Naast het object-oriented programming (OOP) paradigma kent de software engineeringswereld sinds enkele jaren ook het aspect-oriented programming (AOP) paradigma. Het begon ooit als 'hype', maar ondertussen is AOP volwassen geworden en de bijhorende frameworks en tools zijn inmiddels 'proven technology'.

thema

Aspect-Oriented Programming met AspectJ

Krachtige modulariseringstechnieken

Het centrale doel van AOP is het verminderen van de complexiteit van de applicatie. Dit wordt toegepast op gebieden waar de middelen en technieken van objectoriëntatie ontoereikend zijn. Het is dus een aanvulling, geen vervanging van de objectoriëntatie. Dit hoofdstuk beschrijft het centrale idee achter de aspectoriëntatie, zodat men in staat is de AOP-concepten zoals *Separation of Concerns, cross-cutting* en *joinpoints* te begrijpen en in staat is te beoordelen waar en hoe aspecten zinvol en efficiënt kunnen worden ingezet.

Aspect-oriented programming werd ooit ontworpen om bepaalde soorten requirements of features (of in AOP taal: 'concerns') in modules te kunnen implementeren, die met gewone OOP niet of alleen moeilijk te modulariseren zijn.

Separation of Concerns Het modulariseren van code, of zoals het oorspronkelijk door Dijkstra in zijn essay 'On the role of scientific thought' (1974) werd geformuleerd, de *Separation of Concerns*, is een erg belangrijke programmeertechniek om de complexiteit van applicaties zo klein mogelijk te maken: Programmeercode die één bepaald *concern* realiseert, moet ook op één plek als 'eenheid' (of module) in de codebase worden geïmplementeerd. Zo wordt de applicatie gemakkelijker te onderhouden en kunnen concerns gemakkelijk worden toegevoegd, aangepast of verwijderd. In de OOP zijn de kleinste modules klassen en methoden die tot grotere modulen (componenten) of packages kunnen worden samengevat. De resulterende modulestructuur is de klassenhiërarchie.

Code Tangling en Code Scattering Niet alle concerns kunnen gemakkelijk één op één naar functies of klassen worden vertaald. Een bekend voorbeeld is de afhandeling van logging. Hierbij gaat het niet zozeer om debug-logging statements, die programmeurs bijna willekeurig in de code toevoegen, maar meer om de functionele eisen aan de loggingafhandeling van een applicatie, zoals bijvoorbeeld de eis dat 'de doorlooptijd van iedere aanroep van een I/O methode moet worden gelogd'. Op zich is dit een simpele eis, maar met alleen procedurele middelen leidt dit tot code, waarbij de methode zowel de business logica als de code voor het loggen bevat.

```
public String readXml(String fn) throws
IOException {
     // logging logica deel 1
     long startTijd = new Date().getTime();

     // businesslogica
     BufferedReader bfr = new
BufferedReader(new FileReader(fn));
     String out = bfr.readLine();

     // logging logica deel 2
     long doorlooptijd = new Date().getTime()
- startTijd;
```

```
    logger.debug("duur van methode in ms:" +
doorlooptijd);

    return out;
}
```

In een module (in dit geval de methode) worden twee concerns tegelijk gerealiseerd. De methode kan niet zomaar in andere contexten worden hergebruikt, omdat niet alleen de business logica maar tegelijk ook de logging logica wordt overgenomen. Dit fenomeen wordt dan *code tangling* genoemd.

Andersom is de code die de logging-feature realiseert over meerdere functies verdeeld: Wil men de manier van loggen wijzigen, dan moeten er op vele punten in de codebase aanpassingen worden doorgevoerd (*code scattering*). Het is duidelijk dat *code tangling* en *code scattering* het *separation of concerns* principe weerspreken. In de AOP worden deze soort concerns *cross-cutting concerns* genoemd, omdat deze dwars door de traditionele modules heen snijden.

Oplossingen in de OOP-wereld Natuurlijk is dit probleem in de OOP-wereld niet nieuw en is geprobeerd met OOP-middelen voor degelijke gevallen de *separation of concerns* te realiseren. Het volgende codevoorbeeld laat zien hoe met het *decorator pattern* de beide concerns 'inlezen van een bestand' en 'loggen van de doorlooptijd' over twee klassen kunnen worden verdeeld, zodat de *separation of concerns* gewaarborgd blijft.

```
public interface XmlReader {
    public String readXml(String fn) throws
IOException;
}
```

Het interface XmlReader wordt door de klasse FileXmlReader geïmplementeerd, die verantwoordelijk is voor het realiseren van de business-logica.

```
public class FileXmlReader implements
XmlReader {

    public String readXml(String fn) throws
IOException {
        BufferedReader bfr = new
BufferedReader(new FileReader(fn));
        String out = bfr.readLine();

        return out;
    }
}
```

De logging-logica wordt uiteindelijk gerealiseerd door de klasse LoggingXmlReader, die het eigenlijke inlezen aan de XmlReader delegeert en deze dus zogenoemd *decoreert*.

```
public class LoggingXmlReader implements
XmlReader {

    private XmlReader reader;
    private Logger logger = Logger.getLogger(
LoggingXmlReader.class);

    public LoggingXmlReader(XmlReader reader)
{
        this.reader = reader;
    }

    public String readXml(String fn) throws
IOException {
        long startTijd = new Date().
getTime();

        String out = reader.readXml(fn);

        long doorlooptijd = new Date().
getTime() - startTijd;
        logger.debug("duur van methode in
ms:" + doorlooptijd);

        return out;
    }
}
```

De twee concerns zijn nu gescheiden in twee aparte klassen, die uiteindelijk door de client-code op de volgende manier worden samengevoegd.

```
XmlReader reader = new LoggingXmlReader(new
FileXmlReader());
String out = reader.readXml("foo");
```

De *separation of concerns* is gewaarborgd, de logging logica zit in een aparte klasse, maar de oplossing lijkt nogal omslachtig: Om de ene methode uit te kunnen breiden met extra gedrag moet er een interface en een aparte klasse komen. Wanneer het dus gaat om een heel beperkte set van operaties, dan is het *decorator* pattern erg geschikt. Maar voor cross-cutting concerns, die in principe op een groot aantal methodes kunnen worden toegepast, leidt deze oplossing tot een explosie van klassen en interfaces.

Voor essentiële en vaak voorkomende cross-cutting concerns, zoals transaction management of session management, maar ook toegangsbewaking, zijn zogenoemde *runtime containers* geïntroduceerd. In de Java-wereld is het bekendste voorbeeld natuurlijk de EJB-

container. Bij toepassing van bijvoorbeeld *container managed transaction* bevat de eigenlijke programmeercode geen enkele verwijzing naar de transactie. Deze is buiten de businesslogica-code in een `deployment descriptor` vastgelegd, die de container laat weten waar en hoe de feature *transaction* op de code moet worden toegepast.

Maar de EJB-specificatie is niet makkelijk uit te breiden door een programmeur –er kunnen niet zomaar nieuwe types deployment descriptors in de EJB container worden gehangen. Zoals het bovenstaande voorbeeld laat zien, leidt ook het gebruik van *design patterns* vaak tot complexere code. Om het probleem van het modulariseren van cross-cutting concerns op een generieke manier op te lossen, zetten AOP-frameworks bovenop de bekende OOP-modules zoals klassen en functies een nieuw type module: het zogenoemde *aspect*.

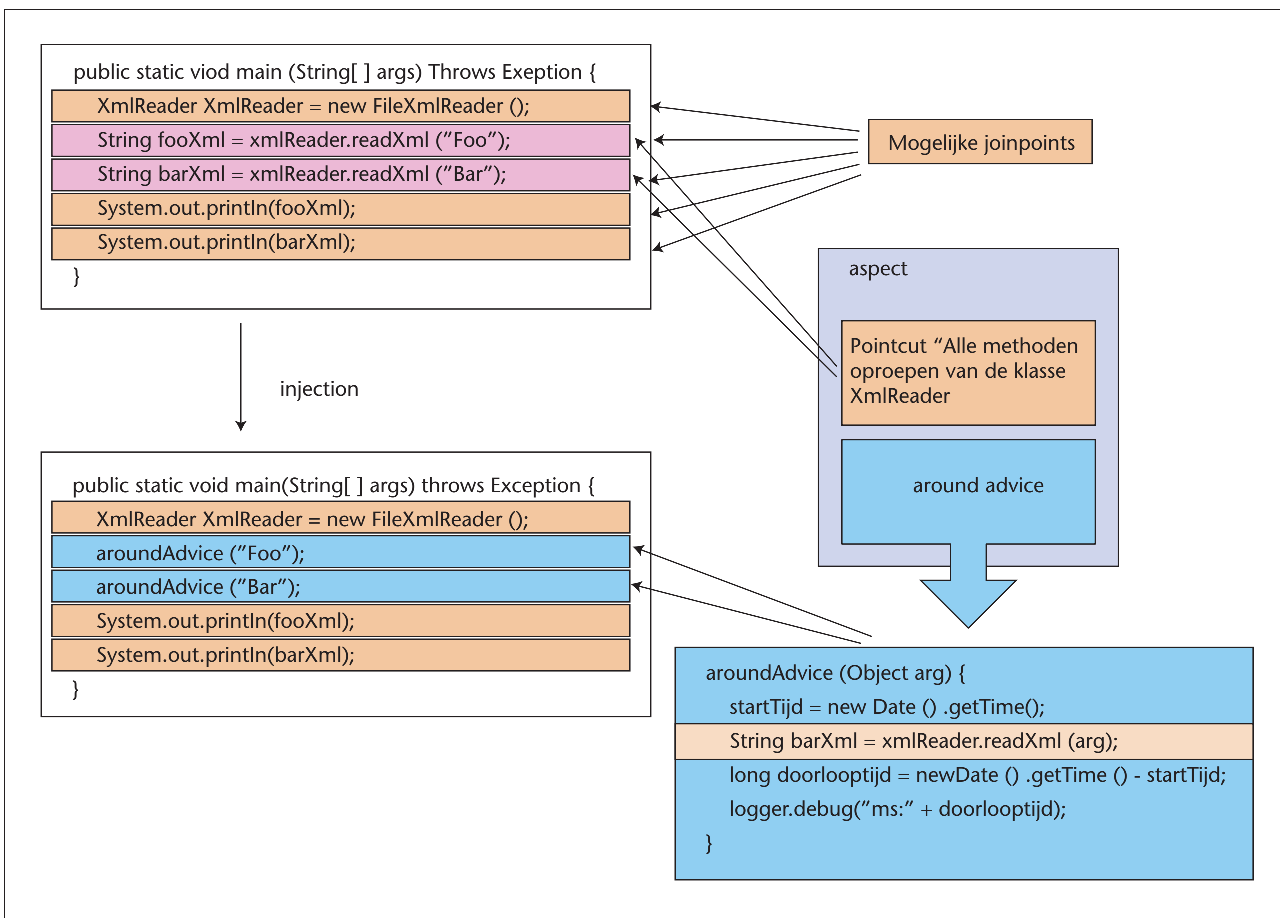
Anatomie van Aspecten Een aspect bestaat in essentie uit drie onderdelen:

- *Advice*: het eigenlijke stukje code dat het cross-cutting concern implementeert, bijvoorbeeld voor het loggen.
- *Joinpoints*: de mogelijke punten in de code waar een advice kan worden ingevoegd.
- *Pointcut*: een definitie die alle punten (*joinpoints*) in de code eruit pikt, waarop het *advice* zal worden toegepast.

Een AOP-framework zorgt ervoor dat het stukje code dat in het advice staat, op basis van de pointcut op de juiste plekken in de codebase wordt uitgevoerd. Omdat dit niet alleen op runtime, maar afhankelijk van het gekozen AOP-framework zelfs op compile time kan gebeuren en de *advice* code dus letterlijk in de bestaande klassen en methoden van de applicatie wordt ingevoegd, spreekt men hierbij in AOP-termen over het *injecteren* of *weaving* van het *advice*.

Natuurlijk is het niet zinvol om op iedere plek in de code het injecteren van een aspect *advice* toe te staan. In het simpelste geval zijn dit bijvoorbeeld alle punten in de code waar methoden worden aangeroepen. Maar er zijn ook AOP-frameworks waar bijvoorbeeld de foutafhandeling (de *catch(){}* clause in Java) een joinpoint kan zijn.

Tenslotte moet in het aspect nog worden vastgelegd of het advice vóór (*before advice*), na (*after advice*) of in plaats van (*around advice*) het joinpoint moet worden uitgevoerd. In het geval van het around advice wordt de joinpoint code compleet vervangen door de advice-code, terwijl de advice-code zelf dan



Figuur 1. Een overzicht van concepten van een aspect.

Adv

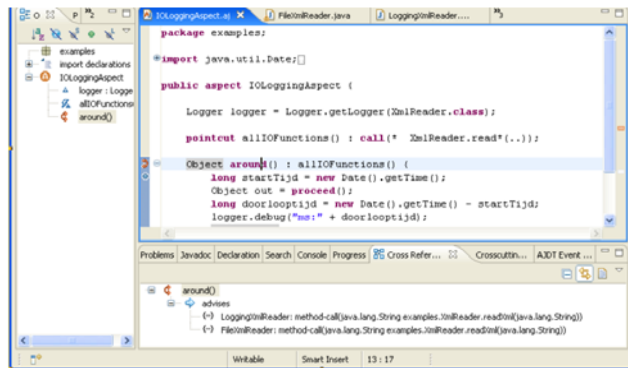
weer een aanroep naar de oorspronkelijke joinpoint-code bevat. Zodoende kan er dus inderdaad code voor en na het joinpoint worden geïnjecteerd. Figuur 1 laat nog eens alle genoemde concepten van een aspect zien.

Uit de mogelijke joinpoints (in blauw) selecteert een pointcut alleen die joinpoints waar een methode van de klasse XmlReader wordt aangeroepen (paars). Hieraan is een zogenoemd *around advice* gekoppeld met een simpel log-statement. Het advice wordt via het AOP-framework in de business-code geïnjecteerd (in groen).

AspectJ Er is een groot aantal AOP-frameworks op de markt, maar het AspectJ framework staat algemeen bekend als het meest krachtige om met cross-cutting structuren om te gaan. In AspectJ zijn aspecten gewoon Java-klassen, maar in plaats van het keyword *class* wordt het keyword *aspect* gebruikt. Hierdoor kan een aspect naast de AspectJ-specifieke pointcut en advice-definitie gewoon Java-code bevatten. Dus ook property's en methoden, zodat de aspectcode zelf naar bekende de OOP design-principes kan worden gestructureerd. De AspectJ keywords worden natuurlijk door de standaard Java-compiler niet begrepen, en daarom wordt er gebruik gemaakt van een aparte compiler 'ajc', die de aspect sources in valide Java bytecode compileert. Het volgende code-fragment laat zien hoe de *cross cutting concern* 'logging logica' voor iedere read methode van de klasse XmlReader als aspect in AspectJ kan worden geïmplementeerd:

```
public aspect IOLoggingAspect {

    private Logger logger = Logger.
getLogger(XmlReader.class);

    pointcut allIOFunctions() : call(*
XmlReader.read*(..));

    Object around() : allIOFunctions() {
        long startTijd = new Date().
getTime();

        Object out = proceed();

        long doorlooptijd = new Date().
getTime() - startTijd;
        logger.debug("ms:" + doorlooptijd);

        return out;
    }
}
```

Het keyword *pointcut* definieert de reeks joinpoints als soort reguliere expressie. In dit voorbeeld matcht de



Figuur 2. De AJDT plugin ondersteunt de ontwikkelaar bij het ontwikkelen van code met betrekking tot aspecten.

pointcut alle aanroepen (*call*) van alle functies van de klasse XmlReader die met 'read' beginnen, ongeacht de parameters. In het *around* advice wordt uiteindelijk het concern 'loggen van de doorlooptijd' geïmplementeerd, waarbij door middel van het *proceed*() statement de gekozen joinpoint (dus de XmlReader.readXml() methode) transparant wordt aangeroepen.

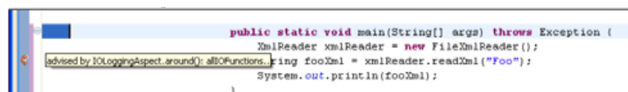
De *separation of concern* principe is gewaarborgd: 'inlezen van xml-bestanden' en 'loggen van de doorlooptijd van het inlezen' staan in twee aparte modules, de XmlReader.readXml() en het aspect. De clientcode zelf bevat geen enkele hint meer dat naast de business logica ook nog het logging-concern wordt gerealiseerd.

```
XmlReader reader = new FileXmlReader();
String out = reader.readXml("foo");
```

Het samenvoegen van de twee concerns vindt dus niet plaats in de programmatuur maar wordt door de aspect-compiler gerealiseerd, die het *advice* op de door de *pointcut* gedefinieerde punten in de Java-programmatuur injecteert. De transparantheid is dus totaal, aan de Java-code is niet meer te zien welke andere concerns hier nog gerealiseerd worden.

Dit is echter ook vaak een punt van zorg voor veel programmeurs. Bij de OOP-oplossing via het *decorator* pattern was tenminste nog aan de client-code te zien dat de FileXmlReader-klasse gewrapped was door de LoggingXmlReader-klasse en er dus blijkbaar logging plaatsvindt. Maar pas door deze transparantie krijgt de applicatie de noodzakelijke flexibiliteit die cross-cutting concerns eigen is.

Stel dat het logging-concern zou worden uitgebreid op bijvoorbeeld 'alle publieke methodes'. Misschien lijkt dit een onzinnige regel maar er zijn snel vergelijk-



Figuur 3. Door dubbelklikken kan de ontwikkelaar direct naar de juiste plek in de code springen.

bare zinnige concerns op te stellen (bijvoorbeeld 'alle excepties moeten worden gelogd'). Bij een op het *decorator* pattern gebaseerde oplossing moeten voor alle klassen met publieke methoden aparte *decorator* klassen worden aangemaakt, interfaces gedefinieerd en bij iedere klasse instantiëring de klasse in de juiste decorator worden *gewrapped*. Dit lijkt bijna niet realiseerbaar.

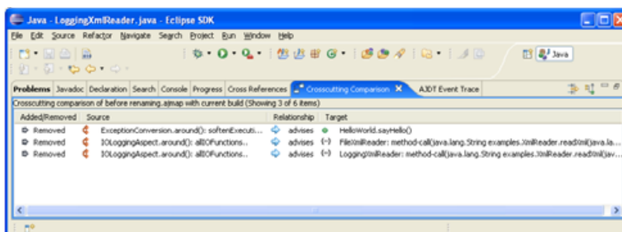
In AspectJ-daarentegen kan het aspect zonder problemen worden uitgebreid door gewoon de expressie van de pointcut aan te passen, zodat alle publieke methodes worden gematcht. De complexiteit verandert niet.

```
pointcut allPublicFunctions() : call(public *
*.*(..))
```

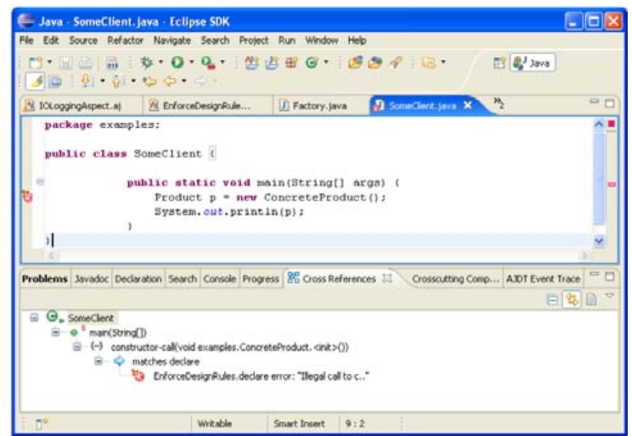
AJDT Zoals net beschreven is de totale transparantheid van aspecten zoals deze door AspectJ wordt bereikt de grootste kracht van het framework, maar ook een groot probleem van de ontwikkelaar. Hij kan aan de code niet zien welke *advices* hierop worden toegepast, de cross-cutting structuur wordt in de code niet weer gespiegeld. Om deze structuur zichtbaar te maken moeten speciale tools worden gebruikt. Voor de Eclipse IDE is het AspectJ Development Tools plugin (AJDT) hiervoor het meest gebruikelijke tool. Figuur 2 laat zien hoe de AJDT plugin de ontwikkelaar bij het ontwikkelen van code met betrekking tot aspecten ondersteunt.

Aan de marker op de linkerkant van het around() *advice* is te zien dat er inderdaad code is waarop de advice wordt toegepast. De crossreference view laat zien welke code, namelijk de twee oproepen naar XmlReader.readXml() functies. Door dubbelklikken op deze regels kan de ontwikkelaar direct naar de juiste plek in de code springen, bijvoorbeeld naar de onderstaande functie. Ook hier is door een marker op de linkerzijkant te zien op welke regels *advices* zijn toegepast. Door een rollover window kan de ontwikkelaar snel herkennen welk *advice* dit is.

Problematisch zijn met de huidige versie van het AJDT-plugin nog refactoringsslagen, omdat de expressies die de pointcut definiëren door het refactoringsmodule van Eclipse niet worden ontdekt. Door het hernoe-



FIGUUR 4. De *cross-cutting map* is een soort snapshot van de cross-cutting structuur, die de gebruiker voor en naar de refactoring kan opslaan.



FIGUUR 5. De AJDT-plugin herkent een fout.

men van de methode readXml() naar bijvoorbeeld inlezenXml() matcht de pointcut expressie *call(* XmlReader.read*(..))* niet meer. Hulp hierbij levert de AJDT-plugin in vorm van een *cross-cutting* map, een soort snapshot van de cross-cutting structuur, die de gebruiker voor en naar de refactoring kan opslaan. Vervolgens kunnen deze twee maps met elkaar worden vergeleken en kan de oorspronkelijke structuur weer worden hersteld. Rest nog te melden dat ook het debuggen van code die verweven is met aspecten, geen probleem is met AJDT.

CROSS-CUTTING CONCERNS Sommige concerns zijn snel als cross-cutting concerns te herkennen. Hierbij gaat het onder andere om *tracing, logging, transaction management, exception handling, security, monitoring en caching*. De implementatie van degelijke evidente concerns gebeurt min of meer analoog aan het logging voorbeeld: Bepaal de pointcut die het set van methoden definieert waarop bijvoorbeeld transaction management moet worden toegepast. Definieer een around() advice, die de transactie start voordat de eigenlijke business logica wordt aangeroepen (before) en die de transactie commit zodra de aanroep retourneert (after).

Het valt natuurlijk buiten de grenzen van dit artikel om op al deze concerns in te gaan, maar er zijn bepaalde concerns waarvan het niet evident is dat aspecten ook hiervoor een elegante oplossing kunnen bieden. Eén daarvan zal afsluitend nog worden toegelicht om de veeltaal mogelijkheden die met AspectJ mogelijk zijn te demonstreren.

BEWAKEN VAN DESIGN INVARIANTEN In de OOP-wereld is het met gebruikelijke middelen meestal niet gemakkelijk om bepaalde design-invarianten te bewaken, bijvoorbeeld dat nieuwe objecten van het type Product alleen door de klasse Factory mogen worden aangemaakt. Met behulp van de volgende aspect kan dit op een simpele manier worden bewaakt:

```
public aspect EnforceDesignRules {

    pointcut allProductConstructorCalls() :
call(Product+.new(..));

    pointcut withinFactoryCalls() :
withincode(static * Factory.*(..));

    declare error :
        allProductConstructorCalls()
        && !withinFactoryCalls() : "Illegal
call to constructor";
}
```

Er worden hier twee pointcuts gedefinieerd: De pointcut allProductConstructorCalls() wordt toegepast op alle aanroepen van de constructor van een klasse van het type Product. Het plusteken in de pointcut achter Product geeft aan dat het dus ook om interface implementaties of subklassen van de type Product kan gaan.

De tweede pointcut definieert alle *joinpoints* binnen methoden van de klasse Factory. De declare error aanwijzing laat de AspectJ compiler uiteindelijk een error gooien als een joinpoint wordt geraakt waar wel een constructor van het type Product wordt aangeroepen, maar deze aanroep niet binnen een methode van de klasse Factory gebeurt. Zoals in Figuur 5 te zien is, wordt dit ook door de AJDT-plugin als fout herkend. Op deze manier laten zich gemakkelijk complexe design-invarianten bewaken, zoals bijvoorbeeld afhankelijkheden tussen software-componenten en architectuur lagen.

Conclusie AOP en met name AspectJ zijn krachtige modulariseringstechnieken die ook voor al bestaande applicaties snelle oplossingen kunnen bieden. Vaak zal de verleiding groot zijn om in plaats van een slecht design te herstellen, achteraf even snel een aspect toe te passen. Maar AOP mag niet als quickfix worden gezien. Juist omdat het zo krachtig is, moet er goed over worden nagedacht welke concerns inderdaad met een AOP-oplossing veel gemakkelijker kunnen worden gerealiseerd en bij welke concerns AOP alleen maar een noodoplossing blijkt te zijn. Daarom hoort AOP een integraal onderdeel van een modern software-design te zijn.

Martin Weidner is als software architect werkzaam bij Ordina J-Technologies.

InterSystems brengt CACHÉ 2007 uit

InterSystems Corporation heeft een nieuwe versie van CACHÉ uitgebracht. Daaraan zijn innovatieve technologieën en hulpmiddelen toegevoegd, die het ontwikkelen van webapplicaties met 40 procent kunnen versnellen. In de vernieuwde postrelationele database CACHÉ heeft InterSystems Zen geïntegreerd, een raamwerk voor het bouwen van platformonafhankelijke webapplicaties met uitgebreide functies. Sneller dan ooit kunnen deze applicaties nu operationeel gemaakt worden. Een andere nieuwe component, Jalapeño genaamd, ontlast Java-programmeurs van object-relationeel mappen; dit verkort de ontwikkeltijd aanzienlijk.

InterSystems is leverancier van database- en integratiesoftware: CACHÉ, een postrelationele database, Ensemble, software voor snelle integratie en HealthShare, een uitwisselingsplatform voor medische informatie.

De nieuw toegevoegde component aan CACHÉ, Zen, is een uitbreidbaar raamwerk, dat geoptimaliseerd is voor maximale ontwikkelsnelheid, schaalbaarheid en installeren van complexe webapplicaties. Zen bevat een uitgebreide bibliotheek van pasklare componenten, zoals grids, tabellen en keuzebomen. Ingewikkelde processen laten zich hiermee snel samenstellen en beheren. Daardoor zijn ontwikkelaars in staat gemakkelijk webapplicaties te bouwen die zich hetzelfde gedragen en dezelfde (gebruikersinterface-)functionaliteit bieden als geavanceerde desktop-programmatuur.

Met Zen beschikt CACHÉ 2007 over een uitbreiding van de AJAX-technologie voor het snel ontwikkelen van internetapplicaties. Het verfijnde beveiligingsmodel van CACHÉ is in de Zen-componenten opgenomen. Bovendien is de objecttechnologie van Zen verantwoordelijk voor uitbreidingsmogelijkheden van de meegeleverde componenten. Daarnaast is het mogelijk om nieuwe, zelf-ontwikkelde code te integreren in oplossingen die een nog hogere graad van maatwerk vereisen. Zen maakt bovendien gebruik van een shared object client/server-datamodel, dat de moeizame ontleding van XML overbodig maakt. Als de applicatie eenmaal is geïnstalleerd, garandeert de nauwe integratie met de database voor prestaties en schaalbaarheid die in een traditionele AJAX-omgeving niet mogelijk zijn.

Een andere nieuwe component in CACHÉ, Jalapeño, biedt de mogelijkheid om bestaande Java-objecten automatisch persistent te maken in de krachtige CACHÉ-database-omgeving zonder dat objectrelationeel mappen nodig is. Jalapeño maakt database-onafhankelijke applicatie-ontwikkeling mogelijk zonder het extra werk dat met SQL altijd gemoeid is. Mocht de toepassing op een relationele database zoals Oracle, SQL Server of andere uitgerold worden, dan behoort dit eveneens tot de geboden mogelijkheden.

CACHÉ 2007 is verkrijgbaar voor Windows, Linux, Mac, UNIIX en Open VMS.