KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculteit
Ingenieurswetenschappen

Ontwerp en implementatie van een raamwerk voor aspect weavers binnen het Chameleon raamwerk

Jens De Temmerman

Thesis voorgedragen tot het
behalen van de graad van
Master in de
ingenieurswetenschappen:
computerwetenschappen

Promotor:
Prof. dr. ir. E. Steegmans

Assessoren:
Prof. dr. D. Clarke
Dr. ir. M. van Dooren

Begeleider:
Dr. ir. M. van Dooren

Academiejaar 2010 – 2011

Voorwoord

Graag wil ik mijn promotor, professor dr. ir. Eric Steegmans bedanken om dit onderwerp aan te reiken en samen met de andere leden van het departement computerwetenschappen mijn interesse te wekken in software ontwikkelingsmethodologie. Ook mijn begeleider, dr. ir. Marko van Dooren dank ik voor de vele raadgevingen en de uitstekende begeleiding bij deze masterproef.

Verder wil ik graag mijn ouders bedanken om mij de kans te geven te studeren en me te blijven motiveren. Tenslotte wil ik ook al mijn vrienden danken om steeds een uitlaatklep te zijn op de stressvolle momenten en in het bijzonder mijn vriendin, Tinne, die er op elk moment voor me was en een voortdurende steun geweest is bij het schrijven van deze masterproef.

Jens De Temmerman

Inhoudsopgave

Samenvatting

Deze masterproef beoogt het ontwerp en de implementatie van een raamwerk voor aspect weavers binnen het Chameleon raamwerk. Er werd onderzocht welke functionaliteit en abstracties nodig zijn om een aspect wever te implementeren. Aan de hand hiervan werd een raamwerk opgesteld binnen het Chameleon raamwerk, dat een taalonafhankelijke houvast biedt voor het implementeren van aspect weavers. Om het raamwerk te evalueren werd eerst een concrete aspect weaver voor Java geïmplementeerd die qua syntax en functionaliteit gebaseerd is op AspectJ. Er wordt gekeken hoeveel herbruik mogelijk gemaakt wordt door het raamwerk en de mate van uitbreidbaarheid wordt getoetst door bepaalde - in AspectJ niet aanwezige - pointcuts te implementeren. Verder werd ook een Eclipse plugin ontwikkeld die, gebruik makende van de in Chameleon aanwezige mogelijkheden, een functionele editor biedt om aspecten te definiëren voor de Java weaver. Uit de evaluatie van deze weaver blijkt dat aan de vooropgestelde doelen - het identificeren van de nodige functionaliteit en abstracties en het maken van een herbruikbare en uitbreidbare implementatie, voldaan is. De ontwikkelde weaver voor Java is zeer functioneel en uitbreidingen zijn eenvoudig te realiseren. Vervolgens werd een tweede weaver geïmplementeerd, voor een uitbreiding van Java: JLo. Het doel hierbij was om te toetsen of de herbruikbaarheid ook stand houdt voor weavers voor verschillende programmeertalen. Ook hier wordt een positief evaluatie gemaakt. Niet alleen is de ontwikkelde weaver voor Java volledig herbruikbaar om ook JLo-code te weaven, bovendien is het uitbreiden van deze weaver met JLo-specifieke elementen eenvoudig en is er ook hierbij veel herbruik mogelijk.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Deel I

Inleiding

Hoofdstuk 1

Inleiding

In de inleiding wordt de probleemstelling geschetst die deze masterproef aanpakt. Ook de verdere structuur van de tekst wordt kort overlopen.

1.1 Probleemstelling

In het paradigma van object geörienteerd programmeren wordt veel aandacht besteed aan het modulariseren van de verschillende facetten van het systeem, de *concerns*. Sommige van deze concerns komen echter op verschillende plaatsen en verwoven met andere concerns voor. Aspect geörienteerd programmeren biedt hier een oplossing en zal deze *crosscutting* concerns modulariseren, wat de onderhoudbaarheid van het software systeem ten goede komt. De concerns worden op een enkele plaats beschreven en waar nodig in de applicatie ingevoegd door een automatisch proces, de aspect weaver. Aspect geörienteerd programmeren wordt verder beschreven in hoofdstuk 2.

Hedendaagse aspect weavers zijn echter slechts geschikt voor een enkele (programmeer)taal. Net zoals andere tools voor softwareontwikkelaars, zoals editors of refactoring tools, bevatten ze (een deel van) de taal specificatie. Het elk afzonderlijk implementeren van deze taal specificatie in al deze tools is niet bevorderlijk voor de correctheid of onderhoudbaarheid ervan. Bovendien zorgt het ervoor dat deze tools enkel kunnen werken op broncode van een bepaalde programmeertaal. Het Chameleon project wil hier verandering in brengen door een raamwerk voor meta modellen aan te bieden. Deze tools werken dan op het meta model, waardoor de taal specificatie slechts op een enkele plaats gedefinieerd is. Bovendien zorgt het ervoor dat er herbruik mogelijk is tussen tools voor verschillende programmeertalen, waardoor deze sneller ontwikkeld kunnen worden. Chameleon wordt verder beschreven in hoofdstuk 3.

Deze masterproef stelt de volgende doelen:

1. Onderzoeken welke functionaliteit en abstracties nodig zijn voor het weaven van aspecten
2. Een ontwerp en implementatie maken voor deze functionaliteit als die nog niet binnen het Chameleon raamwerk aanwezig is
3. Ontwikkelen van een concrete tool voor het weaven van aspecten waardoor het raamwerk geëvalueerd kan worden

1.2 Structuur van deze tekst

In hoofdstuk 2 wordt aspect geörienteerde software ontwikkeling verder geduid. Voor Chameleon wordt hetzelfde gedaan in hoofdstuk 3. Hoofdstukken 4, 5 en 6 schetsen hoe de drie belangrijkste pijlers van aspect geörienteerd programmeren - het aspect, het advice en de pointcut - geïmplementeerd worden binnen het raamwerk. De hoofdmoot van deze tekst, hoofdstukken 7 en 8, gaan over hoe het weaving *proces* - het combineren van de aspecten met de basis code - in het raamwerk voorgesteld wordt. Om dit raamwerk succesvol te kunnen evalueren zijn twee specifieke weavers ontwikkeld, voor Java en JLo. De details van deze implementaties en de evaluatie van het raamwerk, alsook een algemene evaluatie van Chameleon, zijn te vinden in hoofdstuk 9. In hoofdstuk 10 wordt verwezen naar gerelateerd werk. Tenslotte volgt een conclusie in hoofdstuk 11, waar ook eventueel toekomstig werk besproken wordt.

Hoofdstuk 2

Aspect geörienteerde software ontwikkeling

In dit hoofdstuk wordt aspect geörienteerde software ontwikkeling omschreven. Eerst worden de belangrijkste kernwoorden kort geschetst aangezien deze in de hele tekst zullen terugkomen. Vervolgens wordt het probleem geduid waar aspect geörienteerde software ontwikkeling een antwoord op wil bieden en wordt er ingegaan op hoe dit concreet gebeurt.

2.1 Lijst van termen

Hieronder volgt een korte opsomming van de meest gebruikte termen rond aspect geörienteerd software ontwerp, met een korte uitleg. Het doel is om deze concepten, die verder in de tekst vaak gebruikt zullen worden, eenduidig vast te leggen. Deze zijn gebaseerd op de definities gegeven op de *Aspect Oriented Software Development (AOSD)* glossary[1].

Advice Een onderdeel van een aspect dat helpt om *cross-cutting* concerns te modulariseren: code die voorheen gedupliceerd was, staat nu binnen een enkel advice. Bovendien bevat het advice ook alle informatie die nodig is om deze code op de juiste plaats in het programma in te voegen.

Aspect Een bepaalde concern binnen software ontwikkeling. Aspecten zijn, ten opzichte van de dominante decompositie, concerns die *crosscutten*.

Joinpoint[2] De joinpoints zijn de plaatsen waar advice kan ingevoegd worden. Welke joinpoints er beschikbaar zijn hangt af van zowel de taal van het basis model, als de

[1] http://www.aosd.net/wiki/index.php

[2] Volgens de precieze semantiek zijn joinpoints plaatsen in de *control flow* van de applicatie en is de plaats in de code die overeenkomt met dit punt, de statische projectie van het joinpoint, de joinpoint *shadow*. In deze tekst wordt altijd joinpoint gebruikt. De context maakt duidelijk of de *joinpoints* bedoeld worden of de *joinpoint shadows*, maar dit maakt voor het conceptueel begrijpen van deze tekst verder niet uit.

gebruikte aspect taal. Een typisch voorbeeld van joinpoints voor object geörienteerde programmeertalen zijn methode oproepen of het afhandelen van excepties.

Pointcut Een pointcut is een bepaalde set van joinpoints. Bovendien kunnen pointcuts ook de context van het joinpoint blootstellen.

2.2 Probleemstelling

2.2.1 Inleiding

Computersystemen kennen de laatste decennia een enorme groei. De bekende wet van Moore stelt dat het aantal transistors in een geïntegreerde schakeling elke twee jaar verdubbelt. Dit heeft natuurlijk een impact op de snelheid, geheugencapaciteit en andere aspecten van een computersysteem, die ook een bijna exponentiële stijging kennen. Samen met een evolutie van de hardware is er ook een evolutie van de software. Snellere systemen laten complexere applicaties toe. Maar samen met de complexiteit van de applicatie, stijgt ook de complexiteit van het ontwikkelen ervan. Moderne applicaties zoals tekstverwerkers en browsers bevatten miljoenen lijnen code. Daarom zijn er doorheen de tijd verschillende software ontwikkelingsmethodologieën voorgesteld, die ervoor moeten zorgen dat het ontwikkelen en beheren van software niet onmogelijk complex wordt. Het momenteel meest gebruikte paradigma is object geörienteerde software ontwikkeling, met talen zoals Java en C#. Dit paradigma laat toe om het systeem te beschrijven als een collectie van interagerende, herbruikbare objecten, waarbij de verantwoordelijkheden duidelijk gescheiden zijn.

Er zijn echter facetten binnen een software systeem die niet eenvoudig gemodulariseerd kunnen worden binnen het paradigma van object geörienteerde software ontwikkeling. Deze zogenaamde *crosscutting concerns* komen voor op verschillende plaatsen, vermengd met andere concerns. Het is op deze *crosscutting* concerns dat aspect geörienteerd programmeren (*Aspect Oriented Programming* of *AOP*) zich nu toelegt[2].

2.2.2 Scattering en tangling

Het probleem met deze *crosscutting* concerns is tweevoudig: *scattering* en *tangling*. *Scattering* verwijst naar het feit dat een bepaald concern verspreid zit in de code. *Tangling* verwijst naar het feit dat twee concerns verweven zijn met elkaar binnen een enkele module. Hoewel beide problemen vaak samen optreden bij *crosscutting* concerns, moeten ze toch apart bekeken worden. Een eenvoudig voorbeeld van *tangled* code is als volgt. Beschouw een eenvoudig systeem waar gebruikers berichten naar elkaar kunnen versturen:

```
1 public void deliverMessage(User from, String message) {
2         if (from == null || message == null)
3                 throw new IllegalArgumentException();
4
5         messages.add(message);
6 }
```

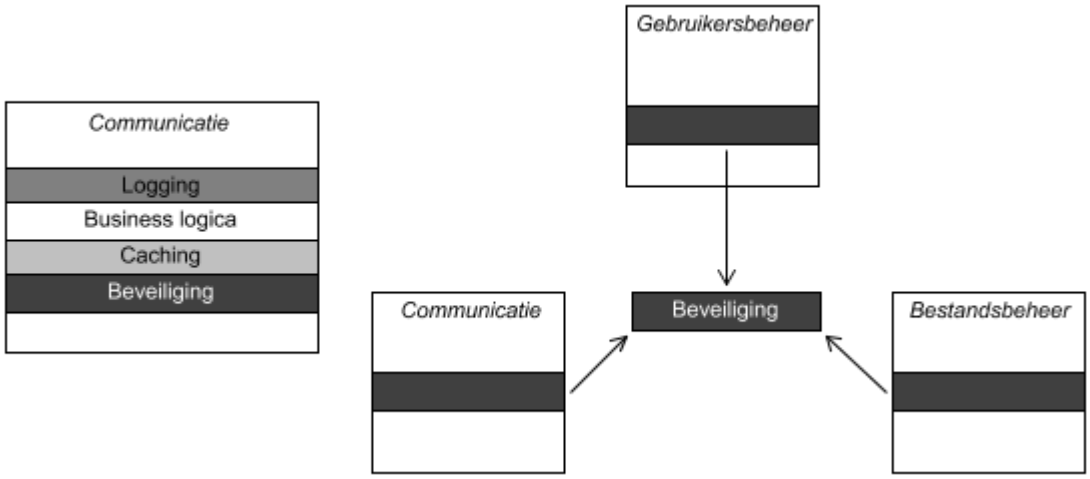
Stel nu dat er twee nieuwe concerns toegevoegd worden: beveiliging, waarbij gekeken wordt of een gebruiker wel de correcte rechten heeft om berichten te ontvangen, en logging. Dit zijn typische voorbeelden van *crosscutting* concerns. De code zal er als volgt uitzien:

```
1  public void deliverMessage(User from, String message) {
2          if (from == null || message == null)
3                  throw new IllegalArgumentException();
4
5          Logger.log("Delivering message");
6
7          if (!SecurityManager.hasPrivillege(this, Privillege.
               RECEIVE_MESSAGES)) {
8                  Logger.log("User has insufficient privilleges
                       to receive messages");
9                  throw new RuntimeException("Insufficient
                       privileges!");
10         }
11
12         messages.add(message);
13
14         Logger.log("Message delivered!");
15 }
```

Het is duidelijk dat dit snel leidt tot onoverzichtelijke code omdat er drie verschillende concerns op een enkele plaats verstrengeld zitten: het versturen van berichten, het uitvoeren van een beveiligingscontrole en het loggen van gebeurtenissen. Een duidelijk voorbeeld van *tangled* code. Het is ook realistisch te denken dat de beveiligingscontroles op andere plaatsen zullen terugkomen, bijvoorbeeld bij het downloaden van bestanden. Dezelfde code om deze controles te doen zal dan op verschillende plaatsen terugkomen. Als hier een aanpassing aan moet gebeuren, moet dit overal gebeuren. Dit komt duidelijk de onderhoudbaarheid van de code niet ten goede. In dit geval spreekt men van een *scattered* concern. Figuur 2.1 op de volgende pagina illustreert deze twee concepten.

2.2.3 AOP op een hoger niveau

In de vorige sectie werd alleen gesproken over broncode. Er wordt een aspect geörienteerde aanpak voorgesteld om de broncode beter te kunnen onderhouden. Het is echter zo dat de *crosscutting* concerns niet alleen op het niveau van de broncode kunnen bekeken worden, maar ook op het niveau van de vereisten en de architectuur. Ook hier zal bijvoorbeeld logging duidelijk voorkomen in verschillende use cases. Door de *crosscutting* concerns op dit niveau te bekijken en niet pas op het niveau van de broncode kan men ervoor zorgen dat de aspecten beter herbruikbaar zijn, omdat deze duidelijk overeen komen met concerns binnen het probleem domein en niet gegroeid zijn uit de broncode. Hier zijn verschillende aanpakken voor te vinden. AspectU [12] definieert een aspect taal voor use cases, analoog aan een aspect taal voor programmeertalen zoals AspectJ. Een andere aanpak [6] is om geen speciale aspect taal voor use cases te definiëren, maar de use cases hiërarchisch te schikken waarbij de *crosscutting* concerns



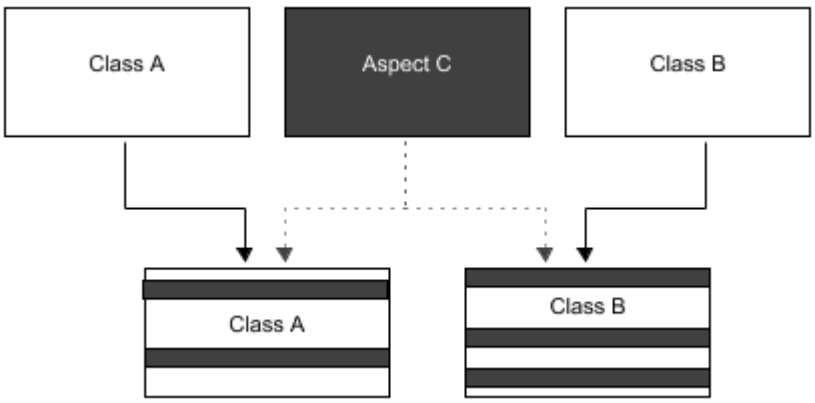
Figuur 2.1: Tangling (links) en scattering (rechts)

enkel voorkomen in de abstracte use cases - slechts op een enkele plaats dus. Er wordt dan gebruik gemaakt van use case overerving en de *extends* relatie om tot de concrete use cases te komen. Indien er naar meer dan enkel broncode gekeken wordt, spreekt men in plaats van over aspect geörienteerd programmeren over aspect geörienteerde software ontwikkeling (*Aspect Oriented Software Development* of *AOSD*).

2.3 Hoe biedt AOP een oplossing?

Het AOP paradigma biedt de mogelijkheid om deze concerns te modulariseren om tot een beter onderhoudbare applicatie te komen. Hiervoor worden de *crosscutting* concerns op slechts een enkele plek beschreven: in een aspect. Binnen een aspect kunnen verschillende elementen gedefinieerd worden die voor een betere modulariteit zorgen. Enerzijds zijn er de *advices*. Deze bevatten de code die voorheen gedupliceerd voorkomt. Anderzijds zijn er de *pointcuts*. Deze kiezen de plaatsen in het programma waar het advice ingevoegd moet worden. Een mogelijke plaats waar advice ingevoegd kan worden is een *joinpoint*. Een pointcut maakt dus een selectie van de joinpoints waar het advice ingevoegd moet worden. Een derde soort elementen die toegevoegd kunnen worden zijn *inter-type declarations*. Op deze manier kan een aspect bestaande klassen wijzigen door er velden of methodes aan toe te voegen, een klasse een bepaalde interface te laten implementeren of deze klasse te laten overerven van een bepaald super type. Er zijn ook nog andere elementen en modellen die binnen AOP aangewend worden om de *crosscutting* concerns te beheren[4], maar deze net beschreven zijn de meest gangbare en vormen dan ook de focus van deze tekst.

Het is dan een ander proces, de weaver, die ervoor zal zorgen dat de *crosscutting* concerns op de juiste manier en op de juiste plaats in het originele programma ingevoegd worden, zoals geïllustreerd in figuur 2.2 op de pagina hiernaast. Een populaire aspect weaver voor Java is bijvoorbeeld AspectJ[3], die ook verder in deze tekst als voorbeeld zal gebruikt worden. Er zijn twee types van weven: het statisch



FIGUUR 2.2: Een wever zal de elementen van het originele programma combineren met de gedefinieerde aspecten

weaven gebeurt bij compile time of load time, terwijl het dynamisch weaven runtime kan gebeuren. Dit heeft als voordeel dat bepaalde aspecten kunnen aan- of uitgezet worden terwijl het programma loopt, zonder de applicatie te moeten herstarten. Bij statisch weaven zou hetzelfde effect kunnen bereikt worden door binnen het advice deze runtime check uit te voeren, maar dit introduceert natuurlijk een overhead. Bij frequent uitgevoerd advice kan runtime weaven tot 26% sneller zijn[11].

Deel II

Een raamwerk voor aspect weavers

Hoofdstuk 3

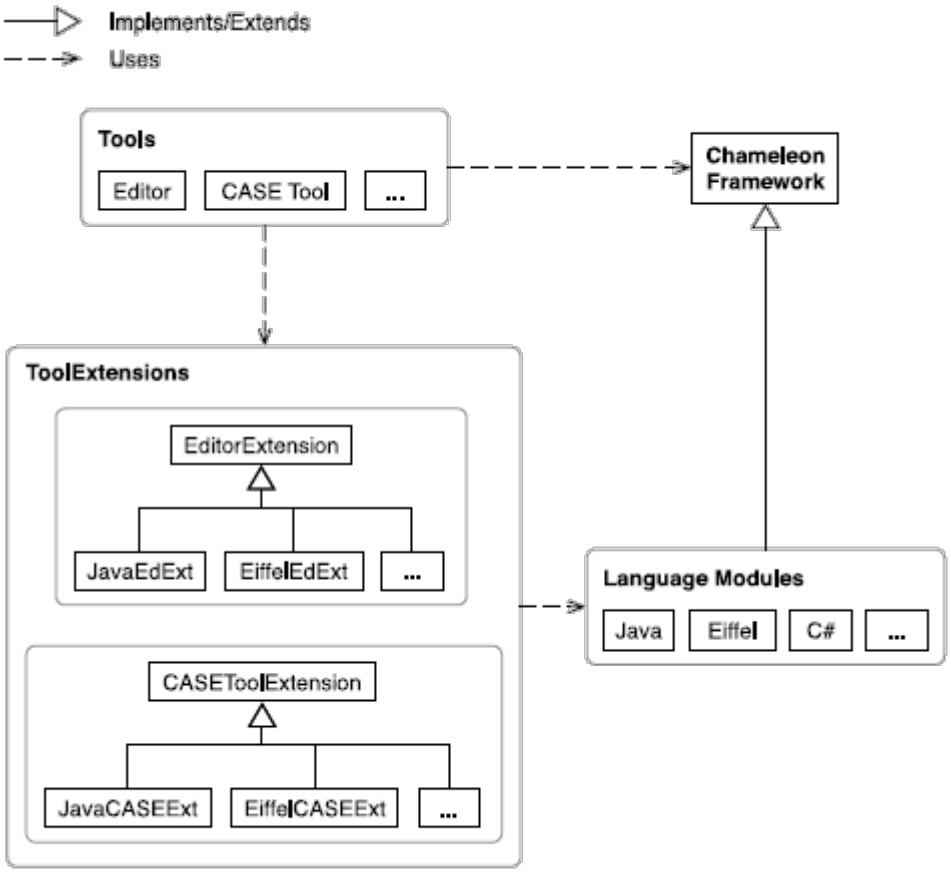
Het Chameleon raamwerk

Deze masterproef is volledig gekaderd binnen het Chameleon raamwerk [14]. Dit hoofdstuk omschrijft kort de probleemstelling, hoe het Chameleon raamwerk dit zal aanpakken en hoe deze masterproef hierbinnen past.

3.1 Probleemstelling

Software ontwikkelaars gebruiken veel hulpmiddelen om een vat te krijgen op de complexiteit van de huidige software: geavanceerde tekst editors, refactoring tools, ... De meeste van deze tools werken niet op de broncode zelf maar parsen deze tot een boom, een zogenaamde abstract syntax tree. In dit opzicht is er tussen de verschillende tools zeker herbruik mogelijk. Maar het probleem is dat deze boom slechts de structuur van de code beschrijft. Het gedrag wordt niet beschreven, wat ook van belang kan zijn. Indien een editor bijvoorbeeld automatische code aanvulling wil aanbieden, moet deze weten welke elementen waar kunnen voorkomen. Zo stelt Eclipse, een populair ontwikkelomgeving voor Java, bij het automatisch aanvullen geen methodes voor die niet toegankelijk zijn, omdat ze bijvoorbeeld de *private* modifier hebben. Deze taal specificatie moet (gedeeltelijk) in elke tool opgenomen worden en wordt zelden herbruikt.

Bovendien is het ook zo dat, door deze taal specificatie op te nemen en te koppelen aan de tool, die tool taal afhankelijk wordt. Dit kan hinderlijk zijn bij het ontwikkelen van een nieuwe taal, aangezien veel software ontwikkelaars twijfelachtig tegenover een programmeertaal staan waar weinig tools voor beschikbaar zijn - dit is voor een deel terecht, aangezien deze tools het een heel stuk makkelijker maken om software te ontwikkelen, maar het is eenvoudig om te zien dat zo er in een vicieuze cirkel terechtgekomen wordt. Weinig tools betekent weinig interesse, weinig interesse bemoeilijkt de ontwikkeling van tools. Een voorbeeld hier is Scala - een object gerichte functionele programmeertaal die op de Java Virtual Machine (JVM) draait. Het was de bedoeling om een IDE te ontwikkelen die, qua mogelijkheden, te vergelijken was met Eclipse voor Java. Deze werd dan ook ontwikkeld als een plugin voor Eclipse. En hoewel Scala veel overeenkomsten heeft met Java was het zeer moeilijk om deze overeenkomsten uit te buiten bij het ontwikkelen van deze plugin [7].



Figuur 3.1: De architectuur van Chameleon[14]

3.2 Oplossing

Chameleon wil een oplossing bieden voor deze problemen door ervoor te zorgen dat tools kunnen werken met abstracties in plaats van concrete specificaties. Chameleon is een raamwerk voor meta modellen van programmeertalen dat gemeenschappelijke entiteiten in programmeertalen zal abstraheren. Taal specifieke zaken worden in aparte taal modules geplaatst. Een analoge constructie is er voor de tools - gemeenschappelijke zaken voor eenzelfde tool voor verschillende talen wordt in een eenzelfde module geabstraheerd, zaken die niet geabstraheerd kunnen worden (omdat ze bijvoorbeeld slechts voorkomen in een enkele taal) staan apart. Figuur 3.1 illustreert deze architectuur.

3.3 Plaats van deze masterproef

Deze masterproef heeft als doel een raamwerk te ontwikkelen voor aspect weavers. Zoals eerder vermeld moet er onderzocht worden welke functionaliteit en abstracties nodig zijn. In de context van de Chameleon architectuur moet er dus gekeken worden wat in de abstracte tool module geplaatst kan worden en wat taal specifiek is. Aangezien er ook weavers worden ontworpen voor Java en JLo, wordt het volledige proces doorlopen. Op het einde van de masterproef zijn dus de volgende modules aanwezig:

- Een module voor de taal onfhankelijke zaken (cfr. de *EditorExtension* uit figuur 3.1)
- Een module die taal specifieke zaken voor de weaver bevat, zowel voor de Java als JLo weaver (cfr. de *JavaEdtExt*)
- Een module voor de Eclipse plugin voor het ingeven van aspecten, zowel voor Java als JLo

Hoofdstuk 4

Aspect

In dit hoofdstuk wordt beschreven hoe een aspect wordt voorgesteld binnen het raamwerk. Verder wordt aangehaald wat de mogelijkheden zijn en hoe deze eventueel uitgebreid kunnen worden.

4.1 Voorstelling

Het aspect is een *top-level* element binnen de hiërarchie van aspect geörienteerd programmeren. Binnen een aspect is het mogelijk om genaamde pointcuts te definiëren waarnaar verwezen kan worden vanuit advice of andere pointcuts - zie sectie 6.1.5 op pagina 28. Bovendien is het natuurlijk ook mogelijk om advices te definiëren. Dit zijn momenteel de enige twee elementen die, binnen een aspect, mogelijk zijn in het raamwerk. *Inter-type* declaraties en aspect variabelen of methodes worden momenteel nog niet ondersteund, aangezien deze masterproef in eerste instantie wou focussen op advices en pointcuts - zie ook sectie 11.2 op pagina 96.

Een aspect is in zekere zin erg te vergelijken met een klasse. Het is een container voor diverse andere elementen - advices, pointcuts - zoals een klasse dit is voor velden en methodes. Het is dan ook niet verwonderlijk dat een groot deel van de functionaliteit reeds aanwezig in het abstracte Chameleon raamwerk, ook zal kunnen herbruikt worden voor aspecten. Een voorbeeld hier is het opzoeken van gedeclareerde elementen: aangezien vanuit het advice, of vanuit andere pointcuts, kan verwezen worden naar de gedeclareerde pointcuts, moet er een mogelijkheid zijn om op te zoeken naar welk element verwezen wordt. Het Chameleon raamwerk maakt dit gemakkelijk: het aspect vermeldt welke elementen gedeclareerd worden - in dit geval de pointcuts - en het element dat de verwijzing doet geeft aan wat de criteria zijn - in dit geval de naam, het aantal parameters en hun type.

4.2 Mogelijke uitbreidingen

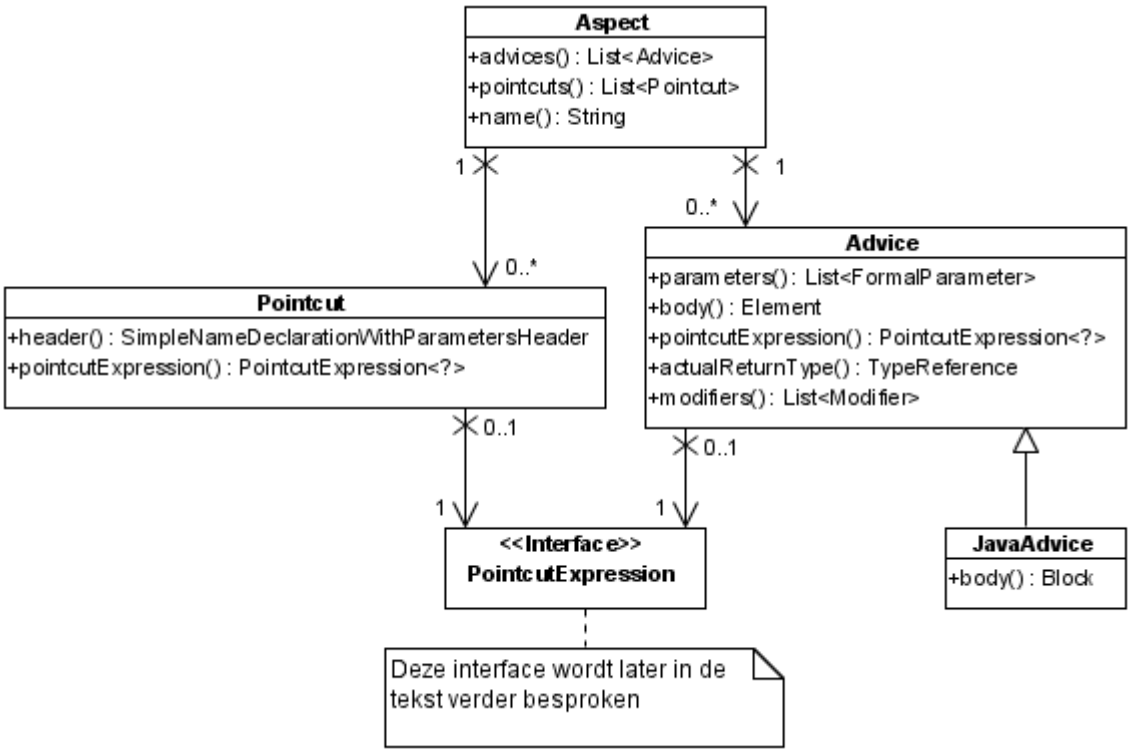
4.2.1 Inter-type declaraties

Met behulp van inter-type declaraties kan een weaver velden en methodes toevoegen aan klassen en de klassehiërarchie aanpassen. Binnen dit raamwerk is dit echter momenteel niet eenvoudig te implementeren. Indien er bijvoorbeeld een nieuwe

methode aan een bepaalde klasse wordt toegevoegd, zal slechts in bepaalde gevallen het mechanisme dat dit type zal opzoeken weet hebben van de nieuwe methode. Er moet verder onderzoek gedaan worden of dit binnen Chameleon mogelijk wordt mits enkele wijzigingen.

4.2.2 Abstracte aspecten, pointcuts, aspect variabelen en methodes

AspectJ laat toe om abstracte aspecten en pointcuts te definiëren, analoog aan abstracte klassen en methodes[1]. Bovendien is het ook mogelijk om variabelen en methodes te declareren in een aspect. Dit valt niet onder de scope van deze masterproef, maar het zo dat er hiervoor vanuit Chameleon ondersteuning is, ook weer omwille van de analogie tussen aspecten en klassen. Het is dus zeker niet zo dat er vanaf nul zou moeten begonnen worden om dergelijke features te voorzien.



Figuur 4.1: Klassendiagram voor Aspect, Advice en Pointcut

[1] Het is niet mogelijk om abstract advice te definiëren omdat advice geen naam heeft, en het dus onmogelijk is om te controleren of het in een subklasse wel geïmplementeerd wordt.

Hoofdstuk 5

Advice

Een advice is gekenmerkt door vier elementen: het type, de parameters, de pointcut waarop dit advice toegepast wordt en de inhoud. In dit hoofdstuk worden deze vier elementen onder de loep genomen. Er wordt gekeken hoe deze gebruikt worden, welke abstracties er voor nodig zijn en hoe ze op een uitbreidbare manier geïmplementeerd kunnen worden in het raamwerk.

5.1 Advice type

Een advice kan op verschillende manieren gecombineerd worden met een bepaald joinpoint. Binnen AspectJ zijn de volgende types beschikbaar:

Before Before advice wordt, zoals de naam het zegt, uitgevoerd voor het uitvoeren van het joinpoint. Indien het advice een exceptie gooit, wordt het joinpoint niet uitgevoerd.

Around Around advice wordt in de plaats van het joinpoint uitgevoerd. Het oorspronkelijk joinpoint kan nul, een of meerdere keren worden uitgevoerd tijdens het advice - met eventueel verschillende parameters - door gebruik te maken van een speciale expressie. In AspectJ is dit **proceed**().

After returning After returning advice wordt uitgevoerd na het joinpoint, maar enkel indien de uitvoering ervan niet voortijdig beëindigd werd door het gooien van een exceptie. After returning advice maakt het ook mogelijk om de return value van de methode die werd uitgevoerd bloot te stellen aan het advice.

After throwing De tegenhanger van after returning is after throwing. Advice van dit type wordt enkel uitgevoerd indien de uitvoering van het joinpoint wel voortijdig beëindigd werd. Bovendien is het mogelijk om te specificeren voor welk type exceptie dit moet gebeuren en kan de exceptie ook worden blootgesteld aan het advice.

After After - ook wel after finally genoemd - combineert after returning en after throwing. Advice van dit type wordt altijd uitgevoerd na het uitvoeren van het joinpoint, ongeacht of de uitvoering ervan voortijdig beëindigd werd of niet.

Al deze types zijn reeds beschikbaar binnen het ontworpen raamwerk. Dit is echter op een uitbreidbare manier geïmplementeerd, zodat nieuwe types eenvoudig toe te voegen zijn. Hiervoor wordt gesteund op functionaliteit die al door Chameleon wordt aangeboden. Elk element binnen Chameleon kan namelijk bepaalde eigenschappen (*Properties*) toegewezen krijgen. Die toewijzing gebeurt onder andere door modifiers - elke modifier wijst een of meer eigenschappen toe aan een bepaald element. Eigenschappen behoren tot een bepaalde taal - zo zijn eigenschappen van methodes zoals *public*, *static* of *final* typisch iets voor object geörienteerde talen. *After, before* en *around* zijn typische eigenschappen die voorkomen in aspect talen.

De set van eigenschappen van een bepaald element kan eenvoudig gecontroleerd worden op geldigheid. Hiervoor moeten er enkele eenvoudige eigenschappen bij het declareren van de eigenschap meegegeven worden, zoals bijvoorbeeld of de eigenschap behoort tot een bepaalde *mutual exclusion* groep of niet. Dit is het geval voor *before, after* en *around* advice: slechts een van deze drie eigenschappen kan toegepast worden op een enkel advice. Deze controles behoren tot de code van het Chameleon raamwerk, hiervoor moet dus erg weinig extra werk gebeuren en er kunnen snel nieuwe eigenschappen toegevoegd worden.

Bij het effectieve weaven zullen de individuele weavers aangeven welke types ondersteund worden. Om een nieuw type advice toe te voegen, moeten dus enkel de volgende stappen gebeuren:

1. Voeg de bijhorende eigenschap toe aan de aspect taal, samen met de eventuele relaties tot andere eigenschappen (*mutual exclusive*, *implies, contradicts*)
2. Voeg een of meer modifiers toe die deze eigenschap toewijzen
3. Voeg nieuwe weavers toe, of pas bestaande aan, die het toegevoegde type ondersteunen

5.2 Parameters

Bij elk advice kunnen ook parameters horen. Dit zijn variabelen die meestal slechts tijdens runtime gekend zijn en voor eenzelfde joinpoint kunnen verschillen tijdens de loop van het programma. Een typisch voorbeeld zijn de parameters van een methode oproep die binnen het advice beschikbaar gemaakt worden, zodat deze bijvoorbeeld gelogd kunnen worden. Deze parameters zijn op te delen in twee soorten: parameters die door het pointcut blootgesteld worden - zoals beschreven wordt in sectie 6.1.3 op pagina 27 en parameters die door het advice zelf blootgesteld worden, zoals de return waarde bij after-returning en de exceptie bij after-throwing advice. Hoe deze parameters precies blootgesteld worden is van weinig verder belang voor het advice - beide soorten zijn bruikbaar bij het uitvoeren ervan.

5.3 Pointcut

De pointcut van een advice bepaalt op welke plaatsen - bij welke joinpoints - dit advice ingevoegd wordt. Bovendien moeten de parameters van het advice ook door het pointcut blootgesteld worden, indien dit niet door het advice zelf gebeurt. Pointcuts en pointcut expressies worden uitvoerig besproken in hoofdstuk 6 op pagina 23.

5.4 Inhoud

De inhoud van het advice bevat de implementatie van de *crosscutting concerns* die gemodulariseerd worden. In het raamwerk wordt opgelaten welk Chameleon element de inhoud vormt. Echter, een concrete weaver zal dit wel moeten vastleggen om correct te kunnen weaven - bij een weaver voor Java, bijvoorbeeld, is het zo dat de inhoud van het advice typisch een *Block* is - een opeenvolging van *Statements*, analoog met de inhoud van een methode. Daarom wordt, zoals de zien is in figuur 4.1 op pagina 18, het advice voor Java geïmplementeerd als een subklasse.

Hoofdstuk 6

Pointcuts

Zoals eerder omschreven is een pointcut niets anders dan een welbepaalde verzameling van joinpoints. Het gedefinieerde advice wordt dan net op die plaatsen ingevoegd. Maar een pointcut kan ook bepaalde context blootstellen die in het advice beschikbaar is. Een pointcut dat elke oproep van een bepaalde methode selecteert kan bijvoorbeeld de argumenten waarmee de methode wordt opgeroepen blootstellen. Een pointcut is dus eigenlijk opgebouwd uit twee elementen: enerzijds de expressie die aanduidt welke joinpoints geselecteerd worden en anderzijds de blootgestelde context - de parameters van het pointcut.

De pointcut expressies worden in sectie 6.1 verder geduid: er wordt een categorisatie voorgesteld, ondersteund door pointcuts aanwezig in AspectJ[1]. In sectie 6.2 is te lezen hoe joinpoints gevonden worden. Tenslotte wordt in sectie 6.3 aangehaald welke operaties op de structuur van pointcut expressies mogelijk moeten zijn en hoe deze uitgevoerd worden.

6.1 Pointcut expressies

Een pointcut expressie definieert welke joinpoints geselecteerd worden. De interface *PointcutExpression* bevat dan ook een methode die, gegeven een *CompilationUnit*, alle joinpoints waar moet gewoven worden zal teruggeven. Er moet echter een onderscheid gemaakt worden tussen drie verschillende soorten van pointcut expressies. Enerzijds zijn er de zogenaamde *statische* pointcut expressies. Dit zijn pointcut expressies waarvan de joinpoints kunnen bepaald worden bij het weaven. Een tweede soort zijn de *runtime* pointcut expressies. Hierbij kunnen de joinpoints niet bepaald worden tijdens het weaven maar is er informatie nodig die pas beschikbaar is tijdens de uitvoer van het programma. Dit impliceert dat een runtime pointcut expressie in de meeste gevallen matcht met alle mogelijke joinpoints en dat het uitvoeren van het bijhorend advice afhangt van de runtime controle. Een derde soort zijn de expressies die context blootstellen. Deze *context exposing* pointcut expressies zorgen ervoor dat de correcte waarden aan de pointcut parameters gebonden worden.

[1]De gegeven lijst van voorbeelden is niet exhaustief. Een volledige lijst ter referentie is te vinden op de AspectJ website, http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html

In de praktijk worden de verschillende soorten pointcut expressies gecombineerd. Dit kan expliciet - door bijvoorbeeld conjunctie - maar eenzelfde pointcut expressie kan ook tot meerdere categorieën behoren. Een voorbeeld hiervan - dat trouwens ook in AspectJ terug te vinden is, zij het met een iets andere syntax - is de volgende pointcut expressie:

```
1 pointcut modifyAccount(Person p): call(void banking.Account.set*()
    ) && thisType(p)
```

Deze pointcut expressie is een conjunctie van twee pointcut expressies: enerzijds een *statische* pointcut expressie, die alle oproepen van methodes waarvan de naam begint met *set* in de klasse *Account* matcht. De tweede pointcut expressie heeft een dubbel doel: enerzijds voegt die een runtime check in die kijkt of het object dat de methode oproep uitvoert, van het type *Person* is. Anderzijds zorgt die er ook voor dat dit object beschikbaar is in advice dat deze pointcut gebruikt en stelt het dus ook context bloot. Aangezien conjuncties, disjuncties en negaties zelf pointcut expressies zijn - zie sectie 6.1.4 op pagina 28 - en deze verschillende types van expressies kunnen combineren, worden deze beschouwd als expressies van elk type. Een laatste apart type is een pointcut expressie die zal verwijzen naar een gedeclareerde pointcut. Dit wordt besproken in sectie 6.1.5 op pagina 28. Deze hiërachie is terug te vinden in het klassediagram in figuur 6.1 op de pagina hiernaast.

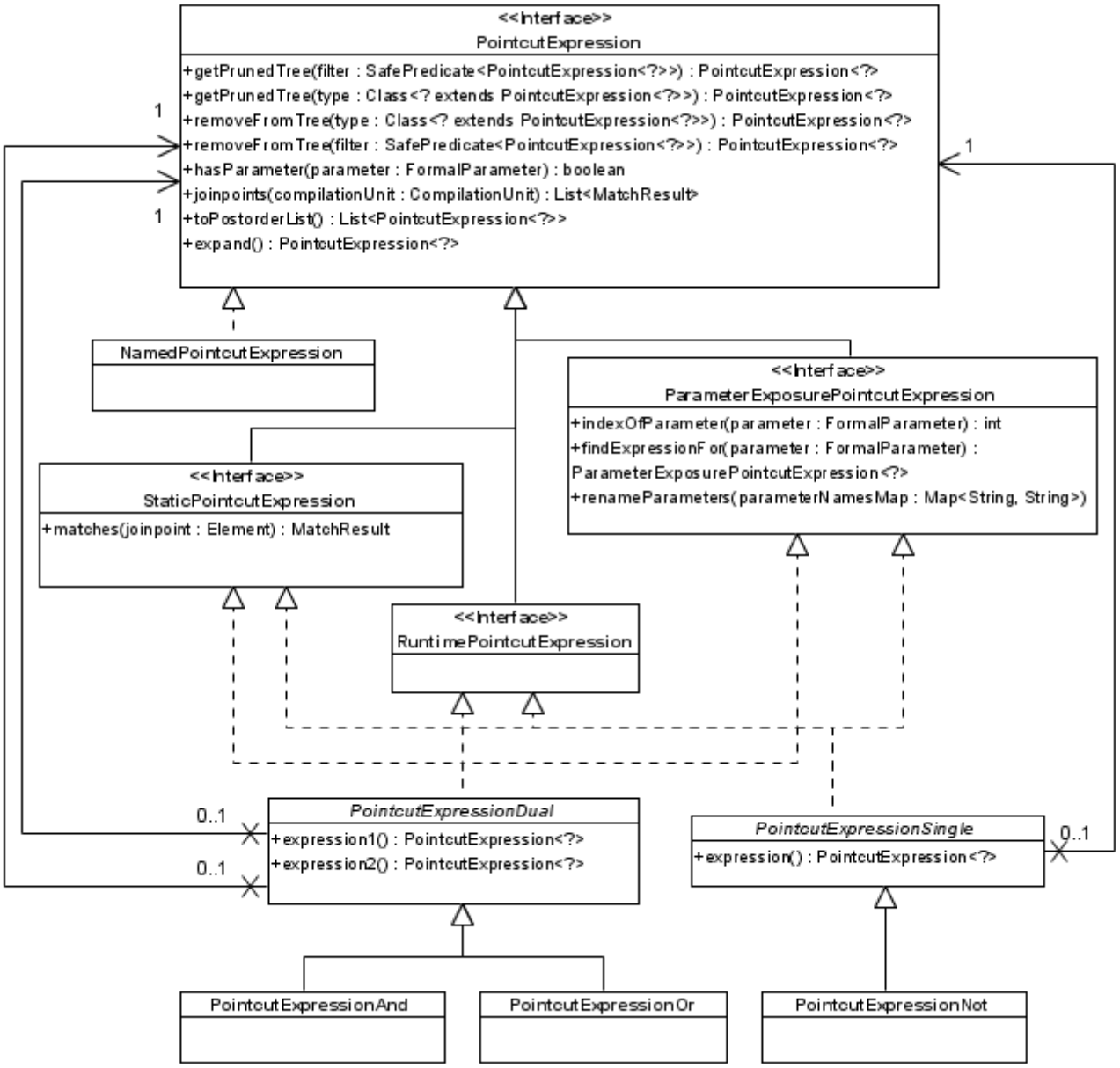
6.1.1 Statische pointcut expressies

Omschrijving

De beslissing of een bepaald joinpoint voldoet aan een statische pointcut expressie kan tijdens het weven bepaald worden, er is geen runtime informatie voor nodig. Om een statische pointcut expressie toe te voegen, volstaat het om de interface *StaticPointcut-Expression* te implementeren. Deze interface definieert, naast de algemene methodes uit de *PointcutExpression* interface, een operatie die iedere statische pointcut expressie moet implementeren - zie figuur 6.1. Deze methode gaat bepalen of het meegegeven element een joinpoint is dat door dit pointcut geselecteerd wordt, of niet.

Een voorbeeld van een statische pointcut expressie die alle methode oproepen met een bepaalde naam zal selecteren ziet er dan als volgt uit:

```
1  public MatchResult matches(Element joinpoint) {
2          if (!(joinpoint instanceof MethodInvocation))
3                  return MatchResult.noMatch();
4
5          MethodInvocation mi = (MethodInvocation) joinpoint;
6
7          // pseudo-code
8          if (mi.signature.name().equals(getNameToMatch())
9                  return MatchResult.match();
10         else
11                 return MatchResult.noMatch();
12 }
```



FIGUUR 6.1: De pointcut hiërarchie

Tot hiertoe werden verschillende statische pointcutexpressies toegevoegd voor de Java implementatie van de weaver. Belangrijk is echter op te merken dat deze pointcutexpressies volledig werken op de abstracties die Chameleon biedt. Bovendien staat de effectieve weaver ook los van de gebruikte expressies. Daardoor is het mogelijk om, voor bepaalde concepten, een compleet taal onafhankelijke expressie te definiëren. Natuurlijk blijft die expressie dan enkel bruikbaar voor talen en modellen waarbinnen dit concept een betekenis heeft.

Voorbeelden uit AspectJ

call(pattern) Alle methode oproepen die voldoen aan het gegeven patroon, bv. **call**(**void** Account.setBalance(**int**));

execution(pattern) Elke uitvoer van een methode die voldoet aan het gegeven patroon, bv. **execution**(String Person.getName());

get(pattern) Lezen van een veld dat voldoet aan het gegeven patroon, bv. **get**(String Person.name);

set(pattern) Schrijven naar een veld dat voldoet aan het gegeven patroon, bv. **set**(String Person.name);

handler(type) Afhandelen van een exceptie van het gegeven type, bv. **handler**(RuntimeException);

within(type) Matcht indien het joinpoint zich bevindt (lexicaal) in het gegeven type, bv. **within**(Loggable);

within(type) Matcht indien het joinpoint zich bevindt (lexicaal) in de gegeven methode, bv. **within**(**void** Account.setBalance(**int**));

6.1.2 Runtime pointcut expressies

Omschrijving

In tegenstelling tot statische pointcut expressies, kan de beslissing of aan een runtime pointcut expressie voldaan is niet genomen worden tijdens het weaven. De methode *matches*, zoals deze te zien was in de *StaticPointcutExpression* interface, is dus ook afwezig aangezien deze geen zin heeft. Merk op dat deze pointcut expressies - in tegenstelling tot de statische - slechts informatie bevatten over *wat* er gecontroleerd moet worden en niet *hoe* dit moet gebeuren. Deze verantwoordelijkheden moeten hier gescheiden worden omdat de manier waarop deze controles gebeuren afhankelijk is van hoe er gewoven wordt. Hoe dit concreet gebeurt wordt later in de tekst besproken (sectie 8.3 op pagina 61).

Voorbeelden uit AspectJ

De meeste pointcut expressies in AspectJ die runtime controles uitvoeren, worden ook gebruikt om context van het joinpoint bloot te stellen. Daarom komen sommige van de voorbeelden hieronder, ook terug als voorbeelden van pointcut expressies die context blootstellen. In deze context wordt enkel het controle facet van de pointcut expressie bekeken.

if (booleaanse expressie) Voert een runtime controle uit waarbij joinpoints enkel matchen als de booleanse expressie waar is. De booleaanse expressie kan gebruik maken van de parameters die blootgesteld werden door de pointcut. Bv. **pointcut** logTransactions(Account from): **call**(**void** Account.transfer(Account, ..)) && **args**(from)&& **if** (Logger.isEnabled())&& **if** (from.owner().isExecutive());

this(type of parameter) Matcht indien het huidige object (hetzelfde als Java's *this*) van het gegeven type is. Kan ook verwijzen naar een parameter in de pointcut definitie of in het advice. Bv. **pointcut** doTransaction(Person p): **call**(**void** Account .transfer(..)) && **this**(p);

target(type of parameter) Matcht indien het doelobject van het gegeven type is. Kan ook verwijzen naar een parameter in de pointcut definitie of in het advice. Bv. **pointcut** doTransaction(SavingsAccount a): **call**(**void** Account.transfer(..)) && **target**(a);

args(type of parameter, ...) Matcht indien het joinpoint argumenten heeft (zoals een methode oproep (*call*) of het uitvoeren van een methode (*execution*), maar ook minder expliciet zoals het afhandelen van een bepaalde exceptie (*handler*)) die van de gegeven types zijn. Er kan ook verwezen worden naar parameters in de pointcut definitie of in het advice. Bv. **pointcut** doTransaction(SavingsAccount from , Account to): **call**(**void** Account.transfer(Account, Account))&& **args**(from, to);

6.1.3 Context exposing pointcut expressies

Omschrijving

De pointcut expressies die ervoor zorgen dat de context van een bepaald joinpoint beschikbaar is in het advice zijn van het type *ParameterExposurePointcutExpression*. Deze interface biedt drie methodes aan, zoals te zien in figuur 6.1 op pagina 25.

De eerste methode geeft terug op welke index de gegeven parameter zich bevindt, indien er meerdere parameters door de expressie blootgesteld worden. De tweede methode laat toe om, in een boom van expressies, op te zoeken welke expressie zorgt voor het blootstellen van een bepaalde parameter. Dit is nodig omdat de manier waarop parameters blootgesteld worden, afhangt van het type pointcut expressie. De laatste methode laat toe om parameters te hernoemen. Deze methode wordt gebruikt bij het ontwikkelen van genaamde pointcut expressies - zie sectie 6.1.5.

Voorbeelden uit AspectJ

Al deze voorbeelden voeren ook een runtime controle uit. Hier wordt enkel het blootstellen van de context beschouwd.

this(parameter) Bindt het huidige object (hetzelfde als Java's *this*) aan de meegegeven parameter. Bv. **pointcut** doTransaction(Person p): **call**(**void** Account.transfer (..)) && **this**(p);

target(type of parameter) Bindt het doelobject aan de meegegeven parameter. Bv. **pointcut** doTransaction(SavingsAccount a): **call**(**void** Account.transfer(..))&& **target**(a);

args(type of parameter, ...) Bindt de argumenten van het joinpoint aan de meegegven parameters. Bv. **pointcut** doTransaction(Account from, Account to): **call**(**void** Account.transfer(Account, Account))&& **args**(from, to);

6.1.4 Pointcut expressies combineren

Om tot een expressief pointcut model te komen moet het natuurlijk ook mogelijk zijn om de verschillende expressies te combineren. Ook binnen het ontwikkeld raamwerk is hier de mogelijkheid voor. Er is zowel ondersteuning voor de conjunctie, disjunctie als negatie van expressies. Bovendien zijn deze opnieuw zelf expressies zodat een willekeurig grote structuur opgebouwd kan worden. Deze structuur vormt een binaire boom, met op de knopen de conjunctie, disjunctie of negatie en op de bladeren de statische en runtime pointcut expressies.

6.1.5 Genaamde pointcut expressies

In bestaande aspect talen, zoals AspectJ, is het ook mogelijk om pointcuts een bepaalde naam te geven en kan er vanuit advice of andere pointcuts naar verwezen worden. Dit maakt de aspecten overzichtelijker en zorgt natuurlijk voor beter onderhoudbare code - aangezien de duplicatie van pointcut definities vermeden wordt. Bovendien speelt een zinvolle naam ook een rol in de herbruikbaarheid van pointcut definities [6].

Naast de mogelijkheid om pointcuts een naam te geven, moet er natuurlijk ook de mogelijkheid zijn om te verwijzen naar dat pointcut vanuit andere expressies. Een dergelijke pointcut expressie is makkelijk te implementeren in Chameleon, omdat het hier gaat om een verwijzing of, in Chameleon termen, een *CrossReference* - net als bijvoorbeeld een methode oproep. Het grootste deel van het werk dat bij deze verwijzingen komt kijken - in het model opzoeken naar welk element er concreet verwezen wordt - is reeds gebeurd binnen het Chameleon raamwerk en kan herbruikt worden. De ondersteuning voor genaamde pointcut expressies kon dus eenvoudig geïmplementeerd worden binnen het raamwerk.

Bij het gebruiken van een genaamde pointcut expressie wordt deze, voor het eigenlijke weaven gebeurt, ontwikkeld. Dit wil zeggen dat de expressie die verwijst naar een bepaald pointcut volledig vervangen wordt door de expressie van deze pointcut. Natuurlijk moet er hierbij op gelet worden dat de namen van eventuele parameters ook worden aangepast. Een voorbeeld:

```
pointcut openAccount(AccountHolder ac): call(void Account.
    open()) && this(ac);
```

```
pointcut doAccountAction(AccountHolder accountHolder):
    openAccount(accountHolder) || ...;

// Na 'ontwikkelen':
pointcut doAccountAction(AccountHolder accountHolder): (call(
    void Account.open()) && this(accountHolder)) || ...;
```

6.2 Het zoeken naar joinpoints

In deze sectie wordt omschreven hoe het zoeken naar joinpoints door de weaver in zijn werk gaat. De weaver zal in alle *CompilationUnits* op zoek gaan naar joinpoints waar advice moet geweven worden. Dit gebeurt door alle advices te overlopen en te kijken welke joinpoints voldoen aan de door het advice gedeclareerde pointcut expressie. Er is al opgemerkt dat het mogelijk is om de verschillende types van expressies impliciet te combineren. Er kan dus een pointcut expressie geschreven worden die zowel een statische als een runtime controle uitvoert. In de praktijk worden dit soort combinaties best vermeden[2], om een duidelijker gescheiden pointcut model op te bouwen. Bovendien kunnen er expliciete combinaties gemaakt worden van verschillende expressies, door bijvoorbeeld conjunctie, zodat er geen expressiviteit verloren gaat. Dit wordt echter niet opgelegd door het raamwerk. De aanwezige interfaces en het standaard gedrag (in de abstracte superklasses) bieden enkel een basis aan die deze regel volgt maar dwingen deze niet af. Het staat de gebruiker van het raamwerk natuurlijk vrij om pointcut expressies te definiëren die zowel statisch als runtime controles uitvoeren. Het raamwerk zal ook deze expressies correct behandelen. Dit standaardgedrag zal echter in de meeste gevallen volstaan - wat aangetoond wordt door de implementatie van een concrete weaver in hoofdstuk 9 - en zorgt voor een snelle ontwikkeling: als een nieuwe statische pointcut wordt aangemaakt moet deze slechts drie methodes implementeren, waarvan twee behoren tot het Chameleon raamwerk (*clone* en *children*) en een specifiek tot het raamwerk voor aspecten (*matches*). Figuur 6.2 op pagina 31 duidt de standaard implementatie.

Zoals gezegd kan een pointcut expressie beschouwd worden als een boomstructuur. De volgende regels zijn het standaardgedrag zoals gedefinieerd in de abstracte superklasses. Ze worden recursief toegepast voor deelbomen:

Voor statische pointcut expressies: Zoals aangegeven wordt er gebruikt gemaakt van de *matches* methode om na te gaan of een bepaald joinpoint voldoet aan de expressie.

Voor runtime pointcut expressies: Alle runtime pointcut expressies matchen sowieso met alle joinpoints.

[2]Een wel veel voorkomende combinatie is om runtime controles te combineren met het blootstellen van context. Hoewel ook in dit geval er een vermenging van verantwoordelijkheden is, is deze beter te verantwoorden: in de meeste gevallen zou het blootstellen van een bepaalde parameter vooraf moeten gaan aan een typecheck om geen fouten te krijgen tijdens de loop van het programma. In AspectJ geldt dit onder andere voor de *args* en *this* pointcut expressies - deze kunnen ofwel een runtime controle doen, ofwel zowel een runtime controle als het blootstellen van bepaalde parameters. Het raamwerk ondersteunt deze pragmatische aanpak ook, wat bewezen wordt door de implementatie van soortgelijke pointcut expressies. Dit wordt verder besproken in hoofdstuk 9.

Voor pointcut expressies die context blootstellen: Volgen dezelfde regel als de runtime pointcut expressies.

Voor conjuncties: Een conjunctief pointcut matcht enkel als beide sub-expressies matchen.

Voor disjuncties: Een disjunctief pointcut matcht als een van beide sub-expressies matchen. Let op, hier mag echter geen gebruik gemaakt worden van short-circuit[3] evaluatie omwille van de runtime expressies. Een eenvoudig voorbeeld dat dit illustreert is het volgende:

```
1  @Deprecated
2  public void someMethod();
3
4  public void someOtherMethod {
5          someMethod();
6  }
7
8  before() : (call(void *.someMethod()) && if(false)) || (
       callAnnotated(Deprecated) && if(true)) {
9          // Do something
10 }
```

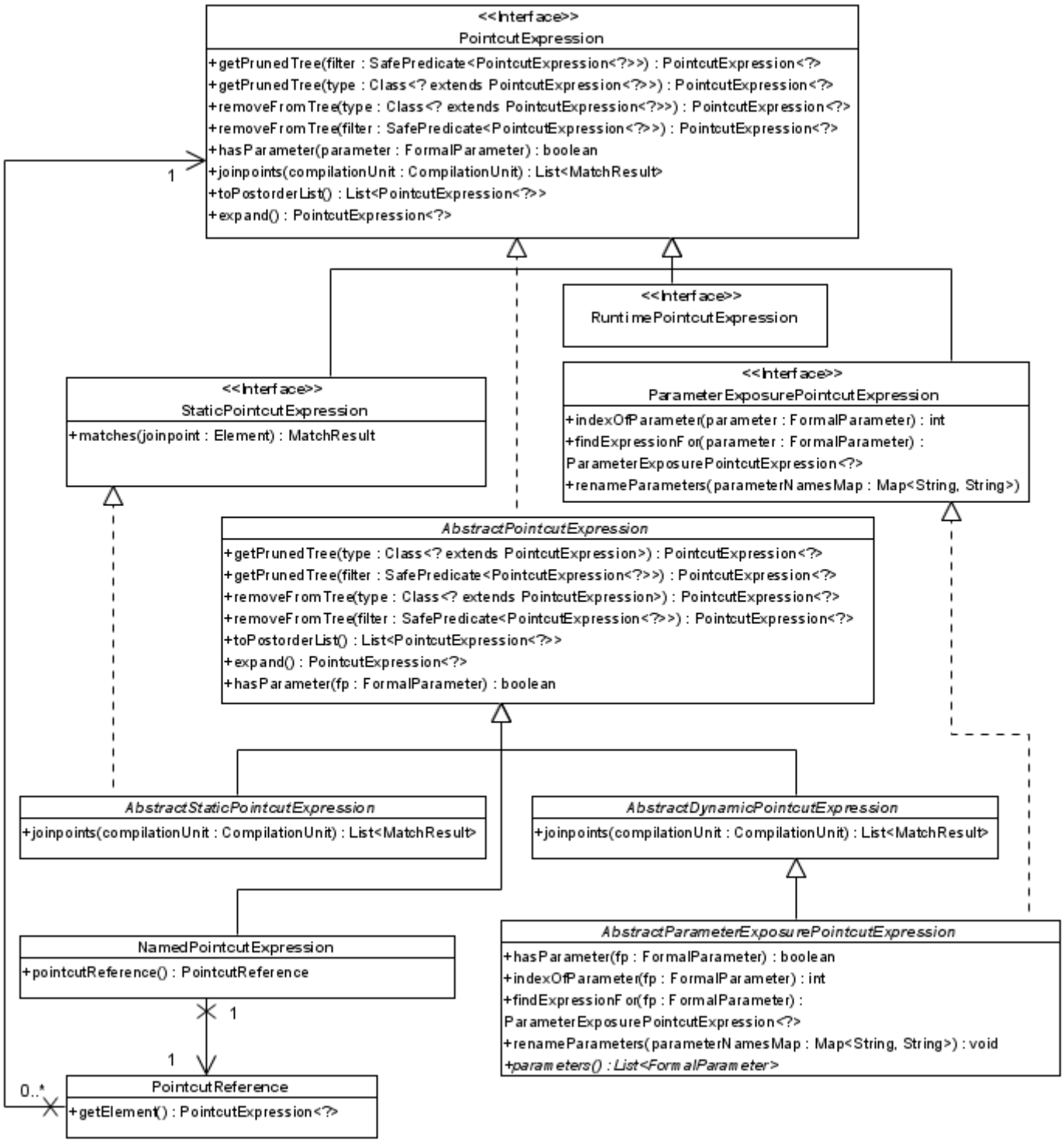
Het eerste deel van deze disjunctie wordt duidelijk gematched op lijn 5. Indien er zou gebruik gemaakt worden van short-circuit evaluatie zou echter enkel het eerste deel van deze disjunctie gebruikt worden door de weaver. Bijgevolg zou enkel de runtime controle **if** (**false**) ingevoegd worden, wat ervoor zorgt dat het advice niet uitgevoerd wordt - een foutief resultaat, want het tweede deel matcht ook op datzelfde punt, en daar slaagt de runtime controle wél.

De juiste strategie is als volgt:

- Indien beide deelbomen matchen, geef dan de disjunctieve pointcut expressie als resultaat. In bovenstaand voorbeeld zou dit er toe leiden dat de runtime controle **if** (**false** || **true**) uitgevoerd wordt, wat het correcte resultaat geeft.
- Indien slechts 1 van beide deelbomen matcht, geef deze deelboom als resultaat.
- Indien geen enkele deelboom matcht, geef aan dat ook deze expressie niet matcht.

Voor negaties: Negaties werken zoals verwacht: het pointcut matcht als het sub-pointcut van de negatie dat niet doet en omgekeerd.

[3]Bij short-circuit evaluatie wordt het tweede argument enkel geëvalueerd indien het eerste niet volstaat om de expressie te bepalen. In Java wordt dit bijvoorbeeld toegepast bij het bepalen van een booleaanse expressie. Indien in de expressie boolean test = method1()|| method2(); het resultaat van method1() **true** is, is de expressie ook **true**. De tweede oproep, method2(), wordt dan niet meer uitgevoerd. Het resultaat beïnvloedt namelijk de expressie niet.



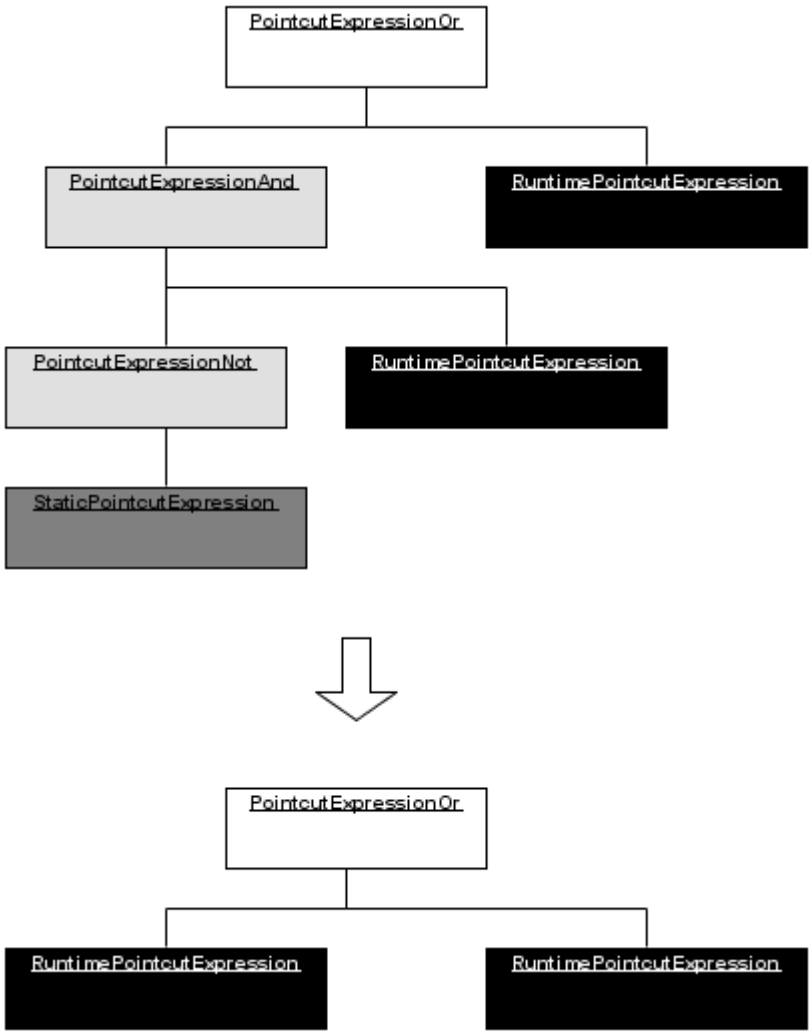
FIGUUR 6.2: De pointcut interfaces en hun standaardgedrag

6.3 Het filteren van expressies

In sommige gevallen moet de boom van expressies gefilterd worden - we willen slechts bepaalde expressies behouden. Een voorbeeld hiervan is dat, om runtime controles in te voegen, we alle runtime expressies uit de boom moeten halen. Om dit echter op een correcte manier te doen moeten we de structuur van de boom behouden, aangezien deze de uiteindelijke controle bepaalt. Concreet wordt de volgende strategie toegepast op de wortel van de boom:

- *And/or* pointcut expressies: filter beide subbomen. Indien deze beide leeg zijn, is deze pointcut expressie ook leeg. Indien slechts een van beide leeg is, geef dan de niet lege deelboom terug. Indien beide niet leeg zijn, geef dan deze pointcut expressie terug met als nieuwe deelbomen de gefilterde oorspronkelijke deelbomen.
- *Not* pointcut expressies: filter de subboom. Indien deze leeg is, is ook deze pointcut expressie leeg. Indien niet, geef dan deze pointcut expressie terug met als nieuwe subboom de gefilterde oorspronkelijke subboom.
- Alle andere expressies: indien de pointcut expressie voldoet aan de filter, geef deze terug.

Dit wordt geïllustreerd in figuur 6.3, waar de boom gefilterd wordt op runtime pointcut expressies. De statische pointcut expressie onderaan de boom (in het donkergrijs) verdwijnt, waardoor de *not* en *and* expressies hoger in de boom (in het lichtgrijs) ook verdwijnen. De rechtse subboom van de *and* expressie en de *or* expressie in de wortel (in het zwart) voldoen echter wel aan de filter, waardoor de *or* expressie in de wortel behouden blijft.

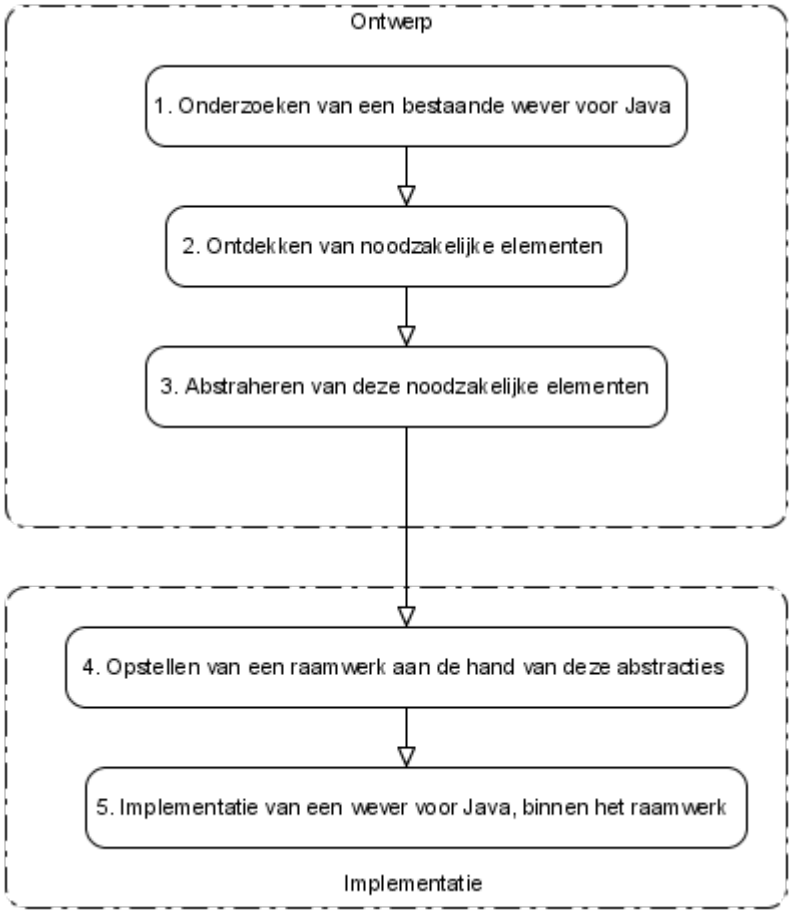


FIGUUR 6.3: Het filteren van de pointcut boom.

Hoofdstuk 7

Het weaving proces

In de vorige hoofdstukken werd beschreven hoe enkele van de elementen van aspect geörienteerd programmeren - aspect, advice en pointcut - worden voorgesteld binnen het raamwerk. In dit hoofdstuk wordt de weaver besproken - het proces dat, gegeven het basismodel en het aspect model, tot een gewoven model komt. Eerst wordt gekeken welke elementen nodig zijn om tot een volledige weaver te komen door enkele voorbeelden te bekijken voor een specifieke taal, in dit geval in Java. Vervolgens wordt in hoofdstuk 8 gekeken welke abstracties nu overeenkomen met deze specifieke elementen en hoe een uitbreidbaar raamwerk opgesteld kan worden aan de hand van deze abstracties. Deze worden elk apart besproken waarbij hun verantwoordelijkheden duidelijk beschreven worden. Er wordt geschetst hoe deze verschillende elementen interageren om tot een resultaat te komen binnen het raamwerk. De evaluatie van het raamwerk, die gebeurt aan de hand van concrete implementaties, volgt in hoofdstuk 9.



Figuur 7.1: Het proces dat gevolgd werd om te komen tot het volledige raamwerk

7.1 Onderzoeken van een bestaande weaver voor Java

In de eerste stap werd een bestaande weaver voor Java onderzocht: AspectJ. Het joinpoint model van AspectJ is relatief eenvoudig, maar laat toch toe om complexe *crosscutting* concerns te beschrijven. Er werden in hoofdstuk 6 reeds voorbeelden gegeven van bestaande pointcuts binnen AspectJ. Het zijn dan ook deze pointcuts, en hun bijhorende joinpoints, die het vertrekpunt vormen van de analyse. Aan de hand van deze pointcuts, en hoe ze gewoven kunnen worden, wordt onderzocht wat noodzakelijk is om tot een functionele weaver te komen.

Deze weaver zal op zich wel verschillen van AspectJ, net omdat het Chameleon raamwerk gebruikt wordt. AspectJ zal namelijk weaven op basis van bytecode, terwijl dit bij deze aanpak niet mogelijk is. De broncode wordt immers omgezet naar een model dat bestaat uit Chameleon abstracties. Dit wordt dan getransformeerd door de weaver en dan opnieuw uitgeschreven naar broncode. De weaver valt dus niet binnen een van de bekende manieren van weaven - het gebeurt namelijk op een hoger niveau

dan de broncode, namelijk het niveau van de abstracties die Chameleon biedt. Het is hierdoor dat het raamwerk de herbruikbaarheid ten goede zal komen. Indien twee programmeertalen syntactisch verschillen maar binnen Chameleon met voornamelijk dezelfde abstracties voorgesteld worden, zal de code voor de weaver ook voor een groot deel herbruikbaar zijn. Echter, het uiteindelijke *resultaat* (de gewoven applicatie) komt sterk overeen met wat een weaver op basis van broncode zou produceren.

7.2 Ontdekken van noodzakelijke elementen

Om te ontdekken welke elementen noodzakelijk zijn voor een weaver, wordt er dieper ingegaan op hoe het weaven zou gebeuren bij de pointcuts uit 6.1. Echter, in plaats van AspectJ te volgen en op bytecode te werken, wordt er gekeken wat er nodig is op het niveau van de broncode om te weaven. Dit valt makkelijker te abstraheren naar wat er nodig is op het niveau van het abstrace model dat door Chameleon aangeboden wordt - wat we uiteindelijk willen bereiken. Eerst wordt een van de meest gebruikte pointcuts bekeken: het aanroepen van een methode (*call*).

7.2.1 Aanroepen van een methode

Een eerste eenvoudig aspect zou als volgt kunnen zijn:

```
1  public Aspect Logging {
2          pointcut sensitiveMethods(): call(boolean Account.
               transfer(..));
3
4          before() : sensitiveMethods() {
5                  Logger.log("Entering␣a␣sensitive␣method");
6          }
7  }
8
9  public class Account {
10         private int balance = 0;
11
12         public Account(int balance) {
13                 this.balance = balance;
14         }
15
16         public static void main(String[] args) {
17                 Account a1 = new Account(100);
18                 Account a2 = new Account(100);
19
20                 a1.transfer(a2, 50);
21         }
22
23         public boolean transfer(Account to, int amount) {
24                 if (amount <= 0)
25                         throw new IllegalArgumentException();
26
```

```
27                  if (balance < amount)
28                          return false;
29
30                  withdraw(amount);
31                  to.deposit(amount);
32          }
33  }
```

Op lijn 20 is er duidelijk een joinpoint dat matcht met het gedefinieerde pointcut. Een eenvoudige implementatie van weaven zou zijn om het advice gewoon voor het joinpoint in te voegen:

```
1  public static void main(String[] args) {
2          Account a1 = new Account(100);
3          Account a2 = new Account(100);
4
5          Logger.log("Entering a sensitive method");
6          a1.transfer(a2, 50);
7  }
```

In dit geval geeft dit een correct resultaat. Maar in andere gevallen zou dit tot een foutief resultaat kunnen leiden. Een voorbeeld:

```
1  public static void main(String[] args) {
2          Account a1 = new Account(100);
3          Account a2 = new Account(100);
4
5          Logger.log("Entering a sensitive method");
6          if (false && a1.transfer(a2, 50)) { // the 'false'
               serves as a trivial example.
7                  // ...
8          }
9  }
```

Hier wordt duidelijk het advice wel uitgevoerd terwijl de methode niet wordt uitgevoerd, opnieuw omwille van het de *short-circuit* evaluatie in Java. Het is wel mogelijk om het weaven zo te laten gebeuren dat een correct resultaat bekomen wordt, maar dan zal de strategie die bij het weaven gevolgd moet worden niet enkel afhangen van het joinpoint en het advice, maar ook hoe en waar het joinpoint voorkomt. Dit willen we absoluut vermijden om de logica bij het weaven zo eenvoudig mogelijk te houden.

Een betere oplossing is dan om de methode oproep integraal te vervangen door een oproep naar een door de weaver gegenereerde methode. Deze methode zal zowel het advice uitvoeren als de originele methode (via reflectie). Ook bij *around* advice, waar het mogelijk is om de methode waar het joinpoint naar verwijst op te roepen - eventueel met andere parameters - vormt dit geen probleem, aangezien dit ook via reflectie kan gebeuren. Een voorbeeld:

```
1  public static void main(String[] args) {
2          Account a1 = new Account(100);
3          Account a2 = new Account(100);
```

```
4
5           Logger.<Boolean>advice_rndname(a1, "transfer", new
                Object[]{a2, 50});
6   }
7
8   public class Logger {
9           public static <T> T advice_rndname(Object src, String
                methodName, Object[] params) {
10                  Logger.log("Entering a sensitive method");
11                  // execute original method through reflection
                        - this is pseudo code!
12                  return src.executeMethod(methodName, params);
13          }
14  }
```

Een nadeel van deze methode is natuurlijk dat ze slechts beschikbaar is indien er reflectie mogelijkheden zijn in de programmeertaal. Ook is de logica iets ingewikkelder dan hier voorgesteld - bijvoorbeeld voor het afhandelen van excepties. Het is echter zeker mogelijk om een dergelijke weaver te implementeren: de uiteindelijke weaver voor Java die ontworpen wordt aan de hand van het raamwerk werkt op deze manier.

Vereiste 1 *Een mechanisme om de advice methode aan te maken, aan de hand van het gegeven advice en het joinpoint waarop het advice toegepast wordt.*

Vereiste 2 *Een mechanisme dat bepaalt welk element met het joinpoint gecombineerd wordt tijdens het weaven. Dit is immers niet altijd het advice zelf zoals dit voorbeeld aantoonde: hier is het een methode oproep die moet aangemaakt worden.*

Vereiste 3 *Een mechanisme om het joinpoint te transformeren: in dit geval het joinpoint, een methode oproep, vervangen door een ander element - tevens een methode oproep.*

7.2.2 Afhandelen van excepties

Een tweede pointcut dat bekeken wordt is het afhandelen van een exceptie. Een eenvoudig voorbeeld dat toont hoe logging kan toegevoegd worden bij het afhandelen van een exceptie is het volgende:

```
1   public Aspect Logging {
2           pointcut illegalOperation(): handler(
                UserNotPriviligedException);
3
4           before() : illegalOperation() {
5                   Logger.log("A user performed an action he
                        wasn't allowed to!");
6           }
7   }
8
9   public class Account {
```

```
10          private int balance = 0;
11
12          public Account(int balance) {
13                  this.balance = balance;
14          }
15
16          public static void main(String[] args) {
17                  Account a1 = new Account(100);
18                  Account a2 = new Account(100);
19
20                  a1.doSafeTransfer(a2, 50);
21          }
22
23          public boolean doSafeTransfer(Account to, int amount)
                {
24                  try {
25                          return a1.transfer(a2, 50);
26                  catch (UserNotPriviligedException e) {
27                          // Failure mode: do nothing
28                          return false;
29                  }
30          }
31
32          public boolean transfer(Account to, int amount)
               throws UserNotPriviligedException {
33                  if (amount <= 0)
34                          throw new IllegalArgumentException();
35
36                  if (!SecurityManager.hasPrivilege(this,
                        Privilege.WITHDRAW))
37                          throw new UserNotPriviligedException
                                ();
38
39                  if (balance < amount)
40                          return false;
41
42                  withdraw(amount);
43                  to.deposit(amount);
44          }
45  }
```

In tegenstelling tot bij methode oproepen, is het hier geen probleem om het advice rechtstreeks in te voegen. Sterker nog, het is niet mogelijk om de inhoud van het catch - block gewoon te vervangen door een methode oproep naar een advice methode, zoals bij *call*-pointcuts wel kan. Het is bijvoorbeeld mogelijk dat in het catch - block gebruik gemaakt wordt van lokale variabelen van de methode. Deze zijn natuurlijk niet beschikbaar binnen een dergelijke advice methode. Het is dan natuurlijk wel opletten dat er geen conflicten optreden tussen de namen van lokale variabelen en variabelen

uit het advice. Eventueel kan de weaver deze conflicten oplossen door binnen het advice te kijken welke variabelen er gedeclareerd worden, en indien deze reeds bestaan binnen dezelfde scope als het joinpoint, deze te hernoemen. Dergelijke opzoekingen en hernoemingen zijn makkelijk mogelijk binnen de abstracties die Chameleon biedt[1].

Merk op dat hier geen nieuwe mechanismen voor vereist zijn. De reeds gestelde vereisten - 2 en 3 op pagina 39 - volstaan om ook deze functionaliteit te ondersteunen. Het is echter duidelijk dat de drie reeds gestelde vereisten afhangen van de manier waarop gewoven moet worden. Het moet dus mogelijk zijn om de elementen die verantwoordelijk zijn voor deze vereisten, eenduidig vast te leggen - afhankelijk van het type joinpoint.

Vereiste 4 *Een mechanisme dat voor een bepaald (type van) joinpoint vastlegt welke elementen verantwoordelijk zijn voor het transformeren van het advice, het bepalen van het element waarmee het joinpoint gecombineerd wordt en de manier waarop dit combineren gebeurt.*

7.2.3 Het uitvoeren van een methode

Een joinpoint dat duidt op het uitvoeren van een methode verschilt weinig van een joinpoint dat duidt op het oproepen van een methode. Het verschil zit in de eventueel blootgestelde context of typecontroles door *this* en *target*. Bij methode oproepen verwijst *this* naar het object van waaruit de oproep gebeurt en *target* naar het object waarop de methode wordt uitgevoerd. Bij methode uitvoer verwijzen beiden naar het object waarin de methode uitgevoerd wordt.

Het uitvoeren van een methode kan ook op een analoge manier behandeld worden als het oproepen van een methode wat betreft het weven. Een mogelijke strategie voor de wever zou als volgt kunnen zijn:

1. Maak een nieuwe methode aan in de klasse van het joinpoint. Deze methode is een exacte kopie van de methode die uitgevoerd wordt, op de naam na om conflicten te vermijden.

2. Maak een advice methode aan, analoog aan wat moest gebeuren bij het oproepen van een methode. Deze advice methode zal de aangemaakte kopie oproepen (met behulp van reflectie) op de plaatsen waar het nodig was om de originele methode uit te voeren, bijvoorbeeld na het uitvoeren van de advice code bij *before* advice.

3. Vervang de inhoud van de originele methode door een oproep van de net aangemaakte advice methode.

Hiervoor hoeven geen nieuwe mechanismes te worden gebruikt.

[1] Een interessante applicatie die ook zou kunnen ontwikkeld worden binnen het Chameleon raamwerk is een refactoring tool. Een dergelijke tool zou deze functionaliteit zeker moeten implementeren. Aangezien het gaat om abstracties die Chameleon biedt is er ook hier zeker ruimte voor herbruik.

7.2.4 Het lezen van of schrijven naar een veld

Het lezen van of schrijven naar een veld kan op dezelfde manier gewoven worden als bij methode oproepen. Voor het lezen van een veld wordt de expressie vervangen door een oproep naar een aspect methode. Deze zal via reflectie de inhoud van het veld inlezen en het advice uitvoeren. Na het uitvoeren van het advice wordt het resultaat gewoon teruggegeven. Het schrijven naar een veld kan analoog.

7.2.5 Runtime controles

De pointcuts die als voorbeelden dienden voor runtime controles - *this, target, args* en *if* - zullen, zoals de naam het zegt, tijdens runtime beslissen of een bepaald advice al dan niet uitgevoerd moet worden. Bovendien kunnen *this, target* en *args* ook gebruikt worden om context bloot te stellen - hierover meer in sectie 7.2.6, hier gaat het enkel over over het uitvoeren van runtime controles. Het weaven moet dus sowieso gebeuren, de runtime checks moeten gebeuren voor het al dan niet uitvoeren van het advice. Maar hierbij hangt de plaats waar deze runtime controles moeten gebeuren natuurlijk af van de manier waarop het weaven gebeurd is. Bij het oproepen van een methode zullen de controles moeten ingevoegd worden in de aangemaakte advice methode. Er kan dan beslist worden om, indien niet voldaan is aan de controles, het advice niet uit te voeren. Bij het afhandelen van excepties daarentegen moeten de controles bij het joinpoint zelf geplaatst worden - er is daar namelijk geen advice methode.

Bovendien hangt de manier waarop de controles moeten gebeuren ook af van de manier waarop er gewoven werd. Neem als voorbeeld een *this* type controle. Bij het afhandelen van excepties kan er gewoon een controle gebeuren op het type van het *this* object. In een advice methode verwijst *this* natuurlijk niet naar het object dat de oproep uitvoerde. Het object dat de oproep uitvoert wordt meegestuurd naar de advice methode en de controle moet daarop gebeuren. Het mechanisme dat bepaalt hoe de controle uitgevoerd wordt moet dus niet enkel bepaald worden door het type van pointcut, maar ook door de manier waarop het weven gebeurt.

Vereiste 5 *Een mechanisme om aan te bepalen hoe een runtime controle moet gebeuren.*

7.2.6 Blootstellen van context

Zoals gezegd kunnen de pointcuts *this, target* en *args* ook gebruikt worden voor het blootstellen van de context van het joinpoint binnen het advice. Ook hier kan weer dezelfde opmerking gemaakt worden als bij de runtime controles: de manier waarop er gewoven wordt is van belang om de juiste context bloot te stellen. In sectie 7.2.5 werd besproken hoe bij een *this* typecheck het object waarvan het type gecontroleerd moet worden afhangt van de gebruikte strategie. Maar dit is ook het object dat moet worden toegewezen aan een parameter! De manier waarop een bepaalde parameter toegewezen wordt hangt dus af van twee zaken: enerzijds door welke pointcut deze blootgesteld wordt, anderzijds de gebruikte strategie bij het weaven.

De keuze om bij pointcuts die zowel een runtime controle doen als een parameter toewijzen, deze verantwoordelijkheden duidelijk te scheiden is bewust. Dit zorgt voor een duidelijkere, meer modulaire structuur van het raamwerk. Er wordt echter niet

op expressiviteit ingeboet en de pragmatische aanpak om deze verschillende types te combineren in een enkel pointcut wordt ondersteund.

Vereiste 6 *Een mechanisme om aan te duiden hoe een parameter moet toegewezen worden.*

7.2.7 Volgorde van runtime toevoegingen

De runtime controles en het toewijzen van de parameters kunnen niet in eender welke volgorde gebeuren. Indien men een bepaalde parameter wil toewijzen, moet de bijhorende controle van het type reeds gebeurd zijn - anders loopt men het risico op een fout tijdens het uitvoeren van het programma waartegen de gebruiker - diegene die het pointcut schreef - bestand moet tegen zijn. Het zou dus logisch zijn om alle runtime controles eerst te plaatsen, gevolgd door de toewijzingen van de parameters. Ook dit is echter geen afdoende oplossing. Een *if* pointcut, bijvoorbeeld, controleert namelijk tijdens runtime een bepaalde booleaanse expressie. Deze expressie kan echter ook een van de parameters bevatten. Om deze controle dus uit te kunnen voeren, moet de parameter natuurlijk eerst toegewezen zijn! We hebben dus een mechanisme nodig dat alle controles en toewijzingen in de correcte volgorde toevoegt. Hierbij is het belangrijk om rekening te houden met de uitbreidbaarheid - het moet altijd mogelijk zijn om nieuw pointcuts te definiëren die op een eenvoudige manier moeten kunnen toegevoegd worden aan de bestaande weavers.

Een tweede punt is dat het invoegen van deze controles en toewijzingen op twee plaatsen kan gebeuren: bij het getransformeerde advice, zoals bij methode oproepen het geval is, of bij het getransformeerde joinpoint zelf, zoals bij het afhandelen van excepties het geval is. Dit moet echter op een uniforme manier kunnen gebeuren.

Vereiste 7 *Een mechanisme om het invoegen van de runtime controles en het toewijzen van de parameters te coördineren. Dit mag immers niet in gelijk welke volgorde gebeuren maar moet op een welbepaalde manier. Deze manier hangt tevens ook af van de manier van weven, waarbij er moet rekening mee gehouden worden dat bepaalde pointcuts niet ondersteund worden door sommige weavers - een voorbeeld hiervan is het* args *pointcut bij het lezen van een veld (*get*) - en dat in de toekomst nieuwe pointcuts makkelijk toegevoegd moeten kunnen worden.*

Vereiste 8 *Het doorvoeren van runtime transformaties moet via een uniforme interface gebeuren.*

7.2.8 Het zoeken naar joinpoints

Voor elk advice dat gewoven wordt, moeten alle joinpoints waar dit moet gebeuren gezocht worden. Maar niet alleen het joinpoint moet bijgehouden worden - ook de pointcut expressie die het joinpoint selecteerde is van belang. Deze kan immers nog pointcut expressies bevatten die aanleiding geven tot runtime controles of het toewijzen van parameters.

Vereiste 9 *Bij het zoeken naar joinpoints voor een advice moeten zowel het joinpoint als de pointcut expressie die dit joinpoint selecteerde, bijgehouden worden opdat het weaven correct zou kunnen gebeuren.*

7.2.9 Meerdere advices voor eenzelfde joinpoint

Het is mogelijk dat verschillende pointcuts eenzelfde joinpoint selecteren. Dat wil zeggen dat meerdere advices op dezelfde plaats moeten gewoven worden. Dit impliceert dat het weaven niet in een enkele fase kan gebeuren. Immers, bij bijvoorbeeld een methode oproep wordt het joinpoint vervangen. Indien de weaver dan verder gaat met het afhandelen van de andere advices, zal geen enkele pointcut meer matchen met het originele joinpoint aangezien dit niet meer bestaat. Dit leidt ertoe dat enkel het eerste advice effectief afgehandeld wordt. Een oplossing hiervoor is om van het weaven een proces in twee fasen te maken. In de eerste fase wordt gekeken welke advices op welke joinpoints toegepast moeten worden. Vervolgens, in fase twee, gebeurt het effectieve weaven.

Indien meerdere advices van toepassing zijn op eenzelfde joinpoint, moeten deze, om een consistent resultaat te bieden, in een vast bepaalde volgorde behandeld worden. AspectJ laat toe om aspecten te rangschikken op prioriteit via een *declare precedence* statement[8]. Advices die behoren tot aspecten met een hogere prioriteit worden eerst gewoven. Indien twee advices behoren tot hetzelfde aspect, krijgt het eerst geplaatste advice voorrang. Omwille van de beperkte scope wordt hier gekozen voor een eenvoudig prioriteitsmodel: Around advice krijgt voorrang op before advice, wat op zijn beurt voorrang krijgt op after advice. Indien twee advices van hetzelfde type zijn is de volgorde onbepaald. Verder gebeurt het weaven als analoog aan AspectJ indien er meerdere advices van toepassing zijn op eenzelfde joinpoint[2]:

Around advice voert bij een *proceed* call het volgende advice of, indien er geen volgende advice is, het joinpoint uit.

Before advice voert eerst het advice uit. Indien dit niet voortijdig beëindigd wordt - bijvoorbeeld door het gooien van een exceptie - wordt het volgende advice uitgevoerd, of indien er geen volgende advice is, het joinpoint.

After returning advice voert eerst het volgende advice of, indien er geen volgende advice is, het joinpoint uit. Als deze code niet voortijdig beëindigd wordt, zal het advice worden uitgevoerd.

After throwing advice voert eerst het volgende advice of, indien er geen volgende advice is, het joinpoint uit. Als deze code wél voortijdig beëindigd wordt door het gooien van een exceptie, zal het advice worden uitgevoerd.

After voert eerst het volgende advice of, indien er geen volgende advice is, het joinpoint uit. Vervolgens wordt het advice uitgevoerd.

Vereiste 10 *Het weaven moet niet meteen gebeuren maar in twee fases. In eerste fase wordt gekeken op welke plaatsen er advice moet worden ingevoegd. Alle nodige informatie om te kunnen weaven wordt hier verzameld en in een enkele entiteit geplaatst. Deze entiteit is dan in de tweede fase verantwoordelijk voor het starten van het weef-proces.*

[2] http://www.eclipse.org/aspectj/doc/released/progguide/semantics-advice.html

Vereiste 11 *Vooraleer het weaven van start gaat, worden de entiteiten per joinpoint geordend. Er moet dus een mechanisme zijn om deze entiteiten te ordenen volgens de beschreven regels.*

7.2.10 Het coördinerend proces

Alle bovenstaande elementen moeten natuurlijk op het juiste moment en in de juiste volgorde gebeuren. Er moet dus een element zijn dat optreedt als coördinerend proces en dat de individuele deelprocessen - het zoeken naar joinpoints, het sorteren van de weavers, het weaven zelf - beheert.

Vereiste 12 *Een coördinerend proces dat de individuele deelprocessen zal verbinden.*

7.2.11 Conclusie

Door een grondige analyse te maken van de mogelijkheden van een bestaande aspect taal, AspectJ, en de manier waarop een relatief eenvoudige maar desalniettemin krachtige statische weaver op basis van broncode zou kunnen geïmplementeerd worden zijn er - naast de eerder beschreven elementen voor aspects, advices en pointcuts voor te stellen - twaalf belangrijke vereisten geïdentificeerd om de weaver op een modulaire en herbruikbare manier op te bouwen. In het volgend deel wordt geschetst welke abstracties aanwezig moeten zijn om aan deze vereisten te voldoen, en worden de functionaliteit en verantwoordelijkheden precies afgelijnd en gespecificeerd.

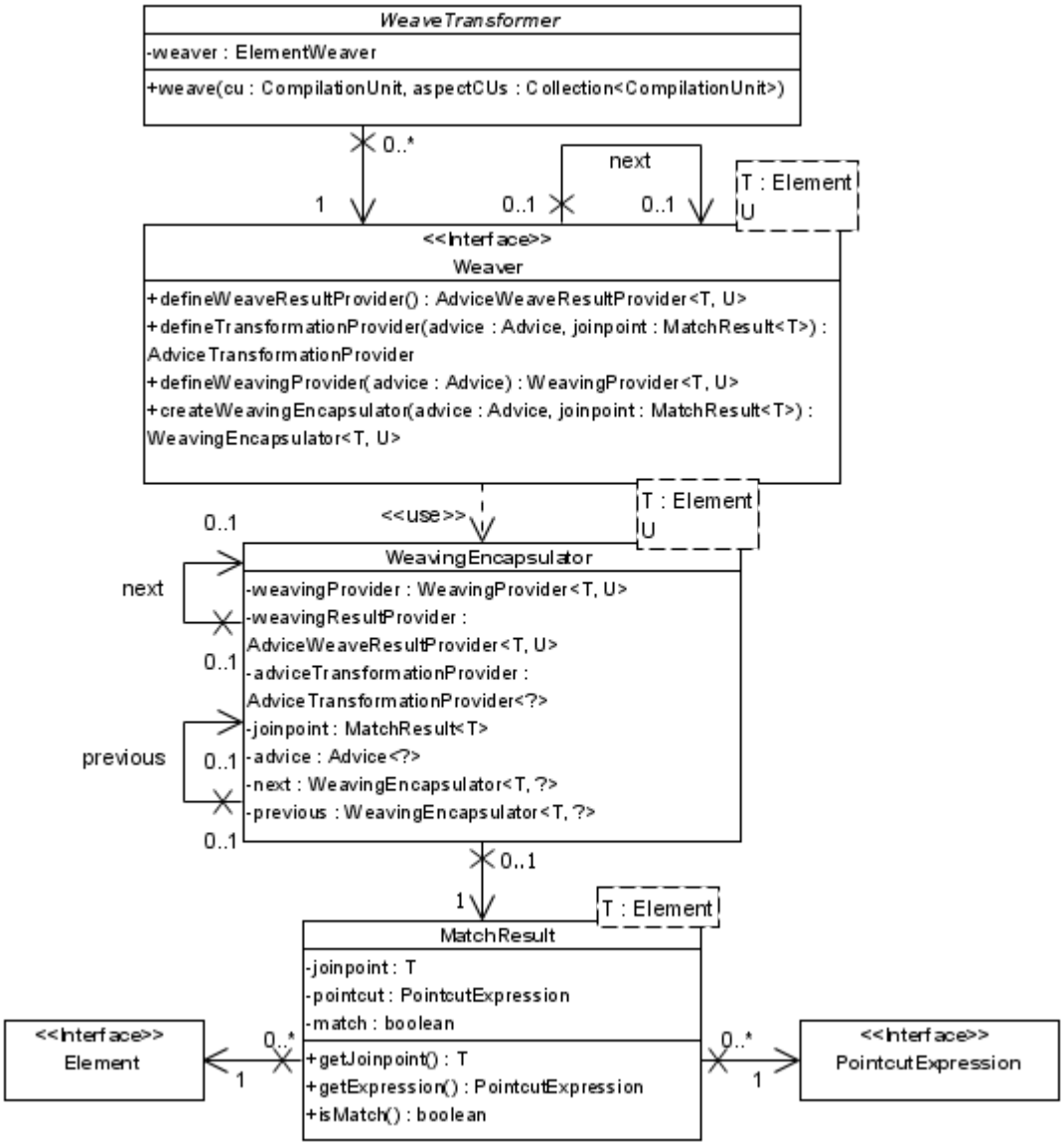
Hoofdstuk 8

Opstellen van het raamwerk

In het vorig deel werden de nodige elementen geïdentificeerd om een krachtige, uitbreidbare weaver te implementeren. De volgende stap is om deze elementen te abstraheren en zoveel mogelijk los proberen te maken van taal-specifieke zaken. Er wordt geprobeerd om alles uit te drukken op het niveau van het de abstracties die Chameleon biedt, wat toelaat om een abstract raamwerk op te stellen. Binnen dit raamwerk kunnen dan de taal specifieke zaken ontwikkeld worden om tot een concrete weaver te komen. Deze stap is opgedeeld in drie subsecties: het zoeken naar het joinpoint en het starten van het weaving proces (sectie 8.1), het transformeren van het joinpoint (sectie 8.2) en de runtime controles en het toewijzen van parameters (sectie 8.3). Bij elke sectie wordt indien nodig een klassediagram gegeven dat de relevante elementen situeert. Hier worden enkel de belangrijkste methodes getoond. Bij de bespreking van elk afzonderlijk element worden alle methodes getoond.

8.1 Het zoeken naar joinpoints en starten van het weaving proces

Figuur 8.1 toont hoe een *WeaveTransformer* het weefproces voor een bepaalde *compilation unit* zal starten. Deze zal alle joinpoints opzoeken waar gewoven moet worden. Deze worden, samen met het pointcut dat dit joinpoint selecteerde, bijgehouden in een *MatchResult*. Om de concrete *WeavingEncapsulator*s te bekomen - die alle nodige informatie om een bepaald joinpoint te kunnen weaven bevatten - wordt een *chain of responsibility* van *Weaver*s gebruikt. Het zijn deze *Weaver*s die de concrete strategie voor een type joinpoint - bijvoorbeeld methode oproepen - vastleggen. Deze processen worden verder in deze sectie uitgebreid besproken.



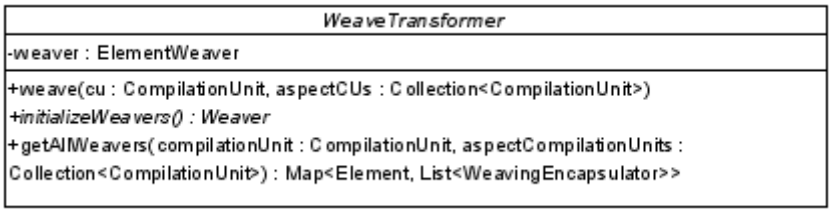
Figuur 8.1: De *WeaveTransformer* start het weaving proces door voor elk joinpoint dat gewoven moet worden, de *WeavingEncapsulator* te creëren.

8.1.1 Het coördineren van het weaving proces

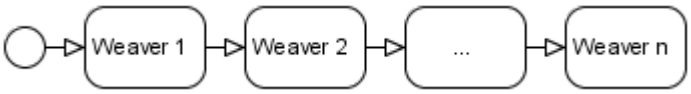
Rationale (vereiste 12 op pagina 45) De twaalf vermelde vereisten zullen door verschillende elementen behandeld worden. Er is natuurlijk een element nodig dat het hele proces zal starten en de taken zal delegeren. Dit is de *WeaveTransformer*. Deze bevat een lijst van individuele *Weaver*s - zie figuur 8.1 - die elk vastleggen welke combinaties van joinpoint en advice ze kunnen weaven. Aan de hand hiervan zal

de *WeaveTransformer* de te weaven joinpoints opzoeken en de bijhorende *Weaver*s selecteren.

Implementatie De *WeaveTransformer* zal een gegeven *CompilationUnit* weaven. Om dit te doen worden alle joinpoints gezocht waar gewoven moet worden, samen met alle informatie die hiervoor nodig is (onder de vorm van *WeavingEncapsulator*s). De *WeaveTransformer* klasse beschikt over drie belangrijke methodes: het starten van het weaven, het opzoeken van alle joinpoints en bijhorende informatie om te weaven en de methode om de afzonderlijke *Weaver*s - die bepalen hoe elke combinatie van joinpoint en advice moet gewoven worden - te initialiseren. Deze vormen een *Chain of Responsibility [1]* - voor elke combinatie van joinpoint en advice dat moet gewoven worden, worden de *Weaver*s in de keten afgelopen om de geschikte te vinden.



FIGUUR 8.2: De *WeaveTransformer*

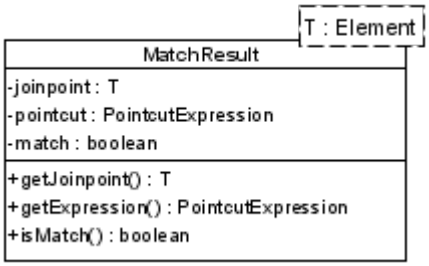


FIGUUR 8.3: De *Chain of Responsibility*

8.1.2 Het zoeken naar joinpoints

Rationale (vereiste 9 op pagina 43) Hoe het zoeken naar joinpoints voor een bepaalde pointcut in zijn werk gaat, werd al uitgebreid beschreven in sectie 6.2 op pagina 29. Als een joinpoint gevonden is dat geselecteerd wordt door een pointcut, wordt zowel het joinpoint als het pointcut in een object opgeslagen - het *MatchResult*. Het joinpoint wordt om evidente redenen bijgehouden: dit moet later door de weaver getransformeerd worden. Het pointcut moet ook echter bijgehouden worden om de juiste runtime controles en het toewijzen van de parameters toe te laten.

Implementatie De wever zal dus, in de eerste stap, per advice alle joinpoints zoeken die gematcht worden door de pointcut van dat advice. Het resultaat is een lijst van *MatchResult* objecten, die zowel het joinpoint bevatten als de specifieke pointcut waarmee gematcht is. Dit is niet noodzakelijk de gehele pointcut van het advice, maar kan ook een deelboom ervan zijn - zie sectie 6.2 op pagina 29!



Figuur 8.4: De *MatchResult* klasse

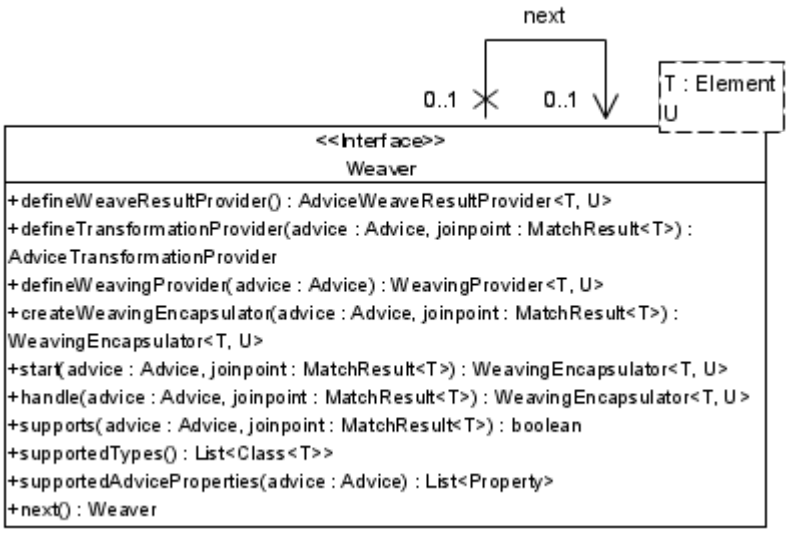
8.1.3 De weaver

Hoewel het volledige proces - vanaf het zoeken naar de joinpoints tot het effectief weaven van de modellen - kan worden geclassificeerd als 'weaven', wordt de weaver in deze context als een kleiner deel van dit geheel beschouwt. Het is het proces dat, voor elk joinpoint dat gewoven moet worden, de volgende zaken zal bepalen:

1. De *WeaveResultProvider*, die vastlegt naar wat het joinpoint moet getransformeerd worden
2. De *AdviceTransformationProvider*, die verantwoordelijk is voor het transformeren van het advice
3. De *WeavingProvider*, die verantwoordelijk is voor het tranformeren van het joinpoint (aan de hand van het resultaat van de *WeaveResultProvider*)

Deze drie elementen worden afzonderlijk besproken in sectie 8.2.

Rationale (vereiste 4 op pagina 41) Het is op deze drie punten dat de weavers voor verschillende joinpoints kunnen afwijken. Het moet dus mogelijk zijn om meerdere van dit soort weavers te definiëren, die elk een bepaalde combinatie van deze drie elementen vastleggen. Vervolgens moet er bij elk joinpoint waar advice moet gewoven worden, de correcte weaver gekozen worden. Deze keuze hangt af van zowel het joinpoint als het advice. Elke weaver beschikt over een *supports* methode die, gegeven het advice en het joinpoint, aanduidt of die combinatie door deze weaver afgehandeld kan worden. Figuur 8.5 illustreert de verantwoordelijkheden van de *Weaver*: het vastleggen van de drie *providers*, het aanmaken van de *WeavingEncapsulator* en de *chain of responsibility*.



FIGUUR 8.5: De *Weaver* interface.

Implementatie Zoals gezegd worden alle afzonderlijke weavers in een *chain of responsibility* geplaatst: indien een weaver een bepaalde combinatie van joinpoint en advice niet kan afhandelen, wordt deze doorgegeven naar de volgende weaver in de keten. Van zodra een enkele weaver de combinatie wel kan afhandelen, stopt de keten daar. Indien dit het geval is, wordt de methode *createWeavingEncapsulator* opgeroepen. Deze methode zal een object van het type *WeavingEncapsulator* aanmaken. Dit object bevat alle nodig informatie om het weaven toe te passen, met een enkele uitzondering: indien er verschillende advices op eenzelfde joinpoint toegepast moeten worden, is op dit moment in het proces nog niet bekend wat het vorige en volgende advice is.

8.1.4 De WeavingEncapsulator

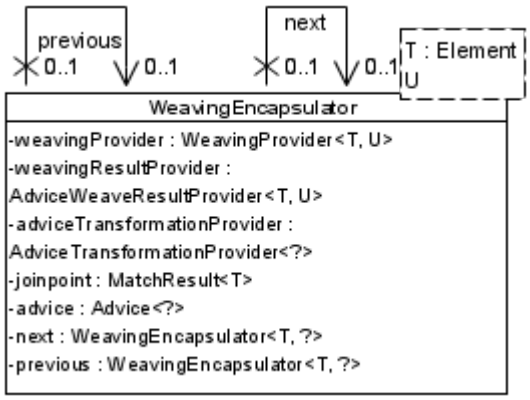
Rationale (vereisten 10 op pagina 44 en 11 op pagina 45) De net besproken *Weaver* legt vast op welke manier het weaven zal gebeuren. Deze mag echter zelf het weaven nog niet starten. Indien er immers meerdere advices van toepassing zijn, moeten deze eerst geordend worden volgens de prioriteitsregels beschreven in sectie 7.2.9 op pagina 44. Bovendien moet elke weaver weten of er een ander advice aan vooraf ging, of op volgt, om tot een correct resultaat te komen. Immers, indien er nog een advice volgt moet dit volgend advice uitgevoerd worden in plaats van de originele code van het joinpoint. Daarom zal de weaver het weefproces niet zelf starten, maar alle nodige informatie om te kunnen weaven in een object encapsuleren - de *WeavingEncapsulator.* Figuren 8.6 op de pagina hierna en 8.9 op pagina 56 illustreren dat dit object over alle nodige informatie beschikt om een bepaald joinpoint en advice te weaven.

Om correct te weaven moet het proces dus in twee fasen verlopen. Er wordt in de eerste fase een lijst opgesteld die, per joinpoint, alle *WeavingEncapsulators* bevat. Daarna worden deze gesorteerd, waarbij de criteria uit sectie 7.2.9 gevolgd worden

om de volgorde te bepalen. Vervolgens gebeurt het effectieve weaven: in de correcte volgorde en waarbij elke weaver op de hoogte is van welke weavers er aan voorafgaan en er op zullen volgen.

Implementatie Aangezien er, voor elke *WeavingEncapsulator*, moet bekend zijn welke andere *WeavingEncapsulator*s er aan voorafgaan en er op volgen, is een dubbel gelinkte lijst de perfecte datastructuur om deze bij te houden.[1]

Zoals vermeld moeten alle *WeavingEncapsulator*s voor een bepaalde joinpoint gesorteerd worden vooraleer het effectieve weaven kan starten. Dit sorteren gebeurt door een *Comparator*. Op deze manier kunnen de reeds aanwezige sorteermethodes in Java[2] gebruikt worden om te sorteren. Bovendien maakt dit het ook eenvoudig om, indien dit later nodig zou zijn, de manier waarop gesorteerd wordt uit te breiden of te vervangen door bijvoorbeeld een expliciete declaratie van de prioriteit, zoals in AspectJ het geval is. Het enige wat dan vervangen of aangepast moet worden is deze *Comparator*.



Figuur 8.6: De *WeavingEncapsulator*. Deze bevat alle elementen die nodig zijn om te weaven en vormt een dubbel gelinkte lijst met alle *WeavingEncapsulators* die op hetzelfde joinpoint toegepast worden.

8.1.5 Interacties

Het weaven begint bij de *WeavingTransformer*. Deze registreert de keten van *Weavers* die gebruikt kunnen worden. Vervolgens wordt, per compilation unit, het volgende gedaan *voor elk advice* (tussen haakjes staat het nummer van de stap in het sequentie diagramma - figuur 8.7):

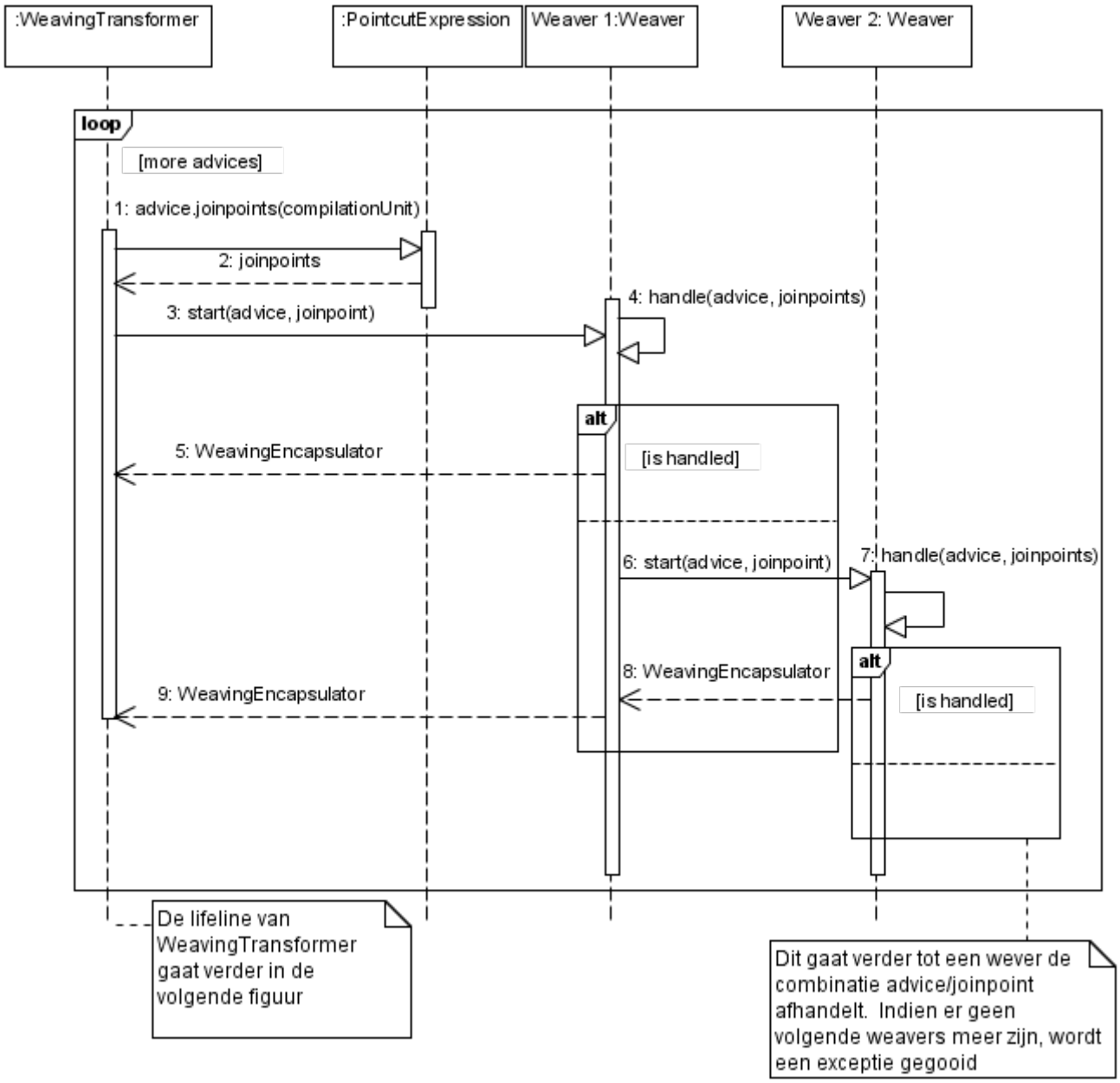
1. Zoek alle joinpoints waar het advice van op toepassing is in het model (1,2).

[1] Omdat voor het praktisch gebruik enkel de *previous* en *next* pointers van elk element nodig zijn, is de klasse *WeavingEncapsulator* geen implementatie van java.util.List. Er is simpelweg een methode om, gegeven een itereerbare collectie van *WeavingEncapsulator*s, deze om te zetten naar een dubbel gelinkte lijst.

[2] Collections.sort, met $O(n \log(n))$complexiteit volgens http://download.oracle.com/javase/6/docs/api/java/util/Collections.html#sort(java.util.List, java.util.Comparator).

2. Voor elk gevonden joinpoint wordt gekeken welke weaver de combinatie van joinpoint en advice ondersteunt. Hierbij wordt gebruik gemaakt van de *chain of responsibility* (3, 4, 6, 7).

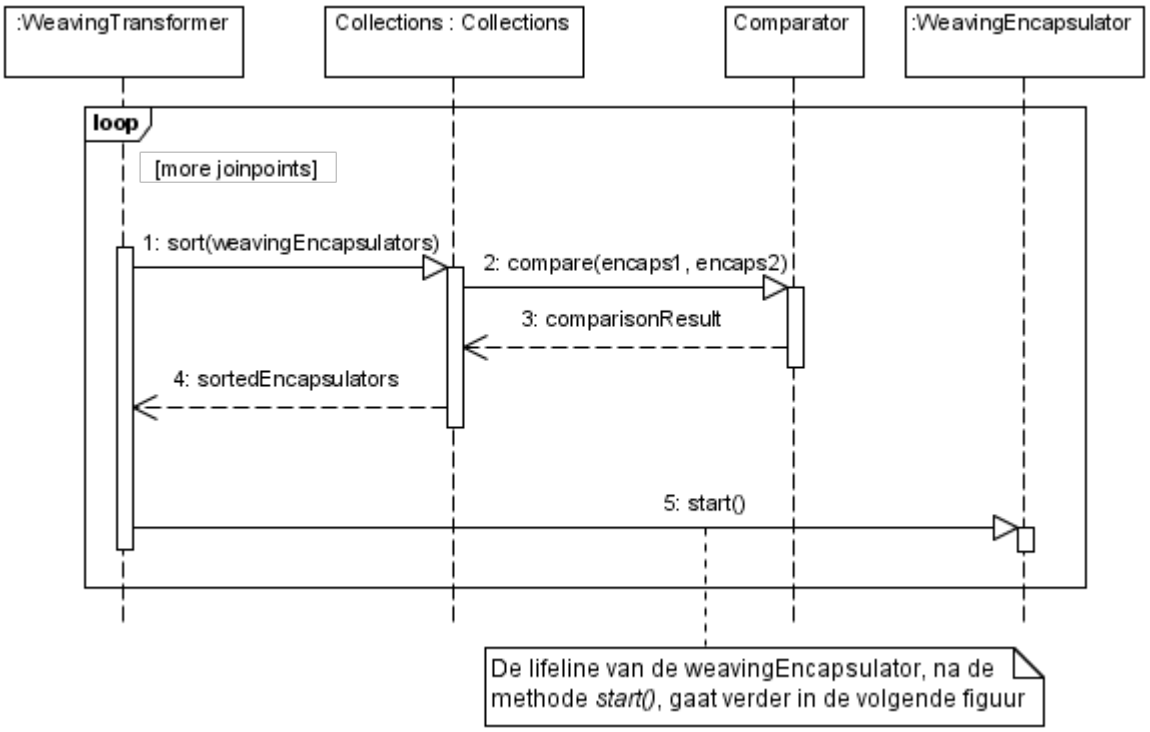
3. Deze weaver geeft de *WeavingEncapsulator* terug die op dat joinpoint zal weaven. Aangezien het mogelijk is dat meerdere advices rond eenzelfde joinpoint weaven, wordt per joinpoint een lijst bijgehouden van alle *WeavingEncapsulator*s die op dat joinpoint van toepassing zijn (5, 8, 9).



Figuur 8.7: Sequentie diagramma voor het vinden van de joinpoints en bijhorende weavers

Na dat dit gebeurd is voor elk advice, gebeurt het volgende, zoals te zien in figuur 8.8:

1. De bijgehouden lijst van *WeavingEncapsulator*s wordt per joinpoint geordend (1, 2, 3, 4).
2. Het effectieve weaven wordt gedelegeerd naar de *WeavingEncapsulator*s. Deze stap wordt later in de tekst nog verder verfijnd (5).

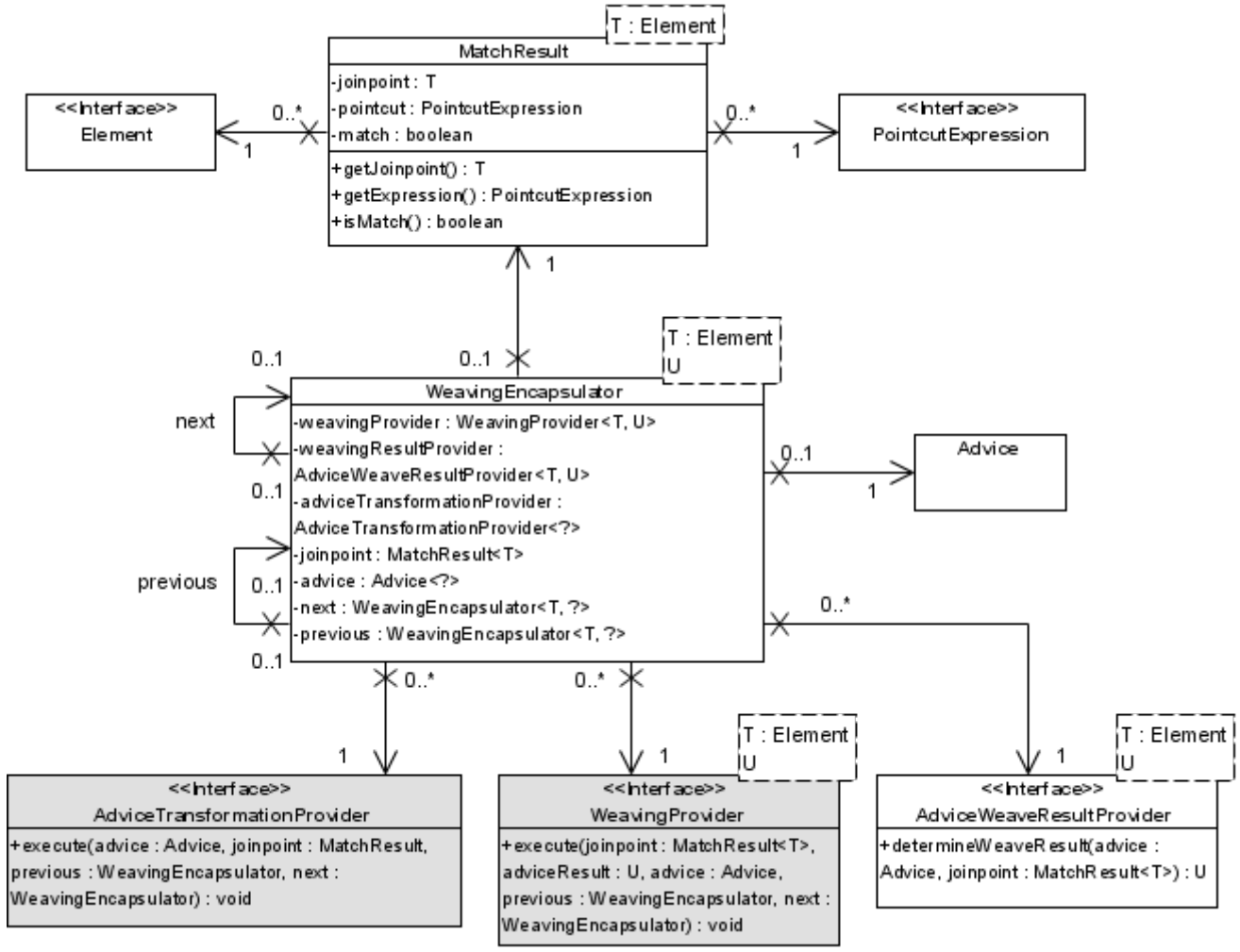


FIGUUR 8.8: Het sequentie diagramma voor het sorteren en het starten van het weaven

8.2 Joinpoint transformatie

Het transformeren van het joinpoint is de meest vanzelfsprekende vereiste. Een advice wordt toegepast op een bepaald joinpoint, dus moet de weaver in staat zijn om joinpoints te transformeren. Dit proces valt uiteen in twee delen, want het tranformeren van het joinpoint komt eigenlijk neer op een combinatie maken van twee elementen: enerzijds het joinpoint zelf, anderzijds een ander element. Eerst moet dus bepaald worden met welk element het joinpoint zal gecombineerd worden. Hiervoor is de *AdviceWeaveResultProvider* verantwoordelijk. Vervolgens moet beslist worden hoe joinpoint en advice gecombineerd zullen worden - hiervoor is de *WeavingProvider* verantwoordelijk. Bovendien is het in sommige gevallen ook nodig om het advice

te transformeren. Dit kan gebeuren door een *AdviceTransformationProvider*. Deze elementen bepalen dus *hoe* het weaven zal gebeuren en worden daarom door de *Weaver* bepaald en door de *WeavingEncapsulator* bijgehouden - zie figuren 8.1 op pagina 48 en 8.9. Het is duidelijk dat de *WeavingEncapsulator* over alle elementen beschikt die nodig zijn om te kunnen weaven: *waar* dit moet gebeuren (het *MatchResult*), met welk *Advice* als hoe het moet (de drie *Provider*s). De *Provider*s worden verder in deze sectie besproken.



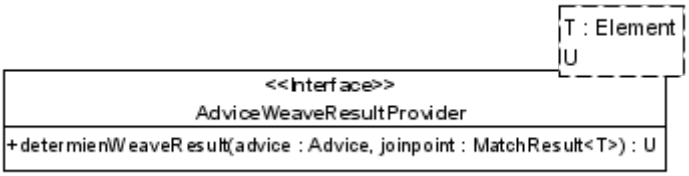
Figuur 8.9: De *WeavingEncapsulator* beschikt over alle nodige elementen om het effectieve weaven correct uit te voeren.

8.2.1 De AdviceWeaveResultProvider

Rationale (vereiste 2 op pagina 39) Zoals gezegd is de *AdviceWeaveResultProvider* verantwoordelijk voor het bepalen van het element dat met het joinpoint

gecombineerd wordt. Dit staat immers niet vast maar hangt af van de manier waarop er gewoven wordt. Dit element kan namelijk het advice zijn, zoals bijvoorbeeld het geval was bij de weaver die het afhandelen van excepties kon transformeren in sectie 7.2.2 op pagina 39. Echter, bij het aanroepen van een methode is het zo dat het joinpoint vervangen wordt door een andere methode oproep, namelijk naar de advice methode - zie sectie 7.2.1 op pagina 37. De weavers voor deze twee verschillende gevallen, zullen dus een verschillende *AdviceWeaveResultProvider* vastleggen.

Implementatie De interface van *AdviceWeaveResultProvider* is eenvoudig en bevat een enkele methode, zoals te zien is in figuur 8.10. De *determineWeaveResult* methode zal het element teruggeven dat gecombineerd wordt met het joinpoint.

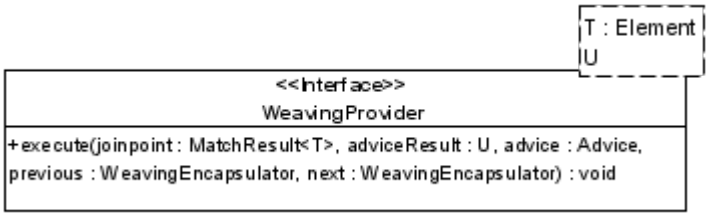


FIGUUR 8.10: De *AdviceWeaveResultProvider* interface

8.2.2 De WeavingProvider

Rationale (vereiste 3 op pagina 39) Het is de *WeavingProvider* die beslist hoe het element bepaald door de *AdviceWeaveResultProvider* gecombineerd wordt met het joinpoint. Dit is immers ook niet vast voor elke weaver. Bij een methode oproep moet het joinpoint integraal vervangen worden door een andere methode oproep. Bij het afhandelen van een exceptie moet het advice voor, na, of rond het joinpoint geplaatst worden - afhankelijk van het type advice. Het is dus de verantwoordelijkheid van de *WeavingProvider* om het *eigenlijke* weaven, het omvormen van het joinpoint, door te voeren.

Implementatie Dit eigenlijke weaven gebeurt in de *execute* methode - zie figuur 8.11. Deze methode neemt natuurlijk het joinpoint en het resultaat van de *determine-WeaveResult* oproep van de *AdviceWeaveResultProvider* als parameters. Het advice zelf wordt ook meegegeven omdat, indien er na het weaven nog transformaties moeten gebeuren om runtime controles uit te voeren of om context bloot te stellen, dit nodig is. Deze transformaties worden uitvoerig besproken in sectie 8.3 op pagina 61. Ook het volgende en vorige te weaven advice op ditzelfde joinpoint worden meegegeven. Deze zijn ook van belang om correct te weaven - bijvoorbeeld voor methode oproepen, mag het joinpoint enkel vervangen worden door een oproep naar de *eerste* advice methode, anders wordt een deel van het advice niet uitgevoerd.

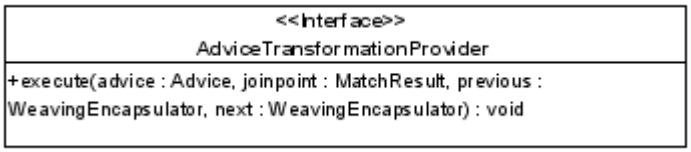


Figuur 8.11: De *WeavingProvider* interface

8.2.3 De AdviceTransformationProvider

Rationale (vereiste 1 op pagina 39) De *AdviceTransformationProvider* beslist op welke manier het model verder moet getransformeerd worden, naast het weaven zelf. Zoals aangehaald is dit nodig bij bijvoorbeeld methode oproepen, aangezien daar een extra advice methode aangemaakt moet worden. In bepaalde gevallen moet er geen transformatie gebeuren, zoals het weaven van het afhandelen van excepties. Het al dan niet transformeren van het advice en de manier waarop moet dus ook door de *Weaver* vastgelegd worden.

Implementatie De transformatie wordt uitgevoerd door door de *execute* methode - zie figuur 8.12. Deze methode neemt als argumenten natuurlijk het advice, en verder het joinpoint - aangezien bijvoorbeeld bij methode oproepen er een advice methode *per joinpoint* moet gemaakt worden om de excepties te kunnen propageren - en het vorige en volgende te weaven advice op hetzelfde joinpoint.



Figuur 8.12: De *AdviceTransformationProvider* interface

8.2.4 Interacties

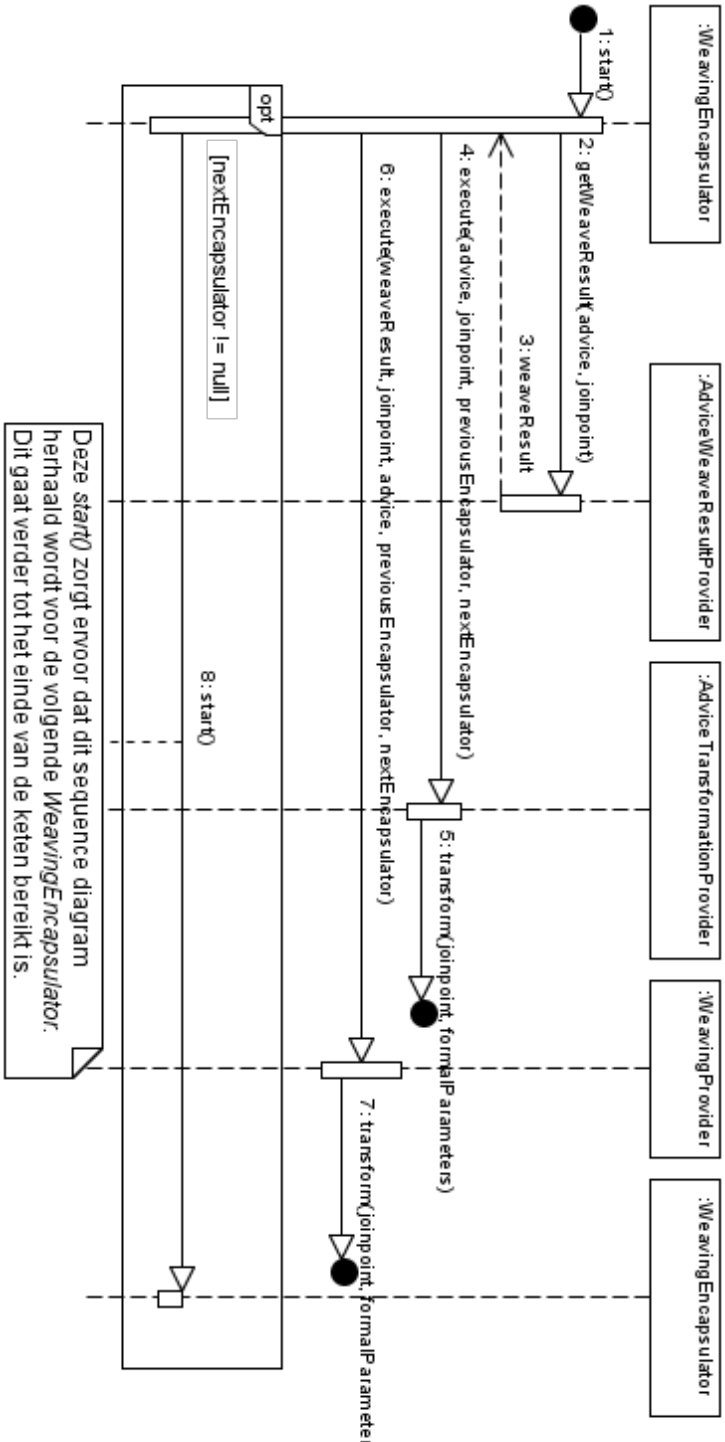
De *WeaveTransformer* dus beschikt nu, per joinpoint, over een lijst van *WeavingEncapsulator*s die dit joinpoint weaven. Het weaven wordt gestart door de eerste *WeavingEncapsulator* (1) en deze doet het volgende, zoals te zien in figuur 8.13:

1. Bereken het resultaat van het weven (via de *AdviceWeaveResultProvider*) (2, 3)
2. Voer bijkomende transformaties op het advice door (via de *AdviceTransformationProvider*) (4)
    a) Voer eventuele runtime transformaties door op het getransformeerde advice (via de *Coordinator*) - deze stap wordt verder in de tekst verfijnd (5)

3. Combineer joinpoint en resultaat (via de *WeavingProvider*) (6)

    a) Voer eventuele runtime transformaties door op het getransformeerde joinpoint (via de *Coordinator*) - deze stap wordt verder in de tekst verfijnd (7)

4. Roep de volgende *WeavingEncapsulator* aan in de keten, indien deze bestaat. Deze begint opnieuw bij stap 1 in deze stappenlijst, maar voor de volgende *WeavingEncapsulator* (8)



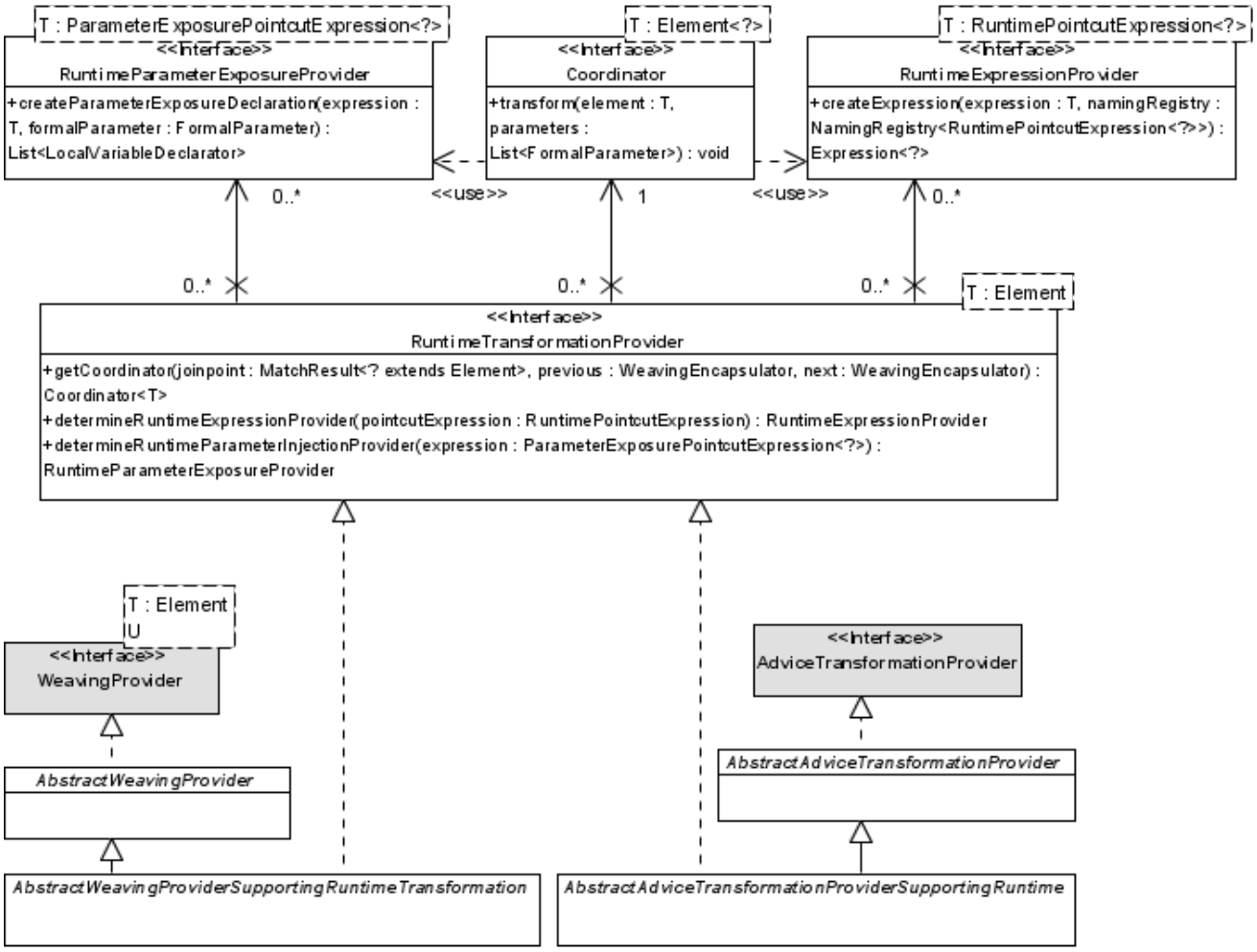
Figuur 8.13: Het sequentie diagramma voor het weven, zonder de transformaties voor runtime toevoegingen

8.3 Runtime controles en het blootstellen van context

De laatste elementen die geïdentificeerd werden voor het opbouwen van een wever, zijn deze die nodig zijn om runtime controles uit te voeren en de context van het joinpoint bloot te stellen naar het advice toe. In secties 7.2.5, 7.2.6 en 7.2.7 werden drie basisvereisten geïdentificeerd: het aanmaken van de expressie die een runtime controle uitvoert, het aanmaken van een expressie die een parameter toewijst en een coördinator die ervoor zorgt dat deze expressies in de juiste volgorde en op de juiste manier ingevoegd worden. In de volgende secties worden deze individueel besproken.

Een bijkomend probleem is echter dat het object dat de runtime transformaties zal starten - door de coördinator aan te roepen - ook niet eenduidig bepaald is. Immers, bij een methode oproep (een *call* pointcut) moeten de runtime transformaties ingevoegd worden in de aangemaakte advice methode, dus na het transformeren van het advice. Het starten van deze transformaties gebeurt dus best door de *AdviceTransformationProvider*, aangezien dit object verantwoordelijk is voor het aanmaken van deze methode. Bij het afhandelen van een exceptie (een *typeHandler* pointcut) moeten deze runtime transformaties echter bij het joinpoint zelf ingevoegd worden, na het weven. Deze starten gebeurt hier best door de *WeavingProvider*, aangezien dit object verantwoordelijk is voor het transformeren van het joinpoint.

Er zijn dus verschillende elementen die moeten interageren om op een juiste manier deze expressies toe te voegen. Figuur 8.14 toont deze interacties. Het is voor zowel *WeavingProvider*s als *AdviceTransformationProvider*s mogelijk om extra transformaties uit te voeren - hiervoor moet de *RuntimeExpressionProvider* geïmplementeerd worden. Deze biedt al mogelijke informatie aan om deze transformaties correct uit te voeren: de objecten die de expressies aanmaken (de *RuntimeParameterExposure*- en *RuntimeExpressionProvider*) en het coördinerend object (de *Coordinator*). Deze laatste gaat de expressies die door de *Provider*s aangemaakt worden integreren in het te transformeren element. Deze concepten worden in deze sectie in meer detail besproken.

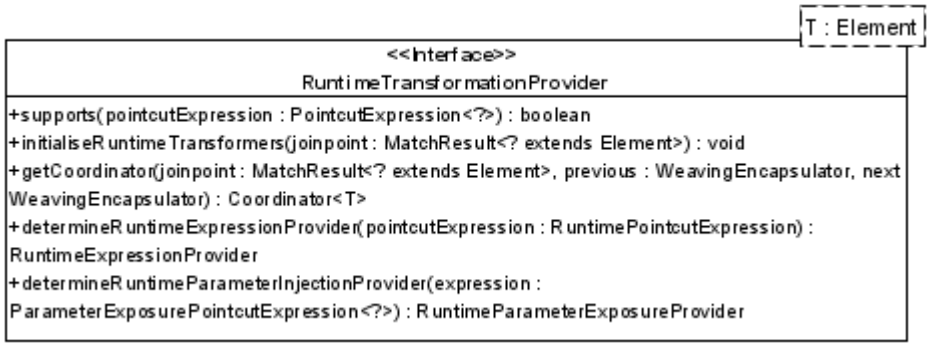


Figuur 8.14: Het toevoegen van runtime expressies gebeurt door een *RuntimeTransformationProvider*. Deze bepaalt hoe de expressies opgebouwd worden (*RuntimeParameterExposureProvider* en *RuntimeExpressionProvider*) en hoe deze ingevoegd moeten worden (*Coordinator*)

8.3.1 De RuntimeTransformationProvider

Rationale (vereiste 8 op pagina 43) Zoals gezegd is het object dat de runtime transformaties start niet vast bepaald. Het kan zowel een *AdviceTransformationProvider* zijn als een *WeavingProvider*. Om ervoor te zorgen dat het starten van de runtime transformaties op een eenduidige en overzichtelijke manier gebeurt, moeten objecten die een runtime transformatie willen starten de *RuntimeTransformationProvider* interface implementeren. Deze interface biedt de nodige methodes om de runtime transformatie te starten en om de coördinator, die de runtime transformatie effectief uitvoert, van de nodige informatie te voorzien.

Implementatie De *RuntimeTransformationProvider* interface declareert vijf methodes die allen dienen om de runtime transformaties correct uit te voeren. De methode *supports* controleert of een bepaalde pointcut ondersteund wordt. *initialiseRuntimeTransformers* Zal de objecten verantwoordelijk voor de runtime controles en het blootstellen van parameters initialiseren indien dit nodig is. De *getCoordinator* methode haalt de correcte coördinator op die de effectieve transformatie zal uitvoeren. De twee laatste methodes, *determineRuntimeExpressionProvider* en *determineRuntimeParameterInjectionProvider*, halen aan de hand van het type pointcut de objecten op die de expressies creëren om de controles uit te voeren en parameters toe te wijzen.

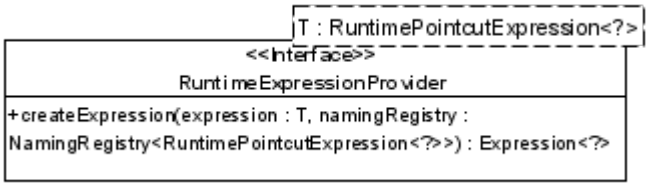


FIGUUR 8.15: De *RuntimeTransformationProvider* interface

8.3.2 Runtime controles

Rationale (vereiste 5 op pagina 42) Zoals gezegd in 7.2.5, is voor eenzelfde pointcut dat een runtime controle uitvoert - bijvoorbeeld het *this* pointcut - de expressie van de controle niet overal gelijk, maar varieert dit naar gelang de manier waarop gewoven wordt. Het is dus het object dat de runtime transformatie initieel start - een *RuntimeTransformationProvider* - dat zal aangeven hoe de expressie om de controle uit te voeren tot stand moet komen. Om de herbruikbaarheid te vergroten, zal dit niet rechtstreeks worden geïmplementeerd in de *RuntimeTransformationProvider*. Deze zal wel voor een gegeven pointcut aangeven welk object, een *RuntimeExpressionProvider,* verantwoordelijk is voor het aanmaken van de expressie. Op deze manier kan bij verschillende gevallen waar de runtime controle op dezelfde manier tot stand moet komen dezelfde *RuntimeExpressionProvider* gebruikt worden.

Implementatie De implementatie van de *RuntimeExpressionProvider* interface is zeer eenvoudig. De interface declareert slechts een enkele methode die, gegeven de pointcut en een register van namen, de correcte expressie genereereert die de runtime controle uitvoert. Het register van namen dient om pointcuts die steunen op resultaten van eerdere controles - bijvoorbeeld conjuncties - te kunnen ondersteunen. Het register bevat dan de namen van de variabelen waarin de resultaten van de eerdere controles opgeslagen zijn.



Figuur 8.16: De *RuntimeExpressionProvider* interface

8.3.3 Het toewijzen van parameters

Rationale (vereiste 6 op pagina 43) Net zoals bij runtime controles kan het toewijzen van de parameters van een advice niet altijd op eenzelfde manier gebeuren. Daarom wordt dezelfde strategie toegepast als bij runtime controles: het aanmaken van de expressies om de parameter toe te wijzen gebeurt in een apart object - een *RuntimeParameterExposureProvider* - niet in de *RuntimeTransformationProvider* zelf. Deze geeft aan welke provider moet gebruikt worden voor welk pointcut.

Implementatie Ook hier weer een analoge en eenvoudige implementatie. Er wordt slechts een methode gedeclareerd die, gegeven de parameter van het advice die moet toegewezen worden en de pointcut die dit doet, de expressies die ervoor zorgen dat de parameter toegewezen wordt opbouwd.

Figuur 8.17: De *RuntimeParameterExposureProvider* interface

8.3.4 Het coördineren van de runtime transformaties

Rationale (vereiste 7 op pagina 43) Zoals vermeld in sectie 7.2.7 is de volgorde van de runtime controles en het toewijzen van de parameters niet eenduidig bepaald. Daarom is er een coördinerend proces nodig dat de expressies op de juiste plaats en in de juiste volgorde invoegt.

Implementatie De implementatie van de *Coordinator* interface is tevens eenvoudig. De interface declareert een enkele methode die, gegeven het te transformeren element - bijvoorbeeld de aangemaakte advice methode - en de parameters van het advice die moeten toegewezen worden, de correcte expressies zal toevoegen aan het element.

FIGUUR 8.18: De *Coordinator* interface

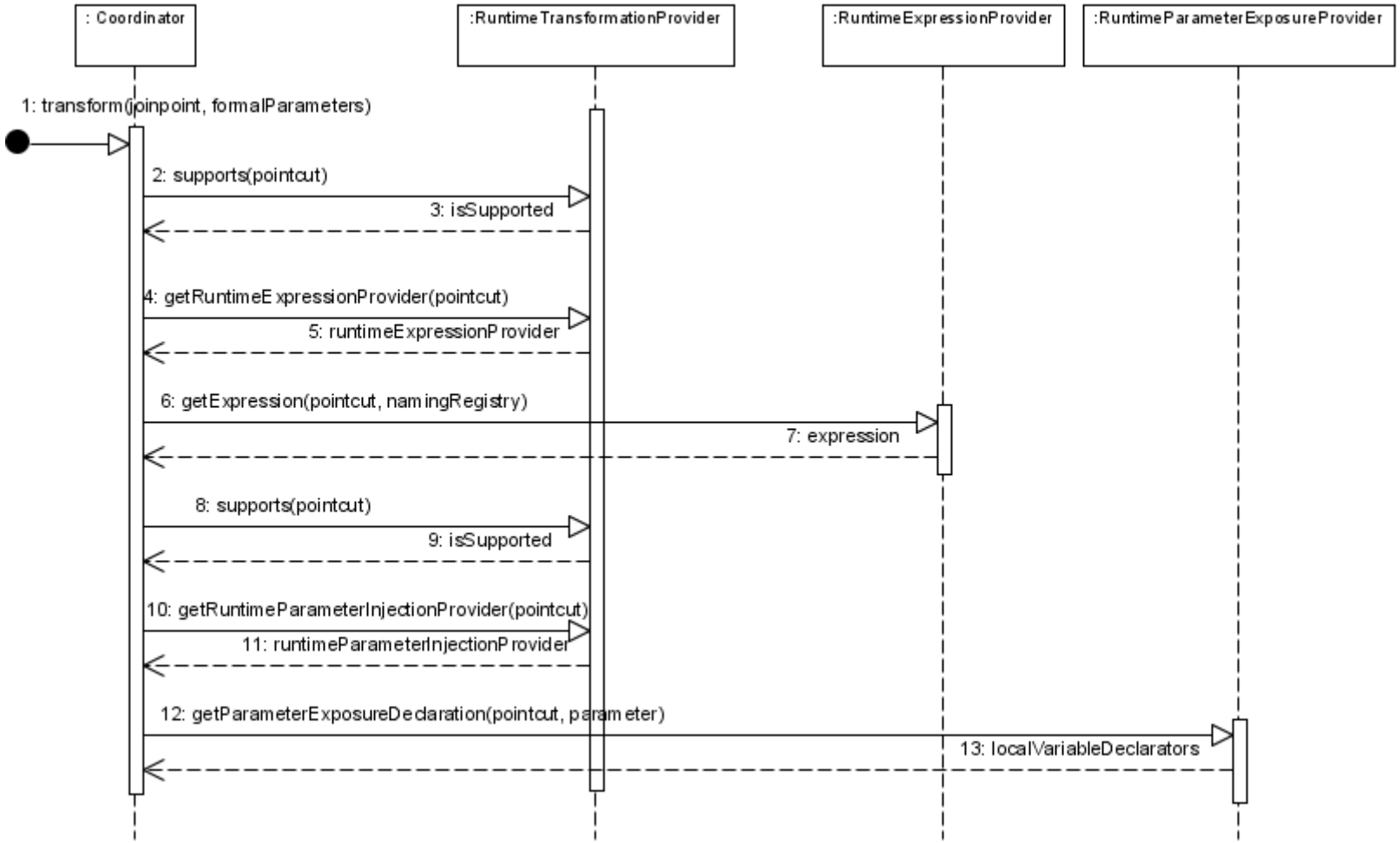
8.3.5 Interacties

Een mogelijk voorbeeld voor hoe een *Coordinator* in zijn werk gaat is te vinden in figuur 8.19. Dit hangt echter af van het type *Coordinator*, onderstaande stappen zijn indicatief en representatief voor het gedrag van een *Coordinator*, maar gelden niet volledig in alle gevallen.

1. De pointcut boom wordt gefiltered - eerst worden alle runtime controles die ondersteund worden door de *RuntimeTransformationProvider* eruit gehaald (2, 3)
2. Vervolgens wordt voor elk van deze pointcuts het object dat de expressie zal opbouwen opgevraagd (4, 5)
3. Dit object maakt vervolgens de effectieve expressie aan, die de *Coordinator* zal invoegen (6, 7)

Dit patroon herhaalt zicht voor het blootstellen van parameters:

1. Het filteren van de pointcut boom voor het blootstellen van parameters (8, 9)
2. Het opvragen van het object dat de expressies aanmaakt (10, 11)
3. Het opvragen van de expressies (12, 13)



Figuur 8.19: Het sequentie diagramma voor het invoegen van de runtime controles en het toewijzen van parameters.

Hoofdstuk 9

Evaluatie

In dit hoofdstuk wordt een evaluatie gemaakt van zowel het Chameleon raamwerk, als van het raamwerk dat ontwikkeld werd voor aspect weavers. Het ontwikkelde raamwerk wordt geëvalueerd door een concrete implementatie te maken van twee weavers - voor Java en JLo.

9.1 Implementatie van een weaver voor Java

Nu het volledige raamwerk opgesteld is, moet er natuurlijk gekeken worden of dit ook effectief voldoet om een functionele en uitbreidbare weaver te implementeren. Om dit te toetsen werd ervoor gekozen een wever voor Java te implementeren, gebruik makende van het raamwerk. Het gaat hier om een statische weaver die zal werken op de abstracties die Chameleon biedt. Aangezien het model uiteindelijk terug omgezet wordt naar broncode, valt deze weaver te vergelijken met een broncode weaver wat betreft de mogelijkheden en performantie, in tegenstelling tot bijvoorbeeld AspectJ waar dit op bytecode niveau gebeurt[1].

9.1.1 Opbouwen van het model binnen Chameleon

De eerste stap die genomen moet worden is om de nieuwe concepten van de aspect taal een invulling te geven binnen Chameleon. Dit zijn alle pointcuts die geïmplementeerd worden door de aspect taal, aangezien de concepten van een aspect en advice al aangewezig zijn binnen het raamwerk. De volgende pointcuts werden geïmplementeerd - de hoeveelheid werk wordt geduidt bij de weaver voor dat pointcut. Figuur 9.1 op pagina 69 schetst de plaats van deze pointcuts binnen het raamwerk.

- Statische pointcuts:
  - Aanroepen van een methode met een bepaalde signatuur (*call*)
  - Aanroepen van een methode met een bepaalde annotatie (*callAnnotated*)

[1] Het is mogelijk om, binnen dit raamwerk, een Java weaver te ontwikkelen die op dit vlak dichter bij AspectJ staat en ook bytecode zal outputten. Echter, hiervoor moet een parser ontwikkeld worden die bytecode omzet naar een model binnen Chameleon en omgekeerd. Voor de meeste tools waar Chameleon ondersteuning voor wil bieden (zoals editors of refactoring tools) is dit nutteloos aangezien deze werken met broncode.

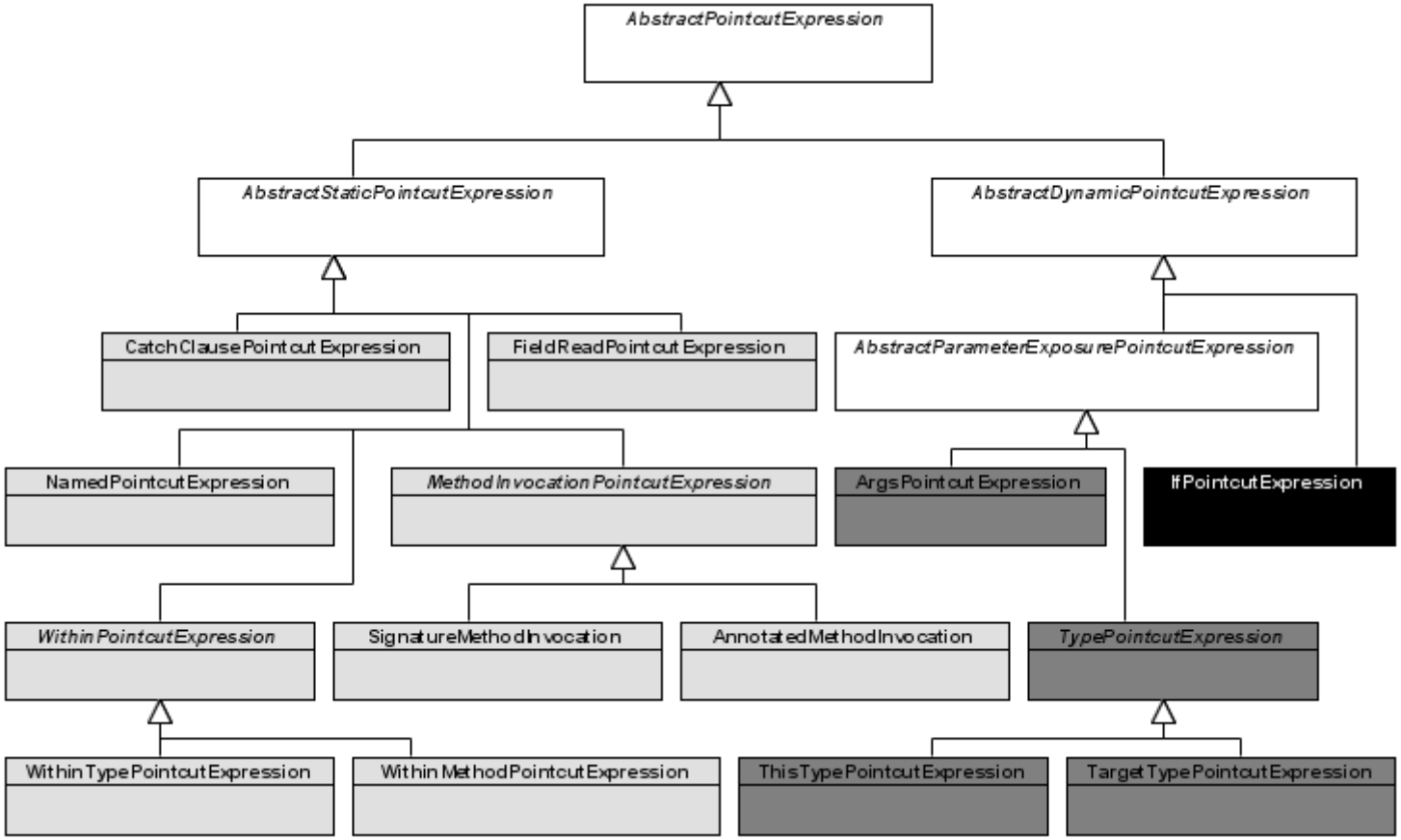
  - Lezen van een veld (*fieldRead*)
  - Afhandelen van een exceptie (*typeHandler*)
  - Afhandelen van een exceptie waarbij geen instructies worden uitgevoerd (*emptyHandler*)
  - Typecasten een expressie (*cast*)
  - Controleren of een joinpoint zich bevindt binnen een bepaald type of een bepaalde methode (*within*)

- Runtime controles en toewijzen van parameters:
  - Controleren op type van argumenten / toewijzen van argumenten aan parameters (*arguments*)
  - Controleren op type van het huidige object / toewijzen van het huidig object aan een parameter (*thisType*)
  - Controleren op type van het doel object / toewijzen van het doelobject aan een parameter (*targetType*)
  - Runtime controle met een booleaanse expressie (*if*)

Advice kan ook op verschillende manieren interageren met het joinpoint. De volgende types - die ook in AspectJ voorkomen - worden gedefinieerd:

- *Before* advice, dat voor het joinpoint wordt uitgevoerd
- *After returnin*g advice, dat na het joinpoint wordt uitgevoerd indien dit joinpoint niet voortijdig beëindigd werd door het gooien van een exceptie
- *After throwing* advice, dat na het joinpoint wordt uitgevoerd indien dit joinpoint wél voortijdig beëindigd werd
- *Afer* advice, dat sowieso na het joinpoint wordt uitgevoerd, ongeacht hoe het joinpoint beëindigd werd
- *Around* advice, dat in plaats van het joinpoint wordt uitgevoerd maar waarbij de mogelijkheid bestaat om het originele advice uit te voeren, al dan niet met andere parameters

De volgende stap is om een syntax te definiëren voor de aspect taal die gebruikt zal worden. Omdat deze weaver in de eerste plaats dient om het raamwerk te illusteren en evalueren, wordt de syntax sterk gebaseerd op AspectJ. Om een aspect dat aan de syntax voldoet om te zetten naar een model binnen Chameleon, moeten er natuurlijk een lexer en parser geïmplementeerd worden. Deze werden niet manueel gecreëerd, er zijn verschillende oplossingen die dit proces sterk vereenvoudigen. Voor dit project werkt gekozen voor *ANother Tool for Language Recogition (ANTLR)* [10][9]. ANTLR laat toe om een bepaalde grammatica te definiëren en kan een lexer en parser genereren die in staat is om, gegeven een tekst die aan die grammatica voldoet, een abstracte syntax boom op te bouwen: in dit geval het model dat bestaat uit de abstracties die Chameleon biedt.



FIGUUR 9.1: De nieuwe pointcuts, binnen het raamwerk. De statische poincuts in het lichtgrijs, runtime controles in het zwart en combinaties van runtime controles en het toewijzen van parameters in het donkergrijs.

9.1.2 Implementatie van de Java weaver

De volgende sectie geeft aan hoeveel werk (gemeten in lijnen code[2]) de implementatie van de weaver kost.

Algemene elementen

Eerst worden de algemene elementen aangehaald in tabel 9.1, die voor alle afzonderlijke elementweavers gemeenschappelijk zijn. Deze elementen zijn dus herbruikbaar voor alle elementweavers maar wel typisch taalspecifiek, al is herbruik tussen verschillende talen mogelijk. Een voorbeeld hiervan is het object dat de runtime controle (een booleaanse expressie) aanmaakt voor een conjunctie. Deze zal gewoon de conjuntie nemen van de runtime controles van de deelbomen - ook booleanse expressies dus. In Java wordt hiervoor natuurlijk de && operator gebruikt. Deze is echter niet expliciet in Chameleon aanwezig, dit is een *InfixOperatorInvocation* met als methodenaam *&&*. Dit object is natuurlijk herbruikbaar voor alle elementweavers, en zou dus ook bruikbaar zijn voor bijvoorbeeld een C# weaver aangezien ook hier de && operator van toepassing is, maar niet voor talen waar een andere operator gebruikt wordt.

Entiteit	Implementatie naam	# regels code
WeaveTransformer	JavaWeaveTransformer	26
RuntimeExpressionProvider	RuntimeOr	22
	RuntimeAnd	22
	RuntimeNot	20
	RuntimeTypeCheck	25
	RuntimeIfCheck	12
	RuntimeArgumentsTypeCheck	56
	RuntimeSingleArgumentTypeCheck	44
RuntimeParameterExposureProvider	MultipleArgsParameterExposure	51
	TypeParameterExposure	36
	MultipleArgsParameterExposure	51
	SingleArgParameterExposure	46
Totaal		**411**

Tabel 9.1: Implementatie details voor de elementen van een Java weaver die gedeeld kunnen worden voor alle element weavers

Het is duidelijk dat er voor elk element relatief weinig regels code nodig zijn om het te beschrijven. In het geval van de *Provider*s gaat het hier om eenvoudige expressies - er moeten dus weinig model elementen aangemaakt worden, waardoor de grootte beperkt blijft. De *JavaWeaveTransformer* is enkel verantwoordelijk om de specifieke *Weaver*s die beschikbaar zijn voor de Java weaver - methode oproepen, afhandelen van excepties, etc. - te registreren. Ook deze is dus beperkt in grootte.

Ter herhaling, de volgende elementen zijn nog niet aanwezig in het raamwerk en zijn wel nodig om een joinpoint te kunnen weaven:

[2]Witruimte en commentaar niet meegeteld maar wel inclusief alle definities (klasses, methodes) en *import statements*. Geteld met CLOC, versie 1.53. http://cloc.sourceforge.net/

1. De *ElementWeaver* zelf, die de drie providers vastlegt
2. De *AdviceWeaveResultProvider* [3]
3. Een of meerdere *WeavingProvider*s, afhankelijk van de gevolgde strategie en het type advice [4]
4. Een of meerdere *AdviceTransformationProvider*s, afhankelijk van de gevolgde strategie en het type advice [5]
5. Een *RuntimeExpressionProvider* per runtime controle (bv. *if*) die door de wever ondersteund wordt
6. Een *RuntimeParameterExposureProvider* per parameter toewijzing (bv. *arguments*) die door de wever ondersteund wordt
7. Een *Coordinator* die deze laatste twee gegenereerde expressies op de juiste plaats en in de juiste volgorde invoegt.

In de volgende secties wordt, bij het beschrijven van het aantal lijnen code voor elk element, telkens aangegeven tot welk van deze zeven entiteiten dat element behoort door het overeenkomstig nummer in deze lijst, tussen haakjes te vermelden. Indien een liggend streepje staat bij het aantal regels code impliceert dit dat dit element niet extra geïmplementeerd moest worden, maar reeds aanwezig was - in het raamwerk of in het taalspecifieke deel (tabel 9.1 op de linker pagina).

[3] met uitzondering van identiteitsprovider, die gewoon het advice zelf teruggeeft en reeds aanwezig is in het raamwerk

[4] met uitzondering van de standaard provider die het joinpoint integraal vervangt door een ander element en reeds aanwezig is in het raamwerk

[5] met uitzondering van de *no-operation* provider die het advice niet zal transformeren en reeds aanwezig is in het raamwerk

Methode oproepen op basis van signatuur

Implementatie Zoals eerder vermeld gaat de strategie van het weaven sterk samen van het type joinpoint. Er werd in eerste instantie gekeken naar het *call* pointcut, dat methode oproepen met een bepaalde signatuur zal selecteren.

Entiteit	Implementatie naam	# regels code
PointcutExpression	MethodInvocationPointcutExpression	12
	SignatureMethodInvocationPointcutExpression	64
	MethodReference	128
ElementWever (1)	MethodInvocationWeaver	64
AdviceWeaveResultProvider (2)	AdviceMethodProvider*	63
	TargetedAdviceMethodProvider*	25
	DefaultReflectiveMethodInvocation	61
WeavingProvider (3)	ElementReplaceProvider	-
AdviceTransformationProvider (4)	CreateAdviceMethodTransformationProvider*	135
	ReflectiveAdviceTransformationProvider*	193
	ReflectiveMethodInvocation*	335
	BeforeReflectiveMethodInvocation	15
	AfterReflectiveMethodInvocation	23
	AroundReflectiveMethodInvocation	70
	AfterReturningReflectiveMethodInvocation	33
	AfterThrowingReflectiveMethodInvocation	66
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeArgumentsTypeCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
	ReflectiveArgsParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	64
	AdviceMethodCoordinator	45
	MethodCoordinator	66
Totaal		**1462**

Tabel 9.2: Implementatie details voor een wever voor methode oproepen met een bepaalde signatuur. Klassen met een asterisk (*) zijn abstract.

Zoals tabel 9.2 aantoont, kost het dus een 1400-tal regels code om, met behulp van het raamwerk, de nodige pointcuts en een weaver voor methode oproepen te implementeren - met ondersteuning voor verschillende runtime pointcuts.

Analyse Om methode oproepen te kunnen weven zijn er dus relatief veel regels code nodig. Dit heeft twee redenen: enerzijds is dit een complexe case omdat met veel gevallen rekening gehouden moet worden (excepties, statische methodes, parameters, ...) wat een weerslag geeft op het model dat gegenereerd moet worden en anderzijds

was het ook het eerste pointcut waarvoor een weaver ontworpen werd. Bij de andere weavers die op een gelijkaardige manier werken kan veel van deze code herbruikt worden.

Bij een diepere analyse blijkt dat de meeste code (in de bestanden *CreateAdviceMethodTransformationProvider*, *ReflectiveAdviceTransformationProvider* en *ReflectiveMethodInvocation* - samen meer dan 600 regels code van de ca. 1450) dient om een model op te bouwen binnen Chameleon dat de hulpmethodes voor de weaver bevat - bijvoorbeeld het oproepen van de originele methode vanuit de gecreëerde advice methode via reflectie. De methode die dit doet bestaat uit zo'n 100 regels code (voor 40 regels uiteindelijk gegenereerde Java code). Verder moet nog de advice methode zelf alsook de nodige methode oproepen gegenereerd worden, waarbij het ratio van aantal regels code om het model van Chameleon abstracties te genereren op het aantal regels uiteindelijke Java code van dezelfde omvang is. Deze overhead is op zich niet te vermijden omdat dit inherent is aan het gebruik van het Chameleon raamwerk. Bij de evaluatie van Chameleon wordt dit probleem uitgebreider besproken - zie sectie 9.5 op pagina 85. Daar wordt ook het implementeren van factories voor veelgebruikte constructies - bijvoorbeeld de klassieke for-loop: **for** (**int** i = 0; i < n; i++) - aangehaald als mogelijke oplossing. De winst in code die te behalen valt met zo'n dergelijke factory wordt in dit geval op meer dan 50% geschat door een snelle analyse te maken van de gebruikte constructies in de code. Het aanmaken van een lege, *private* constructor kost bijvoorbeeld 5 regels, wat met een factory een enkele regel code kan worden. En hoewel de te behalen winst niet in alle gevallen van deze grootte orde is - sommige constructies zijn te specifiek om in een factory te plaatsen - is een vermindering met 50% van de regels code die het model aanpassen of transformeren, zeker haalbaar. De totale regels code kunnen dus zeer waarschijnlijk tot onder de 1000 teruggebracht worden.

Een tweede punt om op te merken is dat het verschil in implementatie tussen de verschillende types van advice erg klein is. Door een goede analyse te maken van de structuur van de advice methode, kan er voldoende gemeenschappelijke code geabstraheerd worden.

Methoden oproepen op basis van annotaties

Implementatie Het volgend pointcut dat geïmplementeerd werd, zijn oproepen van methode met een bepaalde annotatie (*callAnnotated*). Tabel 9.3 toont aan dat hiervoor bijna geen extra werk vereist is.

Entiteit	Implementatie naam	# regels code
PointcutExpression	MethodInvocationPointcutExpression	-
	AnnotatedMethodInvocationPointcutExpression	47
	AnnotationReference	31
ElementWever (1)	MethodInvocationWeaver	-
AdviceWeaveResultProvider (2)	AdviceMethodProvider*	-
	TargetedAdviceMethodProvider*	-
	DefaultReflectiveMethodInvocation	-
WeavingProvider (3)	ElementReplaceProvider	-
AdviceTransformationProvider (4)	CreateAdviceMethodTransformationProvider*	-
	ReflectiveAdviceTransformationProvider*	-
	ReflectiveMethodInvocation*	-
	BeforeReflectiveMethodInvocation	-
	AfterReflectiveMethodInvocation	-
	AroundReflectiveMethodInvocation	-
	AfterReturningReflectiveMethodInvocation	-
	AfterThrowingReflectiveMethodInvocation	-
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeArgumentsTypeCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
	ReflectiveArgsParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	AdviceMethodCoordinator	-
	MethodCoordinator	-
Totaal		**78**

Tabel 9.3: Implementatie details voor een weaver voor methode oproepen met een bepaalde annotatie. Klassen met een asterisk (*) zijn abstract.

Analyse Merk op dat in de meeste gevallen - voor de geïmplementeerde pointcuts *alle* gevallen - geldt dat, bij het definiëren van een weaver, de strategie zal afhangen van welk type joinpoint er gewoven wordt - niet van het type pointcut dat dit joinpoint selecteerde. Omdat we nu een weaver hebben aangemaakt die methode oproepen kan weaven, kan deze meteen ook gebruikt worden om *alle* methode oproepen te behandelen, ongeacht van welk pointcut die selecteert. Voor een pointcut dat alle oproepen van methodes met een bepaalde annotatie selecteert kan ook de vorige wever

gebruikt worden met een zeer kleine hoeveelheid bijkomend werk - het aanpassen van de parser en het implementeren van de pointcuts binnen het model.

Het lezen van een veld

Implementatie Het derde pointcut dat wordt beschouwd is het lezen van een veld (*fieldRead*). In tabel 9.4 staan de details voor de wever. Deze wever ondersteunt *before*, *around, after* en *after returning* advice. Het *arguments* pointcut is niet ondersteund aangezien dit niet van toepassing is op dit het lezen van velden.

Entiteit	Implementatie naam	# regels code
PointcutExpression	FieldReadPointcutExpression	75
	MemberReference	28
ElementWeaver (1)	FieldWeaver	40
	FieldReadWeaver	45
AdviceWeaveResultProvider (2)	AdviceMethodProvider*	-
	TargetedAdviceMethodProvider*	-
	DefaultReflectiveFieldAccess	36
WeavingProvider (3)	ElementReplaceProvider	-
AdviceTransformationProvider (4)	CreateAdviceMethodTransformationProvider	-
	ReflectiveAdviceTransformationProvider*	-
	ReflectiveFieldRead	125
	BeforeReflectiveFieldRead	15
	AfterReflectiveFieldRead	27
	AroundReflectiveFieldRead	57
	AfterReturningReflectiveFieldRead	34
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	AdviceMethodCoordinator	-
Totaal		**482**

Tabel 9.4: Implementatie details voor een weaver voor het lezen van een veld. Klassen met een asterisk (*) zijn abstract.

Analyse De strategie voor het weaven komt hier voor een groot deel overeen met deze van methode oproepen. Ook hier wordt een advice methode aangemaakt die opgeroepen wordt in de plaats van het originele joinpoint, en gebeurt het (eventuele) oproepen van het joinpoint via reflectie. Daarom is het ook niet verwonderlijk dat er veel code herbruikt kan worden - de omvang van deze weaver is slechts ca. 30% van de omvang van de weaver voor methode oproepen. Net als bij de weaver voor methode oproepen, is het ook hier zo dat een groot deel van de code (in bijvoorbeeld

de klasse *ReflectiveFieldRead*) gaat naar het transformeren van het model. Ook hier kan dus het aantal lijnen code teruggebracht worden door het gebruik van factories, naar schatting 20%.

Het afhandelen van excepties op basis van type

Implementatie Als volgend pointcut wordt nu de wever voor het afhandelen van excepties van een bepaald type bekeken (*typeHandler*). Deze wever ondersteunt *before*, *around* en *after* advice. De details van de implementatie zijn te zien in tabel 9.5.

Entiteit	Implementatie naam	# regels code
CatchClausePointcutExpression	CatchClausePointcutExpression	23
	TypeCatchClausePointcutExpression	54
ElementWever (1)	CatchClauseWeaver	54
AdviceWeaveResultProvider (2)	ReturnAdviceProvider	-
WeavingProvider (3)	CatchClauseInsertProvider*	86
	CatchClauseInsertBeforeProvider	18
	CatchClauseInsertAfterProvider	18
	CatchClauseInsertAroundProvider	22
AdviceTransformationProvider (4)	NoOperationTransformationProvider	-
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeSingleArgumentTypeCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
	SingleArgParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	CatchClauseCoordinator	58
Totaal		**333**

Tabel 9.5: Implementatie details voor een weaver voor het afhandelen van een exceptie van een bepaald type. Klassen met een asterisk (*) zijn abstract.

Analyse Het afhandelen van een exceptie moet, zoals eerder aangehaald, volgens een andere strategie gebeuren. Er wordt dus typisch minder herbruik verwacht dan bij bijvoorbeeld het lezen van velden. Dit is in de praktijk ook zo, maar omdat het een eenvoudiger proces is in vergelijking met de weaver voor methode oproepen - er moeten geen extra methodes gecreëerd worden - blijft de hoeveelheid code beperkt. Ook hier is het zo dat de meeste code (in de *CatchClauseInsertProvider* klasse) nodig is voor het opbouwen van nieuwe elementen binnen het model. Aangezien dit echter beperkter is dan bij de vorige weavers wordt ook minder winst verwacht door het gebruik van factories. Verwacht wordt dat deze weaver rond de 300 regels code groot blijft.

Het afhandelen van excepties zonder instructies uit te voeren

Implementatie Het is ook mogelijk om bij het afhandelen van excepties niet te kijken naar het type van de exceptie, maar naar andere voorwaardes. Een mogelijk interessante usecase is bijvoorbeeld om te kijken naar plaatsen waar excepties afgehandeld worden, maar er geen instructies uitgevoerd worden - lege *catch* blokken. Daarom werd ook het *emptyHandler* pointcut gedefinieerd, dat precies deze joinpoints selecteert. Tabel 9.6 toont de implementatie details.

Entiteit	Implementatie naam	# regels code
CatchClausePointcutExpression	CatchClausePointcutExpression	-
	EmptyCatchClausePointcutExpression	30
ElementWever (1)	CatchClauseWeaver	-
AdviceWeaveResultProvider (2)	ReturnAdviceProvider	-
WeavingProvider (3)	CatchClauseInsertProvider*	-
	CatchClauseInsertBeforeProvider	-
	CatchClauseInsertAfterProvider	-
	CatchClauseInsertAroundProvider	-
AdviceTransformationProvider (4)	NoOperationTransformationProvider	-
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeSingleArgumentTypeCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
	SingleArgParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	CatchClauseCoordinator	-
Totaal		**30**

Tabel 9.6: Implementatie details voor een weaver voor het afhandelen van een exceptie waarbij geen instructies uitgevoerd worden. Klassen met een asterisk (*) zijn abstract.

Analyse Hier geldt hetzelfde verhaal als bij de methode oproepen met een bepaalde annotatie. De *volledige* weaver die excepties zal weaven voor types kan hergebruikt worden aangezien het gaat om hetzelfde type joinpoint. Opnieuw is er dus een zeer kleine hoeveelheid werk nodig om dit pointcut ook te ondersteunen.

Het typecasten van expressies

Implementatie Als volgende pointcut werd ervoor gekozen om joinpoints te selecteren die *typecasts* voorstellen (*cast* pointcut). Deze implementatie is geïnspireerd op het werk van Toledo et al. [13] - zie ook hoofdstuk 10 - die het *cast* pointcut gebruiken om aan te tonen dat hun aanpak makkelijk uitbreidbaar is. Om diezelfde reden werd het ook binnen deze weaver geïmplementeerd. De weaver voor casts ondersteunt alle types

advice: *before* advice spreekt voor zich. *After returning* advice wordt slechts uitgevoerd indien de cast correct gebeurt en er dus geen exceptie optreedt. Het resultaat kan ook blootgesteld worden. *After throwing* advice treedt op als een (specifieke) exceptie optreedt bij het casten. De exceptie kan ook blootgesteld worden. *After* advice wordt sowieso uitgevoerd na het casten, of dit nu slaagt of niet. *Around* advice wordt in de plaats van het joinpoint uitgevoerd en kan een andere *return value* opgeven. Ook de originele cast kan uitgevoerd worden, eventueel met andere parameters, met behulp van een *proceed* call. Verder zijn *alle* runtime controles en parameters ook mogelijk, met uitzondering van het *target* pointcut aangezien dit geen betekenis heeft in deze context.

De implementatie van dit pointcut is gegeven in tabel 9.7.

Entiteit	Implementatie naam	# regels code
PointcutExpression	CastPointcutExpression	43
Wever (1)	CastWeaver	65
AdviceWeaveResultProvider (2)	AdviceMethodProvider*	-
	DefaultCastProvider	37
WeavingProvider (3)	ElementReplaceProvider	-
AdviceTransformationProvider (4)	CreateAdviceMethodTransformationProvider*	-
	CastTransformationProvider*	15
	BeforeCastTransformationProvider	15
	AfterCastTransformationProvider	20
	AroundCastTransformationProvider	25
	AfterReturningCastTransformationProvider	34
	AfterThrowingCastTransformationProvider	75
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeSingleArgumentTypeCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
	SingleArgParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	AdviceMethodCoordinator	-
Totaal		**329**

Tabel 9.7: Implementatie details voor een weaver voor het casten van expressies. Klassen met een asterisk (*) zijn abstract.

Analyse Zoals vermeld werd deze pointcut geïmplementeerd om de mogelijkheden van het raamwerk te onderstrepen wat betreft herbruikbaarheid en snelle ontwikkeling. Dit pointcut past immers in het rijtje van methode oproepen en het lezen van velden in die zin dat het ook hier nodig zal zijn om een aparte advice methode aan te maken. Dit impliceert natuurlijk ook mogelijkheden tot herbruik. Het resultaat is een erg functionele weaver voor het casten van expressies die ontwikkeld werd in *een enkele*

mandag. Ook hier zou een Chameleon factory nog voor een vermindering in code kunnen zorgen, maar omdat in dit geval erg weinig elementen moeten aangemaakt of getransformeerd worden, zullen dit slechts enkele regels zijn. De weaver zal rond de 300 regels code blijven.

De locatie van een joinpoint

Implementatie Als laatste toevoeging wordt een speciaal type pointcut beschouwd - het *within* pointcut. Dit pointcut is speciaal in die zin dat het niet een type van joinpoint selecteert - zoals bij het *call* pointcut enkel methode oproepen - maar alle soorten joinpoints. De enige constraint die opgelegd wordt is dat het joinpoint in een bepaald type of in een bepaalde methode gedefinieerd moet zijn. Omdat er weinig gevallen zijn waarbij het nuttig is om *alle* soorten van joinpoints binnen een bepaalde type of methode te weaven, wordt deze pointcut zo goed als altijd gebruikt in combinatie met andere pointcuts.

Dit pointcut zal dus geen bijhorende weaver hebben. De implementatie is dus eerder beperkt en te zien in tabel 9.8.

Entiteit	Implementatie naam	# regels code
PointcutExpression	WithinPointcutExpression	9
	WithinTypePointcutExpression	70
	WithinMethodPointcutExpression	54
	MethodReference	-
Totaal		**133**

Tabel 9.8: Implementatie details voor het *within* pointcut

Analyse Dit pointcut is speciaal in die zin dat er geen eigen weaver voor ontworpen werd. Immers, dit pointcut selecteert elk joinpoint - wat voor deze weaver overeenkomt met elk element van het model. Het pointcut kan dus enkel gebruikt worden voor joinpoints waar reeds een weaver voor voorzien is.

9.2 Implementatie van een weaver voor JLo

Naast de weaver voor Java wordt ook een weaver voor JLo geïmplementeerd. Eerst wordt geschetst wat JLo precies is en waarvoor het gebruikt kan worden, vervolgens worden de details van de implementatie besproken.

9.2.1 Wat is JLo?

Subobject geörienteerd programmeren

JLo is een extensie van Java: alle expressies die geldig zijn in Java zijn ook geldig in JLo, maar niet omgekeerd. Het doel van deze extensie is het implementeren van subobjecten[15] in Java. Subobject geörienteerd programmeren is een uitbreiding op object geörienteerd programmeren die toelaat om klassen op te bouwen uit verschillende bouwstenen. Een eenvoudige implementatie van een radio kan bijvoorbeeld bestaan uit

een frequentie, een volume niveau en een aan-uit knop. De frequentie en het volume zijn beperkt in intervallen, de aan-uit knop is booleaans. Als deze drie eigenschappen worden geïmplementeerd op de conventionele manier, moeten er zowel voor het volume als voor de frequentie drie waardes worden bijgehouden: het laagst en hoogst mogelijke niveau en de eigenlijke waarde. Ook controles of de waardes zich binnen de grenzen bevinden moet telkens gebeuren bij het wijzigen van de waarde. De aanpak met subobjecten is eenvoudiger en ziet er als volgt uit:

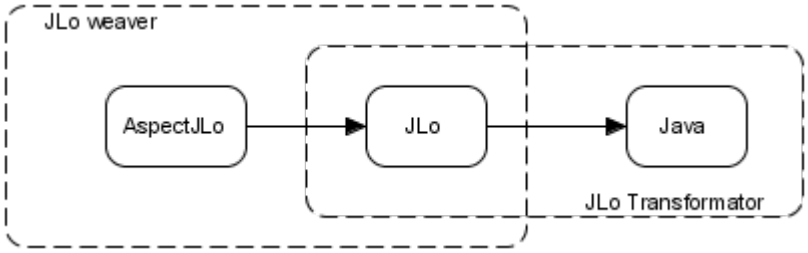
```
class Radio {
        subobject volume BoundedValue<Float> {
                export setValue(Float) as setVolume(Float),
                    getValue as getVolume
        };

        subobject frequency BoundedValue<Float> {
                export setValue(Float) as setFrequency(Float)
                    , getValue as getFrequency
        };

        subobject power BooleanProperty
        {
                export setTrue as powerOn, setFalse as
                    powerOff;
        };
}
```

De eigenschappen van de radio worden dus geïmplementeerd als subobjecten. Via een export clause is het mogelijk om bepaalde methoden van het subobject bloot te stellen in de context van het implementerend object. Zo beschikt de klasse Radio nu over een **public void** powerOn() methode, die gewoon een alias is om de setTrue methode van het power subobject uit te voeren. Deze aanpak heeft verschillende voordelen:

- De interface van de implementerende klasse blijft overzichtelijk en kan enkel de essentiële methodes aanbieden. Zo kan een *BooleanProperty* ook een xor(**boolean** b) methode aanbieden. Dit is in de context van een power knop van een radio echter nutteloos. Omdat het mogelijk is om deze methoden te verbergen, wordt er aangemoedigd om de subobjecten zelf zo functioneel mogelijk te maken, onafhankelijk van de objecten die ze zullen gebruiken.
- Conflicten bij namen - zoals de getValue() methodes - worden eenvoudig opgelost: ze worden hernoemd. Daardoor krijgen ze ook een naam die betekenisvol is zoals getVolume().

JLo

De implementatie van JLo is ook aan de hand van Chameleon gebeurd. Het JLo model wordt dus getransformeerd naar een Java model. Dit impliceert dat JLo specifieke zaken, zoals subobject definities, ook hun overeenkomstig elemenent binnen Chameleon hebben en dus ook een mogelijk target zijn voor nieuwe pointcuts. Bovendien is het



FIGUUR 9.2: De interactie tussen de JLo weaver - die nieuw is - en de JLo transformator, die reeds bestaat

dan ook zo dat de aanwezige JLo transformator herbruikt kan worden. Net zoals de weaver voor Java begint met een bronmodel dat zowel aspect elementen als Java elementen bevat en een puur Java model teruggeeft, start de JLo transformator met een bronmodel dat zowel JLo als Java elementen bevat - aangezien Java een subset is van JLO - en geeft deze een puur Java model terug. Een weaver voor JLo kan dit nu als volgt gebruiken: Het bronmodel bevat zowel aspect, JLo als Java elementen. Dit model wordt gewoven waardoor het enkel nog JLo en Java elementen bevat. Dit model wordt dan gebruikt als bronmodel door de JLo transformator, die er een puur Java model van maakt - zie ook figuur 9.2.

Een interessant punt om hierbij op te merken is dat de JLo transformator bepaalde operaties zal doen die erg gelijken op operaties die een aspect weaver ook uitvoert: het zoeken naar bepaalde punten in het model (cfr. joinpoints) en deze op een bepaalde manier transformeren (cfr. weaven). Het is dan ook waarschijnlijk dat er herbruik mogelijk is tussen het ontworpen raamwerk voor aspect weavers en de JLo transformator!

9.2.2 Implementatie van de weaver voor JLo

Aangezien JLo een extensie is van Java, zijn alle elementen die de ontworpen Java weaver kon transfomeren ook hier aanwezig. Daarom wordt er in eerste instantie gekeken in hoeverre deze herbruikt kunnen worden voor een JLo weaver. Vervolgens wordt een JLo-specifiek pointcut geïmplementeerd en wordt gekeken naar het benodigd werk voor de implementatie.

De resultaten voor het eerste punt zijn zeer positief. Het is niet verrassend dat zaken zoals normale methode oproepen herbruikt kunnen worden - deze zijn immers exact hetzelfde gebleven en de (eventuele) invloed die JLo heeft op deze elementen wordt volledig afgeschermd door het model. Maar ook methode oproepen op subobjecten kunnen *zonder enige aanpassing* gewoven worden. Met het radio voorbeeld zou dit bijvoorbeeld de volgende oproep zijn: radio.volume.getValue(). Hierbij wordt de methode getValue() opgeroepen op het subobject radio.volume. Maar dit is geen geldige Java code, want volume is geen veld van radio! Dit wordt echter toch correct afgehandeld door de weaver net omdat deze werkt op de abstracties gedefinieerd door Chameleon. Een methode oproep kan een bepaald *target* hebben - of dit nu een veld, een andere methode oproep of een subobject is is voor de weaver van geen belang. Het zal de JLo transformator zijn die het lezen van een subobject opzoekt en dit zal

transformeren. Analoog is het ook mogelijk om JLo code - zoals subobject oproepen - te schrijven binnen een advice.

Het tweede punt is het implementeren van een specifieke pointcut voor JLo, met bijhorende weaver. Tabel 9.9 is het JLo equivalent van tabel 9.1 op pagina 70 voor de Java weaver. Het is duidelijk dat zo goed als alle elementen integraal herbruikt kunnen worden.

Entiteit	Implementatie naam	# regels code
WeaveTransformer	JLoWeaveTransformer	30
RuntimeExpressionProvider	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeArgumentsTypeCheck	-
	RuntimeSingleArgumentTypeCheck	-
RuntimeParameterExposureProvider	MultipleArgsParameterExposure	-
	TypeParameterExposure	-
	MultipleArgsParameterExposure	-
	SingleArgParameterExposure	-
Totaal		**30**

Tabel 9.9: Implementatie details voor de elementen van een JLo weaver die gedeeld kunnen worden voor alle element weavers

Het lezen van een subobject

Implementatie Er werd geopteerd voor het lezen van een subobject (*subobjectRead*) - een call zoals radio.volume bijvoorbeeld. Deze pointcut is namelijk specifiek voor JLo, dus een goede kandidaat om aan de hand van een weaver ervoor, een evaluatie te maken. Aangezien het lezen van een subobject geen exceptie kan veroorzaken, wordt enkel *before*, *after*, *around* en *after-returning* advice ondersteund. Op het gebied van runtime controles en het toewijzen van parameters wordt ook het arguments - pointcut niet ondersteund. Er kunnen immers geen argumenten meegegeven worden bij het lezen van een subobject. Verder worden alle pointcuts voor runtime controles en parameter toewijzingen die al in de Java weaver beschikbaar waren, ondersteund.

Ook het nodige werk voor de implementatie van de specifieke weaver voor het *subobjectRead* pointcut is zeer beperkt, zoals tabel 9.10 aangeeft.

Entiteit	Implementatie naam	# regels code
PointcutExpression	SubobjectRead	84
Wever (1)	SubobjectReadWeaver	59
AdviceWeaveResultProvider (2)	AdviceMethodProvider*	-
	TargetedAdviceMethodProvider*	-
	DefaultSubobjectRead	37
WeavingProvider (3)	ElementReplaceProvider	-
AdviceTransformationProvider (4)	CreateAdviceMethodTransformationProvider*	-
	SubobjectReadTransformationProvider*	172
	BeforeSubobjectRead	14
	AfterSubobjectRead	19
	AroundSubobjectRead	45
	AfterReturningSubobjectRead	34
RuntimeExpressionProvider (5)	RuntimeTypeCheck	-
	RuntimeIfCheck	-
	RuntimeOr	-
	RuntimeAnd	-
	RuntimeNot	-
RuntimeParameterExposureProvider (6)	TypeParameterExposure	-
Coordinator (7)	ThreePhasedCoordinator*	-
	AdviceMethodCoordinator	-
Totaal		**464**

Tabel 9.10: Implementatie details voor een weaver voor het lezen van subobjecten. Klassen met een asterisk (*) zijn abstract.

Analyse Het implementeren van een nieuwe pointcut met bijhorende weaver voor JLo is in dezelfde orde van grootte van de implentaties van nieuwe weavers voor Java. Dit is voor een deel te wijten aan het feit dat JLo een uitbreiding is op Java en de uiteindelijke doeltaal ook Java is - de runtime controles, bijvoorbeeld, kunnen exact dezelfde methodes gebruiken. Dit doet echter geen afbreuk aan het resultaat: een weaver die met een heel geringe implementatie - zo'n 150 regels code - de *volledige* featureset van de weaver van Java bevat! Ook het uitbreiden is van dezelfde grootteorde als het uitbreiden van de Java weaver, waarbij dezelfde opmerking kan gemaakt worden omtrent de factories. Aangezien de *SubobjectReadTransformationProvider* klasse voornamelijk code bevat om het model te transformeren en nieuwe elementen aan te maken, zal een factory een winst opleveren in de orde van een 100 regels code.

9.3 Implementatie van de IDE plugins

Voor zowel de aspect taal voor Java als die voor JLo zijn ook plugins beschikbaar voor Eclipse, die toelaten om aspecten te definiëren. Deze IDEs zijn meer dan eenvoudige teksteditors. Er is ondersteuning voor hyperlinks, een outline van de code, keywords, progressie aanduidingen, etc. Een voorbeeld is te vinden in figuur 9.3 op pagina 87. Opnieuw is het zo dat het Chameleon raamwerk hier het ontwikkelen een heel stuk makkelijker maakt. Omdat er al een ruime basis aanwezig is voor het ontwikkelen van

Entiteit	# regels code
AspectJava (1)	34
Aspect.g (2)	250
AspectsJavaModelFactory (3)	51
AspectsBuilder (4)	69
IncrementalJavaTranslator (5)	48
Totaal	**452**

Entiteit	# regels code
AspectJLo (1)	15
AspectsJLo.g (2)	251
AspectsJavaModelFactory (3)	50
AspectsBuilder (4)	70
IncrementalJavaTranslator (5)	86
Totaal	**472**

Tabel 9.11: Implementatie details voor de plugins voor Java (links) en JLo (rechts)

dit soort plugins, is het extra werk dat geleverd moet worden eerder beperkt. Voor elke plugin moeten volgende elementen gedefinieerd zijn:

1. De taal van het model
2. De lexer/parser die de broncode van de aspecttaal kan omzetten naar een model. Aangezien de code voor de lexer/parser automatisch gegenereerd wordt door ANTLR, telt de effectieve code die geschreven moet worden in het ANTLR grammatica.
3. Het object dat een verzameling van files zal omzetten naar het model (met behulp van de lexer/parser)
4. De builder, die ervoor zorgt dat het model getransformeerd wordt en terug wordt uitgeschreven naar bronbestanden
5. De transformator, die het model transformeert alvorens het wordt uitgeschreven

In tabel 9.11 staan de implementatie details voor de Java en JLo plugin. Er wordt bij elk object verwezen naar een entiteit door het nummer in deze lijst te gebruiker. De (kleine) toename in code bij JLo is het gevolg van het feit dat deze nog de JLo transformator moet oproepen. Het is duidelijk dat het schrijven van de grammatica voor de parser die de broncode zal omzetten naar het model het meeste werk is. Natuurlijk is de grootte hiervan volledig afhankelijk van het aantal beschikbare pointcuts en andere elementen binnen de aspect taal.

9.4 Analyse van de taalafhankelijkheid

Het is niet mogelijk om, voor een tool als een aspect weaver, volledig taal onafhankelijk te werken, zelfal wordt er gewerkt op de abstracties die Chameleon aanbiedt. Dit is niet verwonderlijk aangezien er op een bepaald moment wel moeten taal specifieke zaken gebruikt moeten worden. Chameleon ondersteunt dit door toe te laten dat taalafhankelijke zaken in een aparte module worden gedefinieerd. Natuurlijk moet er wel de afweging gemaakt worden wat er binnen het raamwerk taalonafhankelijk geïmplementeerd wordt en wat in een dergelijke taalmodule geïmplementeerd wordt. Het biedt natuurlijk een voordeel om zoveel mogelijk zaken taalonafhankelijk te definiëren aangezien dit de herbruikbaarheid ten goede komt. Er wordt echter nergens in het raamwerk afgedwongen dat de aangeboden implementaties ook gebruikt moeten

worden. Taalmodules kunnen steeds ervoor kiezen om geen gebruik te maken van dit standaardgedrag, maar vermoedelijk blijft dit gedrag hetzelfde voor alle weavers.

Het eerste element waar naar gekeken wordt is de *WeaveTransformer*. Deze steunt volledig op de aangeboden interfaces van andere elementen - geen concrete implementaties - en kan dus taalonafhankelijk geïmplementeerd worden, met een uitzondering: de *chain of responsibility* die de weavers bijhoudt, is wel taalafhankelijk - verschillende talen zullen verschillende weavers aanbieden. Het enige wat een taalmodule moet doen om een *WeaveTransformer* aan te maken is dus aangeven welke *Weaver*s er gebruikt worden. De rest van het gedrag is taalonafhankelijk.

Het zoeken naar pointcuts werd al eerder beschreven. De pointcuts die zich onderaan de boom bevinden zijn deels taalafhankelijk. Een *typeHandler* joinpoint bijvoorbeeld, steunt enkel op een element binnen het taalonafhankelijk deel van het Chameleon raamwerk - niet het Java deel - en zou dus herbruikt kunnen worden bij bijvoorbeeld C#. Indien men echter een weaver zou implementeren voor use cases - waar dit concept niet aanwezig is - heeft het geen zin dit pointcut op te nemen. Een pointcut dat verwijst naar een eerder gedeclareerd pointcut, alsook conjunctie (*&&*), disjunctie (*||*) en negatie (*!*) pointcuts zijn wel volledig taal onafhankelijk.

Alle andere elementen zijn in principe taalafhankelijk. Dit natuurlijk niet weg dat er nog steeds een groot deel van de functionaliteit binnen deze elementen gedeeld zal worden door verschillende - zo niet alle - talen. Daarom definieert het raamwerk ook nog de volgende standaard implementaties:

- Voor zowel de *WeavingProvider* als de *AdviceTransformationProvider* zijn er standaard implementaties voorzien die runtime transformaties al dan niet ondersteunen. Indien deze wel ondersteund worden, zullen deze ook automatisch na het weaven of transformeren uitgevoerd worden. Dit zorgt ervoor dat een gebruiker van het raamwerk runtime transformaties wil uitvoeren, niet verantwoordelijk is om dit zelf te onthouden en op een correcte manier te implementeren. Hij kan zijn *WeavingProvider* een subklasse maken van *AbstractWeavingProviderSupportingRuntimeTransformation* en moet enkel de individuele methodes (voor het weven, het teruggeven van de correcte *Coordinator*, ...) een invulling geven.

- Voor *Coordinator*s is er een standaard implementatie die methodes voorziet om pointcut expressies om te zetten naar runtime controles, aangezien dit gaat om het declareren van lokale variabelen en het uitvoeren van een *if-then-else* test, concepten die in het algemene Chameleon metamodel aanwezig zijn.

- Voor zowel de *AdviceWeaveResultProvider* als de *AdviceTransformationProvider* wordt een standaard implemtatie aan het raamwerk toegevoegd dat geldt als een *no-operation*: er wordt simpelweg geen transformatie uitgevoerd bij de *AdviceTransformationProvider*, en het advice zelf wordt teruggegeven bij de *AdviceWeaveResultProvider*.

9.5 Evaluatie van Chameleon

9.5.1 Voordelen

Het Chameleon raamwerk biedt een uitstekende ondersteuning voor het ontwikkelen van tools zoals een aspect weaver. In eerste instantie zorgt het voor een abstracte laag

bovenop specifieke syntax van een programmeertaal. Zo zal het afhandelen van een exceptie in Java of in C# voorgesteld worden met hetzelfde element in Chameleon - een *CatchClause.* Op deze manier is het zoeken naar pointcuts gescheiden van de syntax, maar gebeurt dit zoeken aan de hand van de abstracties die Chameleon biedt. Bovendien moet er slechts eenmalig een syntax boom opgesteld worden van het programma, dus als dit al reeds gebeurd is - om bijvoorbeeld een refactoring tool te implementeren - hoeft hiervoor geen extra werk te worden gedaan.

Een tweede voorbeeld van hoe de abstracties van Chameleon het ontwikkelen makkelijker maken is bij genaamde pointcuts en de verwijzingen ernaar. Omdat dit soort verwijzingen veelvuldig aanwezig zijn in programmeertalen - denk maar aan methode oproepen - is er binnen Chameleon een eenvoudige manier voorzien om deze aan te maken en binnen het model te zoeken, via de *CrossReference* interface.

Chameleon biedt ook een plugin aan voor de Eclipse IDE, waarin duidelijk te zien is welke voordelen Chameleon biedt - zie ook figuur 9.3 op de rechter pagina:

Syntax highlighting van keywords Syntax highlighting is mogelijk door een enkele regel code toe te voegen aan de parser voor elk keyword.

Hyperlinks Hyperlinks, waarmee binnen een editor eenvoudig kan gesprongen worden naar een bepaald element, zijn mogelijk door gebruik te maken van *CrossReference*s binnen Chameleon.

Foutboodschappen Elk Chameleon element kan zowel fouten als waarschuwingen geven, wat gebruikt kan worden om het element te valideren. Hiervoor beschikt elk element over de methode *verifySelf* die een *VerificationResult* teruggeeft.

Ondersteuning voor een *outline* Een outline toont hoe een element opgebouwd is, bijvoorbeeld voor een klasse in Java toont het onder andere de velden en methodes die in die klasse gedefinieerd worden. Analoog toont een Aspect de pointcuts die in dat aspect gedeclareerd worden.

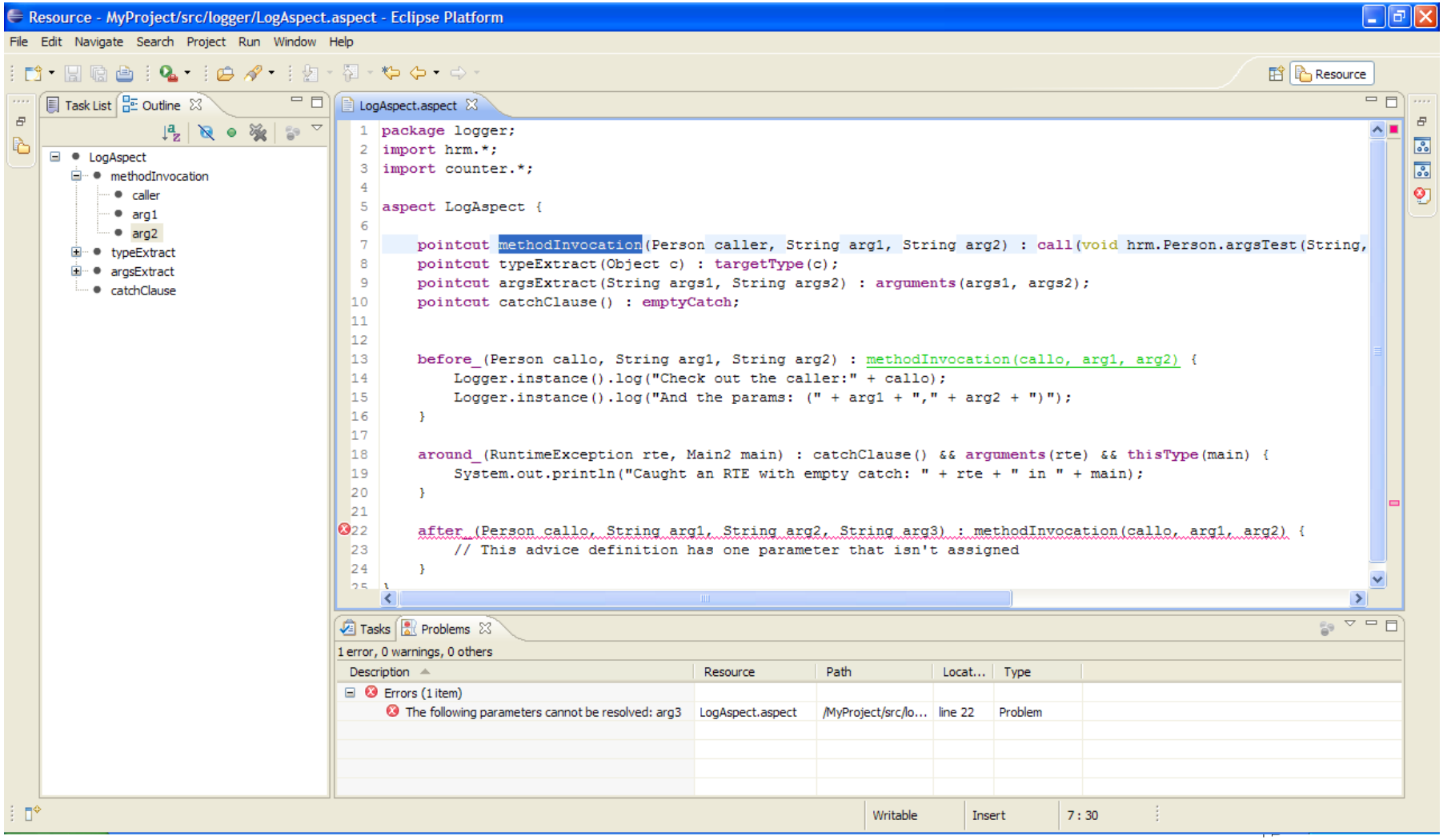
9.5.2 Nadelen

Net als voor alle raamwerken is er natuurlijk ook een leercurve voor het gebruik van Chameleon. Omdat het nog een *work in progress* is, is nog niet alles even goed gedocumenteerd. De opbouw is echter logisch en eens de basisconcepten duidelijk zijn is het makkelijk om het raamwerk zelf te gebruiken of uit te breiden. Bovendien zijn er ook unit tests beschikbaar om eventuele wijzigingen te testen.

Verder is het manueel aanmaken van (een deel van) een model nogal omslachtig. In dit geval was het bijvoorbeeld soms nodig om nieuwe methodes aan te maken. In vergelijking met de Java code die uiteindelijk uitgeschreven werd, omvat de code om het model op te stellen een veelvoud van het aantal regels code. Dit kan leiden tot grote methodes wat de onderhoudbaarheid niet ten goede komt. Om te illustreren, om de volgende regel Java code te genereren:

```
LogAspect.instance().<T>proceed(_$object, _$methodName, _$arguments, new
    Class[]{String.class, String.class});
```

is de volgende methode nodig:



FIGUUR 9.3: De editor met syntax highlighting, hyperlinks, foutboodschappen en de outline van het bestand

```
1  private RegularMethodInvocation createProceedInvocation(InvocationTarget
       aspectClassTarget, Expression objectTarget, Expression
     methodNameTarget, Expression argumentsTarget) {
2          RegularMethodInvocation getInstance = new
               RegularMethodInvocation("instance", aspectClassTarget);
3          RegularMethodInvocation proceedInvocation = new
               RegularMethodInvocation("proceed", getInstance);
4          proceedInvocation.addArgument((new BasicTypeArgument(new
               BasicTypeReference("T"))));
5
6          proceedInvocation.addArgument(objectTarget);
7          proceedInvocation.addArgument(methodNameTarget);
8          proceedInvocation.addArgument(argumentsTarget);
9
10         ArrayCreationExpression typesArray = new ArrayCreationExpression
               (new ArrayTypeReference(new BasicJavaTypeReference("Class"))
               );
11         ArrayInitializer typesInitializer = new ArrayInitializer();
12
13         try {
14                 for (FormalParameter fp : (List<FormalParameter>)
                       getJoinpoint().getJoinpoint().getElement().
                       formalParameters())
15                         typesInitializer.addInitializer(new ClassLiteral
                               (fp.getTypeReference().clone()));
16         } catch (LookupException e) {
17                 // This shouldn't occur in normal usage, only a bug can
                       cause this
18                 e.printStackTrace();
19         }
20
21         typesArray.setInitializer(typesInitializer);
22         proceedInvocation.addArgument(typesArray);
23
24         return proceedInvocation;
25 }
```

Dit kan vanuit het Chameleon raamwerk wel opgelost worden door een factory aan te bieden die bepaalde - veelgebruikte - standaardconstructies kan opbouwen. Voorbeelden hiervan zijn standaardmethodes (zoals standaard constructors) of *for* loops als **for** (**int** i = 0; i < n; i++).

9.5.3 Conclusie

Hoewel het omgaan met de abstracties die Chameleon biedt kan leiden tot een grote hoeveelheid code om eenvoudige elementen te beschrijven, staat het vast dat het gebruik van Chameleon binnen dit project zeer veel voordelen gaf. Het raamwerk voor aspect weavers profiteert maximaal van de abstractielaag die Chameleon biedt. Dit komt bijvoorbeeld zeer sterk naar voor bij de implementatie van de JLo weaver: de reeds ontwikkelde Java weaver kon, zonder aanpassingen, integraal herbruikt worden - zelfs om code te weaven die syntactisch gezien geen Java is. Bovendien kan ook de grote hoeveelheid benodigde code om een model op te bouwen - het grootste nadeel bij het gebruik van Chameleon - sterk gereduceerd worden door het aanbieden van factories voor veelgebruikte constructies, zoals een standaard *for* loop.

9.6 Evaluatie van het raamwerk voor aspect wevers

9.6.1 Voordelen

Zoals geïllustreerd zorgt het raamwerk voor aspect weavers voor een duidelijke houvast voor het implementeren van concrete aspect weavers binnen het Chameleon raamwerk. Door de nodige concepten vooraf te definiëren wordt het makkelijker om voor de gebruiker - diegene die een aspect wever ontwerpt - om tot een implementatie te komen. Bovendien zorgt de scheiding van verantwoordelijkheden ervoor dat code in veel gevallen herbruikt kan worden binnen een aspect weaver, zoals aangetoond in sectie 9.1.2. Verder was een doel bij het ontwerpen ook om herbruik te bekomen voor weavers in verschillende talen. Dit wordt duidelijk aangetoond door de JLo weaver, die verscheidene elementen uit de Java weaver integraal kan herbruiken.

9.6.2 Nadelen

Voor het ontwerpen van een aspect weaver binnen het raamwerk, moet er een parser zijn die de doeltaal omzet naar een model van door Chameleon aangeboden abstracties. Dit is zeker geen triviale taak voor geavanceerde talen - de ANTLR grammatica voor Java bestaat uit zo'n 2000 regels. Het is echter zo dat, om een weaver te laten werken, deze de joinpoints moet opzoeken en dus de taalspecificatie van de doeltaal zal moeten bevatten. Er moet dus ook een parser zijn die de joinpoints kan vinden, al kan deze minder geavanceerd zijn dan een volledige parser. Het is dus niet zo dat de noodzaak voor een parser een vereiste is die enkel in dit geval voorkomt. Dit is net een probleem dat Chameleon wil aanpakken - de taalspecificatie wordt losgekoppeld van de tool, waardoor deze slechts een enkele maal moet geïmplementeerd worden en er herbruik mogelijk is.

Deel III

Conclusie

Hoofdstuk 10

Gerelateerd werk

Deze masterproef is niet het eerste onderzoek naar een makkelijk uitbreidbare en herbruikbare manier om aspect wevers te ontwerpen. Toledo et al. ontwikkelden reeds een methode om AspectJ op een makkelijke en snelle manier te kunnen uitbreiden [13]. Ze maken hiervoor gebruik van drie technieken - SDF, Stratego en Reflex - om te komen tot een uitbreidbare AspectJ implementatie. De aanpak die in deze masterproef voorgesteld wordt bekomt een snelle en makkelijke uitbreidbare implementatie op de volgende manier:

- Joinpoint selectie aan de hand van de abstracties die Chameleon biedt
- Standaardimplementaties binnen het raamwerk
- Herbruik van de code voor een enkele weaver door duidelijke scheiding van verantwoordelijkheden
- Herbruik van code tussen verschillende weavers

In de aanpak van Toledo et al. wordt de extensibiliteit aangetoond door de implementatie van een nieuw type pointcut (*cast*, dat het typecasten van expressies selectereet). Ook binnen de wever voor Java die ontwikkeld werd kan dit pointcut gewoven worden, waarbij gebruik kan gemaakt worden van alle types advice en alle aanwezige pointcuts voor runtime controles of het blootstellen van parameters. De implementatie omvat een 300 tal regels code (zie tabel 9.7 op pagina 78) en er was ongeveer 1 mandag voor nodig - de wever binnen het ontworpen raamwerk is dus duidelijk snel uitbreidbaar. Bovendien stelt dit raamwerk nog een tweede doelstelling: uitbreidbaar zijn over verschillende talen heen door gebruik te maken van het Chameleon raamwerk. Aangezien de aanpak van Toledo et al. gebruik maakt van de Reflex AOP kernel is deze slechts bruikbaar voor Java.

Een tweede onderzoek combineert verschillende bestaande aspect georienteerd extensies [5]. Er wordt een compositie raamwerk opgesteld dat toelaat om deze verschillende extensies te combineren en te definieren in welke volgorde deze uitgevoerd moeten worden. De aanpak die in deze masterproef voorgesteld wordt laat ook toe om verschillende aspect extensies te combineren. Hiervoor moet elke extensie zijn features (zoals bijvoorbeeld specifieke pointcuts) definieren als elementen binnen Chameleon, en moeten de verschillende specifieke wevers aangemaakt worden. Door dan te definiëren

in welke volgorde er moet gewoven worden indien eenzelfde joinpoint door meerdere advices gewijzigd wordt - wat voorzien is binnen het framework - kunnen de *precendece rules* vastgelegd worden. Een nadeel is wel dat elke aspect georienteerde extensie opnieuw zal moeten geïmplementeerd worden binnen het raamwerk, terwijl de aanpak van Kojarski et al. toelaat om met bestaande extensies te werken. Een voordeel is wel dat het zeer eenvoudig wordt om code te delen tussen de verschillende extensies.

Hoofdstuk 11

Conclusie

In dit hoofdstuk wordt een conclusie gemaakt van het geleverde werk, waarbij het bekomen resultaat vergeleken wordt met de vooraf opgestelde doelen. Er wordt ook gekeken naar wat er nog verder zou kunnen gebeuren.

11.1 Conclusie

De vooropgestelde doelen waren:

1. Onderzoeken welke functionaliteit en abstracties nodig zijn voor het weven van aspecten
2. Een ontwerp en implementatie maken voor deze functionaliteit als die nog niet binnen het Chameleon raamwerk aanwezig is
3. Ontwikkelen van een concrete tool voor het weven van aspecten waardoor het raamwerk geëvalueerd kan worden

Al deze doelstellingen zijn gehaald. De functionaliteit en abstracties worden in het raamwerk geschetst. Daar wordt ook een standaard implementatie voorzien die compleet taalonafhankelijk is. Taal specifieke zaken worden in de taalmodules geïmplementeerd, maar het is natuurlijk mogelijk om ook daar code te delen door een *abstracte taalmodule* te creëeren voor bijvoorbeeld object georiënteerde talen.

Om aan te tonen dat de ontdekte functionaliteit en abstracties volstaan om een functionele wever te creëeren, werd in eerste instantie een effectieve weaver voor Java aangemaakt. Deze weaver is gebaseerd op AspectJ wat betreft functionaliteit en syntax, maar er zijn ook verschillende pointcuts die niet in AspectJ aanwezig zijn (*cast*, *emptyHandler*) om aan te tonen dat de weaver, dankzij het raamwerk, makkelijk uitbreidbaar is. Deze weaver is nog in opbouw, maar is al zeer functioneel en kan een groot aantal joinpoints weaven.

Vervolgens werd ook een weaver voor een uitbreiding van Java, namelijk JLo, aangemaakt. Deze toont de voordelen van ontwikkelen met behulp van het Chameleon raamwerk duidelijk aan: het is, zonder de weaver te moeten aanpassen, perfect mogelijk om elementen te weaven die geen geldige syntax hebben in Java, maar wel in JLo. Ook de mogelijkheden tot het herbruiken van elementen door verschillende weavers zijn hier duidelijk: de JLo weaver implementeert de volledig Java weaver én uitbreidingen

hierop met een minimale hoeveelheid extra code. Naast de concrete tools voor het weaven zelf, werd ook een ondersteunde plugin voor de Eclipse IDE ontwikkeld die toelaat om aspecten, advices en pointcuts te definiëren. Deze tool is meer dan een eenvoudige teksteditor en bevat verschillende hulpmiddelen voor de ontwikkelaar: syntax highlighting, hyperlinks, validatie, ...

De gemaakte analyse aan de hand van het aantal lijnen code toont duidelijk de voordelen van het ontworpen raamwerk voor aspect weavers, en het gebruik van het Chameleon raamwerk. Tussen de verschillende pointcuts is er veel herbruik mogelijk, waar gemiddeld slechts een 400 regels code nodig zijn om nieuwe pointcuts en hun weavers toe te voegen. Bovendien is het grootste deel hiervan de code om de modelelementen aan te maken of te transformeren. Dit kan dus ook nog significant verlaagd worden door in Chameleon factories te implementeren voor veelgebruikte constructies.

De geïmplementeerde weavers voor Java en JLo bewijzen dat het mogelijk is - omwille van het opgestelde raamwerk voor aspect weavers binnen het Chameleon raamwerk - om op een eenvoudige en uitbreidbare manier aspect weavers voor talen en taaluitbreidingen te definiëren.

11.2 Toekomstig werk

Aangezien deze masterproef beperkt is in tijd en scope, is het niet mogelijk geweest om alle facetten van een aspect weaver te bekijken. Er werd in de eerste plaats gefocussed op de advice/pointcut mechanismes. Om een volledige evaluatie te kunnen maken van het raamwerk moet de huidige weaver voor Java verder afgewerkt worden qua functionaliteit. AspectJ heeft nog enkele pointcuts die niet geïmplementeerd werden in de Java weaver omwille van tijdsgebrek. Om te staven dat de voorgestelde aanpak ook zou werken voor deze pointcuts en om te evalueren hoeveel werk dit zou kosten moeten deze ook geïmplementeerd worden. Verder moet er ook aandacht besteed worden aan andere mechanismen voor aspect geörienteerd programmeren, zoals het aanpassen van de structuur van de base code, bijvoorbeeld het introduceren van nieuwe methodes. Een eerste analyse gaf al aan dat dit geen triviaal probleem is, aangezien de opzoekmechanismes binnen Chameleon hier waarschijnlijk zouden moeten voor aangepast worden.

De twee weavers die ontworpen zijn, worden gebruikt voor twee relatief gelijkende talen. Het zou zeker interessant zijn om ook een weaver te ontwikkelen voor een taal die veel verder staat van Java, om zo de herbruikbaarheid optimaal te evalueren. Indien het over een andere object gerichte programmeertaal gaat, zoals C#, zal er vermoedelijk veel herbruik mogelijk zijn, aangezien de verschillen met Java dan eerder syntactisch zijn - wat door Chameleon weggeabstraheerd wordt. Bij een andere taal, bijvoorbeeld een logische taal als Prolog, zal dit vermoedelijk minder het geval zijn, aangezien het hier om meer conceptuele verschillen gaat. Aangezien Chameleon niet beperkt is tot programmeertalen, is het ook mogelijk een weaver te ontwerpen voor modelleertalen. Zo zijn use cases ook een mogelijk doel [12].

Bibliografie

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28, December 1996.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[4] Sergei Kojarski and David H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28th International Conference on Software Engineering*, pages 212–221, Shanghai, China, May 20-28 2006. ICSE 2006, IEEE Computer Society.

[5] Sergei Kojarski and David H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 515–534, New York, NY, USA, 2007. ACM.

[6] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen. Domain-driven discovery of stable abstractions for reusable pointcut interfaces. *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 75–86, 2009.

[7] Sean McDirmid and Martin Odersky. The scala plugin for eclipse. *Proc ECOOP Workshop on Eclipse Technology eXchange ETX*, 2006.

[8] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[9] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[10] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995.

[11] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[12] Jonathan Sillito, Christopher Dutchyn, Andrew David Eisenberg, and Kris De Volder. Use case level pointcuts. In *IN PROC. ECOOP 2004*, 2004.

[13] R. Toledo and E. Tanter. A lightweight and extensible aspectj implementation. 14(21):3517–3533, 2008.

[14] Marko van Dooren. Abstractions for improving, creating, and reusing object-oriented programming languages.

[15] Marko van Dooren and Eric Steegmans. A higher abstraction level using first-class inheritance relations. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 425–449. Springer, 2007.

Deel IV

Bijlagen

Bijlage A

Populariserend artikel

Universeel Programmeren met Aspecten

Jens De Temmerman

DistriNet, Dept. Computerwetenschappen
K.U.Leuven, België

31 December 2011

Het populaire besturingssysteem Windows blies onlangs 25 kaarsjes uit en het mag duidelijk zijn dat de informatica in deze tijd een immense evolutie heeft doorgemaakt. Een office-pakket van nu is niet meer te vergelijken met een teksteditor van toen op gebied van functionaliteit. Deze evolutie vindt niet alleen plaats op de PC thuis - een moderne smartphone biedt enorm veel functies. Deze groei is uiteen te halen in twee verschillende luiken: enerzijds de evolutie van de *hardware* als gevolg van de wetenschappelijke vooruitgang. De chips waaruit deze apparaten zijn opgebouwd zijn alsmaar kleiner, sneller én goedkoper te produceren. Anderszijds is er ook de *software.* Aangezien de toestellen krachtiger worden, kunnen ook de applicaties mee evolueren en veel meer functionaliteit bieden. De keerzijde van de medaille is dan dat deze applicaties ook veel complexer zijn. Hoe krijgen software ontwikkelaars vat op deze complexiteit?

1 Geschiedenis van software ontwikkeling

1.1 De eerste generatie: machinecode

Op het laagste niveau werkt een computer enkel met binaire getallen. Deze binaire getallen worden gegroepeerd in *instructies*, bijvoorbeeld per 32 voor een 32-bit processor. Elk programma is uiteindelijk niet meer dan een opeenvolging van instructies (ook *machinecode* genoemd) aangezien dit het enige is wat de processor kan begrijpen en dus ook kan uitvoeren. Machinecode is een programmeertaal van de eerste generatie. De code is zo geschreven dat die meteen kan gelezen worden door een machine.

```
0e 1f ba 0e 00 b4 09 cd 21
69 73 20 70 72 6f 67 72 61
74 20 62 65 20 72 75 6e 20
6d 6f 64 65 2e 0d 0d 0a 24
ec 85 5b a1 a8 e4 35 f2 a8
6b eb 3a f2 a9 e4 35 f2 6b
6b eb 68 f2 bb e4 35 f2 a8
6b eb 6b f2 a9 e4 35 f2 6b
6b eb 6f f2 a9 e4 35 f2 52
```

Figuur 1: Machinecode

De keerzijde hiervan is dat het als mens erg moeilijk is om te begrijpen wat een bepaalde instructie doet, laat staan een opeenvolging van instructies. Een mogelijke instructie is bijvoorbeeld: 1011000001100001. Dit wordt door de processor geïnterpreteerd als: Plaats (10110) het getal 97 (of 01100001 in binair) in het geheugen op locatie 0 (000).

Machinecode is echter specifiek voor een bepaalde processor. Eenzelfde instructie kan een verschillend effect hebben indien die uitgevoerd wordt op twee verschillende types processoren. Een programmeur zou dus, per type processor waar zijn applicatie moet op werken, een ander programma moeten hebben. In figuur 1 staat een voorbeeld van machinecode voor een tekstverwerkingsprogramma. De binaire getallen zijn per acht gegroepeerd in een hexadecimaal getal.

1.2 De tweede generatie: assembleertaal

Het mag duidelijk zijn dat dit geen praktische manier van werken is om een complex programma te creëeren. Om het voor de ontwikkelaars wat makkelijker te maken werd *assembleertaal* ontwikkeld, een programmeertaal van de tweede generatie. In assembleertaal worden nog steeds instructies geschreven, maar men kan gebruik maken van symbolische namen om dit te doen. Hierdoor wordt het voor de ontwikkelaar makkelijker om instructies te schrijven en te begrijpen. De instructie 1011000001100001 zou in assembleertaal bijvoorbeeld als volgt geschreven kunnen worden: MOV 97,0. Ofwel: "MOVe getal 97 naar het geheugen, op locatie 0." Dit is voor een mens al heel wat leesbaarder. Bovendien kunnen instructies in assembleertaal eenvoudigweg omgezet worden naar instructies in machinecode - en omgekeerd - aangezien er een duidelijke mapping is. Voor dit voorbeeld is 10110 ↔MOV. Dit wil echter zeggen dat een programma in assembleertaal enkel geschikt is voor een bepaald type processor. Figuur 2 toont een deel van een tekstverwerkingsprogramma in assembleertaal.

```
cmp edi, esi
je 01001D66
inc edi
inc edi

mov eax, edi
pop edi
pop esi
pop ebp
ret 0004
```

Figuur 2: Assembleertaal

1.3 De derde generatie: mens-vriendelijk programmeren

Assembleertaal is duidelijk ook nog geen ideaal werkmiddel voor software ontwikkelaars. De programmeertalen van de derde generatie zijn een beweging geweest om het programmeren meer mens-vriendelijk te maken. Hierbij worden programma's op een hoger niveau beschreven. Deze code wordt dan door

een ander programma, een *compiler*, omgezet naar assembleertaal of machinecode. Het is dan ook de compiler die alle details gaat afhandelen die voor een programmeur niet van belang zijn. Stel bijvoorbeeld dat de programmeur een eenvoudige som van drie getallen wil berekenen: $x = a + b + c$. Aangezien een computer slechts twee getallen tegelijk kan optellen en niet drie, worden eerst $a + b$ uitgerekend. Vervolgens wordt dit resultaat opgeteld bij c. Moest een programmeur dit in assembleertaal schrijven, zou hij expliciet alle stappen moeten uitschrijven en ook details vermelden die eigenlijk niet van belang zijn, zoals waar in het geheugen het tussenresultaat moet opgeslagen worden. Het achterwege kunnen laten van dit soort zaken zorgt dus voor programma's die overzichtelijker zijn en sneller ontwikkeld worden. Nog een voordeel is dat code geschreven in deze programmeertalen platform onafhankelijk is. Het enige wat de programmeur nodig heeft om zijn code te laten draaien op een bepaald type processor is de juiste compiler. De meest gebruikte programmeertalen tegenwoordig, zoals C++ en Java, vallen onder deze categorie. Figuur 3 toont een klein voorbeeld van C++ code.

```
void AutoComplete::Show(bool show)
{
    lb->Show(show);
    if (show)
        lb->Select(0);
}
```

Figuur 3: C++ code

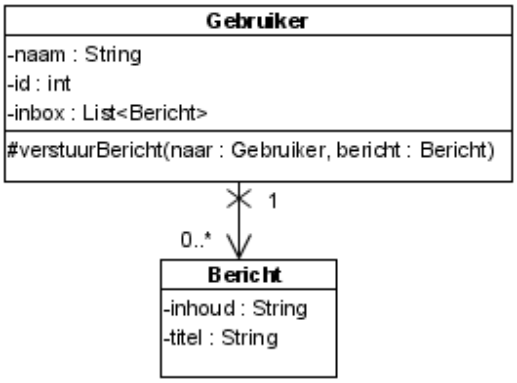
Programmeertalen van de derde generatie zijn dus bedoeld om het de ontwikkelaars makkelijker te maken door details achterwege te laten. Dit principe wordt binnen de computerwetenschappen heel vaak toegepast en wordt *abstractie* genoemd. Zo maakt een besturingssysteem abstractie van de inhoud van de harde schijf: waar de enen en nullen die een bepaald bestand vormen staan is van geen belang voor een gebruiker. Dit detail wordt door het besturingssysteem verborgen en wat de gebruiker ziet zijn bestanden met een naam. Abstractie alleen is echter niet genoeg. Zelfs als men gebruik maakt van derde generatie programmeertalen kunnen programma's groeien tot miljoenen lijnen code. Als men al deze code gewoon lijn per lijn in een bestand neerschrijft zal dit wel werken - een computer heeft hier geen problemen mee. Maar voor de ontwikkelaars die deze code moeten aanpassen en uitbreiden is dit onbegonnen werk. Er is een grote nood aan structuur.

2 Object Gericht Programmeren: Orde in de Chaos?

Object gericht programmeren is momenteel het paradigma dat de beste oplossing biedt voor deze nood aan structuur. Een programma bestaat dan uit *objecten*, elk met hun eigen gegevens, operaties en verantwoordelijkheden. Wat het programma dan uiteindelijk doet wordt beschreven door de interacties tussen de objecten. Neem bijvoorbeeld een eenvoudig systeem om berichten uit te

wisselen. Gebruikers kunnen een contactpersoon selecteren, een bericht typen en dit versturen. Ze kunnen ook berichten lezen die ze van andere gebruikers hebben ontvangen. Elke concrete gebruiker binnen dit systeem is dus een apart object met bepaalde eigenschappen (een naam, een identificatienummer, ...) en bepaalde operaties, bijvoorbeeld een operatie om een bericht te versturen naar deze gebruiker. Ook een bericht is een object, met als eigenschappen bijvoorbeeld de titel en de inhoud. Figuur 4 toont de structuur van dergelijke objecten.

Object gericht programmeren is met reden zeer populair binnen de softwareontwikkeling. Het probleemdomein van een programma is zeer vaak te beschrijven in aparte concepten en hun interacties. Een complex programma is daarom nog niet in een enkele oogopslag te vatten, maar het gestructureerd werken biedt een stevige houvast aan de programmeur.

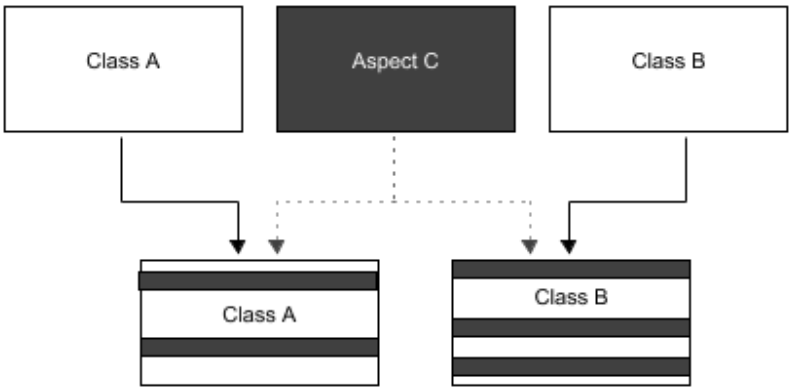


Figuur 4: Het berichtensysteem in object gericht programmeren

Maar ook object gericht programmeren is geen aanpak die op alle problemen een antwoord kan geven. Het is namelijk nog steeds zo dat sommige aspecten van het programma verweven zijn met andere aspecten. Dan komen dezelfde problemen terug boven die object gericht programmeren net probeerde op te lossen. Bepaalde delen code worden op verschillende plaatsen gedupliceerd. Als dat stuk code aangepast moet worden, moet de programmeur dit overal doen. Het is zo goed als onvermijdbaar dat dit op een bepaald moment fout loopt - een programmeur past het op plaats A wel aan, maar vergeet dit op plaats B. Het gevolg is dus dat het programmeren van de applicatie complexer wordt en niet omdat de applicatie zélf complexe vereisten stelt, maar omdat de methodiek tekort schiet.

3 Aspectgeörienteerd programmeren

Als oplossing voor deze problemen wordt *aspectgeörienteerd programmeren (AOP)* naar voor geschoven. Dit is geen nieuw programmeerparadigma, zoals object geörienteerd programmeren dit was, maar vormt een aanvulling op de bestaande paradigma's. Binnen AOP gaat men een oplossing bieden voor de zogenaamde *crosscutting concerns*, handelingen die op verschillende plaatsen terug komen. Hiervoor worden drie concepten aangeboden. De code die het crosscutting concern implementeert wordt *advice* genoemd. Dit is dus de code die zonder AOP op verschillende plaatsen in het programma gedupliceerd zou worden maar hier dus eenmalig wordt geschreven. Deze code moet natuurlijk nog op de correcte plaats(en) ingevoegd worden. Een *joinpoint* is een plaats in het programma waar



Figuur 5: Het weefproces

een *advice* kan ingevoegd worden. Om te weten op welke plaatsen dit moet gebeuren en welke niet, moet er een selectie worden gemaakt van verschillende *joinpoints*. Zo'n selectie noemt men een *pointcut.*

Gewapend met deze concepten heeft men nu een oplossing voor het probleem van de crosscutting concerns. Vooraleer de code omgezet wordt naar machinecode wordt alle *advice* op de gewenste plaatsen in de code ingevoegd. Dit proces heeft de naam *weaven* meegekregen omwille van de overeenkomst met het weven van stof. Het resultaat is dus dat de gedeelde code slechts op een plaats beschreven moet worden en deze door een automatisch proces op de juiste plaats ingevoegd wordt. Figuur 5 illustreert dit proces. Het is dus makkelijker voor een programmeur om deze code te gaan onderhouden.

4 Chameleon: een model voor programmeertalen

Net als veel schrijvers tegenwoordig niet meer op een typemachine tikken maar gebruik maken van een tekstverwerker, met de nodige extra functionaliteit, maken ook programmeurs gebruik van speciale software om hun werk uit te voeren. Het gaat hier bijvoorbeeld om een speciaal soort tekstverwerkers, ontworpen om code in te schrijven, die de programmeur helpen door deze extra informatie te geven en op mogelijke fouten te wijzen. Ook zo'n aspect weavers vallen hieronder - het zijn enkel hulpmiddelen voor de ontwikkelaar. Mensen die het uiteindelijke programma gebruiken zullen dit niet merken.

Zoals gezegd is objectgericht programmeren momenteel het meest gebruikte paradigma binnen software ontwikkeling. Maar dit wil niet zeggen dat er slechts een enkele programmeertaal gebruikt wordt. Integendeel, er zijn tientallen verschillende programmeertalen die gelijkaardige concepten hanteren maar op bepaalde essentiële punten verschillen - en dus ook allemaal hun bestaansrecht

hebben. Er wordt echter zelden voordeel gehaald uit de gelijkenis tussen verschillende talen.

En hier ligt nu net het probleem. Deze hulpmiddelen, die de programmeur helpen vat te krijgen op de complexiteit, zijn heel vaak slechts geschikt voor een enkele programmeertaal. Ondanks het feit dat de programeertalen Java en C# zeer veel gelijkenissen tonen, kan een aspect *weaver* voor Java geen C# code gaan weven of omgekeerd. Als men dus voor beide talen een weaver wil, zal men er twee moeten maken.

Chameleon wil hier nu verandering in brengen door de gelijkenissen in programmeertalen te gaan modelleren. Ook hier wordt weer een vorm van abstractie toegepast - de details van specifieke talen worden niet bekeken maar er wordt geprobeerd om alles zo algemeen mogelijk te beschrijven. De bedoeling is om applicaties - zoals een aspect weaver - dan op te gaan bouwen in twee delen. Een algemeen deel, beschreven volgens het Chameleon model, en een taal-specifiek deel. Indien men dan bijvoorbeeld een weaver wil uitbreiden om ook een andere programmeertaal te ondersteunen, moet men slechts een nieuw taal-specifiek deel gaan ontwerpen - wat veel minder werk is dan een volledige weaver.

Programmeurs worden dus indirect geholpen door Chameleon. Het zorgt ervoor dat het makkelijker wordt om applicaties te gaan ontwikkelen die programmeurs helpen. Het is niet de bedoeling om een enkele, universele programmeertaal te ontwikkelen. Veel programmeertalen ontstaan omdat de makers een bepaalde niche proberen in te vullen en hebben ook hun bestaansrecht. Chameleon wil de vicieuze cirkel - waarbij geen hulpmiddelen worden gemaakt omdat de taal niet populair genoeg is en de taal niet aan populariteit wint omdat er niet genoeg hulpmiddelen voor zijn - gaan doorbreken.

5 Conclusie

De evolutie van de informatica heeft de complexiteit van programma's sterk verhoogd. Zelfs een relatief eenvoudige applicatie als een tekstverwerker bevat duizenden regels code en werken verschillende mensen tegelijk aan. Het is dan ook zaak om het voor de programmeur zo makkelijk mogelijk te maken om zijn code te beheren. Hoe complexer de applicatie, hoe meer hulp de programmeur kan gebruiken bij het ontwikkelen om grip te krijgen op deze complexiteit. Momenteel zijn er al veel hulpmiddelen beschikbaar, maar deze zijn heel vaak enkel geschikt voor een enkele programmeertalen. Chameleon wil hier verandering in brengen door het makkelijker te maken om tools, zoals aspect weavers, te gebruiken in meer programmeertalen. Zo wordt verhinderd dat software ontwikkelaars verdrinken in hun eigen code - een noodzakelijke voorwaarde om de huidige evolutie verder te kunnen zetten.

Bijlage B

Wetenschappelijk artikel

Design of a framework for aspect weavers using the Chameleon framework

Jens De Temmerman

***Abstract*—The Aspect Oriented Programming paradigm increases modularization of crosscutting concerns. However, the aspect weaver depends on the language specification, just like other tools for programmers such as editors or refactoring tools. The Chameleon framework aims at obtaining better reuse for different tools for different (programming) languages by providing a framework for meta models. This paper outlines a framework that allows the fast and extensible development of aspect weavers, using the Chameleon framework. To demonstrate the effectiveness of the designed framework in providing reuse, aspect weavers for Java and JLo were implemented.**

***Index Terms*—Software engineering, aspect oriented programming**

I. Introduction

In the currently most popular programming paradigm, object oriented programming, there is a large focus on modularization and proper separation of concerns and responsibilities. However, not all concerns can be neatly modularized. Some concerns are said to be crosscutting. There are two distinct - although they often appear together - signs of crosscutting concerns in source code. The code that handles these concerns is said to be *tangled* if it is intertwined with code that handles other concerns. It is said to be *scattered* if it is spread across several modules.

Aspect Oriented Programming[2], [4] (AOP) aims at solving the problem of tangled or scattered code by providing a single module where crosscutting concerns are represented - an *aspect*. Different methods are used to accomplish this, for instance in AspectJ[3], [8], a popular AOP extension for Java, the available features are advices, pointcuts and inter-type declarations. Currently, we only consider advices and pointcuts in scope of this project.

AOP has gained in popularity the last decade. There also has been research into the reusability and extensibility of current aspect languages. These approaches however, are often limited to either an existing aspect language or a fixed target language. The approach presented here aims to provide a framework that promotes a more general form of reuse.

This framework will be built on top of the Chameleon framework, discussed in section II. Chameleon aims at providing better reuse when developing tools - such as refactoring tools, integrated development environments and aspect weavers by providing a framework for meta models on which these tools can operate. This way, there is reuse of code between different tools for the same (programming) language, but also between the same tool for different programming languages.

By studying the features of AspectJ and determining how such a weaver could be implemented, we identify the different elements required to design and implement an aspect weaver. We then create abstractions in the Chameleon model representing these features. In the next step, we tie the abstractions together and define a default implementation. The results are outlined in section III.

Finally, the framework is evaluated by using it to develop an actual weaver for two programming languages: Java and JLo. This weaver is then evaluated with regard to extensibility and reusability. These findings are presented in section IV.

II. Chameleon

For modern programming languages, the software engineer has an ever increasing amount of tools available to help design and implement applications. Advanced integrated development environments couple many of these tools: editors, compilers or interpreters, debuggers, re-factoring tools, aspect weavers, etc. However, these tools also introduces two coupled, but distinct problems. The first problem is that, for a certain language, the semantics are repeated in each tool, increasing the development time for each tool as well as the potential amount of bugs. The second problem is that new languages have a difficult time finding adoption as all these productivity enhancing tools are mostly language specific. The Scala plugin for Eclipse[7] is an example of this.

Chameleon aims to solve these problems by providing a framework for meta models of programming languages [12]. Programming tools will then mostly work on this model instead of the actual language specification. Language specific details are handled in smaller language models.

The scope of this project is to design and implement a framework that can be used to build aspect weavers, using the Chameleon framework.

III. A framework for aspect weavers

To design and evaluate a framework for aspect weavers, a simple yet functional aspect weaver for Java was designed, based on AspectJ but implemented in the Chameleon framework. Note that the weaver was only based on AspectJ regarding functionality, not the weaving process. Although we could use how AspectJ weaves as a starting point, we chose not to since the weaver works on bytecode. Since all weavers implemented on the framework will work on Chameleon models, we describe how weaving would happen if source code weaving would be used.

A. Discovery of the required elements

1) Joinpoint transformation: The first pointcut we take a look at is the *call* pointcut. In order to weave this correctly

in source code, we can not just insert our advice statements before, after or around the method call - for instance with this trivial example: **if** (**false** && objectA.methodCall()). The advice code should not be run because the method call will never happen as it is short-circuited[1]. If we would insert our advice code at the joinpoint itself, we would have to take into account where the joinpoint is located - which is something we clearly want to avoid for complexity reasons. A approach that does work is to simply replace the method invocation with a new invocation to a method representing the execution of the advice and the original method. This execution could be done with reflection, for example. Just rewriting the original invocation in the advice method may not work in all cases - for instance if the adviced method is *private* - because the advice method is in a different type. Reflection allows to circumvent the access modifier.

Requirement 1. A mechanism to transform the advice.

Requirement 2. A mechanism to transform the joinpoint.

Requirement 3. A mechanism to determine with what element the joinpoint is combined.

The second pointcut we take a look at is the *handler* pointcut. To correctly weave this pointcut, we can insert the advice code at the joinpoint itself. In fact, it is not possible to replace the entire block of the handler with a call to a advice method, as done for *call* pointcuts, since this block may contain, for example, references to local variables or return statements. This requires no new elements as mechanism 2 can be used for this. For the *execution, get, set* and *within* pointcuts, no other mechanisms are required as they can all be handled the same way as a *call* or *handler* pointcut.

However, it is clear that different joinpoints require different solutions for the first three requirements. This implies that there must be some object that determines which solution will be used for each (type of) joinpoint.

Requirement 4. An element that determines which elements are responsible for transforming the advice, determining what element is combined with the joinpoint and how this is supposed to be done.

2) Runtime transformations: The next pointcuts we take a look at are those that perform transformations to the advice code to incorporate runtime checks, the exposure of parameters, or both. To obtain a proper separation of responsibilities, we separate the insertion of runtime checks and parameter assignments even if they are done by the same pointcut.

Runtime checks depend on the way the weaving is performed. Not only because they have to be implemented differently - for *handler* pointcuts, they have to be inserted at the joinpoint and execute the original block - or the next advice for that joinpoint- if the check fails, for *call* pointcuts they are implemented in the advice method and have to call the original method - or the next advice - if the check fails. To incorporate this we have to decouple the type of pointcut expression and how the expression is inserted.

Requirement 5. A mechanism to determine how a certain runtime check must be performed.

The same applies to parameter assignments. It depends on the way weaving is performed: exposing *this* when weaving the advice in place is simply done by assigning the Java expression *this* to the parameter, however when a separate advice method is used, the object representing *this* must be passed as a parameter to this method and exposed through its name as a parameter to the advice.

Requirement 6. A mechanism to determine how a certain parameter assignment must be performed.

The object responsible for inserting the expressions must also order these. For example, an *if* pointcut may use pointcut parameters in its boolean expression. Therefore, these parameters must be type checked, then exposed and only then may expression performing the *if* check be inserted.

Requirement 7. An arbitration mechanism that orders the expressions and inserts them correctly.

Also note that these runtime checks and parameter assigments can be inserted at two places: a created advice method, as will have to be done for method invocations, or at the joinpoint itself, as will have to be done for catch clauses. In order to obtain a proper separation of concerns, the elements that allow runtime transformations must provide a uniform interface.

Requirement 8. Objects that can perform runtime transformations must have a uniform interface regarding the requirements for these transformations.

The application of runtime transformations has another important implication. Because these can only be performed after weaving a certain joinpoint, the pointcut that selected that joinpoint must be available, since it is that pointcut that contains the runtime checks and parameter assignments that have to be performed.

Requirement 9. When searching for joinpoints for a given advice, both the joinpoint and the pointcut that selected it must be stored. The joinpoint for obvious reasons - it is needed to perform the actual weaving. The pointcut can contain runtime checks or parameter assignments.

3) The weaving process: we need an element that coordinates the weaving. While this element performs no transformations itself, this coordinating process is what would be considered as the weaver. As a last requirement, it must be possible for several advices to apply to the same joinpoint. We need a way to delay weaving until all applicable advices have been found, order these advices and then weave them in the correct order, making sure the correct semantics are applied. Therefore, we weave in two passes: first we iterate over every advice and, for each joinpoint that is advised by it, we encapsulate all required mechanisms and information. In the second pass, for each joinpoint that is advised at least

[1]In Java, the && operator is a short-circuit operator: if in the expression "a && b" a is false, b will never be evaluated since the expression will be false regardless of b.

once, the advices are ordered and the weaving is executed.

Requirement 10. A coordinating process that will connect the individual subprocesses.

Requirement 11. The weaving must be done in a two-phase process. In phase one all advised joinpoints are determined. In phase two, the advices are ordered according to certain precedence rules and the weaving is executed.

Requirement 12. There must be a mechanism to order advices that are applied to the same joinpoint, according to a certain set of precedence rules.

B. Using the discovered elements to create abstractions

In this step, we create abstractions of the defined elements and mechanisms. These abstractions will form the basis of the framework.

The weaving process starts in a *WeaveTranformer* (requirement 10, figure 1). For each compilation unit that has to be woven and for each advice that has been defined, all joinpoints selected by the pointcut defined by the advice are looked up (figure 2). The result is stored as a *MatchResult*, containing both the joinpoint and the pointcut that selected it (requirement 9, figure 3). These results are then passed to the chain of *Weaver*s defined in the *WeaveTransformer* - implemented as a chain of responsibility[1]. Each *Weaver* (requirement 4, figure 4) defines which combination of joinpoint and advice type it supports, and if a given *MatchResult* is supported by the *Weaver*, it builds the three required elements for weaving (figure 5): the *AdviceTransformationProvider* (requirement 1), the *WeaveResultProvider* (requirement 3) and the *WeavingProvider* (requirement 2). Note that the *Weaver* does not perform any actual weaving - it simply returns a *WeavingEncapsulator*, which contains all the info required for weaving (figure 6). It is only after sorting - using a standard Java *Comparator* (requirement 12)- all *WeavingEncapsulator*s that act on the same joinpoint, that the actual weaving is performed (requirement 11).

Each object that wants to perform transformations to add runtime checks and parameter assignments, must implement the *RuntimeTransformationProvider* interface (requirement 8, figure 7). This interface defines the necessary methods to perform the required transformations (figure 8): the used *Coordinator* (requirement 7), the object that will return the correct runtime check expression for the given pointcut (requirement 5) and the object that will return the correct parameter assignment expression (requirement 6).

IV. Evaluation

A. Implementation of a Java weaver

Using the defined concepts, we implemented a functional weaver for Java to demonstrate the effectiveness of the framework. In the first step, we defined a syntax for our aspect language. We chose a syntax very similar to the AspectJ syntax since this is certainly sufficient for our purposes. Since the weaver only works on Chameleon models, we needed a parser that constructs the Chameleon model from the aspect

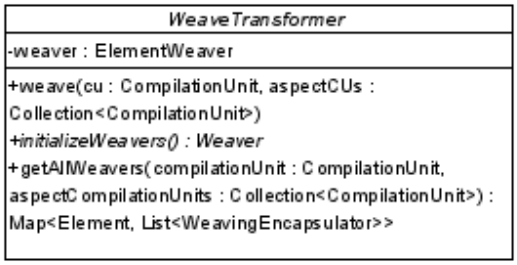


Figure 1. The *WeaveTransformer*

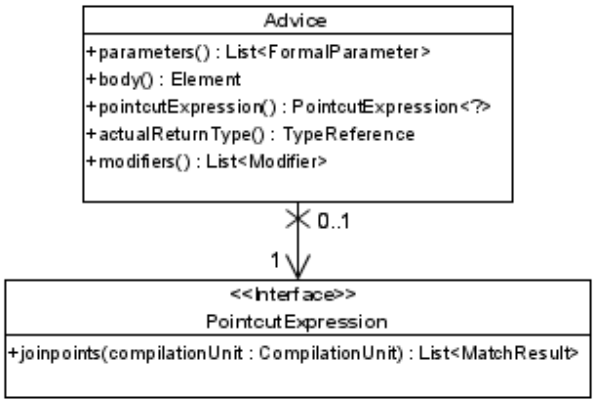


Figure 2. The *Advice* and *PointcutExpression* interfaces

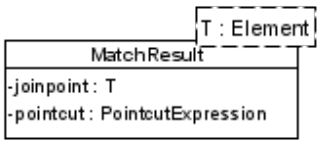


Figure 3. The *MatchResult*

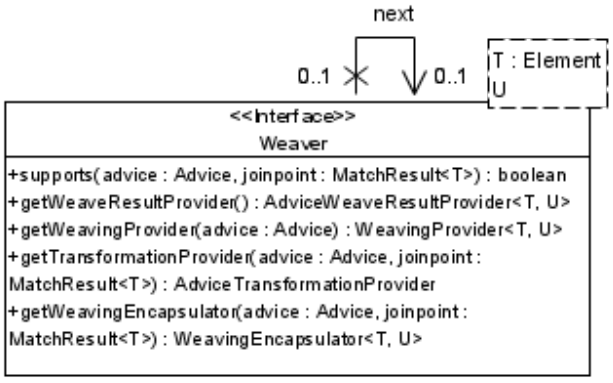


Figure 4. The *Weaver*

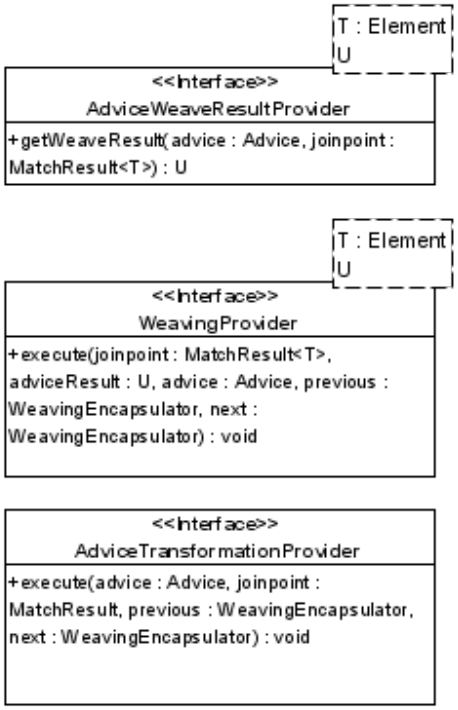


Figure 5. The *AdviceTransformationProvider, WeaveResultProvider* and *WeavingProvider* interfaces

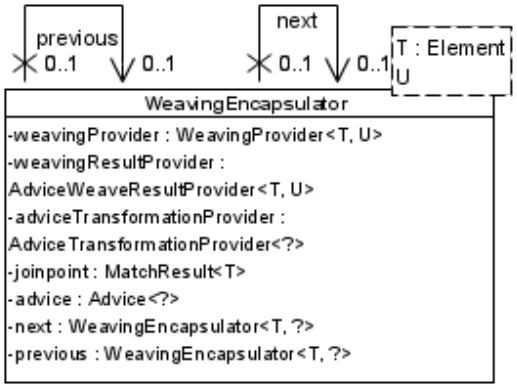


Figure 6. The *WeavingEncapsulator*

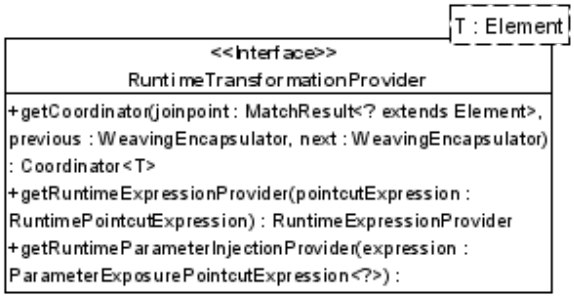


Figure 7. The *RuntimeTransformationProvider*

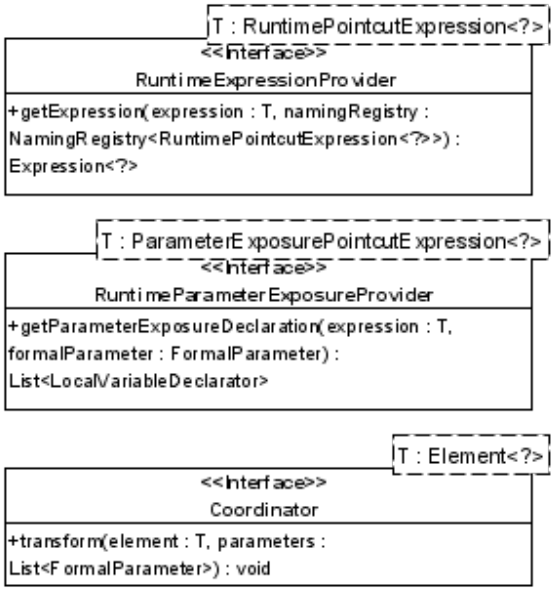


Figure 8. The *Coordinator, RuntimeExpressionProvider* and *RuntimeParameterExposureProvider* interfaces

source code. This was done using ANTLR[9]. Note that the parser that transforms the base source code (Java in this case) is separate from the parser that transforms the aspect code, so reuse here is possible as well. Of course, language specific parts of the model - e.g. concrete pointcuts such as *call* - must be implemented as well. Again, there is a lot of potential for reuse here, as the concept of a method call is present in many languages although the signature may differ. Table I shows how many lines of code[2] it took to implement certain pointcuts and their weaver. This table has to be read chronologically - the first weaver, for method invocations according to signature, was a lot more work in comparison since almost every other weaver can reuse elements defined by it. The common elements for all pointcuts include the *WeaveTransformer* for Java and several *RuntimeExpressionProviders* and *RuntimeParameterExposureProviders*, which are common for all defined weavers for Java.

Note that a lot of the code is used to transform and build models in Chameleon. At the moment this is cumbersome, a lot of code is required to construct simple models. For example in one case, it took 25 lines of code to construct a relatively simple method invocation - one line of output code. This could be improved by implementing factories for constructs that are often used (method invocations, for loops, ...).

[2]Excluding whitespace and comments but including all definitions (classes, methods) and import statements. Counted using CLOC, version 1.53. http://cloc.sourceforge.net/

Type of poincut	Lines of code
All pointcuts listed in table I	30
Subobject reads	464

Table II
COMPARISON OF THE REQUIRED LINES OF CODE FOR EACH TYPE OF POINTCUT (JLO WEAVER)

Type of poincut	Lines of code
Common elements for all pointcuts	411
Method invocations according to signature	1462
Method invocations according to annotations	78
Field reads	482
Exception handeling according to type	333
Exception handling (empty catch block)	30
Typecasts	329
Within a method or type	133

Table I
COMPARISON OF THE REQUIRED LINES OF CODE FOR EACH TYPE OF POINTCUT (JAVA WEAVER)

B. Implementation of a JLo weaver

JLo is a Java extension for implementing subobjects[13]. Since every Java expression is also a valid JLo expression, we expect to be able to reuse the defined pointcuts and weaver for Java for Java expressions. This is indeed the case. But the weaver can also handle JLo expressions that are not valid Java expressions. An example of this is the following method invocation: radio.volume.getValue(). The target of the method invocation is in this case not a field, but a subobject. This implies that radio.volume is not a valid Java expression. The fact that our defined Java weaver can still support this expression *without any need for modification* is due to the use of the Chameleon framework, which abstracts syntactical differences away. Table II shows the results: implementing *every single* pointcut present in the Java weaver required 30 lines of code in total. Implementing a pointcut specific to JLo - in this case, reading a subobject - and its weaver, required about 450 lines of code.

V. FUTURE WORK

The framework that has been implemented supports a basic but functional weaver. While we analyzed the needs of a more functional weaver such as AspectJ and designed the framework so that it is easily extensible, the designed Java and JLo weavers should be extended to provide more features. That way, the weaving framework can be evaluated further.

Furthermore, we should also implement a weaver for other programming languages to see how much reuse the framework provides for different languages as this was also an important concern when designing and implementing the framework.

VI. RELATED WORK

There have been other approaches to creating a extensible aspect weaver. The approach by Toledo et. al [11] uses the syntax of AspectJ and the Reflex AOP kernel to facilitate the extension of AspectJ, especially the addition of new pointcuts. In our method, pointcuts that are already handled by an existing weaver can be implemented easily due to the fact that we operate on the Chameleon meta model. To illustrate this, we implemented a *callAnnotated* pointcut that picks out calls to methods that have a certain annotation. This can be woven by the same weaver as the regular *call* pointcut, the only required changes were to the parser and the classes representing the new pointcut - about 90 lines of code in total. For pointcuts that require a different strategy to weave, the amount of work that has to be done is dependent on how much of the other strategies can be reused. As mentioned, field read pointcuts required about 400 lines of code.

Another approach, proposed by Kojarski et al. [5], is to combine different aspect oriented language extensions. It constructs a composition framework by plugging together independently developed aspect mechanisms. Our method allows this kind of composition, since each different aspect language extension would have its features (such as pointcuts) represented as a Chameleon model. Furthermore, each extension would simply have its own weaver(s). By defining precedence rules for these weavers and combining the Chameleon model, it is possible to compose these different extensions. Of course, this would require that current aspect language extensions are reimplemented to use the framework we developed, while the approach proposed by Kojarski et al. can combine existing extensions.

Furthermore, these approaches only deal with extensibility of existing aspect weavers for a single language. Our approach also provides possibilities for reuse across programming languages, as this is also one of the advantages of the Chameleon framework.

VII. CONCLUSION

This work discusses the implementation of a framework for aspect weavers, using the Chameleon framework. We discussed how the framework was constructed and evaluated it by implementing a weaver for Java. Our approach focuses on reuse, both for a weaver for a single language and for weavers across languages. A lot of this is already accomplished by using the Chameleon framework. A disadvantage of our approach is that it requires a reimplementation of existing aspect extensions to fit the framework. However, for languages where such a weaver is not yet present, or for non - programming languages, such as use cases - which are representable as Chameleon models and for which an aspect-oriented approach has been advocated and used [6], [10] - the framework provides a way for rapid development of a functional aspect weaver. This was partly proven by the JLo example, where it took only 30 lines of code to implement a very functional weaver.

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28, December 1996.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[4] Sergei Kojarski and David H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28th International Conference on Software Engineering*, pages 212–221, Shanghai, China, May 20-28 2006. ICSE 2006, IEEE Computer Society.

[5] Sergei Kojarski and David H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 515–534, New York, NY, USA, 2007. ACM.

[6] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen. Domain-driven discovery of stable abstractions for reusable pointcut interfaces. *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 75–86, 2009.

[7] Sean McDirmid and Martin Odersky. The scala plugin for eclipse. *Proc ECOOP Workshop on Eclipse Technology eXchange ETX*, 2006.

[8] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[9] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator, 1995.

[10] Jonathan Sillito, Christopher Dutchyn, Andrew David Eisenberg, and Kris De Volder. Use case level pointcuts. In *IN PROC. ECOOP 2004*, 2004.

[11] R. Toledo and E. Tanter. A lightweight and extensible aspectj implementation. 14(21):3517–3533, 2008.

[12] Marko van Dooren. Abstractions for improving, creating, and reusing object-oriented programming languages.

[13] Marko van Dooren and Eric Steegmans. A higher abstraction level using first-class inheritance relations. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 425–449. Springer, 2007.

Fiche masterproef

Student: Jens De Temmerman

Titel: Ontwerp en implementatie van een raamwerk voor aspect weavers binnen het Chameleon raamwerk

Engelse titel: Design and implementation of a framework for aspect weavers using the Chameleon framework

UDC: 681.3

Korte inhoud:

Deze masterproef beoogt het ontwerp en de implementatie van een raamwerk voor aspect weavers binnen het Chameleon raamwerk. Er werd onderzocht welke functionaliteit en abstracties nodig zijn om een aspect wever te implementeren. Aan de hand hiervan werd een raamwerk opgesteld binnen het Chameleon raamwerk, dat een taalonafhankelijke houvast biedt voor het implementeren van aspect weavers. Om het raamwerk te evalueren werd eerst een concrete aspect weaver voor Java geïmplementeerd die qua syntax en functionaliteit gebaseerd is op AspectJ. Er wordt gekeken hoeveel herbruik mogelijk gemaakt wordt door het raamwerk en de mate van uitbreidbaarheid wordt getoetst door bepaalde - in AspectJ niet aanwezige - pointcuts te implementeren. Verder werd ook een Eclipse plugin ontwikkeld die, gebruik makende van de in Chameleon aanwezige mogelijkheden, een functionele editor biedt om aspecten te definiëren voor de Java weaver. Uit de evaluatie van deze weaver blijkt dat aan de vooropgestelde doelen - het identificeren van de nodige functionaliteit en abstracties en het maken van een herbruikbare en uitbreidbare implementatie, voldaan is. De ontwikkelde weaver voor Java is zeer functioneel en uitbreidingen zijn eenvoudig te realiseren. Vervolgens werd een tweede weaver geïmplementeerd, voor een uitbreiding van Java: JLo. Het doel hierbij was om te toetsen of de herbruikbaarheid ook stand houdt voor weavers voor verschillende programmeertalen. Ook hier wordt een positief evaluatie gemaakt. Niet alleen is de ontwikkelde weaver voor Java volledig herbruikbaar om ook JLo-code te weaven, bovendien is het uitbreiden van deze weaver met JLo-specifieke elementen eenvoudig en is er ook hierbij veel herbruik mogelijk.

Thesis voorgedragen tot het behalen van de graad van Master in de ingenieurswetenschappen: computerwetenschappen

Promotor: Prof. dr. ir. E. Steegmans

Assessoren: Prof. dr. D. Clarke
Dr. ir. M. van Dooren

Begeleider: Dr. ir. M. van Dooren