lijkt sterk op SQL-queries, maar nu uitgedrukt in objecten i.p.v. in tabellen. Voor alle expressies van de reguliere vorm \w[\.\w]* is het laatste woord ofwel een parameter, ofwel een variabele, ofwel een attribuut van het te zoeken object. De mogelijke operatoren wijken gedeeltelijk af van die uit ANSI-SQL en komen overeen met de bestaande Java-operatoren. Daarnaast zijn er ook nog de ingebouwde functies starts/endsWith(String) voor Strings en contains(...) en

[4] http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html



Figuur 6.8: FOStore's query-mechanisme.

```
String filter =
        "address.street == street && " +
        "salary >= sal && " +
        "department.name.startsWith(deptName) && " +
        "projects.contains(proj) && " +
        "proj.budget > 100000";
Query query = pm.newQuery(filter);
query.setClass(Employee.class);//type van gezochte objecten
query.declareImports("import Project");
query.declareVariables("Project proj");
query.declareParameters(
        "String street, String deptName, int sal");
query.setOrdering(
        "department.deptid ascending, salary descending");
Collection result = (Collection)query.execute(
"Route 66", "Network", new Integer(100000));
```

Figuur 6.9: JDOQL-voorbeeld

isEmpty() voor Collections. We zien dat men echter ook parameters en variabelen kan gebruiken:

- parameters worden bij het uitvoeren van de query gewoon vervangen door een gespecificeerde waarde;
- variabelen worden gebruikt om doorheen collecties te itereren en stellen een onbepaald element voor van een bepaalde collectie. In Fig. 6.9 is "proj" een willekeurig Project-object waarvoor een extra voorwaarde moet gelden. De klassen waartoe de variabelen behoren, moet men importeren.

6.3.5.2 Pointcuts en advice

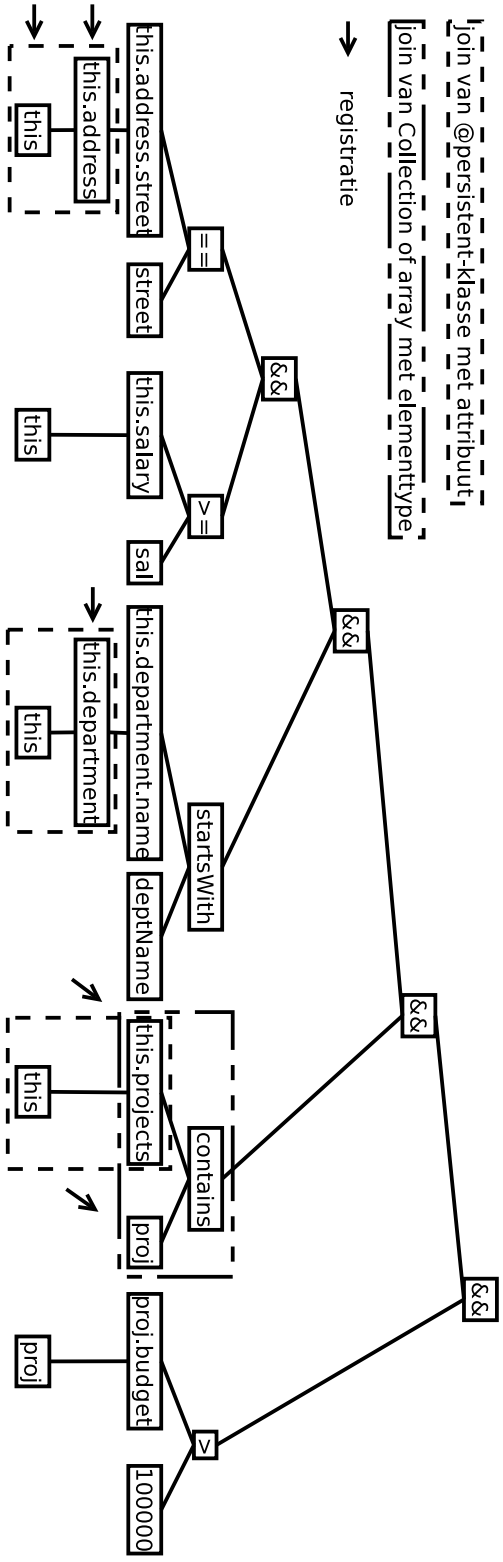
Fig. 6.10 geeft een representatief beeld van welke mogelijkheden er zijn in een AST. Het enige speciale dat er niet in zit, is isEmpty(). Dit vereist een speciale aanpak met outer joins die we niet volledig getest hebben.

Toekennen van alias Voor alle expressies van de reguliere vorm $\backslash w \backslash . \backslash w$ wordt in het advice van Fig. 6.11 gecontroleerd of het attribuut (overeenkomend met de tweede $\backslash w$) wel een @accessor is van de klasse van het object met als alias de eerste $\backslash w$. Zo niet, wordt een JDOQueryException opgeworpen die de volledige query lam legt. Indien wel, dan krijgt de volledige uitdrukking een alias voor verder gebruik.
Verder legt men voor alle expressies van de vorm $\backslash w$, met $\backslash w$ een variabele, parameter of this, een verband tussen de alias en de tabel van de klasse van x. I.p.v. "this -> this.address -> this.address.street" in Fig. 6.10, krijgen we dan "this -> alias0.address -> alias1.street" waarbij alias0 gemapped werd op de tabel voor de klasse van this en alias1 op de tabel voor de klasse Address.

Type-info uit PDL-tags We moeten ook de types bijhouden van de gecreëerde aliassen en bovendien hebben we de types van elementen van Collections, Maps, ... nodig. Het advice uit Fig. 6.12 zorgt hiervoor.

Joins tussen tabellen Ten slotte moeten we joins uitvoeren, enerzijds tussen de tabellen van een klasse en tabellen van arrays, Collections of @persistent-attributen, anderzijds tussen tabellen van Collections en arrays en die van hun elementen (variabelen) bij een oproep van contains (...) . Maps worden niet ondersteund bij JDOQL. Arrays normaal ook niet, maar in onze mapping kan men dit makkelijk oplossen, omdat een Collection min of meer behandeld wordt als een array.
Om de JDOQL-methode "isEmpty" op Collections en arrays uit te voeren, wordt:

- een outer join uitgevoerd van de tabellen horende bij de Collection of array en bij de eigenaar ervan;
- gegroepeerd op de $(ref)OID$-kolom, dit is het ObjectID van het object dat eigenaar is van de Collection of array;



Figuur 6.10: AST van de filter uit het voorbeeld van Fig. 6.9.

```
pointcut register_<klassenaam'>(MyMemoryQuery mem, String thisType,
    String variableName) : this(mem) && call( private void com.sun.
    jdori.common.query.MyMemoryQuery.register(String, String) throws
    JDOQueryException) && args(thisType, variableName) && if (thisType
    .equals("<klassenaam>"));
after(...) throws JDOQueryException: register_<klassenaam'>(...){...}
```

Figuur 6.11: Register-advice. Zelfde opmerking als bij Fig. 6.3.

```
pointcut getCompTypes_<klassenaam'>(MyMemoryQuery mem, String attr,
    String thisType) : this(mem) && call(private void com.sun.jdori.
    common.query.MyMemoryQuery.getCompTypes(String, String)) && args(
    attr, thisType) && if (thisType.equals("<klassenaam>"));
after(...) : getCompTypes_<klassenaam'>(...){...}
```

Figuur 6.12: Type-advice. Zelfde opmerking als bij Fig. 6.3.

```
pointcut getJoins_<klassenaam'>(MyMemoryQuery mem, String attr,
    String thisType, String foreignAttr, boolean outer) : this(mem)
    && call( private void com.sun.jdori.common.query.MyMemoryQuery.
    getJoins(String, String, String, boolean)) && args(attr, thisType
    , foreignAttr, outer) && if (thisType.equals("<klassenaam>"));
after(...): getJoins_<klassenaam'>(...){...}
```

Figuur 6.13: Join-advice. Zelfde opmerking als bij Fig. 6.3.

- enkel de ObjectID's van objecten met een lege groep worden behouden.

Op Fig. 5.4 zouden tabellen a_B en a_B_h ge-outerjoined worden, vervolgens op a_B.OID gegroepeerd worden en enkel die groepen met nul a_B_h.OID's worden verder gebruikt. Dat zijn de objecten met als a_B_h-attribuut een lege Collection of array.

SQL-query assembleren Nadat de query volledig doorlopen werd, moeten alle stukken van de SQL-code nog aaneengelast en uitgevoerd worden. Dit is een algemene functionaliteit die echter een Connection nodig heeft. Om onze code en die van FOStore gescheiden te houden, hebben we dit in een afzonderlijk QueryAspect gestopt. Implementatie is tamelijk triviaal: het goeie aangrijpingspunt specificeren en daar in advice de SQL-query ineensteken en uitvoeren. Het resultaat zal een (mogelijks lege) collectie van ObjectID's zijn.

De resulterende SQL-query voor de AST van Fig. 6.10 is:

```
select distinct alias0.$OID$ from alias53286575 alias1,
    alias1959567594 alias6, alias1716917701 alias0, alias1006596703
    alias4, alias728569514 alias7 where alias6.$projects$=alias7.$OID$
     AND alias0.$department$=alias4.$OID$ AND alias0.$address$=alias1.
    $OID$ AND alias6.$refOID$=alias0.$OID$ AND (((((((alias1.$street$
    ='Route 66') AND (alias0.$salary$>=100000))) AND (alias4.$name$
    LIKE 'Network%')))) AND (TRUE))) AND (alias7.$budget$>100000))
```

Door de keuze voor aparte tabellen voor alle klassen in een overervingshiërarchie, zijn we nu in staat om makkelijk te kunnen antwoorden op queries zoals "Geef alle objecten van type A." op Fig. 6.4. Men hoeft als filter enkel "true" te nemen. Niet alleen pure A-objecten, ook objecten van klassen B of C worden bekomen. Men zou ook eenvoudig een optie kunnen inbouwen om enkel de zuivere A-objecten op te vragen (zie JDO), maar dat hebben we niet gedaan.

6.4 Besluit

In dit hoofdstuk hebben we een primitieve, maar werkende persistentielaag gebouwd. Helaas is er geen interactie tussen de implementor-aspecten: als bij het inlezen of opslaan van een object meerdere aspecten tussenkomen, dan opereren die allemaal naast elkaar. Indien er ergens iets fout loopt, dan reageren de anderen niet. Transacties bestaan dus niet, noch voor één persistentiemethode, noch voor een verzameling. Hergebruik van eenzelfde databankconnectie gebeurt evenmin.
Efficiëntieverhogende voorzieningen zoals een cache of ondersteuning voor meerdere gebruikers, al of niet gedistribueerd, ontbreken vooralsnog.

Toepassing van de AOP-gedachte lost deze hiaten makkelijk op. We kunnen op eenvoudige wijze die ontbrekende functionaliteit, die crosscutting concerns zijn, toevoegen bovenop de geziene aspecten. Deze manier van werken laat een veel grotere controle toe dan het vast includeren van dit alles in bijvoorbeeld de implementor-aspecten.

In volgend hoofdstuk bekijken we enkele mogelijke aspecten die de basisfunctionaliteit uitbreiden.

6.5 Voorbeeld

Fig. 6.14 toont een klein stukje uit het implementor-aspect van de klasse Course, meer bepaald het advice horend bij de write-methode. We zien eerst een oproep voor een databankconnectie, die zal onderschept worden door ConnectionAspect (zie 7.2.1). Vervolgens worden de elementen van Collection-attribuut “prerequisites” opgeslagen, en dan de overige. Van “prerequisites”’s objecten wordt enkel het ObjectID opgeslagen, omdat wegens de associatierelatie een Course-object geen persistentiecontrole heeft over zijn prerequisites (zie 3.2.7.2). Indien er een composite-relatie zou bestaan, zouden de composite-objecten wel één voor één opgeslagen worden.

```
after(casestudy.Course obj) throws persistence.exceptions.
    WriteException: writer_casestudy_Course(obj){
  ConnectionFactory conFac=new ConnectionFactory();
  Connection connection=conFac.getConnection();
  try{
    Statement stmt=null;
    Iterator it=null;
    java.util.Collection obj0=obj.getPrerequisites();
    if(obj0!=null){
      Iterator it0=obj0.iterator();
      int i0=0;
      while(it0.hasNext()){
        casestudy.Course al1=((casestudy.Course)it0.next());
        stmt=connection.createStatement();
        stmt.executeUpdate("insert into alias1616369544 ($refOID$,
            $POSITION$, $prerequisites$) values ('"+obj.getOID().
            toString()+"', "+i0+", '"+((al1!=null)?al1.getOID().
            toString():"null")+"')");
        stmt.close();
        i0++;
      }
    }
    stmt=connection.createStatement();
    stmt.executeUpdate("insert into alias799232454 ($OID$, $courseID$
        , $courseName$, $courseWeight$) values ('"+obj.getOID().
        toString()+"', '"+obj.getCourseID()+"', '"+obj.getCourseName()
        +"', "+obj.getCourseWeight()+")");
    stmt.close();
  }catch(Exception e){
    throw new persistence.exceptions.WriteException(obj.getOID().
        toString(),e);
  }finally{
    conFac.closeConnection(connection);
  }
}
```

Figuur 6.14: Fragment van het write-advice van de klasse Course.

Hoofdstuk 7

Uitbreiding functionaliteit

7.1 Inleiding

Startend van de in vorig hoofdstuk gegenereerde implementor- en introducer-aspecten, kan men nieuwe aspecten ontwerpen. Deze hebben twee doelen:

basis-aspecten: zorgen voor samenwerking en communicatie tussen de egocentrische implementor-aspecten

geavanceerde aspecten: bouwen voort op de bekomen basis om zo een robuuste persistentielaag, rijk aan features, te creëren.

Gemeenschappelijk aan beide soorten aspecten is het feit dat ze, alleen of in clusters samenwerkend, elk voortbouwen op gekende andere aspecten zonder dat die hen kennen. Bij gebruik heeft men bovendien de keuze om naar eigen willekeur bepaalde mogelijkheden uit te schakelen (zelfs at load-time [**?**]), al is dat niet zonder gevaar bij de basis-aspecten.

7.2 Basis-aspecten

Dit zijn aspecten die logistieke steun verlenen aan de geziene aspecten. Ze smeden o.a. de individualistische implementor-aspecten samen tot een hecht team.

7.2.1 ConnectionAspect

Elk implementor-aspect heeft voor zijn SQL-acties een connectie nodig. We hebben o.a. in 6.3.3 en 6.3.4.1 gezien dat deze telkens uit een ConnectionFactory gehaald werd. We willen echter atomische acties maken van write(), delete(), ... Als midden in hun uitvoering iets

```
public pointcut connectionCreation():execution(public Connection
    persistence.type.ConnectionFactory.getConnection()) && cflow(
    PersistenceIntroducer.persistentExecution());
Connection around():connectionCreation(){
  if(connection==null){
    System.out.println(id+": Connection null; creating new one.");
    refCount=1;
    return proceed();
  }else{
    try{
      if(connection.isClosed()){
        System.out.println(id+": Connection closed; creating new one.
            ");
        refCount=1;
        return proceed();
      }else{
        System.out.println(id+": Connection reused.");
        refCount++;
        return connection;
      }
    }catch(SQLException ex){
      System.err.println(id+": Hier nooit geraken; creating new one,
          but almost certainly trouble will follow!");
      refCount=1;
      return proceed();
    }
  }
}
public pointcut connectionClosing(Connection conn):execution(public
    void persistence.type.ConnectionFactory.closeConnection(java.sql.
    Connection)) && args(conn) && cflow(PersistenceIntroducer.
    persistentExecution());
void around(Connection conn):connectionClosing(conn){
  refCount--;
  if(refCount==0) proceed(conn);//nooit bereikt door extra
      closeConnection()
}
```

Figuur 7.1: ConnectionAspect (deel 1)

```
void around(): PersistenceIntroducer.writedeleteupdate() && !cflow(
    PersistenceIntroducer.implementorAdvice()){
  connection=fac.getConnection();
  proceed();
  fac.closeConnection(connection);
  return;
}
Object around(): PersistenceIntroducer.read() && !cflow(
    PersistenceIntroducer.implementorAdvice()){
  connection=fac.getConnection();
  Object ob=proceed();
  fac.closeConnection(connection);
  return ob;
}
```

Figuur 7.2: ConnectionAspect (deel 2)

mislukt, moet het reeds verrichte werk ongedaan gemaakt worden. Hiervoor zullen we een TransactionAspect gebruiken (zie 7.2.2). Er is echter eerst nog een tussenstap nodig.

Eerder is gebleken dat alle implementor-aspecten onafhankelijk zijn van elkaar en elk hun eigen connectie aanvragen. Dit is correct, omdat hun hoofdopdracht erin bestaat de persistentiemethoden van een specifieke @persistent-klasse te implementeren. Zaken als transacties en connection pooling zijn andere concerns en moeten een aparte behandeling krijgen. Om echter transacties te kunnen gebruiken, mag in de volledige uitvoering van bijv. write() slechts één connectie gebruikt worden. Het uitvoeren van een commit of een rollback in JDBC vereist dit namelijk. Daarom hebben we het ConnectionAspect nodig.

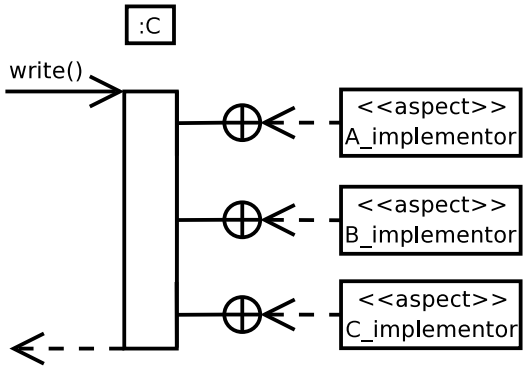
7.2.1.1 Werking

De werking is niet zo moeilijk. Oproepen in write(), update(), delete(), read(...), ... naar een ConnectionFactory worden onderschept. Er zijn twee gevallen mogelijk:

openen connectie: indien nog geen connectie bestaat en er één nodig is, wordt die aangemaakt, anders wordt de bestaande connectie teruggegeven.

sluiten connectie: de connectie mag niet afgesloten worden zolang ze nog nodig is in de control flow van de oorspronkelijke aanroep. Dit betekent dat men bij een write-methode andere write-methoden kan oproepen, maar dat die dezelfde connectie moeten gebruiken.

Om dit nu te bewerkstelligen, werken we in Fig. 7.1 met een reference count die bijhoudt hoeveel keer een connectie al geopend is, omdat evenveel keer de connectie gesloten dient te worden. Dit mogen we rustig onderstellen doordat de implementor-aspecten automatisch gegenereerd worden en dus foutloos (zouden moeten) zijn. Daarnaast zal het TransactionAspect de atomiciteit waarborgen.



Figuur 7.3: Control flow van write(), update() en delete().

7.2.1.2 Problemen

Voor de aanpak van 6.3.4.2 werkt dit mechanisme, voor de andere werkwijze niet. Denken we maar even terug aan het verschillend gedrag dat we zagen in 6.3.3 en 6.3.4.1 inzake overerving. In het eerste geval werd automatisch het advice in de verschillende aspecten van een overervingshiërarchie opgeroepen op basis van de natuurlijke precedentie van de pointcuts onderling. In het tweede geval moesten we echter sleutelen aan het ObjectID en dan als het ware zelf ander advice oproepen. Dit brengt met zich mee dat het voorgestelde mechanisme van reference counting zal werken bij read(ObjectID, boolean), maar niet bij de rest. Alles zit immers in één control flow, nl. die van de oorspronkelijke oproep in de client applicatie (zie Fig. 6.7. Bij de andere methoden hoort bij elk niveau in de overervingshiërarchie een eigen control flow die sequentieel is aan de andere i.p.v. erin vervat te zijn (zie Fig. 7.3.
Op Fig. 6.4 kunnen we dit mooi zien: voor read(ObjectID, boolean) wordt het advice voor C eerst opgeroepen en van daaruit dat voor B en A. Eens het werk gedaan is voor A, keert men terug: het advice voor A en B zit volledig in de control flow van C's advice. Voor de andere methoden wordt eerst het advice voor A volledig uitgevoerd, daarna dat voor B en dan pas dat voor C. Er bestaan hier drie aparte control flows die volledig onafhankelijk zijn van elkaar. Dit betekent dat de reference count nul is na voltooiing van A's advice en dat de connectie dus gesloten wordt. Desondanks wensen we voor alle drie toch dezelfde connectie te gebruiken.

7.2.1.3 Oplossing

De oplossing is echter eenvoudig. Men kan namelijk de levenstijd van aspecten bepalen (zie de percflow-constructie in Fig. 7.4). Standaard wordt één instantie van een aspect gecreëerd voor de geaggregeerde levensperiode van alle klassen waarin het aangrijpt (singleton-gedrag). Dit kunnen we echter veranderen, zodat voor de volledige control flow van een oproep naar write(), read(ObjectID, boolean), ... vanuit de client code het aspect blijft leven. Dit komt overeen met de beoogde levensduur van de connectie die we daarbij kunnen gebruiken. We veronderstellen bij creatie van het ConnectionAspect immers niet dat er aan connectie-pooling gedaan wordt. Dit kan er later gewoon ingeplugd worden.
Aangezien nu elk ConnectionAspect garant staat voor eenzelfde connectie gedurende zijn hele leefperiode, kunnen we meteen na instantiatie van het aspect de connectie daadwerkelijk initia-

```
public aspect ConnectionAspect percflow(PersistenceIntroducer.
    persistentExecution()){
  declare precedence:ConnectionAspect,aspects.imp..*;
  public Connection getConnection(){
    return connection;
  }
  public void closeConnection(){
    fac.closeConnection(connection);
  }
  ...
}
```

Figuur 7.4: ConnectionAspect is geen singleton, maar wordt op speciale momenten geïnstantieerd.

liseren, zodat de reference count altijd minimaal één is. Nadat de bewuste persistentiemethode volledig uitgevoerd is, mag de connectie worden afgesloten.
Dit laatste is echter een probleem. Aspecten kunnen wel een no-arg constructor hebben, een destructor bestaat echter niet (net als in Java zelf). Dit kan men oplossen door de twee advices uit Fig. 7.2. Zij omsingelen de uitvoering (execution) van een door een applicatie opgeroepen persistentiemethode met oproepen voor een connectie. Omdat instantiatie van een aspect op join points voorrang heeft op advice op die plaatsen, krijgen we het beoogde effect.

Dezelfde problemen verhinderen dat we de implementor-aspecten zelf ook een specifieke levensduur zouden geven. Bovendien lijkt het nut hiervoor niet meteen duidelijk. Misschien speelt het een rol in gedistribueerde omgevingen, maar dat werd niet onderzocht.

In Fig. 7.4 zien we ook nog dat precedentie met andere aspecten opgegeven wordt. Tevens werd de connectie toegankelijk gemaakt van buiten uit (zie de getter en de close-methode). Zoals al eerder aangegeven werd, werken ConnectionAspect en TransactionAspect samen, en bovendien is een connectie een basisresource die door meerdere aspecten gebruikt kan worden (zie bijv. 7.3.2).

7.2.1.4 Rekening houden met de onbekende toekomst

Een andere belangrijke kwestie is het openlaten van voldoende ruimte voor toekomstige aspecten. De geziene basisfunctionaliteit kan immers met andere aspecten uitgebreid worden. Het mooie aan AOP is dat elk aspect een eenvoudig klein stukje toe kan voegen aan de bestaande infrastructuur om zo een complexe taak op te lossen. Daarom dienen alle aspecten een welomlijnde, relatief eenvoudige taak te hebben die niet-destructief verderbouwt op reeds aanwezige functionaliteit[1]. Een voorbeeld hiervan zijn ConnectionAspect en TransactionAspect. Hen samennemen zou leiden tot een grote brok code met twee concerns dooreengeweven, die bovendien veel complexer zou zijn dan nu.

Om latere aspecten de kans te geven op zoveel mogelijk plaatsen nog te kunnen ingrijpen, dienen

[1] Dit is sterk vergelijkbaar met de filosofie achter het plugin-mechanisme van het Eclipse-project. Zie http://www.eclipse.org.

```
declare precedence:ConnectionAspect,TransactionAspect;
after() returning(Connection conn):ConnectionAspect.
    connectionCreation(){
  try{
    conn.setAutoCommit(false);
    System.out.println("###"+aspects.ConnectionAspect.aspectOf().id+"
        : transactie OPEN");
  }catch(SQLException ex){...}
}
after() throwing: PersistenceIntroducer.persistentExecution() &&
    within(!aspects.imp.*){
  try{
    aspects.ConnectionAspect.aspectOf().getConnection().rollback();
    System.out.println("###"+aspects.ConnectionAspect.aspectOf().id+"
        : transactie ROLLBACK");
  }catch(SQLException ex){...}
  aspects.ConnectionAspect.aspectOf().closeConnection();
}
before(Connection conn):ConnectionAspect.connectionClosing(conn){
  try{
    conn.setAutoCommit(true);
    System.out.println("###"+aspects.ConnectionAspect.aspectOf().id+"
        : transactie CLOSED");
  }catch(SQLException ex){...}
}
```

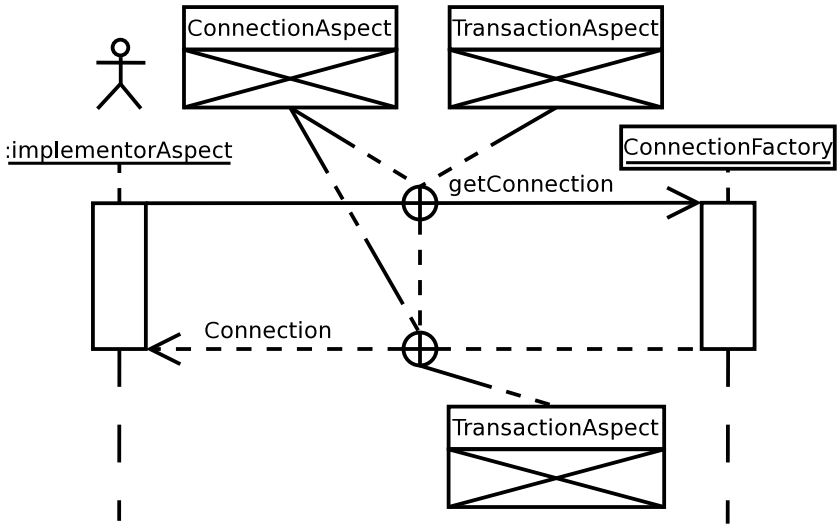
Figuur 7.5: TransactionAspect.

pointcuts voldoende precies te zijn. Alle aspecten bevinden zich bijvoorbeeld in het aspects-package, waarbij de implementor-aspecten zich in een speciaal subpackage hiervan ophouden. Waar mogelijk dient men pointcuts enkel te laten ingrijpen op dit laatste subpackage (of het business model zelf). Restricties op andere basis-aspecten dienen vermeden te worden. Hoewel we nu de functionaliteit ervan nog niet kennen, mogen we hen geen strobreed in de weg leggen. We mogen ook niet speculeren op de plaats en het tijdstip waar ze ingrijpen op de normale werking.

7.2.2 TransactionAspect

De transacties die hier besproken worden, werden eerder al bestempeld als rudimentair. Ze hebben als doel dat een specifieke persistentiemethode als één geheel wordt uitgevoerd: ofwel wordt het object volledig opgeslagen of gelezen of ..., ofwel volledig niet. Dit komt dus overeen met het waarborgen van atomiciteit. Vandaar dat dit ook een basis-aspect is. Het samennemen van meerdere methoden in eenzelfde transactie wordt bekeken in 7.3.1.

In wezen toont Fig. 7.5 een eenvoudig aspect, dat hevig gebruik maakt van de door ConnectionAspect gegarandeerde claim dat dezelfde connectie voor de volledige afhandeling van een



Figuur 7.6: Ketting van aspecten. We gebruiken hier een eigen conventie dat twee verbonden pointcuts eenzelfde pointcut voorstellen, eens vóór en eens achter een methode-oproep. Aangeven van before-, after- en around-advice wordt zo duidelijker.

persistentiefunctie gebruikt kan worden. In JDBC kan men immers enkel transacties gebruiken op eenzelfde connectie. TransactionAspect voert dus een soort piggybacking uit op ConnectionAspect en heeft dezelfde levensduur, maar heeft een lagere precedentie om conflicten te voorkomen.
We krijgen het begin van een ketting van aspecten (zie Fig. 7.6). Zowel in 7.3.1 als in 7.3.2 zullen nog schakels toegevoegd worden.

Ten slotte zit er ook nieuw advice in om bij om het even welke exceptie een rollback uit te voeren.

7.2.3 QueryAspect

Naast de drie vereiste @persistent-afhankelijke advices van 6.3.5.2, is een apart aspect verantwoordelijk voor het aaneenlassen van de verscheidene onderdelen van de te construeren SQL-query. Dit werd eerder al in 6.3.5.2 besproken.

Deze taak is onafhankelijk van enig @persistent-type, dus volstaat een vast aspect.

7.3 Geavanceerde aspecten

Tot nu toe werd niet gesproken over caches, echte transacties, multi-user werking, ... Dit zijn niet-levensnoodzakelijke crosscutting concerns, die een soort middleware-diensten leveren. Implementatie in een aspect ligt dus voor de hand, en in deze sectie zien we enkele voorbeelden daarvan.

```
public interface Persistent {
  ...
  public void write(Transaction trans) throws WriteException;
  public void update(Transaction trans) throws UpdateException;
  public void delete(Transaction trans) throws DeleteException;
  public static class QueryManager {
    ...
    public static Transaction newTransaction() {
      return new TransactionImpl();
    }
    public static Object read(Transaction trans, ObjectID OID,
       boolean deep) {
      return read(OID, deep);
    }
  }
}
```

Figuur 7.7: Aanpassingen aan Persistent.

7.3.1 SuperTransactionAspect

Wanneer men serieuze applicaties bouwt, is het vaak noodzakelijk dat men een aantal persistentie-acties kan groeperen in een transactie. Deze moet voldoen aan de vier ACID-kenmerken [9]:

atomicity Alle acties in de transactie worden ofwel allen samen uitgevoerd, ofwel gebeurt er helemaal niets.

consistency Vóór en na de transactie is de databank in een consistente staat, d.w.z. dat alle constraints voldaan zijn. Tijdens de transactie hoeft dit echter niet.

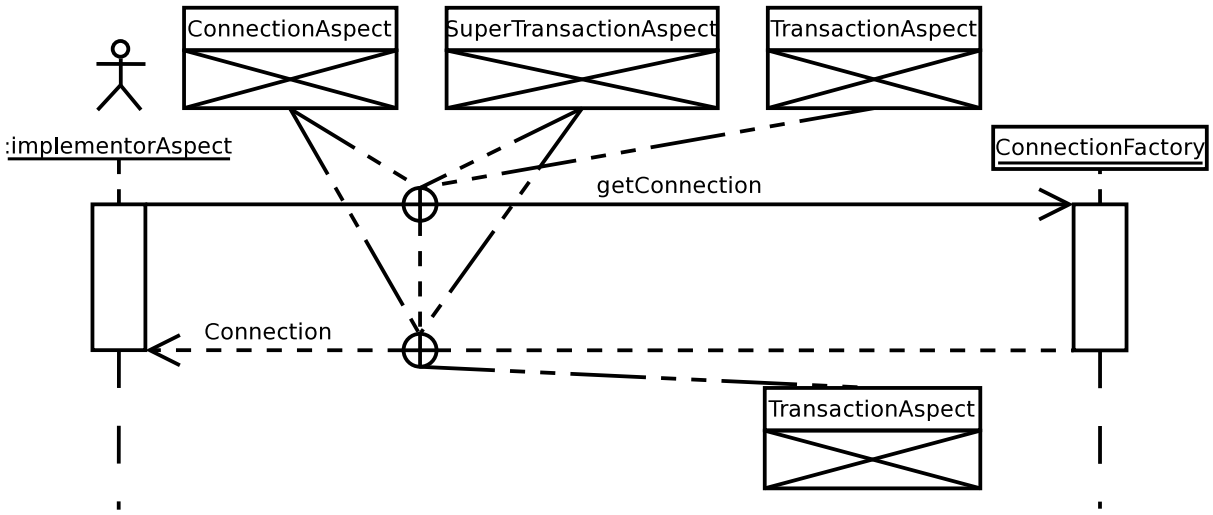
isolation Elke transactie schermt zijn acties af voor andere: niemand kan zijn interne toestand bekijken.

durability Wat er ook gebeurt met de databank (crash, ...), het resultaat van de transactie (commit/rollback) blijft gegarandeerd bestaan.

De Persistent-interface van 6.2.1 is niet voorzien op dergelijke transacties. Fig. 7.7 toont enkele mogelijke uitbreidingen. Alle persistentiemethoden krijgen een “Transaction-aware” equivalent met als extra argument de transactie waarbinnen ze uitgevoerd worden. Ook een factory-methode voor creëren van Transactions is nodig.

In eerste instantie zal no-op code in PersistenceIntroducer ervoor zorgen dat het extra Transaction-argument genegeerd zal worden. Dit is immers een heel gespecialiseerd concern, dat in zijn eigen aspect beschreven moet worden.

Desondanks is zijn werking niet zo moeilijk te begrijpen. Het komt er gewoon op neer dat de Transaction per sessie een vaste connectie toegewezen krijgt. Eerder werd dit al verklaard door de eis van JDBC dat commit of rollback enkel werkt voor alle JDBC-acties die de huidige



Figuur 7.8: Extra schakel in ketting van Fig. 7.6. Zelfde opmerkingen gelden hier.

connectie gebruiken. Cruciaal is de precedentie van het SuperTransactionAspect t.o.v. ConnectionAspect en TransactionAspect, namelijk precies in het midden. Dit breidt de ketting van Fig. 7.6 uit naar Fig. 7.8.
Zo kan men bij een eerste oproep om een connectie te creëren, dit toestaan en de connectie in het Transaction-object opslaan (via ITD). Tevens zal TransactionAspect de "eenheids-transactie" opstarten, maar vanaf dan worden al zijn acties, behalve zijn rollback-advice, in de kiem gesmoord. Hij mag de transactie immers niet middenin committen. Verder moeten alle oproepen die doorgelaten worden door ConnectionAspect om connecties op te starten (typisch bij de start van een aparte write(), ...), beantwoord worden met de connectie van de Transaction zelf. Pas bij een expliciete commit of rollback van de Transaction zelf, mag men de connectie committen of rollbacken.

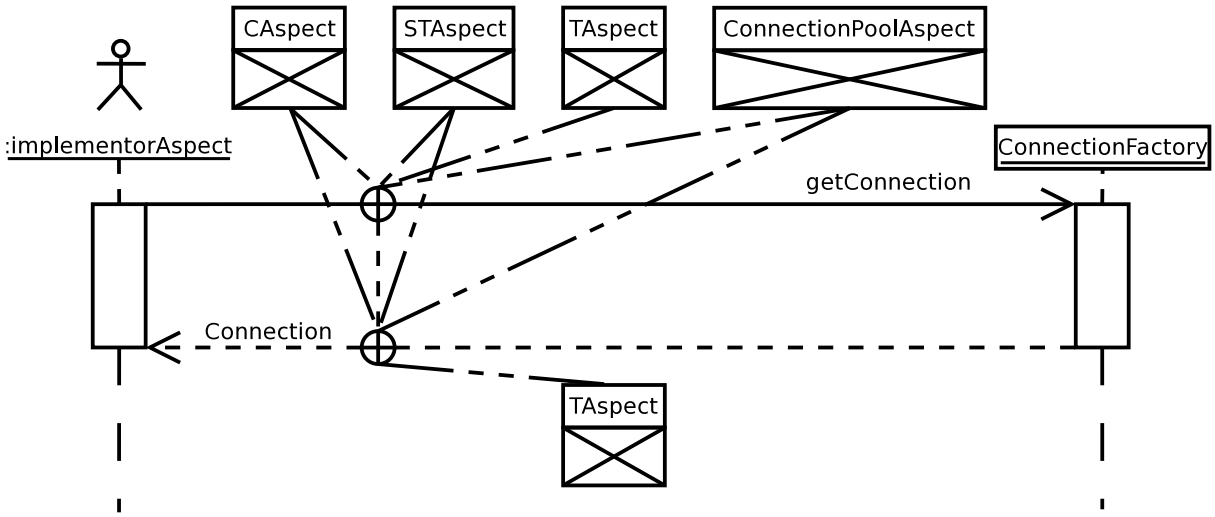
De code neemt te veel plaats in om ze hier letterlijk weer te geven. Daarom kan men ze vinden op http://faramir.ugent.be/thesissen/aspectgenerator/code.zip.

7.3.2 ConnectionPoolAspect

Tot nu toe werden alle connecties nieuw aangemaakt en dan vernietigd. Dit is echter niet efficiënt. Er bestaat echter een wijdverspreide oplossing hiervoor, namelijk het aanleggen van een pool van connecties. Men vist er als het ware een connectie uit en dropt hem er weer in indien hij niet meer nuttig is. Merk op dat ConnectionAspect enkel een verlenging van de levensduur van een connectie bewerkstelligt.

Een oplossing met aspecten ligt voor de hand en is analoog aan het thread pool-voorbeeld uit [18]. Men rijgt gewoon een vierde schakel aan de ketting van Fig. 7.8, zodat men Fig. 7.9 bekomt.

Men hoeft enkel een verzameling connecties aan te leggen en een limiet te lezen uit een configuratiebestand. Vervolgens wordt elke oproep voor een connectie die doorgelaten wordt door TransactionAspect (de voorlaatste in de rij) beantwoord met een gloednieuwe of een gerecy-



Figuur 7.9: Vierde schakel in ketting van Fig. 7.8

cleerde connectie. Afsluiten van de verbinding gaat gepaard met opbergen in de collectie.

7.3.3 CacheAspect

In [21] pleit de auteur dat een cache een essentieel iets is als de databank groter wordt, maar dan vooral bij grote databanken. Bovendien dient ze geoptimaliseerd te zijn voor de taken die de applicatie echt nodig heeft, zoals bijvoorbeeld queries of updates.

Zoals de eerder besproken concerns kan men ook dit in een apart aspect modelleren. Op http://faramir.ugent.be/thesissen/aspectgenerator/code.zip is de code te vinden van CacheAspect, een eenvoudige cache die geen Transactions ondersteunt (enkel "eenheids-transacties"). De @persistent-objecten krijgen via ITD een dirty-attribuut dat "true" is als er niet-opgeslagen wijzigingen aan attributen zijn. Welke attributen in de gaten gehouden worden, moet in een apart aspect dat overerft van het abstracte CacheAspect, gespecificeerd worden. Om het eenvoudig te houden, werden alle attributen geviseerd, maar dit concreet aspect kan ook automatisch gegenereerd worden uit de persistentiebestanden.

De functionaliteit van CacheAspect hangt af van de persistentiemethoden:

write: @persistent-object wordt in de databank en de cache opgeslagen;

update: enkel als het object dirty is door wijzigingen, wordt de update uitgevoerd (object zit intussen al in cache);

delete: verwijderen uit databank én cache, maar het object blijft transiënt bestaan (zoals eerder al besproken werd in 6.2.2);

read: indien het object al in de cache zit, wordt het teruggegeven (dirty of niet), anders wordt de databank geraadpleegd.

Dit laatste is het meest eenvoudige. In werkelijkheid verkiest men misschien om altijd de in de databank opgeslagen versie terug te geven en de andere versie transiënt te laten. Hier belandt men echter in filosofische kwesties omtrent de semantiek van ObjectID's. De hier geïmplementeerde versie is eerder puriteins: "per ObjectID één object".

Talloze uitbreidingen zijn mogelijk, zoals ondersteuning van Transaction's, crash recovery (onopgeslagen wijzigingen), concurrency control, ... Deze dingen kunnen ofwel in het cache-aspect zelf of als een extra aspect geïmplementeerd worden.

7.3.4 DistributedAspect

Tot nu werd geen aandacht besteed aan multi-user werking. Er werd verondersteld dat men alleen werkt, hoewel Transactions al een soort bescherming opleverden. Het ligt buiten de scope van dit werk om echt heel diep in te gaan op deze materie.

Een mogelijke aanpak is om niet meteen een DistributedAspect te creëren, maar om eerst een MultiUserAspect te bouwen, dat toelaat op eenzelfde JVM parallel te werken. Wellicht zullen de bestaande aspecten zoals ConnectionAspect, ... ook enkele wijzigingen moeten ondergaan, maar dit werd niet onderzocht. Er zou extra logica nodig zijn om concurrency control uit te oefenen. Caches zouden dan ook rekening moeten houden met verschillende gebruikers. Dit kan bijvoorbeeld door de Transactions als verschillende gebruikers te zien.

Eens deze functionaliteit werkt, zou men een DistributedAspect kunnen ontwerpen dat de verschillende lokale JVM's verbindt. Of de lokale clients rechtstreeks de databank aanspreken of via een centrale cache, dient echter onderzocht te worden. Wat wel zeker is, is dat men stilaan tot een gedistribueerde, objectgeoriënteerde verpakking rond een relationele databank komt. Dit is een heel uitgebreid onderwerp dat niet in de scope van dit werk ligt.

7.3.5 SyntacticSugarAspect

Het besproken persistentiesysteem kan in sommige applicaties onhandig lijken. Zelf write() of update() oproepen is niet altijd aangewezen. Dit is echter geen onoverkomelijk iets, want zoals reeds verschillende malen vermeld werd, kan men ook een aspect schrijven dat de persistentielaag meer verbergt.

Zo kan men het systeem uit [21] nabootsen (zie eerder) door bijvoorbeeld write() automatisch na de constructie van een @persistent-object op te roepen. Men kan echter ook bestaande persistentiemethoden in legacy-systemen automatisch "vervangen" met around-advice zonder proceed. Het was onze bedoeling een tamelijk algemeen systeem te ontwikkelen dat naar specifieke noden geplooid kan worden.

7.4 Besluit

De hier besproken aspecten consolideren de ruwe persistentielaag uit het vorig hoofdstuk en bouwen er op voort. Gebruikmakend van reeds bestaande functionaliteit kan men dit stap per stap doen. Hoe verder men echter gaat, hoe moeilijker het wordt om voldoende ruimte te laten voor latere toevoegingen en om zich mooi in te passen in het reeds bestaande geheel. Dit is een algemeen gevaar voor de toekomst van AOP.
Dit kwam al aan de oppervlakte bij de "declare precedence : ...; "-lijnen. Hoe complexer de taken die de afzonderlijke aspecten toebedeeld krijgen, hoe ingewikkelder hun onderlinge interactie wordt. Naast het belang van design patterns voor een goed ontwerp, zal er in de toekomst voldoende tool support moeten komen om conflicten tussen verschillende aspecten op te sporen. Het overzichtelijk visualiseren van aspecten-kolonies zou wel eens onontbeerlijk kunnen worden.

Een ander probleem kan snelheid zijn. Niet alleen neemt weaven veel tijd in beslag, de uitvoertijd at run-time neemt ook toe. Hoe meer advice er op hetzelfde pointcut uitgevoerd kan worden, hoe meer tijd arbitrage inneemt. In het geval van dynamische pointcuts geldt dit zeker, omdat er dan geen statische optimalisatie at weave-time kan gebeuren. Desondanks zijn deze vertragingen miniem vergeleken met toegangstijden tot de databank. Dit vereist immers schijftoegang, wat enkele grootteorden trager is dan de uitvoer van extra instructies om het juiste advice te kiezen. Het inpluggen van een cache en een connection pool is veel ingrijpender op de performantie en leidde tot een snellere retrieval dan bij het PDLF-framework.

Hoofdstuk 8

J2SE 1.5

8.1 Inleiding

Sinds zijn doorbraak midden jaren '90, heeft Java flink wat wijzigingen ondergaan: sommige API's werden herwerkt, andere vervangen, nieuwe technologieën geïncorporeerd, ... Vreemd genoeg bleef de taal zelf vrijwel ongewijzigd (afgezien van de invoering van bijvoorbeeld het assert-keyword). Met de invoering van de Java 2 Standard Edition (J2SE) 1.5, codenaam "Tiger", komt hier echter verandering in.

De grootste vernieuwing is zonder twijfel de invoering van Java generics (JSR-14)[1]. Dit is nog het meest te vergelijken met templates in C++. Daarnaast werd ook gehoor gegeven aan de smeekbedes van ontwikkelaars om metadata te kunnen koppelen aan Java-klassen. Dit mondde uiteindelijk uit in de invoering van de Metadata Facility (JSR-175). Andere nieuwigheden zijn o.a. de foreach-constructie, auto-/unboxing, typesafe enumerations, variabel aantal argumenten (zie ellipsis in C++) en static imports. Er bestaan reeds enkele tutorials die alle nieuwigheden kort toelichten, bijvoorbeeld [3].

Enumerations zijn een speciaal soort klassen, en zouden dus ook getagd moeten kunnen worden met PDL.

8.2 Generics

Gedetailleerde bespreking van generics ligt buiten de scope van dit werk, maar wordt bijvoorbeeld in [5] behandeld. De officiële specificatie is JSR-14 [4]. De twee belangrijkste pijlers zijn parameterized types en generic methods, waarvan enkel de eerste onmiddelijk relevant is voor dit werk.

[1] JSR staat voor Java Specification Request. Dit zijn specificaties die voortvloeien uit verzuchtingen van Java-gebruikers aan het Java Community Process (JCP). Daar wordt een specificatie bedacht, die een volledig beoordelingsproces doorloopt alvorens (mogelijks) officieel erkend te worden.

```
Collection<Coin> coins=new ArrayList<Coin>();//generics
coins.add(Coin.EENCENT);//enum
coins.add(Coin.EENEURO);
for(Coin coin: coins){//foreach
  System.out.println(coin.toString());
}
```

Figuur 8.1: Eenvoudig voorbeeld van generics en enkele andere nieuwigheden.

Net zoals bij templates in C++, kan men argumenttypes van functies vervangen door type-parameters om generic methods te bekomen. Deze kunnen aan bepaalde constraints onderworpen worden, zoals opgelegde super- of subklassen. Uiteindelijk doel is het toelaten van meer code reuse.

8.2.1 Parameterized types

Men kan bij de definitie van een klasse werken met bepaalde type-parameters. Deze stellen willekeurige types voor, of moeten aan bepaalde grenzen ("upper/lower bounds") voldoen. Op basis van deze informatie kan de compiler bepaalde problemen met types afvlaggen at compile time, of "unchecked warnings" geven at run-time (later meer hierover).
Pas at run-time krijgen de type-parameters een bepaalde waarde, en een specifieke instantie van de klasse wordt gecreëerd. Elke mogelijke invulling van de parameters komt overeen met een apart type, maar alle types horen bij dezelfde klasse. In Fig. 8.1 zien we een voorbeeld. Collection<Coin>, Collection<Integer>, ... zijn allen verschillende types, maar horen bij dezelfde klasse Collection.

Het grote voordeel van deze voorziening is dat men veiliger zal kunnen werken, omdat de compiler meer fouten zal kunnen detecteren. De voorlopig grootste toepassing ligt bij het Collections-framework. Programmeurs kunnen nu, zoals te zien is in Fig. 8.1, echt afdwingen wat de elementtypes zijn van een Collection. Dit neemt dan ogenschijnlijk de nood weg aan @content-tags. Nochtans dient men nog altijd de gewenste implementatieklasse van een Collection- of Map-interface apart op te geven. Bovendien bestaan er ook nog legacy-applicaties, die *geen* gebruik maken van type-parameters, en is er het probleem van @persistent-attributen met een variabel type.

8.2.1.1 Parameterized Collections en Maps vs. raw types

Oude, niet-geparameteriseerde code kan bijna moeiteloos samenwerken met nieuwe code door invoering van het "raw type"-begrip. Dit is het parameterized type zonder de type-parameters, bijv. Collection i.p.v. Collection<String>. Uiteindelijk komt dit gewoon overeen met niet-generieke code, maar door Tiger's compiler te gebruiken blijft er toch nog een zekere type-controle mogelijk. Objecten met een parameterized type kan men immers altijd probleemloos toekennen aan objecten met een raw type, het omgekeerde kan een unchecked warning opleveren. Bij compileren krijgt men dan de waarschuwing dat er problemen kunnen optreden, maar pas als er at runtime daadwerkelijk een overtreding optreedt van de toegelaten types, wordt een

exceptie opgeworpen. Dit mechanisme maakt het mogelijk dat (goed werkende) legacy code kan blijven interageren met generieke code.
Indien men in nieuwe code dus toch met raw types werkt i.p.v. met geparameteriseerde types, zijn PDL-tags nog altijd nodig om automatische persistentie mogelijk te maken. Dit probleem zal zich vermoedelijk niet vaak stellen.

8.2.1.2 Geparameteriseerde attributen

Een speciaal probleem zijn @persistent-attributen met generiek type, zoals bijvoorbeeld attribuut a in Fig. 8.2. Dit is een algemener geval dan het type T bijvoorbeeld. Om dit aan te pakken kan men op volgende drie manieren ingrijpen:

niet: in de metadata wordt geen extra info over de type-parameters opgenomen. Mapping naar een databank zal dus heel algemeen zijn (BLOB's bijvoorbeeld), en altijd grote dataverplaatsingen genereren[2]. Dit is dus niet echt efficiënt, en querying wordt ook zeer geheugenintensief. De pointcuts van de implementor-aspecten zullen in functie van de raw types uitgedrukt worden, zoals tot nu toe het geval was.

restrictie in annotaties van F: de persistence engineer legt bij het annoteren van F vast welke waarden van T een specifieke mapping krijgen. In het implementor-aspect van F zal dan een algemene behandeling komen, enkel steunend op het raw type, of gewoon een no-op implementatie. Andere aspecten zullen voor de opgegeven waarden van T en U een specifiekere mapping toelaten en moeten hogere precedence krijgen om het algemeen of no-op aspect voor F uit te sluiten.
Deze aanpak is al beter, maar code reuse is beperkt. De persistence engineer kan onmogelijk precies weten waar de klasse ooit gebruikt zal worden, zodat goeie keuzes voor T en U moeilijk zijn.

restricties op plaats van instantiatie van F: de persistence engineer annoteert F's eigen broncode niet speciaal, zodat alle types F<T,U> initieel op zelfde voet behandeld worden (enkel no-op implementatie of heel algemene mapping). Speciale annotaties bij gebruik van een specifiek type F<T_A,U_A> (zie Fig. 8.1 bijvoorbeeld) maken het echter mogelijk om een gedetailleerde, efficiëntere behandeling te verkrijgen met aparte aspecten (zie vorig puntje). Deze werkwijze is flexibeler en laat de keuze voor speciale aspecten over aan diegenen die deze kennis bezitten. Dit houdt echter ook een verlies aan controle in.

Aangezien elke verschillende instantie van T en U een ander type creëert, zou het automatisch mappen van elk type op een eigen tabel overkill zijn. Bovendien zou men het databankscript telkens moeten aanpassen en misschien at run-time tabellen kunnen creëren. Naar alle waarschijnlijkheid zijn objecten van een geparameteriseerd type echter enkel @persistent voor tijdelijke opslag of als composite. Ze zijn immers veel te generiek om gewone business objecten te modelleren. Gedetailleerde queries zijn dan geen echte prioriteit, zodat ook de eerste twee geziene aanpakken aannemelijk zijn.

[2] Werken met BLOB's heeft als groot nadeel dat alle componenten van een opgeslagen object meegeblobbed worden, zodat grote overhead optreedt bij opslaan, lezen en queries, zowel qua rekentijd als qua geheugen.

```
@persistent
private F<T,U> a;
```

Figuur 8.2: @persistent-attribuut met generiek type

8.2.2 Conclusie

Velen hadden veel meer flexibiliteit verwacht van generics. Het blijkt een minder expressieve technologie te zijn dan C++-templates, weliswaar veel veiliger. Sommige "nieuwe" features lijken al mogelijk enkel en alleen door interfaces te gebruiken. Het enige nut dat nu al duidelijk is, waarschijnlijk ook een van de bestaansredenen van generics, is het specificeren van de elementtypes van Collections en Maps. Nu bestaat er een vangnet at compile-time, en code kan veel leesbaarder worden. Geneste aspecten die de elementtypes controleren, iets dat we eerder reeds suggereerden in 3.2.7, zijn niet meer nodig. Wellicht zal het nog wat duren eer andere toepassingen duidelijk zullen worden.

8.3 Metadata

Een populaire toevoeging aan Java is de Metadata Facility, vastgelegd in JSR-175 [2]. "Metadata" betekent eigenlijk "over data" (cf. metafysica), en slaat op alle extra informatie die hoort bij de normale data, hier de broncode. Voorbeelden ervan zijn copyright-informatie, persistentie-annotaties, XDoclet-tags[3] voor EJB-Beans, ... Tot op vandaag moet men dergelijke dingen apart bijhouden, vnl. in XML-, properties-, ...-bestanden, ofwel in javadoc-tags gieten (bijvoorbeeld XDoclet). Dit heeft geleid tot een verscheidenheid aan mogelijke systemen, die elk op andere manieren gebruikt moeten worden. Vaak vereist dit toegang tot het bestandssysteem, waar op een afgesproken plaats de nodige bestanden voor handen zijn. Inlezen is ook niet gevarenvrij. Metadata hangt bovendien nauw samen met de corresponderende broncode en moet dus synchroon blijven ermee. Dit wordt eigenlijk opnieuw bemoeilijkt. Indien javadoc gebruikt wordt, stelt dit probleem zich niet, maar is men de informatie kwijt at runtime.

Tiger's Metadata Facility moet eenduidige verwerking van metadata vergemakkelijken, en zal waarschijnlijk ook leiden tot ontwikkeling van nieuwe standaarden, bijv. voor EJB-Beans[4]. Naast de sterke verbondenheid met de code die het annoteert, kan men ook aangeven wanneer de metadata beschikbaar moet zijn ("retention policy"). Tabel 8.1 geeft de verschillende mogelijkheden aan. Er wordt bij Tiger één gebruiksklaar annotatietype (@Overrides) meegeleverd, dat alle compilers moeten ondersteunen. Automatische ondersteuning voor eigen annotatietypes is niet meteen terug te vinden in de specs, maar we veronderstellen dat dat nog aangepast zal worden.

We gaan kort uitleggen hoe men metadata kan definiëren en gebruiken (niet de technische details uit Tabel 8.1), en dan deze kennis toepassen op PDL.

[3] http://xdoclet.sourceforge.net/xdoclet/index.html

[4] In de nieuwe EJB 3.0 zullen deployment descriptors zelfs volledig vervangen worden door metadata tags!

zichtbaarheid	retention policy			hoe inlezen
	SOURCE	CLASS	RUNTIME	
broncode	ja	nee	nee	javadoc-doclet
class-bestand	ja	ja	nee	enkel annotatietypes inladen in JVM
JVM	ja	ja	ja	reflectie

Tabel 8.1: Verschillende retention policies.

```
@Documented
@Target({TYPE,FIELD,METHOD,CONSTRUCTOR,PACKAGE})
@Retention(SOURCE)
public @interface Deprecated{
  public String since();
  public String why() default "";
}
public interface PostKantoor{
  public void zendBrief(Brief brief);
  @Deprecated(since="1965",why="Vogelpest")
  public void stuurPostDuif(Bericht bericht);
}
```

Figuur 8.3: Definitie van een nieuw annotatietype en gebruik ervan.

8.3.1 Werking

In Fig. 8.3 zien we alle facetten van annotatietypes in actie. We definiëren een annotatietype @Deprecated dat aangeeft dat een bepaald package, type, methode, constructor of attribuut afgeraden wordt (zie @Target-tag).

Een annotatietype heeft ongeveer dezelfde definitie als een interface, maar er zijn enkele beperkingen:

- geen overerving mogelijk. Alle annotatietypes erven automatisch over van de interface Annotation, maar zijn als het ware final. Overgeërfde methoden zoals equals(...) zijn NIET oproepbaar.
- methoden mogen geen parameters hebben en mogen evenmin geparameteriseerd zijn (bounded wildcards zijn toegelaten als Class het return type is).
- return types mogen enkel (arrays van) primitieve types, String, Class, enumeraties en annotatietypes zijn.
- geen verwijzingen naar zichzelf of cirkelverwijzingen (@A->@B->@A, ...).
- geen throws-clause.

De methoden komen eigenlijk overeen met attributen van het annotatietype. Enkel als er een default-waarde opgegeven wordt, is men bij gebruik niet verplicht een waarde op te geven.

voordelen	nadelen
dichtbij de broncode bewaard	enkel aanwezig in broncode
uitbreidbaar	geen standaard voor metadata
makkelijk te verwerken	geïntroduceerde attributen niet te annoteren

Tabel 8.2: Voor- en nadelen van gebruik van javadoc als tagging-mechanisme.

We zien ook dat het nieuwe annotatietype zelf geannoteerd wordt door zogenaamde meta-annotatietypes[5]. Standaard bestaan er drie van:

@Target: geeft aan waar het annotatietype bij kan staan. Een meta-annotatietype heeft ANNOTATION als @Target. Sommige annotatietypes mogen enkel als attribuut van andere gebruikt worden, en dan specificeert men @Target({}). De accolades duiden op initialisatie van een (lege) rij.

@Documented: duidt aan dat javadoc het annotatietype mag opnemen in de standaarddocumentatie.

@Retention: geeft aan waar de annotatie zichtbaar is (zie Tabel 8.1).

In Fig. 8.3 wordt slechts één keer de nieuwe annotatie gebruikt, en dit bij een methode. Allebei de velden worden geïnitialiseerd[6]. De opgegeven metadata is opgenomen in de broncode, zodat een aangepaste doclet deze data kan lezen en bijvoorbeeld waarschuwingen genereren in de documentatie.

8.3.2 Toepassing op PDL

Eerder (in 3.3) hebben we al oppervlakkig de voor- en nadelen van javadoc als tagging-vehikel bekeken. Tabel 8.2 vat alles samen.

Waarom is het een nadeel dat de javadoc-tags door de compiler genegeerd worden en uit het class-bestand weerhouden worden? Ten eerste dient men altijd de persistentiebestanden met de gecompileerde code te bundelen, omdat die enkel uit de broncode gegenereerd kunnen worden. Dit kan wel veel schijfruimte innemen, maar zo'n groot probleem is dit niet. Sterker nog, men verhindert zelfs dat interne details, die ten overvloede aanwezig zijn in de persistentiebestanden, zichtbaar zijn.

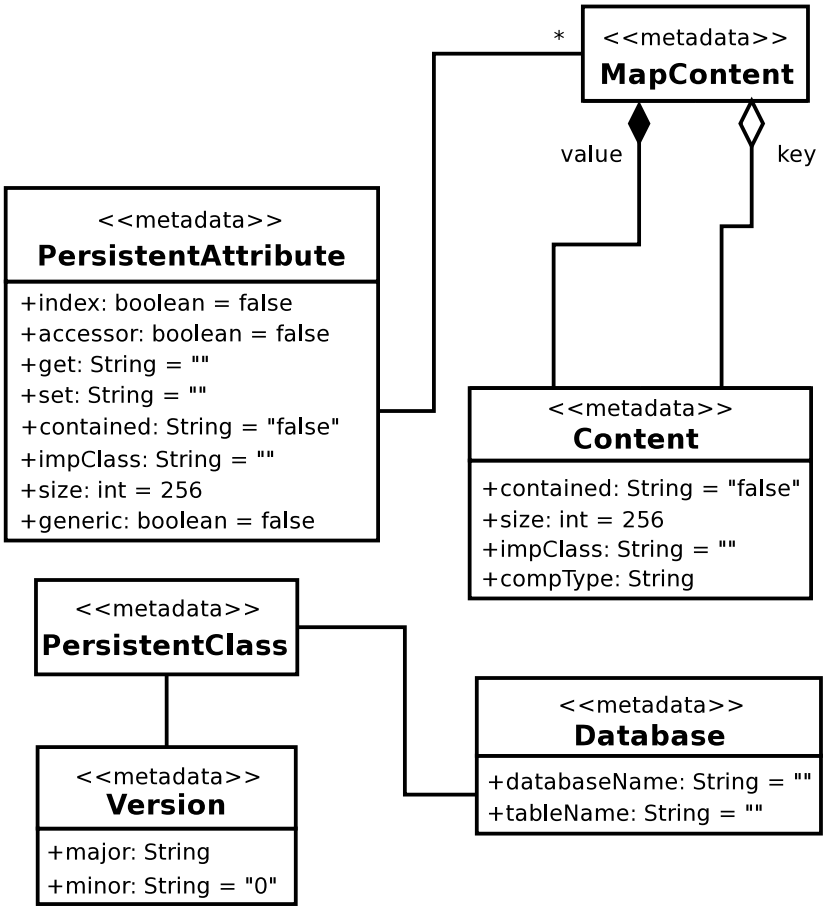
De tweede reden kan echter wel belangrijk zijn[7]. Momenteel kan men via ITD attributen injecteren in klassen. Indien men zo'n attribuut @persistent wil maken, dient men PDL-tags te gebruiken. Gezien javadoc enkel in broncode bewaard blijft, is source code weaving nodig[8].

[5] Deze mogen onderling wél cirkelverwijzingen bevatten.

[6] Hier wordt telkens de naam van het attribuut vermeld en een gelijkheidsteken. Bij zogenaamde single-member annotations, dit zijn annotatietypes met slechts één attribuut dat "value" heet, is dit niet nodig (zie bijvoorbeeld de meta-annotatietypes).

[7] Bij het schrijven van dit werk is men nog volop bezig aan de laatste grote release van AspectJ die J2SE 1.5 nog niet ondersteunt. In een mail [8] op de AspectJ mailinglijst meldde A. Colyer wel dat een "declare annotation"-constructie voorzien wordt.

[8] Tot versie 1.1 beta 3 kon AspectJ dit nog doen, daarna werd (definitief) overgestapt naar bytecode weaving.



Figuur 8.4: Semi-UML schema voor een PDL-implementatie op basis van de Metadata Facility van Tiger. Alle annotatietypes hebben CLASS als retention policy.

```
@persistence.annotations.PersistentClass(version=@persistence.
    annotations.Version(major="01"))
public class Course {
  @persistence.annotations.PersistentAttribute(accessor=true,size=6)
  private String courseID;
  @persistence.annotations.PersistentAttribute(accessor=true)
  private String courseName;
  @persistence.annotations.PersistentAttribute(accessor=true)
  private int courseWeight;
  @persistence.annotations.PersistentAttribute(accessor=true, content
     ={@persistence.annotations.MapContent(value=@persistence.
     annotations.Content(compType="casestudy.Course"))})
  private Collection prerequisites;
  ...
}
```

Figuur 8.5: Broncode van de klasse Course, nu met metadata-tags geannoteerd.

Desondanks komen de @persistent-tags dan niet in de geannoteerde klasse terecht, maar in gegenereerde code voor het aspect. In dit geval is javadoc dus ontoereikend.
Bij bytecode-weaving is het belangrijk dat de PDL-annotaties nog aanwezig zijn in het class-bestand. Pas na het weaven, dus pas in het class-bestand is de metadata aanwezig. Indien we dus PDL inbedden in annotatietypes met retention policy "CLASS", kunnen we dit makkelijk bekomen, en is het probleem opgelost.

Fig. 8.4 stelt deze nieuwe aanpak voor. Principieel zijn er twee soorten metadata-tags, namelijk @PersistentClass en @PersistentAttribute. De eerste annoteert klassen, de tweede attributen, zodat de oorspronkelijke @persistent-tag uit 3.2.3 in tweeën gesplitst is. De overige tags komen overeen met die uit 3.2. Door het ontbreken van overerving, moeten wel overal @MapContent-tags gebruikt worden, waarvan het key-gedeelte enkel zinvolle waarden bevat bij de beschrijving van een Map[9]. Het compType-attribuut van @Content-tags is onnodig als parameterized Collections gebruikt worden.

Voordelen van at run-time zichtbare PDL-annotaties zijn er voorlopig niet:

- run-time weaving is nog niet beschikbaar in AspectJ;
- als de databank niet aangepast is, heeft het geen zin at run-time aspecten te genereren. Bovendien is het toelaten van tabelcreatie en -vernietiging aan jan en alleman niet veilig.

Fig. 8.5 toont opnieuw de klasse Course, nu geannoteerd via Tiger's Metadata Facility.

[9] Null kan niet gebruikt worden als verstekwaarde, dus een @Content-tag met compType "none" lijkt de enige mogelijkheid als default. Dit is niet echt elegant, maar de Metadata Facility laat niets beters toe.

Hoofdstuk 9

Besluit

Het combineren van de sterkste punten van het PDLF-framework en AOP laat inderdaad meer hergebruik van code toe. De PDL-tags zijn declaratief en geven Java net dat tikkeltje meer expressiviteit. De komst van de Metadata Facility vergroot bovendien de kans op een heuse standaard voor annotatie. Generics maken bepaalde tags overbodig.
Er dient verder onderzocht te worden in hoeverre PDL-tags toepasbaar zijn op bijvoorbeeld C++ of C# en in welke mate de droom van een universele persistentielaag van aspecten over verschillende programmeertalen heen, haalbaar is. Toepasbaarheid van de persistentiebestanden op JDO of Hibernate, en in J2EE-omgevingen staat ook nog open voor onderzoek. Het doel van persistentiebestanden is alleszins dat enkel zij zouden volstaan als input van persistentieframeworks.

Aspecten van hun kant laten nu ook een gestructureerde, fysische scheiding van business model en persistentielaag toe. Dit stimuleert hergebruik van de business logic en bevrijdt de programmeurs daarvan volledig van het persistentiejuk. De persistence engineer uit het PDLF-framework neemt die verantwoordelijkheid over.
Door de extractie van de persistentielogica naar aspecten, kan men deze ook beter uitbouwen. Van fundamenteel belang zijn de gegenereerde implementor-aspecten die de persistentiecode bevatten, en het introducer-aspect dat de persistentielaag verbindt met het business model. Ondanks het ontbreken van geavanceerde overervingsmogelijkheden in AspectJ, kan men de gefragmenteerde implementor-aspecten makkelijk georganiseerd doen samenwerken om databankconnecties te recycleren, transacties op te starten, ... Tevens kan men caches inpluggen, ... Of gedistribueerde en multi-user werking ook met aspecten opgelost kan worden is stof voor afzonderlijk onderzoek.
Uiteindelijk heeft men een heel modulair opgebouwd geheel, dat zelfstandig in een jar gebundeld kan worden en door een goeie keuze van pointcut-primitieven (“execution” i.p.v. “call”) met weinig overhead geweven kan worden met het business model in een applicatie.

Het weven van de aspecten met de interface Persistent daarentegen, vergt heel veel tijd. Ook at run-time, wanneer het juiste advice van implementor-aspecten gekozen moet worden, gaat extra tijd verloren. Hoe meer aspecten en advice er zijn, hoe erger het wordt.
Het eerste probleem kan opgevangen worden door slechts op “milestone”-momenten in de ont-

wikkelcycli nieuwe implementor-aspecten te genereren en te weven met Persistent. Zolang in het begin enkel het introducer-aspect bestaat, kan men onder andere werken met mock objects. Het tweede probleem is echter niet zo erg: de toegang tot de databank door bijvoorbeeld de JDBC-driver neemt grootteorden meer tijd in beslag dan de overhead bij advice-selectie.

Andere problemen die we tegenkwamen hadden te maken met de onderlinge precedentie tussen de geavanceerde aspecten. Men moet niet enkel rekening houden met bestaande aspecten, ook de toekomstige moeten genoeg ademruimte krijgen. Nieuwe (visuele) tools die conflicten kunnen opsporen en design patterns of best practices die dit regelen, zullen onontbeerlijk worden in de toekomst.

Wat automatische generatie betreft, kunnen de meeste dingen declaratief gespecificeerd worden ofwel gegenereerd. De TableComponent-hiërarchie zou bijvoorbeeld flexibeler gemaakt kunnen worden afhankelijk van informatie uit extra PDL-tags. De visitors daarentegen, de feitelijke codegeneratoren dus, en sommige geavanceerde aspecten dienen wel nog zelf geïmplementeerd te worden.

Als besluit concluderen we dat AOP een veelbelovende technologie is die veel vooruitgang biedt bij onder andere het persistentievraagstuk. Enkele technologische (snelheid) en semantische (precedentie en overerving) mankementen dienen nog verbeterd te worden.

Bijlage A

Broncode

Op http://faramir.ugent.be/thesissen/aspectgenerator/code.zip staat alle code die nodig is om de besproken resultaten te bekomen. Alles werd ontwikkeld in Eclipse 2.1.3 of 3.0M7 (met plugin voor J2SE 1.5[1]). Er werd getest op een MySQL-databank (versie 4.1.1-alpha-standard).

Er zijn drie afzonderlijke programma's, zie Tabel A.1. Tabel A.2 toont alle packages die daarin gebruikt worden en Fig. A.1 toont andere belangrijke bestanden en directories.

Meer details zijn te vinden in een readme.txt bij de code.

programma	argumenten	doel
PersistenceIntroducer-Generator.java	introducer.xsl␣<relatief pad naar klassenlijst>␣<relatief pad>/PersistenceIntroducer.java	introducer-aspect
PersistenceImplementorGenerator.java	implementor.xsl␣<relatief pad naar klassenlijst>␣<relatief pad naar persistentiebestanden>␣<relatief pad naar implementor-aspecten>	implementor-aspecten
doc.xml	geen (gewoon ant doc.xml)	persistentiebestanden

Tabel A.1: Uitleg over de gebruikte programma's.

[1] Een speciale plugin is nodig om in Eclipse 3.0 Tiger te gebruiken, zie http://www.genady.net/cgi-bin/dl-counter/download.pl?file=jdk15v110 of http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/jdt-core-home/r3.0/main.html#updates.

AspectGenerator	
casestudy	M. Matars casestudy
javapro	D. Jordans JDOQL-query
persistence	parser voor persistentiebestanden, ...
persistence.annotations(*)	annotatietypes voor de PDL-tags
persistence.cache	eenvoudige cache-logica
persistence.cache.exceptions	cache-excepties
persistence.exceptions	persistentie-excepties
persistence.table	TableComponent-hiërarchie
persistence.transaction	supertransactie-logica
persistence.type	Persistent, ConnectionFactory, ...
persistence.visitor	visitors
testApp	extreme testapplicatie
testApp.testPackage	idem
tests	JUnit-tests

Tabel A.2: Alle gebruikte packages. Code van (*) zit in een afzonderlijk zip-bestand, omdat het gebruik maakt van nieuwe features uit J2SE 1.5.

AspectGenerator		
|—	**PersistenceIntroducerGenerator.java**	generatie introducer-aspect
|—	introducer.xsl	XSLT voor introducer-aspect
|—	**PersistenceImplementorGenerator.java**	generatie implementors
|—	implementor.xsl	XSLT voor implementors
|—	Script.java	gegenereerd databankscript
|—	**doc.xml**	doclet voor PDL-tags
|—	aspects	
|	+—...	gegenereerde aspecten
|—	doc	
|	+—...	MainDoclet.java (doclet), ...
|—	templates	
|	+—...	JET-templates
+—	xml	
	+—...	persistentiebestanden

Figuur A.1: Directory-structuur met de bestanden die niet in de package-structuur van Tabel A.2 voorkomen. Onderlijnde namen stellen directories voor, "..." zijn alle bestanden van een directory en de vetgedrukte namen stellen de programma's voor die we gemaakt hebben.

Bibliografie

[1] BASCH, M. en SANCHEZ, A. (2003). Incorporating aspects into the UML. Workshop on aspect-oriented modeling with UML (2003), San Francisco, California.

[2] BLOCH, J. et al. (2003). A program annotation facility for the Java^{TM}programming language: JSR-175 public draft specification [elektronische versie]. Opgehaald in februari 2004 vanaf http://jcp.org/aboutJava/communityprocess/review/jsr175/index.html.

[3] BLOCH, J. en GAFTER, N. (2004). Forthcoming Java^{TM}programming language features [elektronische versie]. Opgehaald in februari 2004 vanaf http://java.sun.com/j2se/1.5/pdf/Tiger-lang.pdf.

[4] BRACHA, G. et al. (2001). Adding Generics to the Java programming language: JSR-14 participant draft specification [elektronische versie]. Opgehaald in maart 2004 vanaf http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html.

[5] BRACHA, G. (2004). Generics in the Java programming language [elektronische versie]. Opgehaald in februari 2004 vanaf http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[6] COLYER, A. (2004a). Re: Incremental and runtime weaving support ? [e-mail]. Gepost op 31 maart 2004 aan de AspectJ mailinglijst (mailto:aspectj-users@eclipse.org).

[7] COLYER, A. (2004b). Re: Reducing code overhead [e-mail]. Gepost op 23 april 2004 aan de AspectJ mailinglijst (mailto:aspectj-users@eclipse.org).

[8] COLYER, A. (2004c). Re: Is it on the cards for PCDs to be defined in terms of annotations? [e-mail]. Gepost op 12 mei 2004 aan de AspectJ mailinglijst (mailto:aspectj-users@eclipse.org).

[9] DATE, C.J. (2000). An introduction to database systems. Reading, Massachusetts, Addison-Wesley, 938 p.

[10] FILMAN, R.E. en FRIEDMAN, D.P. (2000). Aspect-Oriented Programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Minnesota.

[11] GAMMA, E., HELM, R., JOHNSON, R. en VLISSIDES, J. (1995). Design patterns: elements of reusable object-oriented software. Boston, Addison Wesley, 395 p.

[12] GRADECKI, J.D. en LESIECKI, N. (2003). Mastering AspectJ: Aspect-Oriented Programming in Java. New York, John Wiley & Sons, 456 p.

[13] HANNEMANN, J. en KICZALES, G. (2002). Design pattern implementation in Java and AspectJ. ACM SIGPLAN notices, 37(11), 161-173.

[14] JORDAN, D. (2002). JDOQL: the JDO query language. Java Pro, 6(7), 14-21.

[15] KELLER, W. (1998). Object/relational access layers: a roadmap, missing links and more patterns. Proceedings of the 3rd European conference on pattern languages of programming and computing (1998), Bad Irsee, Duitsland.

[16] LADDAD, R. (2002a). I want my AOP!, part 1 [elektronische versie]. Opgehaald in juli 2003 vanaf http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[17] LADDAD, R. (2002a). I want my AOP!, part 2 [elektronische versie]. Opgehaald in juli 2003 vanaf http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html.

[18] LADDAD, R. (2002a). I want my AOP!, part 3 [elektronische versie]. Opgehaald in juli 2003 vanaf http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html.

[19] MONDAY, P. (2002). Hands-on Java Data Objects [elektronische versie]. Opgehaald in mei 2004 vanaf http://www-106.ibm.com/developerworks/edu/j-dw-javajdo-i.html.

[20] MATAR, M. (2002). PDLF: a methodology for object persistence in Java based on a declarative strategy. Doctoraatswerk, Gent, België, Universiteit Gent, 167 p.

[21] RASHID, A. en CHITCHYAN, R. (2003). Persistence as an aspect. Proceedings of the 2nd international conference on Aspect-Oriented Software Development (2003), Boston, Massachusetts.

[22] ROOS, R. (2002). Java Data Objects. London, Addison Wesley, 240 p.

[23] SHARP, R., FANCELLU, D. en STEPHENS, M. (200). EJB's 101 damnations [elektronische versie]. Opgehaald in april 2004 vanaf http://www.softwarereality.com/programming/ejb/index.jsp.

[24] SUN MICROSYSTEMS (1997). JavaBeans^{TM}API specification [elektronische versie]. Opgehaald in maart 2003 vanaf http://java.sun.com/products/javabeans/docs/spec.html.

[25] VANDENBORRE, K., MATAR, M. en HOFFMAN, G. (2002). Orthogonal persistence using Aspect-Oriented Programming. 1st AOSD workshop on aspects, components, and patterns for infrastructure software (2002), Enschede, Nederland.