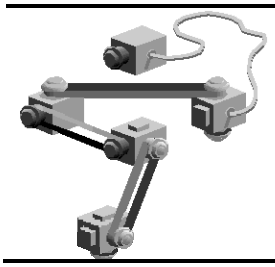


Aspect Oriented Programming in Domain Driven Design

ing. Srinivasan Tharmarajah

Scriptie

Master Software Engineering
Universiteit van Amsterdam

Universiteit van Amsterdam

Sogyo

Afstudeerdocent:	Drs. Hans Dekkers
Stagebegeleider:	Stefan Bookholt
Opdrachtgever:	Sogyo, De Bilt
Versie:	1.0
Datum:	19 augustus 2007
Publicatiestatus:	Openbaar

Inhoudsopgave

Figuren en Tabellen

Figuren

Tabellen

Samenvatting

In dit onderzoek bestuderen we de toepasbaarheid van Aspect Oriented Programming in Domain Driven Design applicaties.

Domain Driven Design(DDD) is een software ontwikkelconcept waarin het ontwikkelen van businesslogica, genaamd 'het domein', een centrale rol speelt. Informatie over het domein wordt in een model gestructureerd en het domeinmodel leidt het design en de softwareontwikkeling. De niet-businessspecifieke onderdelen zoals persistentie, presentatie en autorisatie worden *services* of *infrastructuur* genoemd en hebben een ondersteunende rol. Volgens dit concept probeert men een domein te ontwikkelen dat geen of zo min mogelijk kennis van het bestaan van de services heeft en het domein en de services hebben een minimale koppeling met elkaar.

DDD applicaties van Sogyo bestaan uit drie onderdelen. Namelijk het domein, de services en de tussenlaag die voor de koppeling tussen het domein en de services zorgt. Hierbij functioneren de op het adapter pattern gebaseerde adapters als tussenlaag. Hoewel de adapters in de praktijk redelijk functioneren, hebben ze ook een aantal tekortkomingen. Zo heeft bijvoorbeeld de tussenlaag een sterke koppeling en veel code. Aangezien DDD op Object Oriented Programming gebaseerd is, vormen de cross-cutting concerns ook een probleem.

Aspect Oriented Programming(AOP) richt zich op separation of concerns, hierbij voornamelijk op cross-cutting concerns zoals logging. Daarmee probeert AOP de modulariteit van software te verhogen en duplicatie van code te verminderen. De cross-cutting concerns, genaamd *aspects* in AOP, worden in aparte modules gedefinieerd en door middel van een weaver op verschillende invoegpunten in de logicacode toegevoegd.

Sogyo wil voornamelijk weten of AOP gebruikt kan worden voor de DDD applicaties om het domein en de services met elkaar op een nette manier te koppelen. Hierbij vervangt AOP de huidige tussenlaag en we proberen de tekortkomingen van de adapters op te lossen. Daarnaast is het bedrijf ook geïnteresseerd in de algemene toepasbaarheid van AOP in de .NET omgeving.
Het onderzoek heeft twee hoofdonderzoeksvragen.

- *Is AOP toepasbaar om het Domain Driven Design concept te realiseren?*
- *Verbetert AOP de ontkoppeling van Domain Driven Design applicaties in vergelijking met de huidige implementatie en worden de huidige implementatieproblemen met behulp van AOP opgelost?*

We gebruiken een proof of concept om te laten zien hoe men AOP binnen DDD kan toepassen om de koppeling tussen het domein en de services te realiseren. Het proof of concept wordt ook gebruikt om de onderzoeksresultaten te evalueren en is gebaseerd op de bestaande Sogyo Bankcase[24]. Ons onderzoek beperkt zich tot de Microsoft.NET omgeving en de implementatieproblemen van de Sogyo Robot[5] en Bankcase.

We hebben drie AOP tools die in de .NET omgeving beschikbaar zijn met elkaar vergeleken en daaruit blijkt dat PostSharp een betere oplossing biedt. De andere twee tools zijn Rapier-Loom.NET[14]en Aspect.NET[15].

Het is mogelijk om met behulp van AOP een DDD applicatie te ontwikkelen waarbij het domein geen kennis van het bestaan van de services heeft. Uit de resultaten blijkt dat de AOP versie van de Bankcase minder regels code en minder complexiteit heeft. De koppelingen tussen de domeinklassen en tussenlaagklassen zijn bij de AOP applicatie minder dan bij de adapterapplicatie. Dat geldt ook voor de koppelingen tussen de tussenlaagklassen en de serviceklassen. Bij de AOP applicatie is er echter een sterkere koppeling van de services aan het domein.

De meerwaarde van AOP ligt vooral in het gebruiken van de cross-cutting services. Indien men geen gebruik van de cross-cutting concerns maak, kan men beter adapters gebruiken. Een AOP applicatie heeft een externe tool (b.v. PostSharp) nodig terwijl de adapters alleen de OO constructies gebruiken. Het gebruik van AOP vereist ook extra kennis over dat concept.

Een combinatie van AOP en adapters is mogelijk, waarbij de cross-cutting services met behulp van AOP en de andere services met behulp van adapters geïmplementeerd worden.

Voorwoord

Voor u ligt de scriptie van Srinivasan Tharmarajah. Dit afstudeeronderzoek werd uitgevoerd als afsluitend deel van de eenjarige opleiding Master Software Engineering aan de Universiteit van Amsterdam. De onderzoeksperiode duurde ongeveer 4 maanden.

Het schrijven van deze scriptie was niet mogelijk geweest zonder een aantal personen. Graag wil ik Drs. Hans Dekkers bedanken voor zijn begeleiding zowel gedurende de afstudeeropdracht als het gehele studiejaar. Daarnaast ben ik Prof. dr. Jan van Eijck, Prof.dr. Paul Klint, Dr. Jurgen Vinju en Prof. dr. Hans van Vliet van de opleiding erg dankbaar voor de begeleiding gedurende het studiejaar.

Graag wil ik Sogyo bedanken voor het aanbieden van de afstudeermogelijkheid. Graag dank ik Sogyo medewerkers, in het bijzonder Stefan Bookholt, Gerard Braad, Edwin van Dillen, Ralf Wolter, Thomas Zeeman en Arnoud van Zoest. Ook dank aan medeafstudeerders, vooral Bart Koot, voor een leerzame en gezellige onderzoeksperiode.

Ook dank ik de medestudenten van de opleiding, vooral Anton Lycklama à Nijholt voor zijn kritische feedback tijdens het afstudeerproject. Ik ben de personen dankbaar die deze scriptie gereviewd hebben en feedback op gegeven hebben, met name Sharron Letterboom.

Ik ben zeer mijn familie dankbaar voor de blijvende ondersteuning.

Srinivasan Tharmarajah
Amsterdam, 19-08-2007

Hoofdstuk 1

Inleiding

Domain Driven Design
Software ontwikkelmethoden en concepten veranderen regelmatig. Software-experts proberen methoden te vinden die betere kwalitatieve software met minimale kosten kunnen produceren. Domain Driven Design(DDD), Service Oriented Architecture (SOA) en Table Driven Design(TDD) zijn enkele voorbeelden van de huidige software ontwikkelconcepten.

Domain Driven Design[3,10,25] wordt de laatste jaren populairder, vooral sinds Eric Evans het boek *Domain Driven Design: Tackling Complexity in the Heart of Software*[25] heeft geschreven. Het voornaamste idee van Domain Driven Design is dat businesslogica het primaire onderdeel van softwareontwikkeling is. Het (complexe)probleemdomein waarvoor software ontwikkeld wordt, moet goed begrepen worden. Daaruit maakt men een *domeinmodel* en dat model leidt het design en de verdere softwareontwikkeling. Het *domein* bevat alleen de businesslogica. Niet domeinspecifieke onderdelen zoals presentatie, persistentie of autorisatie worden *services* of *infrastructuur* genoemd en worden zo veel mogelijk los van businesslogica ontwikkeld. Volgens dit concept heeft het domein geen of zo min mogelijk kennis van het bestaan van de services en beide onderdelen moeten zo min mogelijk koppeling met elkaar hebben. [25] zegt dat het domein in een DDD applicatie wel kennis van de services mag hebben, terwijl Sogyo streeft naar een 'schoon' domein dat geen kennis van andere onderdelen heeft. Door de goed gedefinieerde, verdeelde verantwoordelijkheden van het domein en de services, en de minimale koppeling tussen deze onderdelen zou de complexiteit van software minder worden. Daarnaast zou software makkelijker te testen, her te gebruiken en te onderhouden zijn. In 2.1 wordt Domain Driven Design verder behandeld.

Aspect Oriented Programming
Cross-cutting concerns vormen een probleem dat door het huidige Objectgeoriënteerde(OO) paradigma niet op een nette manier op te lossen is. Sommige concerns, bijvoorbeeld logging, kunnen niet in een bepaalde module geplaatst worden en deze concerns worden door verschillende klassen herhaaldelijk gebruikt. Aspect Oriented Programming(AOP)[1,6,7,8,911,30] probeert hiervoor een oplossing te vinden door deze cross-cutting concerns in modules op te vangen. Daarmee probeert men de modulariteit van software te verhogen en de duplicatie van de code te verminderen. AOP wordt gezien als aanvulling op de huidige programmeertechnieken als OOP. Paragraaf 2.2 behandelt Aspect Oriented Programming verder.

Probleemomschrijving
Een probleem bij het toepassen van DDD is dat het domein en de services in de praktijk vaak sterk aan elkaar gekoppeld zijn. Ontwikkeltechnieken zoals design patterns en layered architecture kunnen gebruikt worden om de koppelingen van Domain Driven Design applicaties te verminderen. [3] en [25] presenteren een set van design patterns waarvan software-experts gebruik kunnen maken.
De huidige twee cases Robot[5] en Bank[24] die binnen Sogyo in om loop zijn bestaan uit het domein, de services en de tussenlaag. De tussenlaag is gevormd door op het adapter pattern gebaseerde adapters en deze adapters verzorgen de koppeling tussen het domein en de services. Deze implementatie heeft een aantal tekortkomingen. De tussenlaag heeft over het algemeen veel code en een sterke koppeling met het domein en/of de services. Aangezien DDD op OOP gebaseerd is, is er geen goede oplossing voor de cross-cutting services. In hoofdstuk 3 worden deze tekortkomingen verder beschreven.

Opdrachtomschrijving
Sogyo wil onderzoek doen naar de mogelijkheid om met behulp van AOP de koppelingen tussen het domein en de services te verbeteren. De huidige adapterlaag wordt hierbij door de AOP laag vervangen en de toepasbaarheid van AOP in de .NET omgeving wordt op basis van de Sogyo Bankcase[24] bestudeerd. Hiernaast bestuderen we ook de huidige AOP tools in de .NET omgeving.

Men is geïnteresseerd in antwoorden op diverse vragen.

- In hoeverre is AOP toepasbaar binnen DDD?
- Is AOP beter dan de huidige oplossing?
- Hoe kunnen we beide oplossingen met elkaar vergelijken?
- Wat zijn de huidige AOP tools in de .NET omgeving?

1.1 Bedrijfsomschrijving

Sogyo[32] is in 1995 opgericht en gevestigd op landgoed Sandwijck in De Bilt. Het bedrijf is zich gespecialiseerd in het ontwikkelen van IT-talenten en IT-toepassingen. Er werken circa 75 medewerkers.

Sogyo kent drie afdelingen:

- *Sogyo academy*
  Verzorgt opleidingen voor medewerkers en klanten.
- *Sogyo professionals*
  Biedt verschillende rollen, variërend van tijdelijke inzet op consultancy- of detacheringbasis, tot trajecten die tot een vastverband bij opdrachtgevers kunnen leiden.
- *Sogyo development*
  Ontwikkelt IT-toepassingen. De meeste applicaties worden in Java of .NET omgeving ontwikkeld.

Sogyo geeft presentaties en houdt workshops over diverse onderwerpen, onder andere over Domain Driven Design. Alle drie de afdelingen zijn geïnteresseerd in dit onderzoek en zijn ook de stake holders van dit onderzoek.

1.2 Relevantie van dit onderzoek

Vrijwel alle medewerkers van Sogyo zijn bekend met Domain Driven Design. Op dit moment beschikt Sogyo over weinig kennis van de toepasbaarheid van AOP in de .NET omgeving. Enkele medewerkers zijn bezig met het experimenteren van AOP tools, maar een echt onderzoek is nog niet uitgevoerd. Sogyo ontwikkelt de applicaties over het algemeen volgens het DDD concept.

Sogyo academy
Aangezien Sogyo presentaties en workshops geeft over onder andere Domain Driven Design kunnen de resultaten van dit onderzoek voor deze doeleinden gebruikt worden. Sogyo biedt ook een drie maanden durende opleidingstraject voor de instromende nieuwe medewerkers en DDD is een onderdeel van deze zogenaamde 'Master course'. Middels dit onderzoek kunnen de studenten mogelijk (meer) kennis over de toepasbaarheid van AOP verwerven.

Sogyo professionals
Deze professionals werken voornamelijk bij de opdrachtgevers op consultancy- of detacheringbasis. Indien AOP een in het bedrijfsleven bruikbaar concept is, kunnen de professionals hun klanten aanraden om AOP te gaan gebruiken of in sommige gevallen ook daadwerkelijk toepassen.

Sogyo development
De resultaten van dit onderzoek kunnen door de ontwikkelaars van Sogyo gebruikt worden. Sogyo ontwikkelt applicaties in opdracht van klanten. Indien de onderzoeksresultaten positief zijn, kunnen ze mogelijkerwijs AOP voor de ontwikkeling van de softwareproducten gebruiken.

Algemeen
Uit de gesprekken met de Sogyo medewerkers en gelezen literatuur blijkt dat er op dit moment weinig onderzoeken zijn uitgevoerd, die de bruikbaarheid van AOP in Domain Driven Design bestuderen, vooral wat betreft de .NET omgeving. AOP schijnt vooral meer in de wetenschappelijke wereld en in mindere mate in het bedrijfsleven toegepast wordt.

Wat AOP betreft heeft de Java wereld voorsprong op de .NET wereld. AspectJ[1,33] is zelfs een onderdeel/plugin voor *Eclipse*, een bekende Java ontwikkelomgeving. Op dit moment is er geen standaard

AOP tool voor .NET applicaties en er zijn meer op AspectJ gebaseerd voorbeelden of artikelen te vinden dan op .NET. Het kan veroorzaakt zijn door het feit dat de oorsprong van het AOP concept meer vanuit de Java hoek komt en de bedenkers van dit concept zich meer met AspectJ bezighouden. De laatste jaren is men bezig met het verspreiden van het AOP concept in andere programmeertalen, onder andere in de .NET programmeertalen. Aan de andere kant probeert men applicaties te ontwikken waarin diverse onderdelen minder strek aan elkaar gekoppeld zijn. Dit onderzoek kan mogelijk een bijdrage leveren om deze doelen te kunnen bereiken.

Indien dit onderzoek een positief resultaat oplevert, wordt de mogelijkheid om de .NET applicaties met mindere koppelingen te ontwikkelen groter. Dat kan mogelijk resulteren in betere kwalitatieve software, die een kortere ontwikkel- en onderhoudstijd vergt. Ook als het onderzoek niet zou slagen, kan de kennis die in dit onderzoek is opgebouwd voor Sogyo waardevol zijn.

Twee jaar geleden is al een onderzoek binnen Sogyo met betrekking tot AOP in Java uitgevoerd. Omdat op dit moment binnen Sogyo nog geen onderzoek in de .NET omgeving uitgevoerd is, is er nu voor gekozen om het AOP concept in de .NET omgeving te bestuderen.

1.3 Scope

Het onderzoek bespreekt twee thema's; Domain Driven Design en Aspect Oriented Programming. De nadruk ligt meer op de toepasbaarheid van Aspect Oriented Programming in Microsoft.NET omgeving. We onderzoeken niet de basis ideeën van DDD, maar het DDD concept wordt voornamelijk gebruikt om de problemen van de huidige implementatietechniek, namelijk de adapters, te herkennen en de richtlijnen waaraan de AOP applicatie(proof of concept) moet voldoen te definiëren.

Het onderzoek richt zich op de problemen die Sogyo aantreft bij het ontwikkelen van DDD applicaties waarbij de adapters gebruik worden. Dit onderzoek is gebaseerd op de Sogyo Bankcase. Het is mogelijk dat elders vergelijkbare problemen in mindere mate voorkomen of reeds opgelost zijn.

1.4 Overzicht

Dit hoofdstuk is een inleidend hoofdstuk. Hoofdstuk 2 geeft achtergrondinformatie over Domain Driven Design en Aspect Oriented Programming. De probleemstelling en onderzoekvragen worden in hoofdstuk 3 behandeld. Vervolgens wordt in hoofdstuk 4 de onderzoeksmethode behandeld.

De huidige AOP tools in de .NET omgeving worden in hoofdstuk 5 besproken en hoofdstuk 6 beschrijft de proof of concept, namelijk de Bankcase. In hoofdstuk 7 worden de validatie- methoden van dit onderzoek besproken en in hoofdstuk 8 worden de onderzoeksresultaten geëvalueerd. De scriptie wordt afgesloten met de conclusie en aanbevelingen, future work en reflectie, en de bronnenlijst.

Hoofdstuk 2

Achtergrond

Dit hoofdstuk geeft achtergrondinformatie over de twee hoofdthema's van dit onderzoek, namelijk Domain Driven Design en Aspect Oriented Programming.

2.1 Domain Driven Design

Domain Driven Design(DDD) is een ontwikkelconcept en is de laatste jaren populair aan het geworden[3,10,25]. DDD is grotendeels gebaseerd op object georiënteerd design. We kunnen DDD ook als 'Best Practices van Object Oriented Programming' zien. DDD biedt een denkwijze hoe software ontwikkeld kan worden met behulp van OOP.

[10] beschrijft het volgende over Domain Driven Design:
** For most software projects, the primary focus should be on the domain and domain logic; and*
** Complex domain designs should be based on a model.*

Bij Domain Driven Design wordt, gefocust op het begrijpen van het (complexe) domein en ontwikkelen van het bijbehorende domeinmodel. Kennis over het domein wordt gestructureerd in een domeinmodel en het model leidt het design en de softwareontwikkeling.

Een van de redenen waarom softwareprojecten mislukken is dat het businessdomein en de probleemen niet goed begrepen worden. Als men dus een softwareoplossing voor een businessdomein wil ontwikkelen, moet men eerst het domein en daarbij behorende problemen goed begrijpen. Kennis over het domein moet gestructureerd en vastgelegd worden. De domeinkennis moet de softwareontwikkeling leiden. De moeilijkheid van de meeste softwareprojecten is dan ook het begrijpen van het probleemdomein. Het belangrijkste principe van DDD is dat de complexiteit van het domein duidelijk gedefinieerd wordt en niet de technische infrastructuur. De infrastructuuronderdelen als GUI en database staan los van het domeinmodel en worden als services gezien die het domein ondersteunen (aan de gebruikers tonen of in een database opslaan). Hiermee worden de diverse verantwoordelijkheden van software apart gehouden, genaamd *separation of concerns*. Dit voorkomt 'spaghetti' code in de applicaties waarin diverse verantwoordelijkheden in een klasse geïmplementeerd zijn.

In Domain Driven Design zitten de domeinexpert(s) en software expert(s) samen en de domeinexperts dragen de kennis over het domein aan de software-experts over. Niet alleen de software architecten en informatieanalisten nemen aan de gesprekken deel maar ook de ontwikkelaars van het systeem. Deze kennis wordt gemodelleerd in het domeinmodel. De software-experts geven ook feedback over hoe men het domeinmodel kan veranderen. De software-experts weten over de mogelijkheden en beperkingen van de softwaretools die het domeinmodel naar een applicatie moeten omzetten. Een domeinmodel dat niet met de software implementeerbaar is, heeft geen nut. Met behulp van deze feedback wordt het domeinmodel aangepast. Een domeinmodel wordt niet in één keer volledig ontwikkeld, dit is een iteratief proces.

The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model [25 p.70-71]

Een Domain Driven Design applicatie bestaat uit het domein en verschillende services of infrastructuuronderdelen. Het domein staat centraal en beschrijft alleen businesslogica (core concern). De verantwoordelijkheden als presenteren, persisteren of beveiligen van het domein worden aan de services gedelegeerd.

Mogelijke voordelen
In deze aanpak hebben het domein en de services duidelijk gedefinieerde verantwoordelijkheden. Hierdoor word de complexiteit van het ontwerpen en bouwen van de applicatie geminimaliseerd. Aangezien het domein en services minder met elkaar gekoppeld zijn en een duidelijke scheiding hebben, wordt de applicatie beter onderhoudbaar, herbruikbaar en testbaar. De businesslogica is alleen op één plek gedefinieerd. Wanneer een domein infrastructuuronafhankelijk is, kan het domein makkelijker voor meerdere infrastructuren gebruikt worden(Windows of Web, MS SQL Server of Oracle). Een domeinmodel dat met een service(b.v. GUI) sterk gekoppeld is, is moeilijker te testen.

2.1.1 Ubiquitous Language

Het is bij het modelleren van het domein belangrijk dat de software-experts en de domeinexperts een begrijpbare taal gebruiken. Als de software-experts vaktermen als inheritance of interface gebruiken, zullen de domeinexperts dit niet begrijpen. Daarnaast heeft elk mens een andere communicatiestijl. Dus er moet een algemene woordenlijst ontwikkeld worden die voor iedereen toegankelijk is en die taal moet ambiguïteit voorkomen.

The vocabulary of that Ubiquitous Language includes the names of classes and prominent operations. The language includes terms to discuss rules that have been explicit in the model[25 p.25].

Ubiquitous Language[10,25] is een taal die rondom het domeinmodel wordt ontwikkeld en deze taal wordt gebruikt door alle leden van het team om de softwareactiviteiten te verbinden. Deze taal is specifiek voor het bijbehorende domein en wordt ook gebruikt in het domeinmodel, design, code en daarbij behorende documenten. Hierdoor kunnen de software-experts en de domeinexperts makkelijk met elkaar communiceren zonder dat er een vertaling nodig is.

2.1.2 Domein en Domeinmodel

Domein
The subject area to which the user applies a program is the domain of the software[25].

Het domein voor een bankapplicatie is de bankwereld. De bankwereld bestaat onder andere uit klanten, rekeningen en transacties.

Domeinmodel
It is a rigorously organized and selective abstraction of that (domain) knowledge [25].

Een domeinmodel kan door middel van een diagram beschreven worden, maar ook door middel van geschreven zinnen in een taal. Het domeinmodel bevat informatie over het desbetreffende domein(*real world*). Het design en de verdere softwareontwikkeling moeten afgeleid zijn van dat domeinmodel. Het domeinmodel van een bankapplicatie kan de conceptuele klassen als Klant, Rekening en Transactie bevatten. Doordat het domeinmodel samen door de domeinexperts en de software-experts ontwikkeld wordt, zorgt dit voor het beter begrijpen van de requirements en voorkomt miscommunicatie tussen die twee partijen. Het model moet het design en de code weerspiegelen. Wanneer de code of het design verandert, moet ook het domeinmodel aangepast worden. Het domeinmodel moet zodanig gemaakt zijn dat het model door de programmeertechnieken makkelijk te implementeren is. Het domeinmodel bevat geen informatie over technische implementatie (ArrayList, Interface, Adapters).

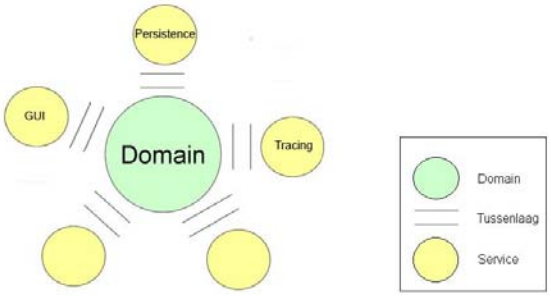
2.1.3 Services / Infrastructuur

Een service is in principe alles wat niet tot het domein behoort en heeft een ondersteunde rol voor het domein. Services kunnen bijvoorbeeld gebruikerinterface, persistentie, logging of authenticatie zijn. De services geven input aan het domein of reageren op verandering in het domein. Een ideale service is zo min mogelijk afhankelijk van andere onderdelen van de applicatie en herbruikbaar voor verschillende domeinen.

2.1.4 DDD Implementatie

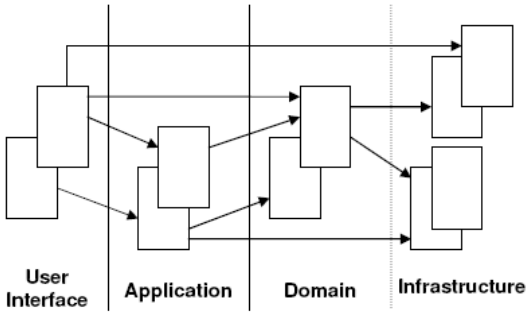
DDD applicaties kunnen verschillende architecturen hebben en diverse Design patterns gebruiken, zolang ze aan de DDD richtlijnen voldoen. Figuur 1 geeft de architectuur van een DDD applicatie weer zoals door Sogyo[5,24] voorgesteld wordt. Figuur 2 toont de Layered Architecture voor de DDD applicaties die [25] voorstelt.

Binnen Sogyo verdeelt men de DDD applicaties in drie onderdelen/lagen, namelijk de domeinlaag, de tussenlaag en de servicelaag.
Domeinlaag: Bevat de businesslogica.
***De servicelaag*:** Bevat verschillende services die het domein ondersteunen. Ze kunnen bijvoorbeeld GUI, persistentie of logging zijn.
De tussenlaag: Zorgt voor de koppeling tussen het domein en de services. Dit biedt geen extra functionaliteit.



Figuur 1: DDD Architectuur - Sogyo [5,24]

[25] stelt een gelaagd model voor, dat uit vier lagen bestaat.
User Interface: Verantwoordelijk voor het tonen van informatie aan de gebruiker en interpreteren van de gebruikersopdrachten.
Application*:* Definieert de taken die de software moet uitvoeren. Deze laag bevat geen businesslogica en coördineert applicatieactiviteiten.
Domain: Bevat informatie over het domein, het hart van software.
Infrastructure: Deze laag biedt ondersteuning aan de bovenliggende laag en bestaat uit services.



Figuur 2: Layered Architectuur[25]

Zoals hierboven te zien is, hebben Sogyo en [25] een aantal verschillen in implementatie. Sogyo streeft naar de volledige onafhankelijkheid van het domein, terwijl de andere architectuur de mogelijkheid biedt om de domeinlaag kennis van de infrastructuur te hebben. De layered architectuur heeft de application layer voor het coördineren van applicatieactiviteiten, terwijl deze verantwoordelijkheid in de Sogyo architectuur door presentatieservice (en domeinlogica) opgenomen wordt.

In dit onderzoek richten we ons op de architectuur van Sogyo. De richtlijnen waaraan de DDD applicatie moet voldoen is dan ook gebaseerd op deze architectuur. Dat betekent dat de domeinklassen geen kennis van de serviceklassen mogen hebben.

Onder 'kennis hebben of gebruiken' verstaan we het volgende. Klasse A heeft kennis van of gebruikt de klasse B, indien:

- Klasse A heeft een attribuut dat refereert naar de klasse B.
- Klasse A roept een methode van de klasse B aan.
- Klasse A heeft een methode die heeft referentie naar de klasse B (via return type of parameter).
- Klasse A is een subklasse van de klasse B.

2.1.5 Richtlijnen

Dit gedeelte geeft een lijst van de richtlijnen van het domein(klassen) en de service(klassen)[4,25]. Volgens deze richtlijnen wordt de proof of concept(Bank case) gemaakt.

Domein

- Er is geen directe aanroep of referentie naar een object buiten het domein.
- Geen additionele bronnen als extra bestanden, externe operaties nodig om te functioneren.

- Het moet niet framework(ADO.NET, NHibernate, etc.) specifiek zijn.

Service
- De operatie die door een service wordt uitgevoerd behoort niet tot een domeinobject.
- De service mag kennis hebben van de domeinobjecten.
- De service heeft geen toestand(stateless) die eigen gedrag kan beïnvloeden.

2.1.5 Discussie

De software-experts die de Objectgeoriënteerde technieken (OO analyse, OO modelling, Design patterns en N-tier architectuur) toepassen, zullen de DDD ideeën herkennen. DDD geeft een aantal richtlijnen en technieken om een software te ontwikkelen, waarbij het domein(model) de softwareontwikkeling leidt. Met OOP programmeert men op verschillende manieren. In OOP worden over het algemeen persistentie, tracing en andere services in de businessobjecten opgenomen. Maar wanneer men over DDD praat, praat men over een schoon domein waar geen servicetaken inzitten en over diverse services beschikt die goed gedefinieerde verantwoordelijkheden hebben. Met DDD probeert men de koppeling tussen het domein en de services te minimaliseren, welke in de meeste OO applicaties sterk gekoppeld zijn. Het gebruik van de Ubiquitous Language zal voor sommige software-experts nieuw zijn. De ontwikkelaars zijn meestal geïnteresseerd in technische details en minder in het leren van een probleemdomein. In DDD is expliciete aandacht voor het leren van het probleemdomein en de ontwikkelaars hebben actief contact met de domeinexperts.

Domain Driven Design verschilt bijvoorbeeld van Table(Database) Driven Design. Bij DDD maakt men eerst het domeinmodel en van daaruit vindt de softwareontwikkeling plaats. Bij Table Driven Design wordt eerst een databasemodel gemaakt en daarin wordt ook businesslogica geplaatst(met behulp van *stored procedures* bijvoorbeeld) en dat leidt de softwareontwikkeling.

DDD aanpak kan niet voor alle problemen een betere oplossing zijn. Het is voornamelijk geschikt voor (complexe) businessdomeinen waarin veel gedrag voorkomen. Er moet daarom per project gekeken worden of DDD wel geschikt is voor dat specifieke project. Een applicatie met simpele CRUD(Create, Read, Update en Delete) operaties kan bijvoorbeeld met Table Driven Design gemaakt worden.

2.2 Aspect Oriented Programming

Op dit moment is er veel aandacht voor Aspect Oriented Programming (AOP)[1,2,6,7,8,9,11,30]. Het eerste paper[30] over AOP is in juni 1997 op *European Conference on Object-Oriented Programming (ECOOP)* gepubliceerd en geschreven door Gregor Kiczales et al. Eén van de fundamentele principes van Software Engineering is *separation of concerns* waarin diverse onderdelen van software gedefinieerde verantwoordheden(presentatie, persistentie, etc.) en min mogelijk overlapping bevatten. AOP is bedoeld om separation of concerns, voornamelijk, *cross-cutting concerns* te verbeteren. Zie 2.2.1 voor meer informatie over cross-cutting concerns. AOP probeert de modulariteit van de software systemen te verbeteren door deze concerns in modules/klassen op te vangen. AOP introduceert hiervoor een nieuw concept, genaamd 'aspect'. Aspecten in AOP zijn in principe de cross-cutting concerns. AOP is een uitbreiding op de bestaande programmeertechnieken als OOP.

AspectJ[1,33] is een bekende AOP taal en een uitbereiding op Java. De eerste versie van AspectJ Development Tools (AJDT), de bekendste Java AOP implementatie, is in maart 1998 verschenen. AJDT is een plugin voor de Eclipse ontwikkelomgeving en op dit moment min of meer de standaard AOP tool in de Java omgeving. AspectJ biedt uitgebreide functionaliteit. Aspect Oriented Programming in de .NET omgeving is niet zo ontwikkeld als de Java omgeving. Er is op dit moment nog geen standaard tool voor AOP in de .NET omgeving.

AOP tools zorgen ervoor dat de cross-cutting concerns oftewel aspecten in de logicacode ingevoerd worden. De ontwikkelaar definieert invoegpunten waarin de aspectcode in de logicacode ingelast moet worden. Tijdens het samenvoegen(weaving) door de *weaver* wordt in deze gedefinieerde invoegpunten(*join-points*) de aspectcode toegevoegd. Een invoegpunt kan bijvoorbeeld een aanroep naar een methode zijn. De verzameling van de invoegpunten wordt *point-cut* genoemd. Weaver is een tool die de cross-cutting concerns en de logicacode combineert. Deze aspecten kunnen los van de logica ontwikkeld worden en de businesslogica heeft

geen kennis van het bestaan van deze aspecten. AOP zou daarom de mogelijkheid bieden om de businesslogica en de services in Domain Driven Design op een nette manier te scheiden.

AOP heeft zowel voor- als nadelen.[36] identificeert op basis van een casestudy de volgende voor- en nadelen.

- *An AOSD approach improves code quality by minimizing code duplication, improving uniformity and understandability and reducing scattering and tangling.*
- *An AOSD approach introduces additional maintainability risks. A potential risk when separating the aspect code from the base code is that the two get out of sync: when a component evolves, its associated aspects do not.*
- *Adoption of AOSD requires different adoption scenario's and change management.*

2.2.1 Cross-cutting concerns

Object Oriented Programming(OOP) is een bekend, veel gebruikt paradigma voor softwareontwikkeling. OO heeft geen nette oplossing voor de cross-cutting concerns. Een cross-cutting concern is een concern dat over verscheidende plekken van de programmacode verspreid ligt. Aangezien deze concerns door meerdere klassen en methoden aangeroepen worden, zijn deze moeilijk in een eigen module te vangen. Zo wordt bijvoorbeeld een service als logging door vele klassen aangeroepen. Dit leidt tot herhaling van dezelfde code op verschillende punten van de applicatie. Subhoofdstuk 2.2.2 beschrijft een voorbeeld van logging. Authenticatie, exceptiehandeling en tracing of logging zijn bekende voorbeelden van cross-cutting concerns.

Met behulp van AOP zou het ook mogelijk zijn om de services als persistentie en presentatie te implementeren. Maar aangezien deze concerns geen cross-cutting concerns zijn, zijn ze niet de voornaamste doelgebieden van aspectoriëntatie.

2.2.2 Motivatie

Zoals hierboven beschreven, worden de cross-cutting concerns in Object Oriented Programming door meerdere methoden en klassen herhaaldelijk toegepast. Als een bankapplicatie een klant moet weergeven, zal men een `Klant` klasse met bijbehorende methoden en properties maken. En niet-klant gerelateerde logica als logging, exceptiehandeling, etc. mag niet in de `Klant` klasse geïmplementeerd worden. Dat maakt het onderhouden, hergebruiken en testen van een applicatie moeilijk.

Hiernaast op Figuur 3 is een voorbeeld van een `Calculator` klasse te zien. Deze class bevat alleen de logicacode code die nodig is om de calculatietaken uit te voeren. Dit is een voorbeeld voor een 'nette' klasse. Deze klasse bestaat uit vier simpele methoden en implementeert de `ICalculator` interface.

```
public class Calculator : ICalculator {
    public int Add(int n1, int n2) {
        return n1 + n2;
    }
    public int Subtract(int n1, int n2) {
        return n1 - n2;
    }
    public int Divide(int n1, int n2) {
        return n1 / n2;
    }
    public int Multiply(int n1, int n2) {
        return n1 * n2;
    }
}
```

Figuur 3: Calculator klasse

Stel dat men de gebruikte methoden, argumenten en terugkeerwaarde van deze klasse wil bijhouden, namelijk *tracing*, moet extra code aan de methoden toegevoegd worden. Figuur 4 geeft de voorbeeldcode weer.

```
public class Calculator : ICalculator
{
    public int Add(int n1, int n2) {
        Console.WriteLine("Entry:Calculator.Add (" + n1 + ", " + n2 + ")");
        int result = n1 + n2;
        Console.WriteLine("Exit: Calculator.Add(" + n1 + ", " + n2 + ") returned " + result);
        return result;
```

```
    }
    public int Subtract(int n1, int n2) {
       Console.WriteLine("Entry: Calculator.Subtract(" + n1 + ", " + n2 + ")");
       int result = n1 - n2;
       Console.WriteLine("Exit: Calculator.Subtract(" + n1 + ", " + n2 + ") returned " + result);
       return result;
    }
    public int Divide(int n1, int n2) {
       Console.WriteLine("Entry: Calculator.Divide(" + n1 + ", " + n2 + ")");
       int result = n1 / n2;
       Console.WriteLine("Exit: Calculator.Divide(" + n1 + ", " + n2 + ") returned " + result);
       return result;
    }
    public int Multiply(int n1, int n2) {
       Console.WriteLine("Entry: Calculator.Multiply(" + n1 + ", " + n2 + ")");
       int result = n1 * n2;
       Console.WriteLine("Exit: Calculator.Multiply(" + n1 + ", " + n2 + ") returned " + result);
       return result;
    }
}
```

Figuur 4: Calculator klasse met tracing –zonder AOP

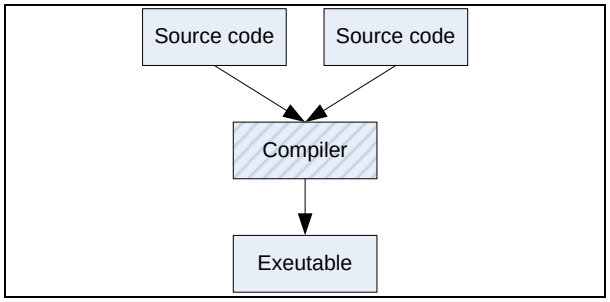
Problemen:

- De **Console.WriteLine("message")** wordt nu acht keer herhaald. Dat maakt de leesbaarheid en onderhoudbaarheid van deze class lager, vooral wanneer men tracing code wil aanpassen of verwijderen.
- De `Calculator` klasse heeft naast de calculatietaken ook de tracingtaak. Dus twee taken in een klasse/methode. Dat vermindert de onderhoudbaarheid van de code.
- Deze oplossing overtreedt het Domain Driven Design principe. Businesslogica moet immers geen service(niet businesslogica) verantwoordelijkheid bevatten.

Aangezien het huidige OO concept geen nette oplossing voor de bovenstaande problemen biedt, probeert AOP hiervoor een oplossing te vinden.

2.2.3 0bject Oriented implementatie

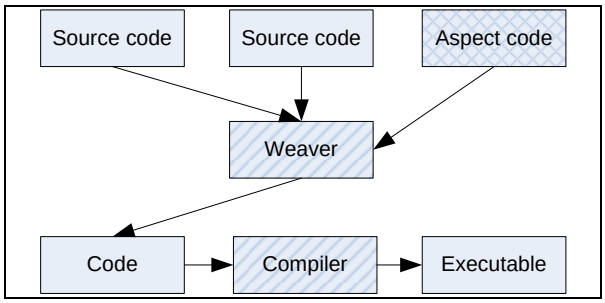
Hiernaast is een plaatje(figuur 5) van een 'algemene' simpele OO applicatie die geen AOP gebruikt. Verschillende source bestanden worden gecompileerd en de compiler maakt een assembly bestand(.exe, .dll). De broncode bevat zowel kern functies als cross-cutting functies.



Figuur 5: Algemene (0bejct Oriented) implementatie

2.2.4 AOP implementatie

In een AOP implementatie worden de businesslogica en de cross-cutting concerns apart gehouden. De broncode beschrijft alleen de kern functies. De aspectcode beschrijft cross-cutting functies. De aspectcode en de broncode worden door middel van een weaver samengevoegd. Deze samengevoegde code wordt door de compiler gecompileerd en er wordt een assembly bestand genereert. Zie Figuur 6.



Figuur 6: AOP implementatie

Hoewel het bovenstaande figuur een algemene AOP implementatietechniek weergeeft, hanteert elke AOP implementatie/tool een eigen implementatiemechanisme. Bij sommige AOP tools wordt eerst de broncode gecompileerd, daarna vindt pas de samenvoeging van de aspectcode plaats.

Over het algemeen bestaat een AOP applicatie uit drie onderdelen. Zie Figuur 7 voor een voorbeeld, dit voorbeeld is gemaakt in C# met behulp van PostSharp.

- Target/domeinklasse – `Calculator.cs`
- Serviceklasse (concern) - `TraceMethod.cs`
- Aspectklasse – `TraceAspect.cs`

Het is ook mogelijk om de implementatie van de tracing service in de aspectklasse zelf te implementeren i.p.v. een apart klasse voor de service. Op die manier heeft de aspectklasse echter meer verantwoordelijkheid en de serviceklasse is minder makkelijk te hergebruiken.

Tracing aspect:

```
public TraceAspect {
  TraceMethod trace = new TraceMethod();
  public void OnEntry(MethodExecutionEventArgs args){
    traceMethod.MethodEntry(args);
  }
  public void OnExit(MethodExecutionEventArgs args){
    traceMethod.MethodExit(args);
  }
}
```

Calculator logica:

```
[Trace]
public Calculator {
   public int Add(int n1, int n2)
   {
        return n1 + n2;
   }
       ....
}
```

Tracing service:

```
public TraceMethod {
    .....
  public void MethodEntry(MethodExecutionEventArgs args){
    Console.WriteLine("Entry: " + Format(args));
  }
  public  void MethodExit(MethodExecutionEventArgs args){
    Console.WriteLine("Exit: " + Format(args));
  }
  private string  Format(MethodExecutionEventArgs args){
    // Format
 }
}
```

Figuur 7: Calculator en tracing met AOP- Voorbeeldcode

De servicemethoden (`MethodEntry` en `MethodExit`) worden via de aspectklasse aan de targetklasse toegevoegd.

2.2.5Weaving

In principe kan weaving (samenvoegen) op basis van het tijdstip dat de weaving plaatsvindt, worden verdeeld in verschillende typen. De volgende typen zijn de meest besproken typen[30,31]:

- *Statische weaving*
  Deze weaving vindt plaats tijdens het compileren van de code naar het executabel bestand. De broncode van een targetklasse wordt tijden de compile- time aangepast door een of meerdere aspecten aan de klasse toe te voegen. AspectJ en PostSharp zijn static weavers.
- *Dynamische weaving*
  Deze weaving vindt plaats tijdens het uitvoeren van de executabel bestand. De toevoeging van de aspectklassen aan de targetklassen vindt in run-time plaats. Rapier-LOOM.NET[14] is een dynamische weaver.

Hoofdstuk 3

Probleemstelling

3.1 Probleemomschrijving

We kunnen het onderzoeksprobleem in tweeën verdelen. Het abstracte gedeelte beschrijft algemene problemen die we in de huidige DDD implementaties tegenkomen. Het concrete gedeelte beschrijft specifieke problemen van de huidige implementaties binnen Sogyo.

3.1.1 Abstract

Een probleem bij het toepassen van DDD is dat het domein en de services in de praktijk vaak sterk aan elkaar gekoppeld zijn. Soms hebben de domeinklassen kennis van de serviceklassen. En de serviceklassen gebruiken meerdere klassen of methoden uit het domein direct. Bij een applicatie met sterke koppelingen wordt de code minder onderhoudbaar en herbruikbaar. Daarnaast is het ook moeilijk om de code te testen.
Het verbeteren van deze kwaliteitaspecten (onderhoudbaarheid, herbruikbaarheid en testbaarheid) zou resulteren in betere kwaliteit van de code en kortere ontwikkel- en onderhoudstijd. Om dit doel te bereiken, moeten de verschillende onderdelen van een applicatie zoveel mogelijk onafhankelijk van elkaar zijn, zogenaamde *low coupling*. Maar die onderdelen moeten wel met elkaar communiceren om een werkbare applicatie te realiseren.
Er zijn een aantal technieken, waaronder Layered architecture en Design patterns zoals Factory, Repository, Composite en Strategy die het DDD concept kunnen ondersteunen. Hiernaast gebruikt men ook *event mechanisme*. Deze technieken proberen diverse verantwoordelijkheden van een applicatie te scheiden. Maar deze technieken hebben tekortkomingen. Zo is het domain layer in een layered architecture[25] afhankelijk van de onderliggende infrastructure layer. Maar Sogyo streeft naar een architectuur waarbij het domein totaal geen referentie naar de services heeft en die architectuur is moeilijk met deze technieken te realiseren. Sogyo stelt een *tussenlaag* voor om de communicatie tussen het domein en de services te bewerkstelligen.

In 2005 werd binnen Sogyo een onderzoek uitgevoerd naar de mogelijkheden om het domein en de services in Domain Driven Design op een betere manier te koppelen. Uit dat onderzoek is voorgesteld dat Aspect Oriented Programming mogelijk een oplossing kan bieden voor de koppeling van de service aan het domein. Dit onderzoek beperkte zich tot de Java omgeving. Hiernaast werden ook nog twee Design patterns, namelijk Adapter en Proxy (in combinatie met andere patterns), als mogelijke oplossingen gepresenteerd.

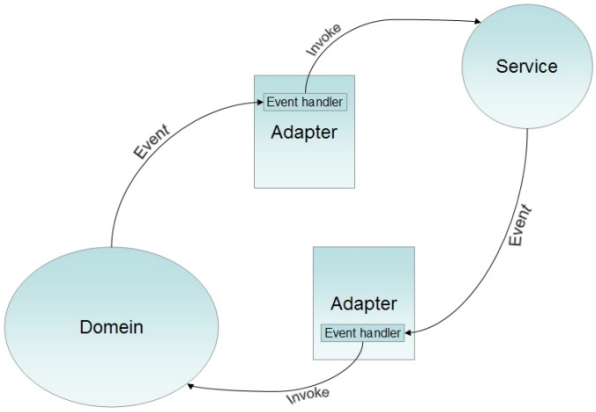
AOP zou dus de mogelijkheid bieden om de koppeling tussen het domein en de services op een betere manier te bewerkstelligen. In een dergelijk geval werkt AOP als laag tussen het domein en de services en vervangt de huidige adapterlaag.
Omdat Sogyo op dit moment weinig kennis over de toepasbaarheid van AOP in de .NET omgeving beschikt, wil het bedrijf hier meer over weten.

3.1.2 Concreet

Er zijn een aantal casestudies met betrekking tot Domain Driven Design binnen Sogyo aanwezig. Deze casestudies worden gebruikt om de medewerkers op te leiden en presentaties te geven bij diverse instellingen. Hieronder worden twee casestudies besproken. Hoewel beide implementaties van adapters gebruikmaken, zijn ze op verschillende manieren geïmplementeerd. Een adapter verzorgt de vertaling van de communicatie tussen verschillende (niet compatibele) objectsoorten. Hoewel het adapter pattern wel een oplossing biedt om het DDD concept te realiseren, heeft dit pattern ook tekortkomingen. Dit onderzoek is op de Bankcase gebaseerd, maar we zullen hieronder ook de Robotcase bespreken.

Robotcase
Figuur 8 toont de globale architectuur van de Robotcase. De koppeling tussen het domein en de services wordt met behulp van de adapters geregeld. De communicatie vindt via *events* plaats. Bij een verandering in het domein of een service wordt een event gestuurd en dat event wordt door één of meerdere geabonneerde adapter(s) ontvangen. De *Event-handler* in de adapters reageert op dat event en doet rechtstreeks aanroepen op serviceobjecten. Het domein heeft geen enkele kennis van de adapters en de services, en vuurt alleen events(met eventuele argumenten) bij een verandering af. De adapters hebben zowel de kennis van de domein- als van de serviceklassen. De serviceklassen kunnen kennis van het domein hebben, maar het is niet noodzakelijk(afhankelijk van een service).

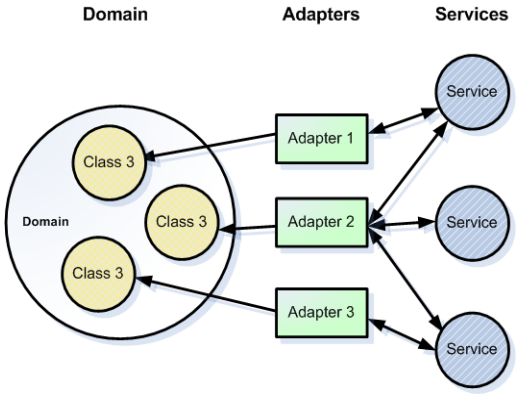
Voorbeeld
In de Robotcase vuurt de `Motion` klasse bij een verandering de event `Changed(this)` af en dit event wordt door twee geabonneerde adapters(`MotionPresentationOutputAdaptor` en `MotionMotorOutputAdapter`) ontvangen. Alle adapters hebben de methode `Update(object sender)`. Deze methode leest het meegestuurde `this,` oftewel `Motion` object uit en roept servicemethoden aan. Deze servicemethoden gebruiken (de properties van) `Motion` object als input.
Op dezelfde manier worden ook de service-events richting de domeinobjecten afgehandeld. Domeinspecifieke functionaliteit wordt door adapters aangeroepen.



Figuur 8: Sogyo Robotcase m.b.v. adapter pattern

Er is ook een alternatieve manier[5] die erg complex is. Daarbij wordt naast het adapter pattern ook Command pattern en Broker pattern toegepast. Het gebruik van meerdere Design patterns kan voor de (beginnende)ontwikkelaars voor minder begrijpbare code zorgen, bovendien verhoogt deze patterns de hoeveelheid code. Dit alternatief beschouwen de domeinexperts van Sogyo als een puristische implementatie die in de praktijk minder geschikt is. In een grote applicatie kunnen deze patterns voor extra complexiteit zorgen, waardoor de begrijpbaarheid van de code minder zal zijn.

Bankcase
Figuur 9 geeft een globale structuur van de Bankcase weer. De oorspronkelijke versie van de Bankcase is in Java gemaakt. Ook de Sogyo Bankcase[24] maakt gebruik van de adapters. Hierbij wordt geen gebruik van de Command of Broker pattern gemaakt en wordt ook geen event-mechanisme gebruikt. Per domeinklasse wordt een adapter gemaakt. Deze adapterklasse werkt als *proxy* voor de domeinklasse, maar heeft niet dezelfde interface als die van de domeinklasse. De adapterklasse voegt de benodigde services aan de bijbehorende domeinklasse toe. De serviceklassen maken gebruik van adapters en vice versa. Maar de domeinklassen maken geen gebruik van adapter- en serviceklassen. Deze architectuur wordt verder in 6.3 toegelicht.



Figuur 9: Sogyo Bankcase m.b.v adapter pattern

3.1.3 Problemen

Dit gedeelte beschrijft de praktische tekortkomingen die bij het implementeren van de huidige DDD implementaties (gebaseerd op de Bankcase en Robotcase) optreden.

3.1.3.1 Duplicatie van code door cross-cutting concerns

Aangezien de cross-cutting concerns als exceptiehandeling, logging en authenticatie niet tot het domein behoren, kunnen deze concerns als services gezien worden. Deze services worden door verschillende adapterklassen/methoden herhaaldelijk gebruikt. Dat leidt tot duplicatie van de code en daarmee ook voor meer regels code. Omdat dit probleem een algemene tekortkoming van de huidige programmeertechnieken als OO is, kan dit ook voor de DDD applicaties gelden.

3.1.3.2 Sterke koppeling

Robotcase

De robotcase heeft deze tekortkoming in mindere mate waardoor Sogyo deze niet als een tekortkoming ziet.

Bankcase

Elke adapter gebruikt één domeinklasse en meerdere adapter- en serviceklassen. Een aanpassing aan een domein- of serviceklasse resulteert in de meeste gevallen tot aanpassing in de bijbehorende adapterklasse. Indien een nieuwe public methode of property aan een domeinklasse toegevoegd wordt, wordt ook de bijbehorende adapterklasse aangepast: een sterke koppeling.

3.1.3.3 Het domein of de services wordt vervuild.

Robotcase

Om de communicatie tussen het domein en de services te kunnen realiseren past men de domeinklassen en/of de serviceklassen aan. In de Robotcase worden *delegates* en *events* aan de domeinklassen toegevoegd om de events-handling te realiseren. Elke methode/property die een verandering naar buitenwereld bekend wil maken, bevat een event(b.v. `Changed` of `Notify`). Deze aanpassing kan als vervuiling van het domein gezien worden. De serviceklassen bevatten ook de event-mechanisme om de veranderingen in service kenbaar te maken.

Bankcase

De bankcase heeft deze tekortkoming niet.

3.1.3.4 Extra code en mogelijk minder begrijpbare code

De tussenlaag zorgt voor extra code, naast de domein- en servicecode. Indien de tussenlaag niet gebruikt wordt (dus een directe koppeling tussen het domein en de services), kunnen we met minder aantal regels code de applicaties ontwikkelen.

Robot case

De structuur van de tussenlaag waar meerdere design patterns(alternatieve manier[5]) en *events* gebruikt worden, kan ingewikkeld zijn. Voor de (beginnende)ontwikkelaars die deze technieken niet kennen, kan dit wellicht probleem opleveren voor het begrijpen van de code. Deze case bevat 7 adapterklassen(393

coderegels) en 4 domeinklassen(513 coderegels). Bovendien is er een extra helperklasse(56 coderegels) nodig om het abonneren van de adapters op events te regelen.
Bankcase
In de bankcase heeft de tussenlaag meercode dan de domeincode. In die tussenlaag worden ook meerdere klassen uit het domein en de services aangeroepen. De domeinklassen hebben in totaal 355 regels code, terwijl de adapterklassen 399 regels code hebben. Deze extra code kan de leesbaarheid van de applicatie verminderen.

3.2 Onderzoeksvraag

Doelen
Sogyo wil nieuwe ontwikkelingen en innovaties op IT gebied snel leren en als het mogelijk is, ook toepassen. Aangezien ze op dit moment over weinig kennis van AOP in de .NET omgeving beschikken, hoopt het bedrijf middels dit onderzoek meer kennis over de toepasbaarheid van AOP te verwerven. Aangezien de meeste applicaties binnen Sogyo volgens het DDD concept ontwikkeld worden, wil het bedrijf de problemen die zich op dit moment tijdens het ontwikkelen van de DDD applicaties voordoen, oplossen. Daarnaast wil het bedrijf een proof of concept waarin de mogelijkheden en beperkingen van AOP zichtbaar zijn.

We zullen onderzoeken of AOP als tussenlaag gebruikt kan worden om de koppeling tussen het domein en de services te realiseren. Hierbij vervangt de AOP laag de huidige adapterlaag en we zullen alle services van de huidige Bankcase ook in de AOP applicatie implementeren.
Ondanks een aantal tekortkomingen werkt het adapter pattern in de praktijk wel. We zullen onderzoeken of de AOP oplossing beter is dan de huidige adapter oplossing.

Proof of concept
De onderzoekresultaten moeten voor de geïnteresseerden zichtbaar zijn en moeten gevalideerd worden.
We zullen met behulp van proof of concept laten zien hoe men AOP in Domain Driven Design kan toepassen.
In de komende hoofdstukken wordt het proof of concept verder uitgebreid behandeld.

Hoofd onderzoeksvragen
De bovenstaande doelen leiden tot twee hoofdonderzoeksvragen.

- Is AOP toepasbaar om het Domain Driven Design concept te realiseren?

- Verbetert AOP de ontkoppeling van Domain Driven Design applicaties in vergelijking met de huidige implementatie en worden de huidige implementatieproblemen met behulp van AOP opgelost?

Subvragen
We kunnen de volgende subvragen definiëren.

- Wat zijn de huidige AOP tools?

Er zijn een aantal AOP tools in de .NET omgeving te gebruiken. Elke tool heeft voor- en nadelen en om de proof of concept te kunnen ontwikkelen, moeten we eerst een geschikte AOP tool vinden. We zullen onderzoeken welke tools beschikbaar zijn en welke tool de beste oplossing voor ons proof of concept biedt?

- Wat is de definitie van kwalitatieve software en hoe gaan we dat meten?

Om de kwaliteit van de twee applicaties te kunnen vergelijken, moeten we eerst bepalen wat een kwalitatief goede software is. Welke kwaliteitskenmerken gaan we meten en hoe? En hoe gaan we de beide applicaties met elkaar vergelijken?

- Hoe goed zijn de implementatietechnieken die binnen Sogyo gebruikt worden?

Op dit moment wordt het DDD concept gerealiseerd met het adapter pattern. We bestuderen aan de hand van de bovengenoemde cases hoe het adapter pattern gebruikt wordt en de bijbehorende problemen.

Hoofdstuk 4

Onderzoeksmethode

[17] beschrijft een methode om wetenschappelijk onderzoek te verrichten. Deze onderzoeksmethode kent zes stappen en we hebben geprobeerd in dit onderzoek deze methode toe te passen. In dit hoofdstuk bespreken we de onderzoeksmethode.

4.1 Onderzoeksmethode

4.1.1 Observeren

Literatuurstudie
De literatuurstudie wordt gebruikt om de kennis, die voor dit onderzoek relevant is, op te doen. De problemen die dit onderzoekt noodzakelijk maken worden geanalyseerd. De implementatieproblemen van de twee cases(Robot en Bank) worden bestudeerd en geconcretiseerd. Achtergrondinformatie over Domain Driven Design en Aspect Oriented Programming worden verschaft door middel van relevante papers, boeken en internetsites. Tevens worden de huidige AOP tools en validatie- mogelijkheden bestudeerd.

Plan van Aanpak
Deze fase wordt ook gebruikt om het plan van aanpak voor dit onderzoek op te zetten. We hebben gesprekken met de DDD experts van Sogyo gehouden om achter de huidige problemen te komen en de evaluatiecriteria op te stellen.

4.1.2 Hypothese

Aan de hand van de literatuurstudie van de vorige stap kunnen er een of meer hypotheses opgesteld worden. Tijdens een onderzoek worden deze hypotheses getoetst op correctheid.
Op basis van bestudeerde literatuur hebben we de volgende hypothese gedefinieerd.

AOP biedt een goede oplossing om het domein en de services te koppelen in Domain Driven Design.

Het vorige onderzoek[4] geeft aan dat er mogelijkheden zijn om de hulp van AOP tussenlaag te gebruiken en in de future work staat beschreven dat de AOP mogelijkheid nog verder onderzocht moet worden. In [20] wordt een mogelijkheid gepresenteerd waarin AOP binnen Domain Driven Design gebruikt wordt. Die mogelijkheden werden met behulp van op Java gebaseerde AspectJ gerealiseerd. In deze stap is het niet mogelijk om te zeggen of AOP beter dan de huidige adapteroplossing zou zijn. Hierover kunnen we geen literatuur vinden.

4.1.3 Design

De volgende stap van het onderzoek is het ontwerpen en bouwen van een experimenteersysteem. Dit systeem wordt als proof of concept gebruikt. De verbeterde versie van het proof of concept wordt als één van de eindproducten opgeleverd. Om het proof of concept te kunnen maken, hebben we de huidige AOP tools in de .NET omgeving bestudeerd en aan de hand van een aantal criteria drie tools met elkaar vergeleken. In hoofdstuk 5 worden deze tools verder behandeld.

4.1.3.1 Proof of concept

Dit design is gebaseerd op de Domain Driven Design theorie, waarin de domein- en serviceklassen zo min mogelijk met elkaar gekoppeld moeten zijn. Het proof of concept zal aantonen hoe men AOP binnen Domain Driven Design kan gebruiken. Daarnaast zal dat als een validatie middel voor dit onderzoek dienen.

Het proof of concept is voornamelijk gerealiseerd met behulp van de Microsoft technieken, namelijk C#.NET, Visual Studio 2005, .NET Framework 2.0. MS Access 2003 en Windows XP. Daarnaast is PostSharp 1.0 Beta gebruikt om het AOP onderdeel(tussenlaag) te realiseren. Deze proof of concept is gebaseerd op de Bankcase die door Sogyo gebruikt wordt.
Het proof of concept bestaat uit drie onderdelen: Domein, Service en AOP tussenlaag. De domeinklassen van beide applicaties(Adapter en AOP) zullen dezelfde zijn en de serviceklassen hebben een kleine aanpassing. Bij de AOP applicatie wordt de tussenlaag(adapterklassen) door de aspectklassen vervangen. In hoofdstuk 6 wordt dit proof of concept uitgebreider behandeld.

4.1.4 Experiment

Hierbij zullen we een vergelijking maken tussen de huidige adapterapplicatie en de AOP applicatie. We gebruiken software meeteenheden[18,19,23] (Software Metrics) om de applicaties te meten. De volgende meeteenheden van de beide implementaties worden gemeten en met elkaar vergeleken.
We maken gebruik van metriek-tools NDepend[21] en SourceMonitor[26].

- Lines Of Code (LOC)
- Cyclomatic Complexity (CC)
- Coupling Between Object classes (CBO)
- Response For a Class (RFC)

Dit proces is een iteratief proces, waarbij meerdere keren het experiment uitgevoerd wordt om de proof of concept steeds te verbeteren. We vragen ook feedback van de DDD experts van Sogyo.

4.1.5 Analyse

In deze stap worden de resultaten van het experiment geanalyseerd en gevalideerd. Om het proof of concept te evalueren moeten er evaluatiecriteria opgesteld worden. Deze criteria wordt in hoofdstuk 8 verder besproken. We zullen evalueren of de AOP applicatie aan deze criteria voldoet.

Validatie
Het proof of concept is een middel om aan te geven in hoeverre het onderzoek geslaagd is. We zullen de gedefinieerde criteria, die uit de Domain Driven Design principes en de huidige implementatieproblemen afkomstig zijn, toetsen op het proof of concept.

De gemeten resultaten van de software meeteenheden van de vorige stap worden gebruikt om dit onderzoek te valideren. Daarnaast definiëren we een aantal aanpassingsscenario's en die scenario's worden zowel op de AOP applicatie als op de huidige adapterapplicatie uitgevoerd. De derde manier is de Sogyo experts op het gebied van Domain Driven Design vragen of de oplossing aan hun eisen voldoet. Deze drie validatie methoden worden in hoofdstuk 7 verder uitgebreid behandeld.

4.1.6 Rapport

Het rapporteren van dit onderzoek vindt op twee manieren plaats.

Scriptie
Deze scriptie is openbaar. Alle geïnteresseerden hebben toegang tot deze scriptie. Deze scriptie zal in ieder geval beschikbaar zijn via Sogyo en de Universiteit van Amsterdam.

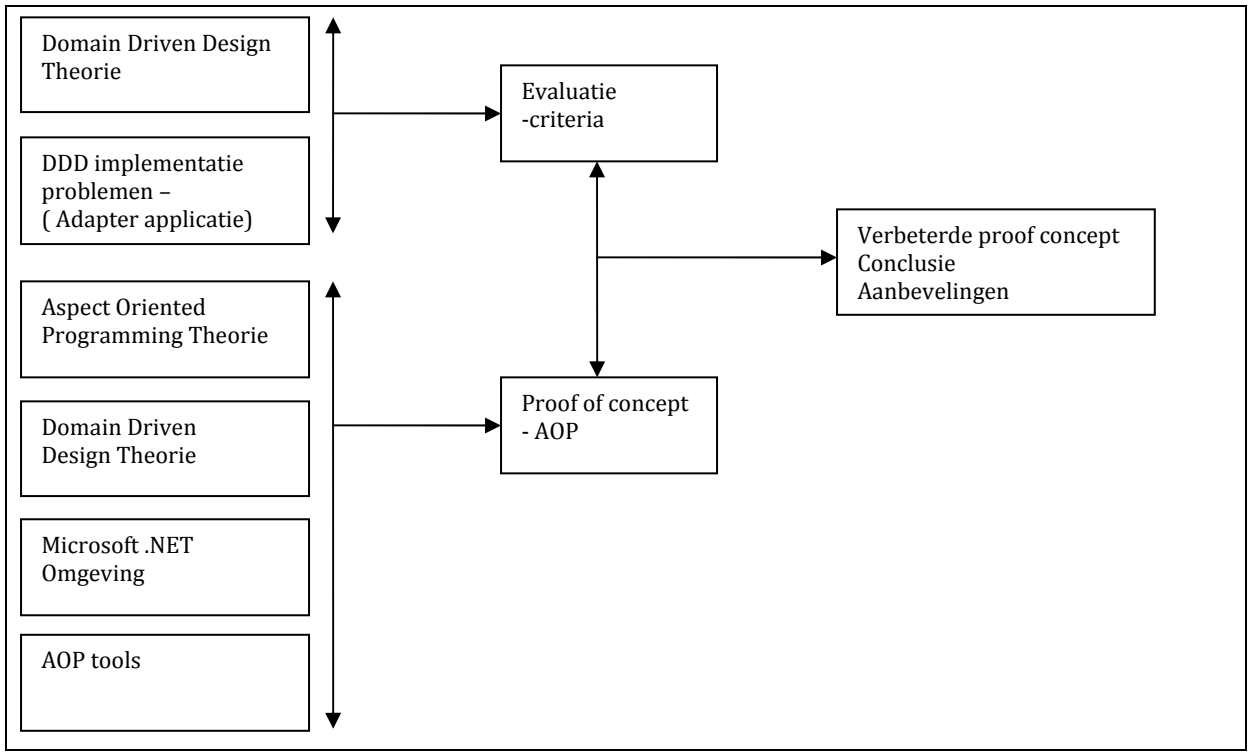
Presentatie
Daarnaast wordt ook een presentatie gehouden over de onderzoeksresultaten. De presentatie vindt zowel bij Sogyo als op de Universiteit van Amsterdam plaats.

4.2 Onderzoeksmodel

[22] heeft een methode beschreven om een onderzoek te modelleren. Het onderzoeksmodel geeft stappenplan tot het onderzoek. Hieronder geven we een korte toelichting over het gebruikte onderzoeksmodel(Figuur 10).

De evaluatiecriteria zijn gebaseerd op de Domain Driven Design theorie en de DDD implementatieproblemen (van de adapterapplicatie). Het proof of concept is gemaakt op de basis van de Aspect Oriented Programming theorie, Domain Driven Design theorie, Microsoft.NET omgeving en AOP tools.

De evaluatiecriteria worden op het proof of concept getoetst en er wordt een vergelijking gemaakt tussen de huidige applicatie en het proof of concept. Daaruit worden het verbeterde proof of concept, conclusie en aanbevelingen als de uiteindelijke producten van dit onderzoek geproduceerd.



Figuur 10: Onderzoeksmodel

Hoofdstuk 5

AOP tools

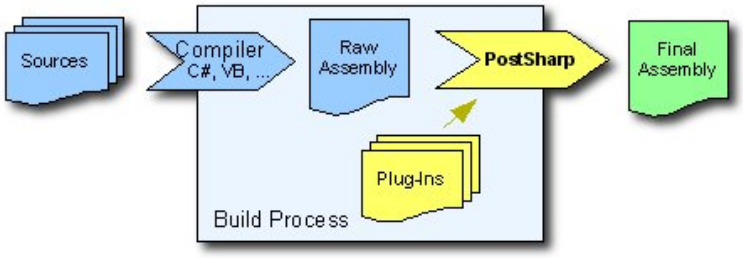
Dit hoofdstuk bespreekt drie AOP implementaties/tools die in de .NET omgeving beschikbaar zijn. Deze implementaties werden met elkaar vergeleken met behulp van een tracing voorbeeld. Alle drie de voorbeeldapplicaties bestaan uit een domeinklasse(`Calclulator`), aspectklasse(`TraceAspect`) en serviceklasse (`TraceMethod`).

De Java omgeving heeft een ontwikkelde AOP tool, namelijk AspectJ die in diverse artikelen te vinden is[1,11,20,30,31]. Maar in de .NET omgeving is er nog geen goed ontwikkelde tool. Alle tools die onderzocht zijn, hebben tekortkomingen.
Na een snel onderzoek op het internet en de gesprekken met een aantal Sogyo medewerkers bleek dat de volgende drie tools mogelijkerwijs betere tools zijn. Deze drie verschillende tools zullen een indruk schetsen over verschillende implementatietechnieken die op dit moment gebruikt worden, en de mogelijkheden en beperkingen die ze hebben. Het is echter wel mogelijk dat een niet onderzochte tool beter is dan deze tools. In 5.4 wordt een vergelijking gemaakt tussen deze drie tools.

5.1 PostSharp

PostSharp[13] is een bekende AOP tool voor de .NET omgeving en de tool wordt door één persoon ontwikkeld. Dit is een open source project en heeft een actieve gebruikersgroep op het internet.

Huidige versie:
1.0 Beta 2
04-03-2007



Figuur 11: PostSharp [13]

PostSharp is een statische weaver. Broncode wordt gecompileerd door een C#, VB of andere .NET gebaseerde compiler. De compiler maakt een assembly(.exe, .dll) bestand van de broncode. PostSharp voegt de aspectcode op de gedefinieerde invoegpunten (Join-points) in het target-bestand toe. Het uiteindelijke bestand bestaat zowel uit de broncode als aspectcode.

5.2 Aspect.NET

Aspect.NET[15] is een initiatief van de Microsoft Research Group. Auteur van deze implementatie is Vladimir Safonov, professor aan de St. Petersburg University. Naast de auteur zijn er ook een aantal studenten bezig met het ontwikkelen van deze implementatie.

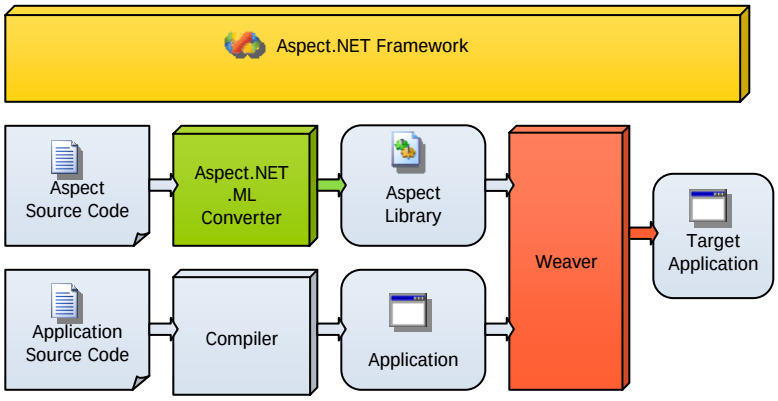
Aspect.NET is een statische weaver, de eerste versie (versie 1.0) van Aspect.NET is september 2005 uitgekomen. Aspect.NET maakt gebruik van de Microsoft Platform Research Development Kit(Phoenix). Aspect.NET kan geïnstalleerd worden als een add-on van Visual Studio 2005.

Figuur 12 toont het Aspect.NET Framework. De aspectcode wordt door middel van de Aspect.NET.ML converter gecompileerd. De applicatiecode wordt door middel van een .NET (C#, VB, etc) compiler

gecompileerd. Met behulp van de Aspect.NET weaver kunnen de aspects uit de aspect assembly in de applicatie assembly ingelast worden. De weaver genereert hiermee een nieuwe uiteindelijke assembly.

In tegenstelling tot andere AOP implementaties wordt hier een nieuwe assemby gegenereerd door de weaver. In de meeste AOP implementaties worden de aspects aan de bestaande targetapplicatie toegevoegd in plaats van een nieuwe samengestelde targetapplicatie te maken.

Huidige versie:
2.1
23-04-2007



Figuur 12: Aspect.NET Framework[15]

5.3 LOOM.NET

LOOM.NET[14] kent twee soorten implementaties.

- Rapier-LOOM.NET
  Dynamisch aspect weaver.
- Gripper-LOOM.NET
  Dit is een statische weaver en wordt niet verder ontwikkeld.

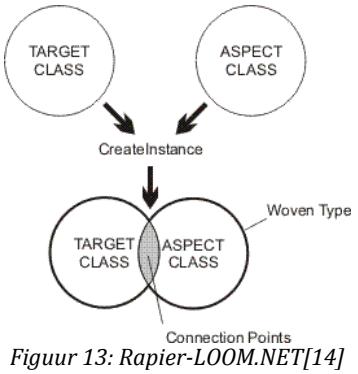
Deze implementaties worden ontwikkeld door Hasso-Plattner-Institute van University Potsdam.

Rapier-LOOM.NET
Bij Rapier Loom.NET worden de aspects in run-time aan de logicacode toegevoegd. Ook deze implementatie maakt gebruikt van Phoenix, maar de gebruiker hoeft Phoenix niet geïnstalleerd te hebben.

De weaver heeft een factory methode, genaamd `CreateInstance`. Deze methode levert een samengevoegd(targetklasse en aspectklasse) object op. In plaats van de `new` operator wordt dus de `Weaver`. `CreateInstance` methode gebruikt om een nieuw samengevoegd object te instantieëren.

Met de connection points van Rapier Loom.NET kan men join-points definiëren, waarin de aspectcode in de targetklasse ingevoegd moeten worden (figuur 13).

Huidige versie:
Rapier-LOOM.NET: 2.0
05-06-2007
Gripper-LOOM.NET: Technical Preview 0.1
06-12-2004



Figuur 13: Rapier-LOOM.NET[14]

5.4 Overige implementaties

De volgende implementaties zijn ook mogelijke kandidaten voor het implementeren van aspectoriëntatie in de .NET omgeving. Deze tools zijn niet onderzocht.

Tool	Laatste versie	Datum	Website
Puzzle.NAspect	Alpha release		www.puzzleframework.com
AspectDNG	0.9	21-09-2006	aspectdng.tigris.org
AspectSharp	2.1.1	15-05- 2005	sourceforge.net/projects/aspectsharp
DotSpect (.SPECT)	1.0	19-01-2006	dotspect.tigris.org
Spring.NET Framework	1.1 RC 1	10-08-2007	www.springframework.net
Wicca	2.0 alpha	26-05-2007	www1.cs.columbia.edu/~eaddy/wicca
EOS Compiler Tool	0.3.4	30-03- 2006	ect.jate.hu

Tabel 1: Overige AOP tools

5.5 Vergelijking

In dit gedeelte zal een vergelijking gemaakt worden tussen de drie AOP implementaties. Hoewel sommige eigenschappen van de drie implementaties gelijk zijn, is het onderliggende mechanisme per implementatie verschillend.

Issues

- Bij de vergelijking is alleen rekening gehouden met een simpele voorbeeldapplicatie. Bij grotere en complexe applicaties kan dit eventueel leiden tot verandering in de evaluatie.
- Elke implementatie heeft eigen naamgeving voor de adviesvormen en join-points en dat maakt de vergelijking moeilijk.

	PostSharp	Aspect.NET	Rapier LOOM.NET
Laatste versie	1.0 Beta 2	2.1	2.0
Laatste release datum	28-05-2007	23-04-2007	05-06-2007
Eerst release	-	18-09-2005 (versie 1.0)	11-03-2003(Versie 1.0)
Type weaver	Statische	Statische	Dynamische
Ondersteunende constructies	Constructor Method Property	Constructor Method	Method
Open source	Ja	Nee	Ja
Aanpassing aan logicacode / servicecode nodig	Nee	Ja. - Servicemethoden moeten static zijn.	Ja - Logicamethoden moeten overerfbaar zijn.(virtual) 0f een interface overerven
Debuggen	Ja	Nee	Ja
Release verschil	2 a 3 builds per week. Huidige versie is nog steeds een beta versie.	Gemiddeld elke halfjaar.	Het verschilt, gemiddeld per jaar.

Tabel 2: Vergelijking - AOP tools

Adviesvormen

	PostSharp	Aspect.NET	LOOM.NET
Voor (Before)	+	+	+
Na (After)	+	+	+
In plaats van (Instead)	+	-	+
Exceptiehandling (AfterThrowing)	+	-	+ [1)]
Na goed afloop (AfterReturning)	+	-	+ [1)]

Tabel 3: Adviesvormen - AOP tools

\+ = wordt ondersteund

\- = wordt niet ondersteund

[1)] Aspectmethode moet het type van de terugkeerwaarde / exceptie van de targetmethode als parameter hebben. Bij Postsharp is dit niet nodig.

Voorbeeld:
```
public void Foo2([JPRetVal] string retval, string name)
```
Targetmethode resulteert de type **string** als terukeerwaarde.

5.5.1 Criteria

1. Service of logicacode moet zo min mogelijk aangepast worden om AOP te ondersteunen.
Voor de PostSharp voorbeeldapplicatie hoeft men de logicacode of de servicecode niet aan te passen. Bij het Aspect.NET voorbeeld moeten alle servicemethoden *static* zijn. Bij het Loom.NET voorbeeld moet de logicamethoden overerfbaar (*virtual*) zijn of de domeinklasse moet een interface overerven.

2. Debuggen
Bij PostSharp en Rapier.LOOM.NET is het mogelijk om te debuggen. Het is mogelijk om op een willekeurige plek in een klasse een breakpoint te zetten. Het is niet mogelijk om een Aspect.NET applicatie te debuggen. We hebben hierbij de Visual Studio debugger gebruikt.

3. Open source
Aangezien de huidige tools nog niet doorontwikkeld zijn, kunnen we 'closed source' als een issue zien. PostSharp en Loom.NET zijn open source projecten. We kunnen de broncode downloaden en naar onze wens de implementatie aanpassen. Daardoor is de gebruiker niet volledig afhankelijk van de ontwikkelaars van de implementaties. Wanneer de ontwikkelaars met de ontwikkeling van hun projecten stoppen, kunnen we alsnog deze implementaties verder ontwikkelen en gebruiken. Broncode van Aspect.NET is niet beschikbaar. Bovendien heeft PostSharp een actieve gebruikersgroep op het internet.

4. Stabiliteit
PostSharp en Loom.NET schijnen stabieler te zijn dan Aspect.NET. We hebben tijdens het experiment van deze twee implementaties geen onbekende neveneffecten of errors ondervonden. Daarentegen is Aspect.NET niet stabiel. Het programma crasht tijdens het invoegen van aspecten aan de logicacode, vaak door onbekende redenen en resulteert in het opnieuw starten van de applicatie en/of Visual Studio ontwikkelomgeving.

5.6 Issues

Dit gedeelte bespreekt een aantal algemene issues van Aspect Oriented Programming, waarmee men rekening moet houden.

1. Tooling
Een betrouwbare tool is misschien het grootste issue dat we nu op dit moment hebben. Er zijn talloze tools in de .NET omgeving, maar die staan allemaal in de onderzoekfase en zijn niet productierijp. Wellicht is PostSharp een productieklaar product, maar de huidige versie is nog steeds in de beta fase. In vergelijking met AspectJ hebben deze tools minder functionaliteit.

2. Adaptatie
Adaptatie van AOP zal komende paar jaren een issue zijn. Het AOP concept is redelijk nieuw. Er zijn zeer weinig concrete casestudies en documentatie van AOP tools in de .NET omgeving te vinden. Er is ook op dit moment minder actieve participatie van gebruikersgroepen. De gebruikers zullen niet zo snel durven om AOP te gebruiken, aangezien er minder adaptatiemogelijkheden (documentatie, casestudies en tooling) zijn.

3. Kennis
Om AOP te kunnen toepassen moet men kennis over het concept beschikken. De meeste ontwikkelaars kunnen in het begin moeilijk hebben met het begrijpen van dit concept. Daarnaast moet men precies weten hoe de AOP tools toegepast kunnen worden. Daarbij is bijvoorbeeld te denken aan het definiëren van point-cuts(met behulp van expressies) en het implementeren van de daarbij behorende adviescode.

5.7 Conclusie

We hebben in dit hoofdstuk drie verschillende implementaties in de Microsoft.NET omgeving bekeken. Elke tool heeft voor- en nadelen. PostSharp biedt op dit moment relatief betere productieklare tools. Daarom kiezen we PostSharp om het proof of concept te ontwikkelen. Zoals eerder is aangegeven is de huidige versie nog steeds een beta versie. Het kan wel duren voordat een productieklare tool op de markt komt. Tot die tijd moeten we de ontwikkelingen rond AOP tools in de gaten houden.

Hoofdstuk 6

Proof of Concept

In dit hoofdstuk bespreken we het gemaakte proof of concept. Dit proof of concept biedt ons de mogelijkheid om de onderzoekresultaten te valideren en zichtbaar te maken.

De proof of concept, Bankcase, is gemaakt met behulp van de volgende technieken.

- Microsoft.NET Framework 2.0
- Visual Studio 2005
- Microsoft Access 2003
- C#.NET
- PostSharp[13] (AOP tool)

Het proof of concept is gemaakt op basis van het Domain Driven Design theorie. Deze applicatie is een consolegebaseerde desktop applicatie. Deze kleine en simpele applicatie wordt binnen Sogyo vooral voor opleidingsdoeleinden gebruikt.

6.1 Domein

Het domein van de bankcase bestaat uit vier klassen en alle vier klassen implementeren interfaces.

- Bank
- Client
- Account
- Transaction

Figuur 14 geeft het klassendiagram van de domeinklassen van de bankapplicatie. Deze domeinklassen bevatten alleen businesslogica functionaliteit. De gebruikers van de applicatie kunnen nieuwe klanten en rekeningen maken. Er kan geld gestort, opgenomen of overgemaakt worden en de transacties worden bijgehouden. De gebruikers kunnen overzicht van de rekeningen en de transacties opvragen.

6.2 Service

Serviceklassen bevatten niet- businessspecifieke onderdelen van de applicatie. De volgende services zijn geïmplementeerd.

- **Logging**

Deze service houdt alle uitgevoerde methoden van de domeinklassen bij. Deze informatie van de methode bevat de methodenaam, argumenten en de eventuele return value. Naast de methoden zijn ook de constructors voorzien van deze service, maar voor de simpelheid tellen we de properties(getters en setters) niet mee. De logging service was niet geïmplementeerd in de huidige adapterapplicatie. Aangezien logging één van de meest gebruikte AOP voorbeelden is, is besloten om deze alsnog te implementeren.

- **Security**

Om de applicatie te kunnen gebruiken moet de gebruiker geautoriseerd zijn. Nadat de applicatie opgestart is krijgt de gebruiker de mogelijkheid om het gebruikersnummer en wachtwoord in te vullen. Deze beveiligingservice is beperkt. Over het algemeen kan voor een uitgebreide service gekozen worden, onder andere met diverse rollen(*Rolset*) te werken.

- **Persistentie**

De businessobjecten worden in een MS Access database opgeslagen. De database bestaat uit drie tabellen. Client, Account en Transaction.

- **Exceptiehandeling**

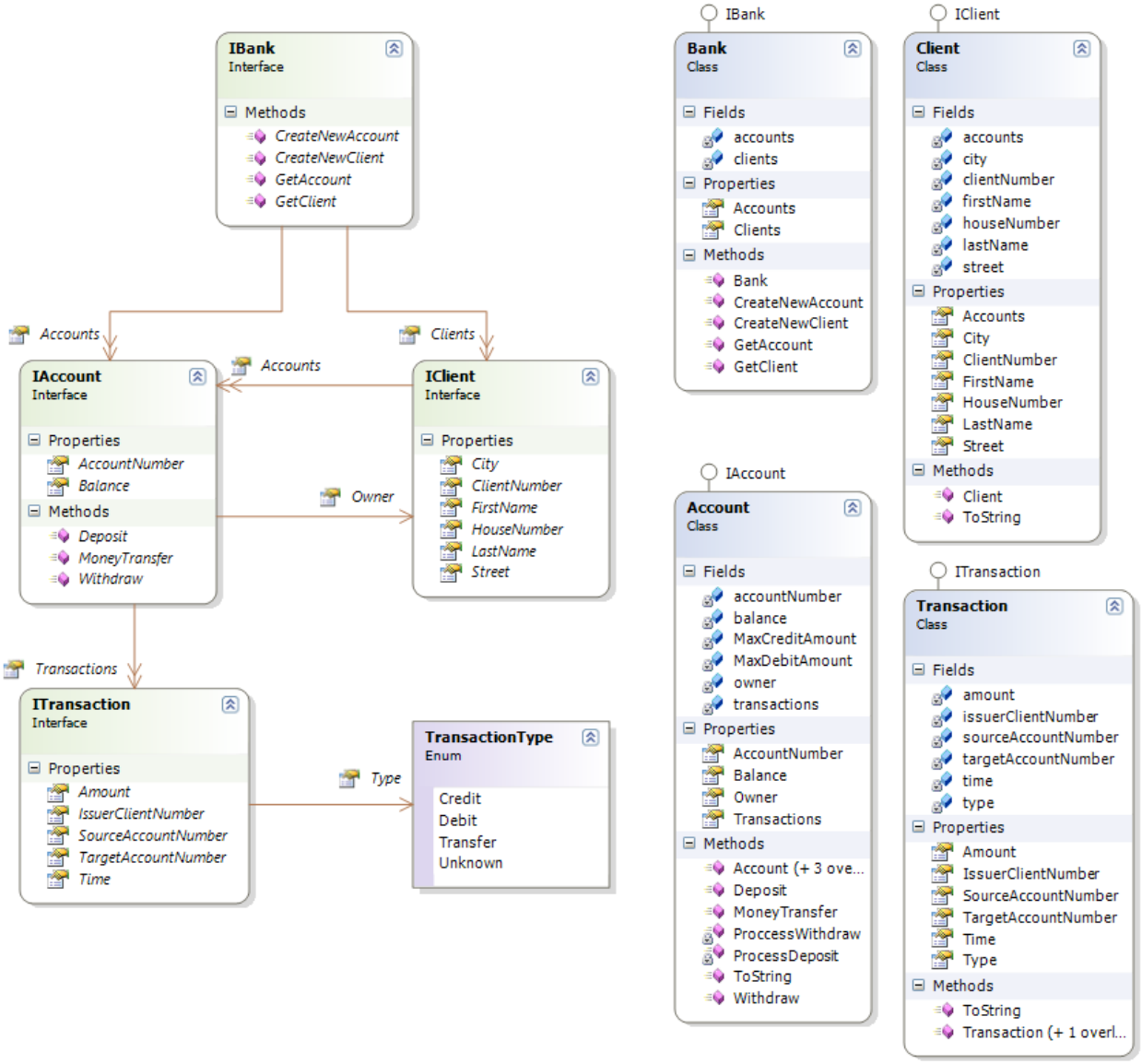
De exceptiehandeling vindt op een hoger niveau plaats. De excepties die vanuit een domeinklasse of een andere serviceklasse gegooid worden, worden in de presentatieservice opgevangen en daarin afgehandeld. De presentatieservice maakt gebruikt van de exceptieservice om deze excepties af te handelen.

- **Presentatie**

Het domein wordt via de presentatieservice beschikbaar gesteld aan de gebruikers. Zoals eerder is aangegeven wordt dit met behulp van een console applicatie gerealiseerd.

- **Validatie**

De domeinobjecten worden via deze service gevalideerd. Zo wordt gecheckt of de voornaam van een klant minimaal 2 letters heeft.



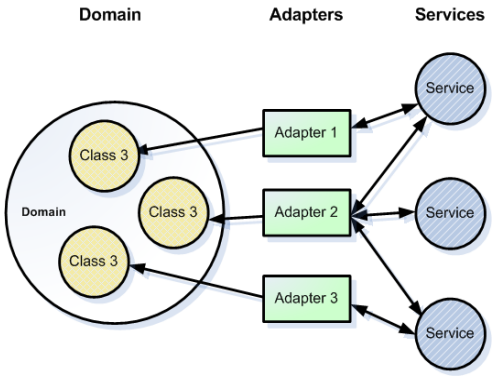
Figuur 14: Class diagram – Het domein van de Bankcase

6.3 Adapter architectuur

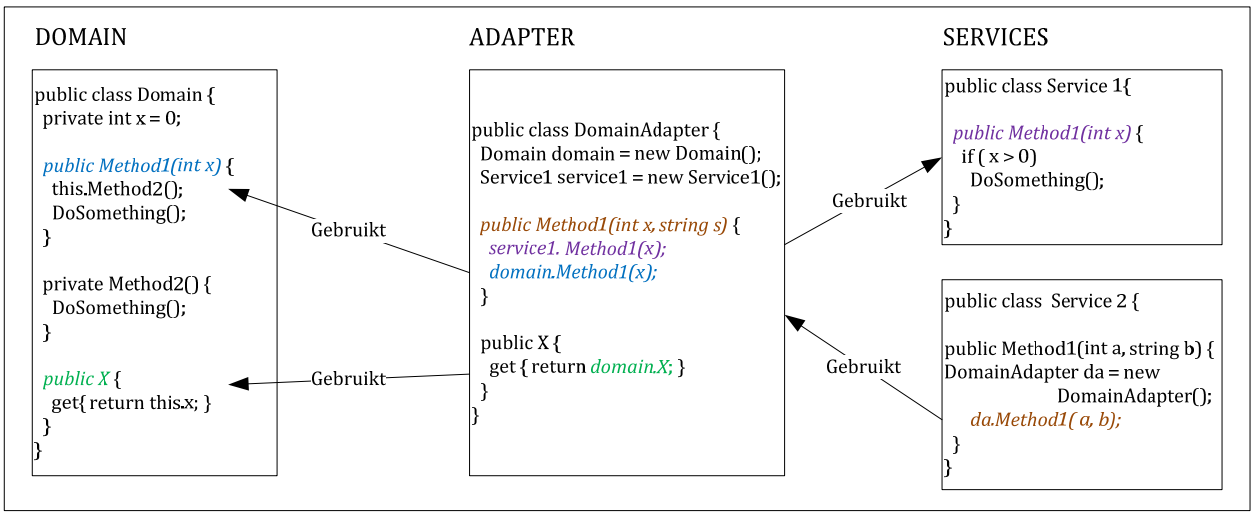
In dit gedeelte wordt de huidige architectuur van de Bankcase besproken. Hoewel de tussenlaagklassen als adapters genoemd worden, hebben ze overeenkomsten met het Proxy pattern. Net als het Proxy pattern

wordt bij de Bankcase de domeinklassen binnen de adapterklassen gebruikt. Bij het Proxy pattern heeft zowel het echte object en het proxy-object dezelfde interface, maar bij de Bankcase is dit niet het geval.

Elke domeinklasse heeft een adapterklasse (b.v. `Client` klasse heeft de adapterklasse `ClientAdapter`), in totaal zijn er dus vier adapters in de adapterapplicatie. Elke adapterklasse gedraagt zich als 'proxy' voor de bijbehorende domeinklasse. De serviceklassen gebruiken de domeinklassen niet direct, in plaats daarvan worden de domeinklassen via de adapterklassen aan de serviceklassen beschikbaar gesteld. De adapterklassen gebruiken de serviceklassen om de services aan het domein te koppelen. Figuur 15 geeft een globale architectuur van de adapterapplicatie weer. En figuur 16 geeft de architectuur met behulp van de programmeercode, op het klassenniveau weer.



Figuur 15: Adapter architectuur1

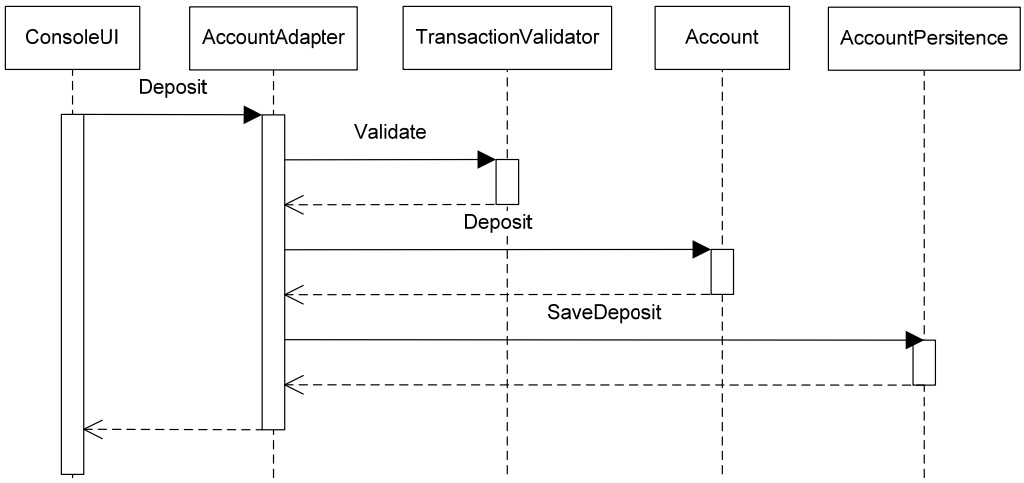


Figuur 16: Adapter architectuur 2

6.3.1 Voorbeeld – Geld storten

In dit gedeelte bespreken we de huidige bestaande adapterarchitectuur aam de hand van een voorbeeld. Hierbij zullen we de actie 'geld storten' met behulp van een sequence diagram(figuur 17) toelichten.
Bij het geldstorten wordt eerst de `Deposit` methode van de klasse `AccountAdapter` vanuit de presentatieservice aangeroepen en deze activiteit bestaat uit drie delen.

- *Transactie gegevens valideren*
  De `AccountAdapter` klasse roept de `Validate` methode van de `TransactionValidator` klasse om de gegevens te valideren.
- *Geldstorting*
  Na de succesvolle validatie wordt de `Deposit` methode van de domeinklasse `Account` aangeroepen. Deze methode voert de geldstorting in het domein uit.
- *Transactie gegevens opslaan*
  Als derde actie wordt de `SaveDeposit` methode van de serviceklasse `AccountPersistence` aangeroepen om de gegevens te persisteren.



Figuur 17: Sequence diagram - Adapter voorbeeld

6.4 AOP architectuur

Domain Driven Design streeft naar zo min mogelijke koppelingen tussen het domein en de services. In deze architectuur proberen we deze richtlijn zoveel mogelijk te bewerkstelligen.

Mogelijkheid1: Een aspectklasse per domeinklasse
Zoals in 6.3 staat beschreven, bestaat de tussenlaag van de adapterapplicatie uit vier adapterklassen die de vier domeinklassen vertegenwoordigen. Kunnen we dan hetzelfde aanpak ook voor de AOP applicatie gebruiken waarbij elke aspectklasse een domeinklasse vertegenwoordigt? Daarbij voegt een aspectklasse de benodigde services aan de bijhorende domeinklasse toe. Bijvoorbeeld: De '`ClientAspect`' klasse zou dan bijvoorbeeld de validatie- en logging services aan de `Client` klasse koppelen. Dit zou betekenen dat een cross-cutting service als logging in alle vier aspectklassen apart gebruikt moet worden om de service aan de domeinklassen te koppelen. Maar daarmee lopen we het voornaamste doel en voordeel van AOP mis, namelijk de modulariteit en minder duplicatie van de broncode. Daarom gebruiken we deze aanpakt niet.

Mogelijkheid2: Een of meerdere aspectklassen per service
In het proof of concept wordt per service één of meerdere aspectklassen ontwikkeld en de AOP layer bestaat uit 7 aspectklassen en een utility klasse.

Niet- domeinspecifieke service
Bij iedere niet-domeinspecifieke service wordt één aspectklasse ontwikkeld. Zo hebben exceptiehandeling, authenticatie en logging eigen aspectklassen die de bijbehorende services aan het domein koppelen.

Per domeinspecifieke services wordt één of meer aspectklassen ontwikkeld.
Een aspectklasse per service
De validatieservice heeft bijvoorbeeld maar één aspectklasse en wordt voor alle domeinobjecten gebruikt. Deze aspectklasse heeft wel de extra verantwoordelijkheid 'filteren'. Per domeinobject wordt in de aspectklasse gekeken welke bijbehorende serviceklasse/methode gebruikt moet worden(figuur 18). Deze aspect kan voor meerdere domeinobjecten hergebruikt worden.

```
if (eventArgs.Instance.GetType() == typeof(Client))
{
  Client client = (Client)eventArgs.Instance;
  ClientValidator cv = new ClientValidator(client);
  if (!cv.Validate())
    throw new Exception("Validation has failed!");
 }

 if (eventArgs.Instance.GetType() == typeof(Account))
 {
  // ...
```

Figuur 18: Broncode - ValidationAspect

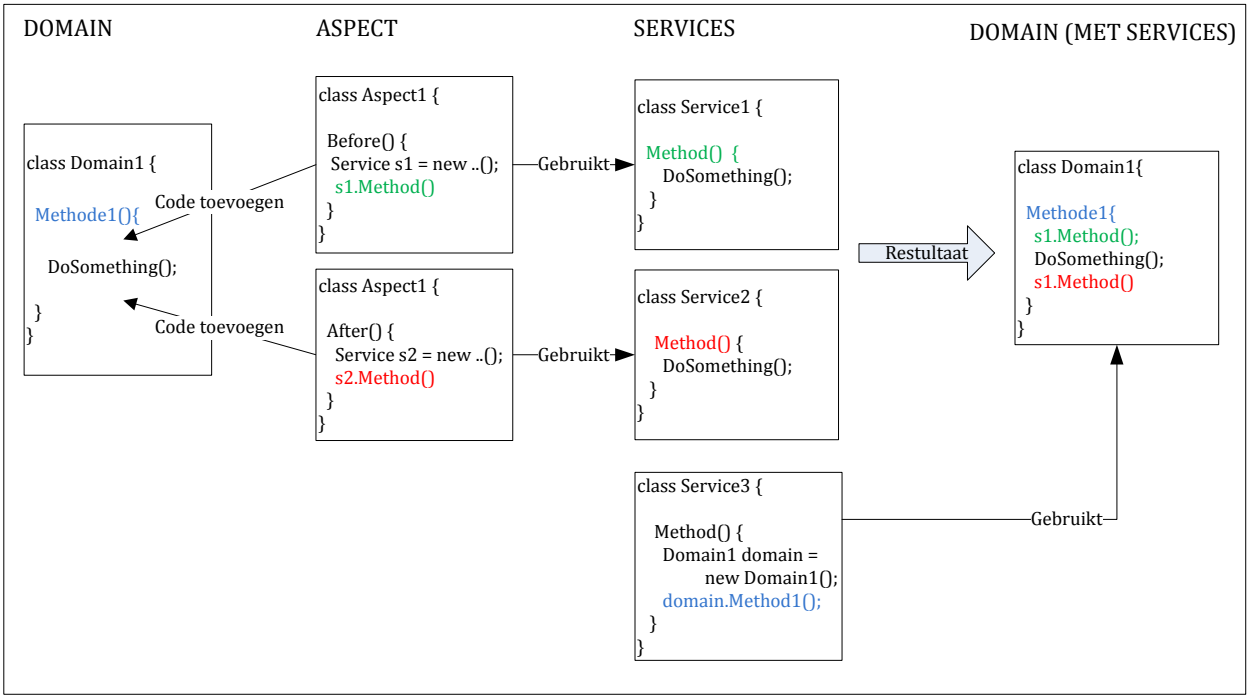
Deze aanpak zorgt wel voor minder aspectklassen en maar wel voor meer complexiteit (door meer decision points).

Meerdere aspectklassen per service
Er kan ook gekozen worden voor een combinatie van service- en domeinklassen om hier vervolgens een aspectklasse voor te ontwikkelen. Het voordeel hiervan is dat het filteren van domeinobjecten niet nodig is, maar dit kan wel weer leiden tot meer aspectklassen. Indien een nieuwe domeinklasse ontwikkeld wordt, die een bepaalde service nodig heeft, moet dan ook een nieuwe aspectklasse ontwikkeld worden.

We hebben gekozen om een combinatie van beide te gebruiken. Sommige services hebben maar één bijbehorende aspectklasse en sommige services hebben meerdere aspectklassen.

Architectuur
Figuur 19 schetst een architectuurdiagram op het klassenniveau waarbij twee servicemethoden via twee aspectklassen aan een domeinmethode toegevoegd worden. De domeinklasse waarin die twee services verwerkt zijn, wordt door een andere service (b.v. presentatieservice) gebruikt.

Bij het creëren van een klant hebben we bijvoorbeeld twee services nodig:
1 – validatie waarbij de door gebruiker ingevulde klantgegevens gecheckt worden.
2 – persistentie om de klantgegevens in de database op te slaan.
We gebruiken daarom de aspectklasse `ValidationAspect` voor de validatieservice en `CreateNewAspect` voor de persistentieservice. De aspectklassen zorgen voor de koppelingen van die twee services aan het domein.

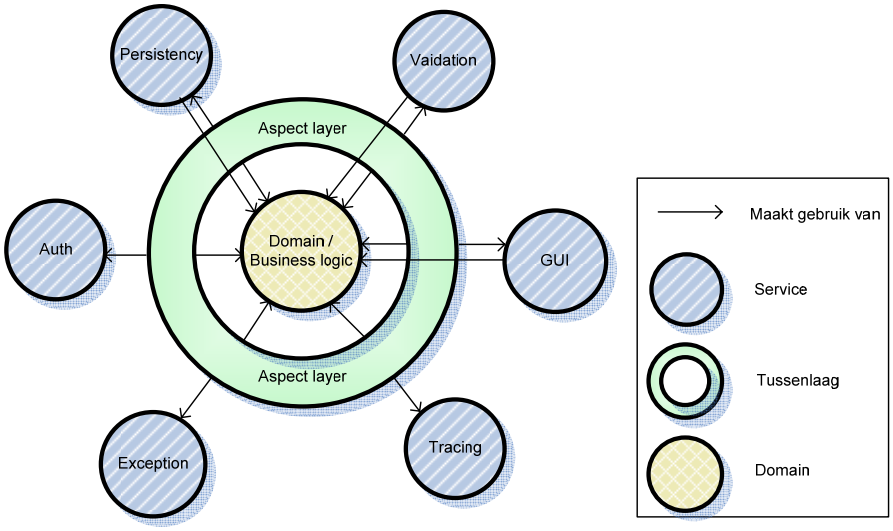


Figuur 19: AOP Architectuur 1

Domeinklasse
Zoals op de figuren 19 en 20 te zien is, gebruiken de domeinklassen van de AOP Bankcase noch de aspectklassen noch de serviceklassen. De domeinklassen hebben geen kennis van het bestaan van de services. De aspectklassen voegen functionaliteiten van services aan domeinklassen toe.

Aspectklasse
Een aspectklasse gebruikt maar één of meerdere serviceklassen van hetzelfde servicetype. Wanneer bijvoorbeeld een exceptie in een domeinklasse optreedt, wordt deze exceptie door een

aspectklasse(`ExceptionAspect`) met behulp van een serviceklasse(`ExceptionHandler`) afgehandeld. De `ValidationAspect` gebruikt twee serviceklassen. De aspectklassen kunnen kennis van domeinklassen hebben, maar dit is niet altijd noodzakelijk. Wanneer een niet- domeinspecifieke taak uitgevoerd moet worden, hoeven de aspectklassen de domeinklassen niet te gebruiken. Tracing, authenticatie en exceptiehandeling kunnen zonder specifieke kennis van het domein uitgevoerd worden.



Figuur 20: AOP Architectuur 2

Serviceklasse

De serviceklassen gebruiken de domeinklassen. Als een service een domeinspecifieke taak moet ondersteunen, moet de service hoe dan ook kennis van het domein hebben. De `ClientValidator` klasse heeft de kennis van de `Client` nodig om de klantinformatie te valideren. In de adapterapplicatie gebruiken de serviceklassen de domeinklassen via de adapters, terwijl in de AOP applicatie de domeinklassen direct gebruikt worden.

De serviceklassen maken van elkaar gebruik. Zo wordt bijvoorbeeld de presentatieservice door de exceptieservice gebruikt. Tijdens het afhandelen van een exceptie wordt de `ShowError` methode van de `ConsoleUserInterface` klasse uitgevoerd om de opgetreden fout aan de gebruiker te tonen. Deze afhankelijkheden worden echter niet op dit diagram getoond. De onderlinge relaties tussen de services zullen wel blijven.

Koppeling van aspects naar targetklassen

PostSharp biedt twee mogelijkheden om deze koppeling(*point-cuts*) te realiseren.

- **Attribuut**
  De aspecten kunnen als attributen boven de target-member geplaatst worden. Het voordeel is dat op die manier duidelijk te zien is welke aspecten aan een bepaalde target-member toegevoegd worden. Het nadeel is dat door deze aanpak het domein wordt 'vervuild'.
- **AssemblyInfo.cs bestand**
  De koppeling wordt in `AssemblyInfo.cs` geregeld. Deze manier maakt gebruik van expressies (b.v. `*.*`).

```
[TraceAspect]
public string FirstName
{
    get { return this.firstName; }
    set { this.firstName = value; }
}
```

```
[assembly: AopBank.Aspect.TraceAspect(
  AttributeTargetTypes = "Domain.*",
  AttributeTargetMembers = "*.*")]
```

Figuur 21: Koppeling van Aspects aan targetklassen

Er is voor gekozen om de 2de manier te gebruiken. Bij de 2de manier worden alle aspect-koppelingen(point-cuts) in één bestand geplaatst en dat is overzichtelijker. Tevens hoeven bij de 2de manier de cross-cutting aspecten niet meerdere keren aan de target-members gekoppeld te worden. Een nadeel van deze aanpak is wel dat `AssemblyInfo.cs` door de aspectdefinities 'vervuild' wordt. En indien er meerdere diverse

domeinspecifieke services aan het domeinklassen gekoppeld worden, kan het `AssemblyInfo.cs` bestand behoorlijk lang worden. Indien de naam van een domeinmethode aangepast wordt, moet deze aanpassing ook in dit bestand doorgevoerd worden, anders zal de desbetreffende service voor de domeinmethode niet beschikbaar zijn.

6.4.1 Voorbeeld – Geld storten

In dit gedeelte zullen we met behulp van een concreet voorbeeld de AOP applicatie bekijken. Bij het geldstorten wordt twee aspecten gebruikt.

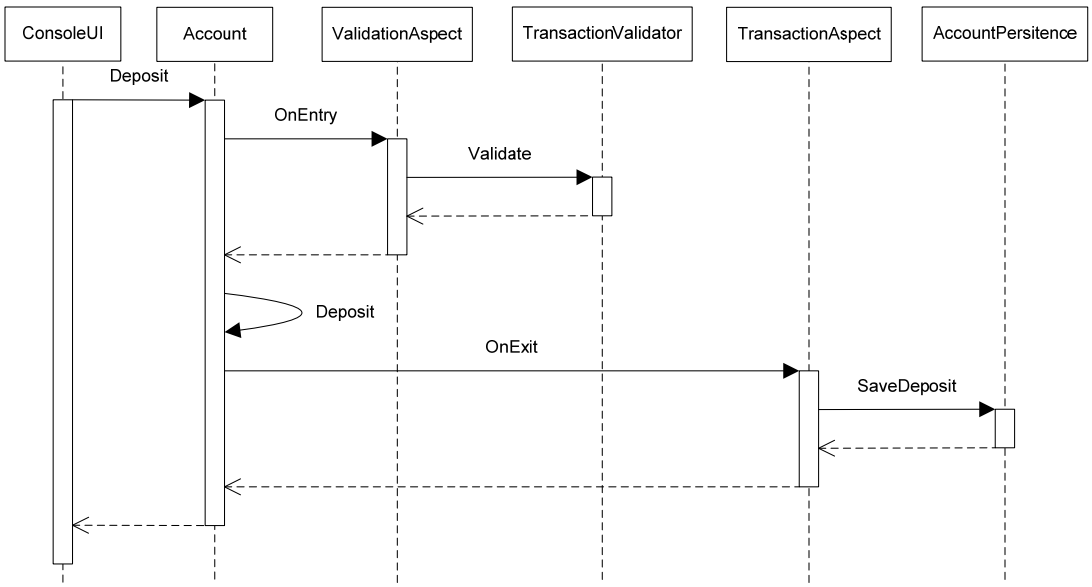
- ValidationAspect
  Deze aspectklasse controleert of een transactie wel uitgevoerd mag worden. Bijvoorbeeld, een gebruiker mag alleen op zijn/haar eigenrekening geld storten.
- TransactionAspect
  De ze aspectklasse slaat een transactie op in de database. Hierbij maakt deze aspectklasse gebruik van `AccountPersistence` klasse.

Deze twee aspecten kunnen naast voor geldstorting ook voor geldopname of overboeking gebruikt worden.

Sequence diagram

Hieronder geven we het hierboven beschreven proces met behulp van een sequence diagram weer(figuur 22). Bij het geld storten wordt de `Deposit` methode van de `Account` klasse door de `ConsoleUI` klasse aangeroepen. Zoals bij het adapter voorbeeld worden bij deze aanroep ook drie acties uitgevoerd.

- *Transactie valideren*
  Voordat de `Desposit` methode uitgevoerd wordt, wordt de `OnEntry` methode van de `ValidationAspect` aangeroepen. Deze roept op zijn beurt de `Validate` methode van de `TransactionValidator` klasse aan. Deze methode voert de validatie uit. Bij een invalide uitvoering wordt een exceptie gegooid en vervolgens wordt de volgende stap(geldstorting) niet uitgevoerd.
- *Geld storten*
  Bij een succesvolle validatie wordt de `Deposit` methode van de `Account` klasse uitgevoerd.
- *Transactie opslaan*
  Als de derde actie wordt de `OnExit` methode van de `TransactionAspect` klasse aangeroepen. Deze methode roept weer de `SaveDeposit` methode van de `AccountPersistence` klasse aan om de transactie op te slaan.



Figuur 22: Sequence diagram - AOP voorbeeld

6.5 Discussie

In de 6.4 gedefinieerde architectuur hebben we een poging gedaan om met behulp van AOP niet alleen de cross-cutting services maar ook andere domeinspecifieke services aan het domein beschikbaar te stellen. Over het algemeen wordt AOP alleen voor de cross-cutting concerns gebruikt. Met deze architectuur hebben we een bredere toepasbaarheid van AOP aangetoond. Zowel in de adapterapplicatie als in de AOP applicatie houden we het domein schoon. Het grote verschil is dat in de AOP applicatie de domeinklassen door de serviceklassen direct gebruikt worden, terwijl bij de adapterapplicatie de adapterklassen door de serviceklassen gebruikt worden.

Zoals eerder is aangegeven, is deze architectuur met behulp van PostSharp in C# gemaakt. Aangezien PostSharp ook ondersteuning voor Visual Basic.NET(VB.NET) biedt, zou het ook mogelijk zijn om VB applicaties met behulp PostSharp te ontwikkelen. Maar deze mogelijkheid hebben we in dit onderzoek niet bestudeerd en ook de AOP mogelijkheden in de overige .NET talen als C++.NET en J#.NET zijn ons niet bekend.

Hoewel de gemaakte AOP Bankcase tool-afhankelijk is, is het architectuur idee wel generiek. Het toevoegen van de services aan het domein zou met elke AOP tool mogelijk zijn. AOP tools bieden de mogelijkheid om met behulp van *Aspect* constructie benodigde concerns aan target-members(methode, property, constructor, etc.) toe te voegen.

Hoe aspects in een applicatie precies gebruikt worden, kan per project of per persoon verschillen. Deze verschillen hebben we ook in de Bankcase en Robotcase gezien. Beide applicaties gebruiken adapters, maar ze worden op verschillende manieren geïmplementeerd. Indien een bepaald framework als NHibernate of Webservices gebruikt wordt, zou men de aspectklassen op een andere manier organiseren dan in het proof of concept. Het zou ook mogelijk zijn om een tussenlaag te creëren die naast de aspectklassen ook adapters en/of event-mechanisme combineert.

Een mogelijk probleem bij het toepassen van deze architectuur in de praktijk is het gebrek aan AOP kennis. Bij het ontwikkelen van een op AOP gebaseerd systeem moet men ook over kennis van het AOP concept en de bijbehorende AOP tool(s) beschikken. Bovendien vereist AOP extra kennis voor de onderhoudsprogrammeurs van het systeem.
Wanneer een domein- of een serviceklasse aangepast wordt, moet dat ook in de point-cut definities verwerkt worden. Tijdens het ontwikkelen van het proof of concept hebben we de namen en de argumenten van meerdere methoden aangepast en vervolgens waren we vergeten om de aanpassing ook in `AssemblyInfo.cs` door te voeren. De compiler gaf geen foutmelding, maar de services waren voor de aangepaste methoden niet beschikbaar.

Hoofdstuk 7

Meten van de kwaliteit

7.1 Kwaliteitskenmerken van software

Kwaliteit is een subjectieve term. Een snelle kijk in het woordenboek resulteert in het volgende: *Bepaalde gesteldheid, hoedanigheid, mate waarin iets geschikt is om voor een bepaald doel gebruikt te worden.* Het probleem is dan ook: hoe gaan we de kwaliteit zo objectief mogelijk meten? Kwaliteit van software kan door een aantal kwaliteitsattributen gekenmerkt worden en non-functional requirements van een software zijn voornamelijk gebaseerd op deze kwaliteitsattributen. Door de meeteenheden die bij deze attributen behoren te meten, kunnen we in zekere mate een uitspraak doen over de kwaliteit van bepaalde software. Tevens is het ook mogelijk om twee applicaties met dezelfde functionaliteit met elkaar te vergelijken.

Met Domain Driven Design probeert men de koppeling tussen het domein en de service te verminderen. Dit onderzoek richt zich voornamelijk op het verminderen van de koppeling in DDD applicaties. Een applicatie met minder koppelingen zou makkelijk te onderhouden(onderhoudbaarheid), te testen(testbaarheid) en te hergebruiken(herbruikbaarheid) zijn. Naast deze kwaliteitskenmerken zijn er meer kwaliteitskenmerken zoals leesbaarheid, stabiliteit, security, performance en uitbreidbaarheid.
We passen de domeinklassen van de huidige adapterapplicatie, in het proof of concept niet aan. De serviceklassen worden alleen voor één doeleinde aangepast. In het proof of concept gebruiken de serviceklassen de domeinklassen direct, terwijl de serviceklassen in de adapterapplicatie de adapterklassen gebruiken. Het grote verschil in de twee applicaties is het vervangen van de tussenlaag die de koppeling verzorgt. Bij dit onderzoek zijn we niet geïnteresseerd in de kwaliteit van de domein- of serviceklassen, maar we zijn geïnteresseerd in de mindere koppeling tussen verschillende onderdelen van de applicatie. Daarom zullen we ons richten op de koppeling en de bijbehorende kwaliteitsattributen. De andere kwaliteitsattributen vallen buiten de scope van dit onderzoek.

Een applicatie met een sterke koppeling kan een negatieve invloed op de volgende kwaliteitattributen hebben.

- *Onderhoudbaarheid*
  Door de afhankelijkheden tussen de onderdelen(businesslogica en verschillende services) van een applicatie, wordt het moeilijk om aanpassingen aan te brengen. Aanpassing aan één onderdeel heeft effect op de andere onderdelen. Wanneer code met verschillende aspecten (b.v.: businesslogica, log en authenticatie) in één methode of klasse opgenomen wordt, wordt de desbetreffende class/methode complexer en onoverzichtelijker.
- *Herbruikbaarheid*
  Doordat de onderdelen sterk aan elkaar gekoppeld zijn, zijn ze applicatieafhankelijk. Daardoor kunnen stukken programmatuur niet voor een andere applicatie hergebruikt worden. Een domein dat niet infrastructuuronafhankelijk is, is makkelijk te hergebruiken, bijvoorbeeld zowel voor de Web als Windows omgeving.
- *Testbaarheid*
  Door de afhankelijkheden kan men niet afzonderlijk een deel(unit) van een applicatie testen. Dit maakt unit-tests moeilijk. Bijvoorbeeld een applicatie met een sterke koppeling tussen GUI en businesslogica is moeilijk te testen.

Naast de koppelingen meten ook het aantal code regels en de complexiteit van de tussenlaag.

7.1.1 Koppeling

Coupling:

> *Two objects are coupled if and only if at least one of them acts upon the other [19].*

Koppeling (Coupling)[19,27] wordt vooral gebruikt als een symbool om de kwaliteit van software te meten. Koppeling betekent in een objectgeoriënteerd design, de mate van afhankelijkheid tussen twee onderdelen (modules, klassen of methoden). *Low coupling* houdt in dat een bepaalde klasse of object zo weinig mogelijk weet van een andere klasse of object. Dat zou betekenen dat de klasse zo min mogelijk referentie naar de andere klassen heeft en zo weinig mogelijk methoden van een andere klasse aanroept, maar er is geen norm voor het aantal koppelingen. Een aanpassing aan een klasse moet weinig tot geen invloed op de andere klassen hebben. Low coupling heeft als doel het verbeteren van onderhoudbaarheid, leesbaarheid en herbruikbaarheid van software. Een applicatie met sterke koppeling is daarentegen moeilijk te onderhouden, testen en her te gebruiken. Daarom is de sterke koppeling ongewenst. Het concept Low coupling heeft een relatie met hoge samenhang (high cohesion), waarin de onderdelen van een klasse met elkaar zeer gerelateerd en op één taak gefocust zijn. In een ideale situatie heeft software een lage koppeling en hoge samenhang.

Er zijn diverse meeteenheden voor het meten van een koppeling voorgesteld. [19] maakt een lijst van vier koppeling meeteenheden. Deze meeteenheden worden in 7.2.1 uitgebreid behandeld.

Name	***Definition***
CBO	*Classes are coupled if methods or instance variables in one class are used by the other. CBO for a class is number of other classes coupled with it. [18]*
RFC	*Count of all methods in the class plus all methods called in other classes. [18]*
CF	*Classes are coupled if methods or instance variables in one class are used by the other. CF for a software system is number of coupled class pairs divided by total number of class pairs.*
DAC	*Data abstraction coupling. DAC for a class is the number of attributes having other classes as their types.*

Tabel 4: Koppeling meeteenheden

Een koppeling tussen twee klassen (klasse A en klasse B) kan om de volgende redenen ontstaan.

- Klasse A heeft een attribuut dat refereert naar de klasse B.
- Klasse A roept een methode van de klasse B aan.
- Klasse A heeft een methode die heeft referentie naar de klasse B (via return type of parameter).
- Klasse A is een subklasse van de klasse B.

Koppelingen kunnen beide richtingen van twee klassen zijn. Bijvoorbeeld, klasse A gebruikt klasse B, maar het is ook mogelijk dat klasse B klasse A gebruikt.

7.2 Validatie methoden

Onderzoeksresultaten kunnen op verschillende manieren gevalideerd worden. We zullen drie verschillende methoden toepassen, namelijk meeteenheden, aanpassingsscenario's en experts interviewen. Deze meeteenheden alleen zijn niet voldoende, ze geven maar een bepaalde indicatie. Men zegt bijvoorbeeld 'more code more errors', maar een applicatie met minder code kan wel weer complexer zijn omdat er bijvoorbeeld veel decision points (if, else, while, etc) zijn. Daarom is er naast de meeteenheden ook voor de andere twee validatie-methoden gekozen.

7.2.1 Meeteenheden

We zullen vier bekende meeteenheden gebruiken. We gebruiken één van de oudste metrieken, namelijk het aantal regels code(Lines of Code) en de bekende metriek Cyclomatic Complexity van Thomas McCabe[12] om de complexiteit van de tussenlagen van beide applicaties te vergelijken. Chidamber en Kemerer [18] hebben een verzameling van meeteenheden geïntroduceerd. Coupling Between Object classes (CBO) en Response For a Class (RFC) zijn twee meeteenheden die gebruikt worden om de koppelingen van software te meten en worden door diverse papers[19,23,27] gerefereerd. Hiernaast zijn er ook nog twee meeteenheden, CF en DAC, die door [19] opgesomd zijn. CF is te verglijken met CBO, maar CF wordt op systeem of namespace niveau gemeten, i.p.v. klassenniveau. DAC telt alleen de data types die in een klasse als attribuut gedefinieerd staan,

terwijl de CBO metriek 'alle type koppelingen' meetelt (zie hier verder bij b), de 3de metriek), waardoor de CBO metriek voor dit onderzoek meer nauwkeurigheid kan opleveren.

a) Lines Of Code (LOC)

Het aantal regels code van een software kan gemeten worden door een bekende en traditionele metriek *Lines Of Code*[27]. LOC is een van de oudste software meeteenheden en wordt vooral gebruikt om een indicatie te geven hoe groot een software project is. Software projecten kunnen variëren van 100 regels tot soms wel 100 miljoenen regels. LOC van een klasse geeft een indicatie van de grootte van de klasse.

Hoe meten

LOC klinkt als een simpele metriek. Maar afhankelijk van wat men wilt meten kan de waarde van deze meeteenheid verschillen.
LOC kan op verschillende manieren gemeten worden. We hanteren dezelfde codestandaard (lay out) zowel voor de adapter applicaties als de AOP applicatie. Wanneer verschillende lay-outs gehanteerd worden, kan het aantal regels code variëren en leiden tot misinterpretatie.

```
using System;
using System.Text;

namespace Example
{
    public class Calculator
    {
        // Add two numbers
        public int Add(int n1, int n2)
        {
            return n1 + n2;
        }
    }
}
```

Figuur 23: NOC voorbeeld

- Source Lines of Code

Deze metriek wordt ook als 'Physical Lines of Code' genoemd en telt het aantal code regels van een applicatie. Dit type LOC bestaat uit statements, commentaarregels en blanke regels.

Source Lines of Code	14
Statements	6

Tabel 5: LOC voorbeeld - waarden

- Statements

Deze metriek wordt ook 'Logical Lines of Code' genoemd. In deze metriek worden alleen de statements van een applicatiecode opgeteld. In ons geval bestaat dit type LOC waarde uit C# statements. Blanke regels en commentaarregels worden niet meegeteld.

Deze 'Physical' en 'Logical' LOC waarden kunnen afhankelijk van de gebruikte lay-out van de code, erg verschillen. Deze waarden zijn ook afhankelijk van de ontwikkelaars en de gebruikte programmeertaal. Een programma met dezelfde functionaliteiten kan door een ervaren programmeur met minder regels geschreven worden, terwijl een onervaren programmeur meer regels code nodig heeft.

Motivatie

Een klasse met een hoog aantal code regels is over het algemeen moeilijker te begrijpen en te onderhouden dan een klasse met minder coderegels. Het LOC getal van de beide lagen (adapter en AOP applicaties) worden met elkaar vergeleken. Een applicatie met een hoger aantal LOC is mogelijk minder leesbaar en onderhoudbaar. De `AssemblyInfo.cs` waarin de point-cuts gedefinieerd staan, wordt ook meegeteld.

b) Cyclomatic Complexity (CC)

Cyclomatic Complexity[12] is ontwikkeld door Thomas McCabe om de complexiteit van software te kunnen meten. De CC waarde van een methode is gemeten door het tellen van het aantal mogelijke paden van de methode.

Hoe meten

Bij het meten van de CC waarde wordt elke decision point (`if`, `else`, `for`, `while` en `case`) meegeteld. Daarnaast wordt 1 opgeteld voor het entry point van een methode.
De voorbeeldcode(figuur 24) heeft twee decision points, `if` en `else`. Daarnaast wordt 1 opgeteld voor de entrypoint. De CC waarde van de voorbeeldcode is daarom 3.

```
public int Add(int n1, int n2){
  int returnValue = 0;
  if ( n1 == 0 || n2 == 0 )
    DoSomething();
  else
    returnValue = n1 + n2;
  return returnValue;
}
```

Figuur 24: CC - voorbeeld

Motivatie

Een methode met een hoge CC waarde zou complexer zijn dan een methode met een lage CC waarde. Een complexe code is moeilijk te begrijpen en te onderhouden. Indien er meer paden in een programma zijn, moet men ook meer testcases schrijven om alle padden te kunnen testen. Een methode met een lage CC waarde is daarom makkelijker te testen. Aangezien de `AssemblyInfo.cs` geen entry- en decision points heeft zullen we de klasse niet meetellen.

c) Coupling Between Object classes (CBO)

Deze metriek is ontwikkeld door Chidamber and Kemerer[18,19,23] en is één van de meest gebruikte metrieken om de koppeling tussen objectklassen te meten. Een object is gekoppeld aan een ander object indien een van die objecten de andere gebruikt. De subklassen van de te bereken klasse worden niet meegeteld.

> *Classes are coupled if methods or instance variables in one class are used by the other. CBO for a class is number of other classes coupled with it. [18]*

Hoe meten

Deze metriek wordt gemeten door het tellen van het aantal classes(types) dat in attributen, parameters, return types en throw declaraties van de te bereken klasse gebruikt worden. Primitieve typen en system typen worden niet meegerekend.

We gebruiken de tool NDepend[21] om CBO van de klassen van beide applicaties te meten. NDepend biedt de mogelijkheid om queries uit te voeren op applicatie-assemblies.

Hieronder is een voorbeeld van een dergelijke query. Dit voorbeeld query resulteert alle klassen vanuit de namespaces die met '`Domain`' beginnen en die door de '`TraceAspect`' klasse gebruik worden. Zo is het mogelijk om per klasse het CBO getal te berekenen. Dit voorbeeld berekent uiteraard de koppeling van een serviceklasse(`TraceAspect`) richting de domeinklassen en niet richting de tussenlaag, dat wordt weer apart berekend.

```
SELECT TYPES FROM NAMESPACES "Domain.*" WHERE IsDirectlyUsedBy   "AopBank.Aspect.TraceAspect"
```

Op die manier is het mogelijk om de koppeling tussen de drie verschillende lagen (domeinklassen, serviceklassen en tussenlaagklassen) te berekenen.

Motivatie

Hoe hoger het CBO getal is, hoe sterker de koppeling is. Een sterke koppeling is ongewenst. Een klasse met een hoge CBO aantal is moeilijk te onderhouden en te testen. Een sterkere koppeling maakt de herbruikbaarheid van een desbetreffende klasse lager. Hoe meer een klasse onafhankelijk is van andere klassen hoe groter de kans dat de klasse hergebruikt kan worden. De `AssemblyInfo.cs` klasse wordt ook hierbij meegeteld.

d) Response For a Class (RFC)

Ook deze metriek is door Chidamber and Kemerer[18,19,23] ontwikkeld. RFC meet het aantal verschillende methoden die uitgevoerd kunnen worden wanneer een object van een bepaalde klasse, een bericht ontvangt. Deze metriek telt alle methoden van de klasse plus alle methoden uit andere klassen die in die klasse aangeroepen worden.

Response set of a class of objects = { set of all methods that can be invoked in response to a message to an object of a class } [18]

RS = {M} $_{Uall\ i}$ *{Ri} where {Ri}=set of methods called by method i {M} = set of all methods in the class [18]*

RFC bestaat namelijk uit twee meeteenheden.

RFC = NLM + NRM
NLM = Number of Local Methods in the class
NRM = Number of Remote Methods called by methods in the class

Hoe meten

- NLM
  Een *local method* is een methode die binnen een klasse of interface gedeclareerd is. Bij NLM wordt het aantal methoden, inclusief constructors, van een klasse geteld.

  De onderstaande voorbeeld query resulteert de methoden van de klasse `Account`. Zoals eerder is aangegeven is deze query ook met behulp van NDepend uitgevoerd.

```
SELECT METHODS FROM TYPES "Domain.Account.Account" WHERE NbMethods >= 0
```

- NRM
  Een *Remote Method* is een methode die door een klasse wordt gebruikt, maar niet gedeclareerd is het volgende:
  - de klasse zelf
  - een klasse of interface, dat door de klasse overerft of geïmplementeerd worden (super klasse).
  - een klasse of methode dat de klasse overerft of implementeert (subklasse).

  Bij het meten van de NLM waarde van een klasse worden naast de 'remote methoden' ook de 'remote constructors' meegeteld.

  De volgende voorbeeld query geeft alle methoden van de serviceklassen die door de *TraceAspect* klasse gebruikt worden.

```
SELECT METHODS FROM NAMESPACES "Service.*" WHERE IsDirectlyUsedBy
"AopBank.Aspect.TraceAspect"
```

De getters en setters van de properties worden ook meegeteld. Dus een property met *get* en *set* verhoogt het aantal met 2.

Motivatie

Een klasse met een hoge RFC waarde wordt geacht complexer te zijn dan een klasse met een lagere RFC waarde. Dat zal ook weer mogelijk het testen en debuggen van de klasse bemoeilijken. Hoe hoger het aantal methoden die uitgevoerd kunnen worden bij het ontvangen van een bericht, hoe hoger de complexiteit van de klasse is. Bij een aanpassing moet men met meerdere methoden rekening houden. Aangezien de `AssemblyInfo.cs` zelf geen methode heeft en geen andere methoden aanroept, zullen we de klasse niet meetellen.

7.2.1.1 AOP meeteenheden

Naast het ontwerpen van AOP talen en tools is men ook bezig met het ontwikkelen van de meeteenheden die de kwaliteit van AOP applicaties kunnen meten. Op die manier introduceert [28] een aantal meeteenheden die, op de huidige OO meeteenheden gebaseerd zijn. De meeste meeteenheden zijn gebaseerd op de meeteenheden van Chidamber en Kemerer[18]. Een metriek, genaamd 'Response For a Module(RFM)' is bijvoorbeeld gebaseerd op de OO metriek Response For A Class (RFC). Er zijn ook tools zoals AOP metrics(versie 0.3)[29], die de AOP meeteenheden ondersteunen.
Deze meeteenheden en tools zijn gebaseerd op AspectJ. Er is op dit moment geen meeteenheid of een metric-tool voor de andere AOP implementaties(vooral .NET implementaties) te vinden. Hoewel AspectJ op Java gebaseerd is, heeft AspectJ een eigen taalconstructie. De 'woorden' als *aspect*, *point-cut*, *beftore* en *after* zijn speciale keywords in AspectJ. Daarentegen gebruiken de .NET AOP implementaties(PostSharp, Apect.NET, etc) over het algemeen 'gewoon' de OO constructie van de programmeertalen(C#). Aangezien PostSharp geen speciale (AOP)uitbereidingen als nieuw taalconstructies bevat en het Proof of concept is met behulp van de huidige OO meetheiden wel te meten is, gebruiken we de OO meeteenheden. De *point-cuts/join-points* worden wel in onze OO meeteenheden meegenomen. De voorgestelde nieuwe AOP meeteenheden schijnen nog niet veel gebruikt worden en zitten nu meer in de onderzoekfase. Het is ons niet bekend hoeverre de AOP meeteenheden betrouwbaar zijn. Wellicht zou men in de toekomst deze speciale AOP meeteenheden kunnen bestuderen.

7.2.2 Experts

De tweede manier is het onderzoeksresultaat laten valideren door experts. Binnen Sogyo zijn er IT Professionals aanwezig die zich met Domain Driven Design bezig houden. Deze experts kunnen de AOP applicatie valideren of de applicatie ook daadwerkelijk voldoet aan de Domain Driven Design principes. Daarbij kijken we onder andere naar de correctheid, begrijpbaarheid en verbeterpunten van de applicatie.

De DDD experts van Sogyo kennen de Bank case vrij goed. We lopen de architectuur van de AOP applicatie met ze door en stellen een aantal vragen aan de experts. Aan de hand van hun feedback wordt het proof of concept verbeterd.

7.2.3 Scenario's

De derde manier die we gebruiken om de onderzoekresultaten te valideren zijn scenario's. We definiëren een aantal scenario's die zowel op de adapterapplicatie als op de AOP applicatie getoetst worden. Dit is echter geen primaire validatie-methode van dit onderzoek en kan gezien worden als aanvulling op de andere twee methoden. Er zijn vele verschillende scenario's mogelijk, afhankelijke van deze scenario's kunnen de resultaten verschillen. Een bepaald aanpassingscenario kan alleen maar één methode van een klasse beïnvloeden, terwijl een andere aanpassing meerdere methoden van meerdere klassen kan beïnvloeden. Daarom kunnen we deze methode niet als een primaire validatie-methode gebruiken.

Scenario's moeten zo objectief mogelijk opgesteld worden en zijn in vergelijking met de meeteenheden minder objectief. Indien de scenario's niet op een juiste manier opgesteld worden, kan het leiden tot misinterpretatie.

Keuze motivatie

Aangezien de domeinklassen en de serviceklassen het meest aangepast worden, hebben we ook gekozen om ons meer op de veranderingscenario's van deze klassen te richten. Over het algemeen wordt de tussenlaag niet veranderd. Deze laag verandert alleen wanneer domeinklassen of de serviceklassen aangepast worden. De aanpassingen in de domeinklassen hebben zowel invloed op de tussenlaag als op de servicelaag. De serviceklassen hebben alleen de ondersteunde rol voor de domeinklassen en de veranderingen in de serviceklassen hebben geen invloed op de domeinklassen, wel op de tussenlaag. We verdelen services in domeinspecifieke en niet- domeinspecifieke services. Een domeinspecifieke service heeft mogelijk meer invloed op de tussenlaag dan een niet- domeinspecifieke service.
We zullen scenario's creëren waarbij domeinklassen en/of de serviceklassen aangepast worden. Daarbij zullen we bestuderen hoeveel klassen en methoden aangepast moeten worden.

Abstracte scenario's

We hebben de volgende scenario's gedefinieerd.

	Mogelijke acties	Laag
1	Nieuwe methode/property toevoegen	Domain
2	Een bestaande methode/property aanpassen	Domain
3	Een domeinspecifieke service aanpassen	Service
4	Een niet-domeinspecifieke service aanpassen	Service

Tabel 6: Abstract scenario's

Concrete scenario's

We hebben de volgende scenario's gebruikt om de effecten van de aanpassingen te meten.

	Scenario's	Mogelijke acties
1	Postcode toevoegen - `Client`	1, 2
2	Aanpassing in validatie - `TransactionValidator`	3
3	Aanpassing in exceptiehandeling - `TraceMethod`	4

Tabel 7: Concrete scenario's

Hoofdstuk 8

Evaluatie

8.1 Meeteenheden

In dit gedeelte zullen we de resultaten van de in 7.2.1 gedefinieerde meeteenheden bespreken. Deze meeteenheden worden zowel op de adapterapplicatie als op de AOP applicatie uitgevoerd. De uitgebreide resultaten van de gemeten waarden van de beide applicaties zijn in de bijlage A te vinden.

8.1.1 Lines Of Code

Lines – Coderegels met blanke en commentaarregels
Statements – alleen de C# statements, exclusief blanke en commentaarregels.

Er is weinig verschil in het aantal regels tussen beide applicaties. AOP heeft meer klassen(8 tegen 4). Elke klasse bestaat uit de declaraties zoals geïmporteerde namespeces(`using`), naam van de namespace en de klasse. Dat leidt tot een hoge LOC waarde bij de aspectklassen. Desondanks blijft de LOC waarde van AOP onder de waarde van de adapterapplicatie.

Lines of code

	Lines	Statements
Adapterklassen	448	224
Aspectklassen	365	215

Tabel 8: LOC

Een hogere LOC waarde van de adapterapplicatie is voornamelijk te danken aan tracing service. Bij alle methoden van de domeinklassen van de adapterapplicatie worden twee tracing methoden (`OnEnrty` & `OnExit`) toegevoegd.

Het is moeilijk om te zeggen welk van de twee applicatie over het algemeen meer regels code heeft. Dat is grotendeels afhankelijk van de services. Een service die herhaaldelijk gebruikt wordt, kan beter met AOP geïmplementeerd worden. Een service die specifiek voor een bepaalde domeinklasse of domeinmethode is, kan ook met behulp van adapters geïmplementeerd worden. Indien de tracing service niet toegepast wordt, heeft de adapterapplicatie maar 176 statements.

Aan de hand van de gemeten LOC waarde van de beide applicaties kunnen we vastleggen dat AOP minder code regels heeft.

8.1.2 Cyclomatic Complexity

In dit gedeelte zullen we resultaten van Cyclomatic Complexity van de beide applicaties analyseren.

De totale CC waarde van de adapterapplicatie is vrij hoger dan de CC waarde van de AOP applicatie. De voornaamste reden voor de hogere waarde is dat de adapterklassen meer methoden bevatten, waardoor ze ook meer entry points hebben. Maar de decision points(`if`, `while`, etc.) van de adapterklassen zijn lager met 16, terwijl de aspectklassen 21 decision points hebben.

Cyclomatic Complexity

	CC
Adapterklassen	63
Aspectklassen	38

Tabel 9: CC

Op basis van de gemeten CC waarde van de beide applicaties kunnen we vastleggen dat de aspectklassen minder complexer zijn dan de adapterklassen.

8.1.3 Coupling between Object classes

In dit gedeelte analyseren we de resultaten van de CBO metriek. Zoals in 7.2.1 is aangegeven worden de CBO waarde van de domeinklassen niet gemeten. In de domeinklassen worden zowel de serviceklassen als de

aspectklassen niet gebruikt. De CBO waarde van de domeinklassen is dan ook 0, daardoor is het niet interessant om te meten.

Tussenlaagklassen

De totale CBO waarden van de tussenklassen van de beide applicaties zijn bijna evenveel. Een hoge CBO waarde van de aspectklassen is te danken aan het feit dat de AOP applicatie veel aspectklassen heeft, in totaal zijn er 7 aspectklassen en een util klasse. Elke aspectklasse maakt gebruikt van 1 of 2 serviceklasse(n).

CBO - Tussenlaagklassen

	CBO – Service	CBO – Domein	Totaal
Adapterklassen	8	14	**22**
Aspectklassen	12	8	**20**

Tabel 10: CBO - Tussenlaagklassen

De minimale individuele CBO waarde bij de adapterklassen is 4 (`BankAdapter`) en maximale waarde is 9(`ClientAdapter`). Bij de AOP klassen ligt de CBO waarden tussen 1 en 5. Hoewel de adapterklassen int totaal maar 4 klassen zijn, hebben ze over het algemeen als individu een grotere CBO waarde.
De niet-domeinspecifieke aspectklassen (`TraceAspect`, `ExceptionAspect` en `AuthAspect`) hebben geen koppeling met de domeinklassen, terwijl alle adapterklassen van meerdere domeinklassen gebruik maken.

Serviceklassen

- Service - tussenlaag

Ruim de helft van de serviceklassen maken gebruik van de adapterklassen, in totaal zijn er 18 koppelingen. De niet-domeinspecifieke klassen (DatabaseManager, TraceMethod, ExceptionHanlder, etc.)maken geen gebruik van de adapterklassen. Bij de AOP applicatie worden de aspectklassen door de serviceklassen niet gebruikt. Het verschil is dan ook groot, 18 tegen 0.

CBO- Serviceklassen

Serviceklassen	CBO – tussenlaag	CBO – Domein	Totaal
Adapterapplicatie	18	0	**18**
AOP applicatie	0	21	**21**

Tabel 11: CBO - Serviceklassen

- Service – Domain

De serviceklassen bij de adapterapplicatie maken geen gebruik van de domeinklassen en in plaats daarvan wordt de adapterklassen door de serviceklassen gebruikt. Maar in de AOP applicatie worden de domeinklassen direct gebruikt. Het verschil is dan 0 tegen 21.

De waarden van beide berekeningen verschillen niet veel, 21 tegen 18. Bij de adapterapplicatie is er een sterke koppeling van de serviceklassen richting de adapterklassen en bij de AOP applicatie is er een sterke koppeling van de serviceklassen richting de domeinklassen.

CBO waarde

Op basis van de CBO waarden kunnen we concluderen dat de koppelingen in beide applicaties ongeveer evenveel zijn. Dit is een relatief kleine applicatie met weinig klassen. In grote projecten kunnen deze waarden mogelijk meer verschillen.

8.1.4 Response For a Class

In dit gedeelte analyseren we de resultaten van de RFC metriek.

Tussenlaagklassen

- NLM

De NRM waarden van de beide applicaties verschillen veel. Zoals hiernaast(Tabel 12) te zien is, bevatten de adapterklassen in totaal 47 lokale methoden, terwijl de aspectklassen maar 17 lokale methoden bevatten.

RFC - Tussenglaagklassen

	NLM	NRM – Service	NRM – Domein	RFC
Adapterapplicatie	47	28	29	**99**
AOP applicatie	17	25	2	**44**

Tabel 12: RFC - Adapterklassen van de adapterapplicatie

- Tussenlaag – Service (NRM)

Er is weinig verschil in de NRM waarden van de beide applicaties, 28 tegen 25. De tussenlagen van de beide applicaties gebruiken ongeveer evenveel methoden uit de serviceklassen.

- Tussenlaag – Domein (NRM)

Hier verschillen de beide NRM waarden aanzienlijk. De adapterklassen maken gebruikt van 29 methoden uit domeinklassen, terwijl de aspectklassen maar 2 methoden daarvan gebruiken. Alle public methoden van de domeinklassen worden door de adapterklassen gebruikt, wat leidt tot een hogere NRM waarde van de adapterapplicatie.

- Totale RFC waarde van de tussenlaagklassen

De RFC waarde van de adapterapplicatie is tweemaal groter dan de RFC waarde van de AOP applicatie. Dus de aanpassingen aan de domein- of serviceklassen kunnen grotere invloed op de adapterklassen hebben dan op de aspectklassen.

Serviceklassen

- NLM

Aangezien we voor de beide applicaties dezelfde serviceklassen gebruiken, zijn de NLM waarden van de beide applicatie hetzelfde. De serviceklassen bestaan uit 80 lokale methoden.

RFC - Serviceklassen

	NLM	NRM – Tussenlaag	NRM – Domein	RFC
Adapterapplicatie	80	47	0	**127**
AOP applicatie	80	0	39	**119**

Tabel 13: RFC - Serviceklassen van de Adapterapplicatie

- Service - Tussenlaag (NRM)

De serviceklassen van de adapterapplicaties maken gebruikt van 47 methoden uit de adapterklassen. In de AOP applicatie maken de serviceklassen echter geen gebruik van tussenlaagklassen(aspectklassen).

- Domein - Tussenlaag (NRM)

In de adapterapplicatie worden geen enkele domeinmethoden door de serviceklassen gebruikt. Maar in de AOP applicatie worden de domeinmethoden door de serviceklassen direct gebruikt, namelijk 39.

- Totale RFC waarden van de serviceklassen

De totale RFC waarden van beide applicaties zijn vrijwel hetzelfde, hoewel de AOP applicatie iets beter blijkt te zijn. Bij één applicatie worden de tussenlaagklassen door de serviceklasse gebruikt, bij de andere applicatie gebruiken de serviceklassen direct de domeinklassen. Daarmee is dat ook meer verplaatsing van de koppelingen.

RFC Waarde

In totaal heeft de AOP applicatie minder RFC waarde dan de adapterapplicatie. Op basis van deze resultaten heeft de adapterapplicatie een sterkere koppeling.

8.2 Experts

In dit gedeelte worden de resultaten van de evaluatiegesprekken met de DDD experts over de AOP architectuur besproken. De architectuur is met drie DDD experts van Sogyo doorlopen. Het ontwikkelen van AOP applicatie is een iteratief proces. Aan de hand van de feedback van experts wordt de AOP applicatie verbeteerd.

DDD richtlijn

Volgens de experts voldoet het proof of concept aan de richtlijnen van Domain Driven Design.

Leesbaarheid

Deze architectuur is goed te begrijpen en dit komt grotendeels door dat deze applicatie redelijk simpel is en de experts hebben al ook enige kennis van het AOP concept. Ze hebben aangegeven dat het gebruik van AOP voor ontwikkelaars zonder voorkennis van AOP, moeilijk kan zijn.

Verbeterpunten

Volgens een expert mogen de aspecten nog generieker zijn, zodat ze ook herbruikbaar zijn. In de AOP architectuur wordt bijvoorbeeld voor alle vier CRUD (Create, Read, Update en Delete) acties vier aspectklassen gemaakt. Indien een generieke 'PersistenceAdapter' gemaakt wordt, zal de adapter voor alle type CRUD acties bruikbaar zijn.

Een andere expert gaf aan dat het gebruik van Attributen(annotaties) i.p.v. `AssemblyInfo.cs` beter zou zijn, zo wordt het bestand niet te groot. Volgens hem kunnen de Attributen niet als een vervuiling van het domein gezien worden.

8.3 Scenario's

In dit gedeelte kijken we naar de resultaten die op basis van 7.2.3 uitgevoerd zijn. De scenario's worden niet als primaire validatie-methode toegepast.

8.3.1 Scenario 1 – Domeinklasse aanpassing

Op dit moment wordt in de domeinklasse `Client` geen 'postcode' gegevens bijgehouden. We passen de `Client` klasse aan en voegen de postcode attribuut aan de klasse toe.

Tabel 14 geeft de resultaten van de aanpassingen die we uitgevoerd hebben om de bovenstaande aanpassing door te voeren. Bij deze aanpassing wordt bijvoorbeeld 1 klasse en 2 methoden van de AOP tussenlaag aangepast. Zoals te merken is, zijn er geen grote verschillen in de aangepaste klassen en methoden.

	Tussenlaag		Service	
	Kla-ssen	Meth-oden	Kla-ssen	Metho den
AOP	1	2	4	6
Adapter	2	2	4	6

Tabel 14: Scenario 1

8.3.2 Scenario 2 - Serviceklasse aanpassing, domeinspecifiek

In dit scenario wordt de `TransactionValidator` klasse aangepast. Op dit moment kunnen alle gebruikers, van een willekeurige rekening geld opnemen en op die rekening geld storten. Met deze aanpassing willen we dat een gebruiker alleen van diens eigen rekening geld mag opnemen en storten.

Tabel 15 geeft de resultaten van de aanpassingen die we uitgevoerd hebben om de bovenstaande aanpassing door te voeren. De gevolgen van deze aanpassing is in beide applicaties hetzelfde.

	Tussenlaag		Service	
	Kla.	Met.	Kla.	Met.
AOP	1	2	1	2
Adapter	1	2	1	2

Tabel 15: Scenario 2

8.3.3 Scenario 3 – Serviceklasse aanpassing, niet- domeinspecifiek

In dit scenario hebben we de tracing service uitgeschakeld.

Tabel 16 geeft de resultaten van de aanpassingen die we uitgevoerd hebben om de bovenstaande aanpassing door te voeren. Deze aanpassing heeft invloed op alle vier adapterklassen en weinig invloed op de AOP applicatie. Bij de AOP applicatie wordt alleen de `AssemblyInfo.cs` klasse aangepast.

	Tussenlaag		Service	
	Kla.	Met.	Kla.	Met.
AOP	0	0	0	0
Adapter	4	22	0	0

Tabel 16: Scenario 3

8.4 Evaluatiecriteria

Net als de probleembeschrijving zijn deze evaluatiecriteria samen met de DDD experts van Sogyo opgesteld.

1. *Het domein moet niet aangepast worden om AOP te ondersteunen.*
   Het domein moet onafhankelijk van de gebruikte infrastructuur zijn, er moet dus geen referentie naar de infrastructuur in het domein aanwezig zijn, want het domein moet bruikbaar zijn voor verschillende infrastructuren(Web of Windows) of technieken(AOP of adapter). Een domeinklasse moet niet een bepaalde interface overerven om AOP te ondersteunen. Er moet geen nieuwe methode/attribuut toegevoegd worden of een bestaande methode aangepast worden om AOP te ondersteunen. Indien het domein geen enkele kennis van tussenlaag of services heeft, houden we het domein schoon. Zoals in 3.1.3.3 is aangegeven wordt in de huidige Robotcase de domeincode aangepast om de koppeling met services te realiseren.

2. *Service moet niet veranderd worden om AOP te ondersteunen.*
   Een service moet niet afhankelijk zijn van de gebruikte koppelingstechniek(adapters, AOP of iets anders). Een serviceklasse moet niet een bepaalde interface overerven om AOP te ondersteunen. Er moet geen nieuwe methode/attribuut toegevoegd worden of een bestaande methode aangepast worden om AOP te ondersteunen. Een aanpassing aan een service kan tevens de objectiviteit van dit onderzoek verlagen. Bij de vergelijking van beide tussenlagen moeten de overige omstandigheden gelijk zijn, zodat we zeker zijn dat eventuele verschilen in de resultaten door de AOP tussenlaag is veroorzaakt en niet door een aanpassing in een service.

3. *De AOP layer heeft minder koppelingen met het domein.*
   We gebruiken meeteenheden en scenario's om de aanwezige koppelingen/afhankelijkheden tussen de AOP tussenlaag en de domeinklassen en de kwaliteit van deze koppelingen te bepalen. De meeteenheden zijn gebaseerd op de kwaliteitsattributen. We onderzoeken of de AOP implementatie minder koppelingen en betere kwaliteitlieve tussenlaag heeft.

4. *De AOP layer heeft minder koppelingen met de services.*
   We gebruiken dezelfde manier als bij punt 3.

5. *Het domein en de services hebben minder koppelingen.*
   We gebruiken dezelfde manier als bij punt 3.

6. *De AOP layer lost de in 3.1.3 beschreven problemen op.*
   6.1 Duplicate van code door cross-cutting concerns
   6.2 Sterke koppeling
   6.3 Het domein of de services wordt vervuild.
   6.4 Extra code en mogelijk minder begrijpbare code

8.5 Evaluatie

In dit gedeelte worden de evaluatiecriteria die we in 8.4 hebben gedefinieerd, per criterium behandelt.

a) Het domein moet niet aangepast worden om AOP te ondersteunen.
Het domeinmodel is niet veranderd. De domeinklassen van de adapter applicatie en de AOP applicatie blijven dezelfde. Hierdoor is aan dit criterium voldaan.

b) Services moeten niet veranderd worden om AOP te ondersteunen.
De serviceklassen worden wel aangepast. Serviceklassen van de adapterapplicatie maken gebruik van adapterklassen, terwijl de serviceklassen van de AOP applicatie direct de domeinklassen gebruiken. Maar deze aanpassing is niet gemaakt om AOP te ondersteunen maar omdat we in de AOP applicatie geen adapterklassen hebben. Verder zijn er geen aanpassingen in de serviceklassen. Hieronder zijn twee voorbeelden uit de adapter- en AOP applicatie.

Adapter voorbeeld

```
…
public static void PrintClientRecords(BankAdapter bank)
…
```

AOP voorbeeld

```
….
public static void PrintClientRecords(IBank bank)
..
```

Afgezien van enige aanpassing voldoet de applicatie aan dit criterium.

c) De AOP layer heeft minder koppelingen met het domeinmodel.

Om dit te kunnen berekenen, vergelijken we de koppelingen tussen de tussenlaag- en de domeinklassen van beide applicaties. De CBO en RFC waarden van de domeinklassen richting de tussenlaagklassen is in beide applicaties 0. De CBO waarde van de adapterapplicatie(van de tussenlaagklassen richting de domeinklassen) is 14 en van de AOP applicatie is 8. De RFC waarde is 76 tegen 19.
Op basis van deze meeteenheden kunnen we vastleggen dat de AOP layer minder koppeling met de domeinklassen heeft. Daarmee is aan dit criterium voldaan.

d) De AOP layer heeft minder koppelingen met de services.
Om dit te kunnen berekenen, vergelijken we de koppelingen tussen de tussenlaag- en de serviceklassen van beide applicaties. De CBO waarde van de adapterklassen richting de serviceklassen is 8 en van de andere richting is 18. De CBO waarde van de aspectklassen richting de serviceklassen is 12 en van de andere richiting is 0. De RFC waarde van de adapterklassen richting de serviceklassen is 75 tegen 42 voor de aspectklassen. En de RFC waarde van de serviceklassen richting de adapterklassen is 127 tegen 80 voor de aspectklassen.
Op basis van deze meeteenheden kunnen we vastleggen dat de AOP layer minder koppeling met de serviceklassen heeft. Daarmee is aan dit criterium voldaan.

e) Het domein en de services hebben minder koppelingen.
Om dit te kunnen berekenen, vergelijken we de koppelingen tussen de domein- en de serviceklassen van beide applicaties. De CBO en RFC waarden van de domeinklassen richting de serviceklassen is in beide applicaties 0. De CBO waarde van de serviceklassen richting de domeinklassen in de adapterapplicatie is 0 tegen 21 in de AOP applicatie. De RFC waarde van de serviceklassen richting de domeinklassen in de adapterapplicatie is 0 tegen 39 in de AOP applicatie.
Op basis van deze meeteenheden kunnen we vastleggen dat de serviceklassen in de AOP applicatie sterkere koppeling met de domeinklassen hebben. Daarmee is aan dit criterium niet voldaan.

f) AOP laag lost de implementatieproblemen van de huidige Bankcase op.
In dit gedeelte zullen we de van te voren gedefinieerde vier problemen bestuderen of deze problemen met behulp van AOP opgelost zijn.

1. Duplicatie van code door cross-cutting concerns
Met behulp van de AOP implementatie hebben we de codeduplicatie verminderd. Exceptiehandeling en tracing zijn niet meer herhaaldelijk in de applicatie opgenomen. Deze services worden centraal op één plek als modules gedefinieerd.

2. Sterke koppeling
In vergelijking met de adapterklassen hebben de aspectklassen minder koppeling (c) en d)). Er is wel een sterkere koppeling van services aan het domein e). De aspectklassen zijn kort, hebben minder methoden en heeft duidelijke taken. De aanpassingen aan het domein of services hebben minder invloed op aspectklassen.

3. Het domein of de services wordt vervuild
Zie de criteria a) en b).

4. Veel extra code en ingewikkelde structuur

- *Veel code*
  Hoewel de AOP applicatie iets minder aantal regels code heeft, is het verschil niet zo groot (Tabel 6).

- *Ingewikkelde structuur*
  De Cyclomatic Complexity van de aspectklassen zijn vrij lager(40%) dan van de adapterklassen. Maar als we alleen de decision-points tellen is de AOP applicatie complexer. In de gesprekken met de DDD experts van Sogyo is aangegeven dat de structuur van de proof of concept niet moeilijk is. Maar het begrijpen van AOP concept kan echter voor velen in het begin moeilijk zijn. Deze applicatie is redelijk klein, daardoor is het ook minder moeilijk te begrijpen.

Hoofdstuk 9

Conclusie en Aanbevelingen

9.1 Conclusie

Op basis van de geanalyseerde resultaten van het uitgevoerde onderzoek kunnen we een aantal conclusies trekken. We gaan per onderzoekvraag kijken of er een antwoord daarop gevonden is.

Is AOP toepasbaar om het Domain Driven Design concept te realiseren
Met behulp van AOP is het mogelijk om de DDD richtlijnen te realiseren. De voornaamste richtlijn is dat de domeinklassen geen kennis mogen hebben van de omliggende serviceklassen. AOP applicatie voldoet aan deze richtlijn. Om AOP te ondersteunen is het ook niet noodzakelijk om zowel het domein als service aan te passen. Omdat in de AOP applicatie geen adapterklassen zijn, gebruiken de serviceklassen de domeinklassen direct. Volgens de vastgestelde richtlijnen mogen de serviceklassen wel kennis van het domein hebben. Maar er is echter een sterkere koppeling van service aan het domein.

Verbetert AOP de ontkoppeling van Domain Driven Design applicaties in vergelijking met de huidige implementatie en worden de huidige problemen met behulp van AOP opgelost.
Op basis van de resultaten kunnen we concluderen dat behulp van AOP de koppelingen minder sterk zijn. In de koppelingen op klassenniveau(CBO) is niet veel verschil te merken, beide koppelingen zijn ongeveer evenveel. Op methodeniveau(RFC) is er zeker een groot verschil in de koppelingen van beide applicaties. De koppelingen op methodeniveau zijn in de AOP applicatie minder dan in de adapterapplicatie.

AOP lost een probleem op dat niet met behulp van de adapters opgelost kan worden, oftewel de cross-cutting concerns. Hiermee hebben we de duplicatie van de code, en daarmee ook het aantal regels code verminderd. Zonder gebruikmaking van deze cross-cutting services zou AOP niet veel voordelen hebben in vergelijking met de adapterapplicatie. Zoals bij de aanpassingsscenario's te zien is, vereist alleen de cross-cutting concern(tracing) geen aanpassing in de overige klassen. Indien de tracing service niet toegepast wordt zou het aantal statements(LOC) bij de adapterapplicatie minder zijn dan bij de AOP applicatie.

Het 'Adapter concept' is vrij algemeen. Dit concept kan ook in de overige talen als Java geïmplementeerd worden. AOP applicaties zijn tool-afhankelijk. Voor een AOP applicatie heeft men een AOP tool zoals PostSharp of AspectJ nodig. Bovendien dienen de ontwikkelaars bekend te zijn met het AOP concept. De aspecten en de join-point definities moeten tijdens de evolutie van software actueel zijn. Het gevaar bestaat dat ze niet meer actueel zijn waardoor de applicatie niet naar behoren functioneert. Om die redenen kan men beter de adapters gebruiken om de koppeling te realiseren.

Een combinatie van AOP en adapters is mogelijk. Men gebruikt de adapters als tussenlaag voor de domeinspecifieke functionaliteiten en de cross-cutting/niet-domeinspecifieke services kunnen met behulp van AOP aan deze adapters toegevoegd worden.

9.2 Aanbevelingen

Sogyo academy
Het AOP concept is een interessant onderwerp. Hoewel dit concept op dit moment niet zo populair als andere paradigma's als Object Oriented Programming, wordt dit concept steeds populair. De laatste jaren worden meer onderzoeken op dit onderwerp gepleegd. Het is aan te raden om een introductie in de 'Master course van Sogyo' over AOP te geven, zodat de medewerkers een idee kunnen krijgen over dit concept. De Sogyo senior medewerkers geven regelmatig interne presentaties voor de 'Master studenten' en AOP kan ook hier

een onderdeel van uitmaken. Uit informele gesprekken met de Sogyo medewerkers blijkt dat ze in dit onderwerp geïnteresseerd zijn en in de toekomst met de AOP tools willen gaan experimenteren.

Sogyo professionals
Als IT professional is het belangrijk om kennis te hebben over de nieuwe ontwikkelingen op IT gebied. Op die manier is het ook raadzaam om over kennis te beschikken over de toepasbaarheid van AOP. De IT professionals kunnen de klanten aanraden om AOP te gaan gebruiken, maar dit kan van een aantal factoren afhangen.
In de volgende gevallen is het niet raadzaam om een dergelijke applicatie te ontwikkelen;

- Een applicatie die niet tool-afhankelijk moet zijn, want voor AOP heeft men een AOP tool nodig. Bovendien zijn de huidige tools in de .NET omgeving niet goed doorontwikkeld.
- Beveiliging of performance gevoelige applicaties. Er zijn op dit moment geen casestudies te vinden die de stabiliteit, security en goede performance van een AOP applicaties garanderen.
- De ontwikkelaars van software moeten kennis van AOP hebben, dat geldt ook voor de toekomstige onderhoudsontwikkelaars. Het toepassen van AOP in een project is daarom ook afhankelijk van het kennisniveau van de (onderhouds)ontwikkelaars.

Sogyo development
AOP kan gebruik worden om applicaties te ontwikkelen. Gezien de minder betrouwbaarheid van de huidige tools, is het op dit moment niet aan te raden om de volledige tussenlaag met behulp van AOP te ontwikkelen. Maar de cross-cutting onderdelen kunnen wellicht met behulp van de tools zoals PostSharp ontwikkeld worden. Daarmee kunnen de ontwikkelaars van Sogyo mogelijk applicaties ontwikkelen die minder regels code nodig hebben. Minder regels code produceren meestal ook minder errors.

AOP wereld is een bewegende wereld met ontwikkelingen. Er komen nieuwe tools bij of sommige tools worden niet meer ontwikkeld. Sommige tools worden verder ontwikkeld. Het is raadzaam dat Sogyo rondom AOP aandacht blijft vestigen, onder andere door middel van verdere (afstudeer)onderzoeken.

Hoofdstuk 10

Future work en Reflectie

10.1 Future work

Kwaliteitattributen
We hebben in dit onderzoek de ontkoppeling van een AOP applicatie bekeken. Testbaarheid, onderhoudbaarheid en herbruikbaarheid zijn de gerelateerde kwaliteitaspecten. Daarnaast zijn er ook nog meer kwaliteitattributen als performance, security en stabiliteit. Deze aspecten kunnen mogelijk in een AOP applicatie slecht geregeld zijn. Een toekomstig onderzoek kan hierover meer duidelijkheid verschaffen.

AOP meeteenheden
In dit onderzoek werden de OO meeteenheden gebruikt om de koppelingen van diverse onderdelen van een applicatie te meten. Indien in de toekomst betrouwbare AOP meeteenheden gepubliceerd worden, met bijbehorende tool ondersteuning, kan men de AOP applicaties met behulp van deze meeteenheden meten.

Grote applicatie
Het gebruikte proof of concept in dit onderzoek is een kleine en simpele applicatie. Bij het gebruik van een grotere applicatie is de kans groter dat er grote verschillen in de resultaten zijn, waardoor nauwkeurigere conclusies getrokken kunnen worden.

10.2 Reflectie

Dit onderzoek is gebaseerd op een kleine case. Het is moeilijk te zeggen in hoeverre deze architectuur voor een grotere en/of complexe applicatie toepasbaar is. We hebben de meest voorkomende services als presentatie, persistentie, validatie en authenticatie in de Bankcase succesvol geïmplementeerd. Daardoor is er wel een grote mogelijkheid dat deze architectuur in vergelijkbare applicaties wel toepasbaar is.
De punten die in 10.1 genoemd zijn, zijn niet onderzocht. In de toekomst zou men met betrekking tot deze punten onderzoeken kunnen uitvoeren.

Aanpassingen in de opdracht
In de eerste versie van het plan van aanpak lag de nadruk meer op de algemene toepasbaarheid van AOP en het bestuderen van de huidige AOP tool in de NET omgeving. Maar bij het afbakenen van de opdracht ging de focus meer naar de tussenlaag die de huidige adapters in Domain Driven Design applicaties moet vervangen. Oorspronkelijk wilde de onderzoeker twee proof of concepts voor beide cases (Robot- en Bank case) applicaties ontwikkelen en de resultaten meten. Maar uiteindelijk was er maar tijd voor een case.

Opgeleverde producten
Naast deze scriptie is er ook het proof of concept en een AOP document waarin vier AOP tools besproken worden opgeleverd. De proof of concept is wel afgerond. Maar door tijdsgebrek was er geen mogelijkheid om het AOP document af te maken.

Verbeterpunten
Het proof of concept werd vrij laat met meerdere Sogyo DDD experts geëvalueerd en daarbij hebben we geen structurele aanpak gevolgd. Aangezien alle drie medewerkers reeds kennis van AOP hebben is het moeilijk te zeggen over de mogelijke tekortkomingen van de AOP architectuur vanuit het oogpunt van de medewerkers zonder voorkennis van AOP.

Referenties

[1] Adrian Colyer, Andy Clement, George Harley en Matthew Webster. *Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison Wesley, 2005

[2] Ivar Jacobson & Pan Weing, Aspect-*Oriented Software Development with Use Cases***,** Addison Wesley, 2004

[3] Jimmy Nilsson, *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*, Addison Wesley Professional, 2006

[4] Ralf Wolter, *Domain Driven Design*, Sogyo, Master thesis, De Bilt, November 2005

[5] Soygo, *Robot case*, Code & documentatie, April 2007

[6] Wolfgang Schultz and Andreas Polze, *Aspect-Oriented Programming with C# and .NET*, Hasso-Plattner-Institute at University Potsdam, 2002

[7] Robert E. Filman , *What Is Aspect-Oriented Programming, Revisited*, Research Institute for Advanced Computer Science, NASA Ames Research Center, 2003

[8] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, B. Natarajan. *An Approach for Supporting Aspect-Oriented Domain Modelling*, Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, 2003

[9] R. J. Walker, E.L.A. Baniassad en G.C. Murphy, *An Initial Assessment of Aspect-oriented Programming*, In Proceedings of the 21st International Conference on Software Engineering, Mei 1999, USA

[10] Domain Driven Design, *domaindrivendesign.org*

[11] Aspect-Oriented Software Development, *aosd.net*

[12] Thomas J. McCabe and Charles W. Butler. *Design Complexity Measurement and Testing*. Communications of the ACM, 32, pp. 1415-1425, December 1989.

[13] PostSharp, *www.postsharp.org,* Versie 3.0

[14] Rapier-Loom.NET, *www.dcl.hpi.uni-potsdam.de/research/loom,* Versie 2.0

[15] Aspect.NET**,** *www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6801,* Versie 2.1

[16] Begrippenlijst, *www.sei.cmu.edu/str/indexes/glossary,* april 2007

[17] Research plan, *web.cs.wpi.edu/~claypool/research/plan.html,* April 2007

[18] Shyam R. Chidamber and Chris F. Kemerer, A *Metrics Suite for Object Oriented Design*, IEEE Transaction on Software Engineering, juni 1994

[19] Gui Gui, Paul. D. Scott*, Reusability Ranking of Software Components by Coupling Measure,* 10th International Conference on Evaluation and Assessment in Software Engineering (EASE), Keele University , UK, april 2006

[20] Ramnivas Laddad, *Domain Driven Design with AOP and DI*, *www.bejug.org/confluenceBeJUG/display/PARLEYS/Domain+Driven+Design+with+AOP+and+DI,* april 2007

[21] NDepend*, www.ndepend.com*, versie 2.2

[22] P. Verschuren en Hans Doorewaard, *Het ontwerpen van een onderzoek*, derde druk, 2005

[23] *Software Quality Metrics for Object Oriented System Environments*, Software Assurance Technology Center, NASA, juni 1995

[24] Soygo, *Bank case*, Code & documentatie, april 2007

[25] Eric Evans, *Domain Driven Design : Tackling Complexity in the Heart of Software*, 2003

[26] Sourcemonitor, *www.campwoodsw.com/sourcemonitor.html,* versie 2.3

[27] Hitz en Montazeri, *Measuring Coupling en Cohesion In Object-Oriented Systems*, University of Vienna, 1995

[28] Mariano Ceccato and Paolo Tonella, *Measuring the Effects of Software Aspectization*. 1st Workshop on Aspect Reverse Engineering (WARE 2004). November, 2004. Delft, Nederland

[29] Aopmetrics, *aopmetrics.tigris.org,* Versie 0.3

[30] Gregor Kiczales et al, *Aspect-Oriented Programming,* European Conference on Object-Oriented Programming (ECOOP), juni, 1997

[31] Michal Forgáč, Ján Kollár, *Static and Dynamic Approaches to Weaving*, 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics, 2007

[32] Sogyo, *www.sogyo.nl*

[33] AspectJ, *www.eclipse.org/aspectj*, Versie 1.5.3

[34] Jingyue Li, Axel Anders Kvale, Reidar Cintradi. *A Case Study on Building COTS Component-Based System Using Aspect-Oriented Programming,* 20th Annual ACM Symposium on Applied Computing (ACM SAC 2005), mart 2005

[35] Pavel Hruby, *Ontology-Based Domain-Driven Design*, OOPSLA Workshop on Best Practices for Model-Driven Software Development*, 2005*

[36] Magiel Bruntink, Arie van Deursen, Tom Tourwe, *Isolating Idiomatic Crosscutting Concerns,* Proceedings of the IEEE International Conference on Software Maintenance (ICSM), September 2005

Bijlage A

Meeteenheden

1.1 Lines Of Code

Hieronder zijn de resultaten van de gemeten LOC waarden van de beide applicaties.
Lines – Coderegels met blanke en commentaarregels
Statements – alleen de C# statements, exclusief blanke en commentaarregels.

Adapter applicatie

Adapterklasse	Lines	Statements
ClientAdapter	136	65
AccountAdapter	134	72
BankAdapter	137	68
TransactionAdapter	41	19
Totaal	**448**	**224**

AOP applicatie

Aspectklassen	Lines	Statements
AuthAspect	19	10
CreatNewAspect	76	47
ExceptionAspect	30	17
LoadDatAspect	36	20
TraceAspect	26	13
TransactionAspect	64	37
ValidationAspect	60	40
Util	41	23
AssemblyInfo(aspectgedeelte)	15	14
Totaal	**365**	**215**

1.2 Cyclomatic Complexity

Hieronder zijn de resultaten van de gemeten CC waarden van de beide applicaties.

Adapterklasse	Entry points	Decision points
ClientAdapter	20	2
AccountAdapter	13	7
BankAdapter	10	7
TransactionAdapter	4	-
Totaal	**47**	**16**

Aspectklassen	Entry Points	Desion points
AuthAspect	1	-
CreatNewAspect	4	5
ExceptionAspect	2	-
LoadDatAspect	1	2
TraceAspect	2	-
TransactionAspect	2	8
ValidationAspect	3	4
Util	2	2
Totaal	**17**	**21**

1. 3. Coupling between Object classes

1.3.1 Tussenlaagklassen

Hieronder zijn de resultaten van de CBO metriek van de tussenlagenklassen. De eerste kolom bevat de klassennamen van de tussenlagen. De tweede kolom geeft per tussenlaagklasse aan hoeveel serviceklassen door de tussenlaagklassen gebruikt worden. Zo is bijvoorbeeld, de `ClientAdapter` klasse maakt gebruik van 1 serviceklasse. De derde kolom geeft per tussenlaagklasse aan hoeveel domeinklassen door de tussenklassen gebruikt worden. De vierde kolom bevat som van de 2de en 3de kolom, tevens de CBO waarde van een tussenlaagklasse. Zo heeft de `AccountAdapter` klasse de CBO waarde van 7.

Adapterapplicatie

Adapterklassen	CBO – Service	CBO – Domein	Totaal
ClientAdapter	1	3	4
AccountAdapter	3	4	7
BankAdapter	5	4	9
TransactionAdapter	1	3	4
Totaal	**8**	**14**	**22**

AOP Applicatie

Aspectklassen	CBO – Service	CBO – Domein	Totaal
AuthAspect	2	0	2
CreatNewAspect	2	3	5
ExceptionAspect	1	-	1
LoadDatAspect	2	2	4
TraceAspect	1	-	1
TransactionAspect	1	1	2
ValidationAspect	2	3	5
Util	1	-	1
AssemblyInfo	**-**	**-**	**0**
Totaal	**12**	**8**	**20**

1.3.2 Serviceklassen

De volgende twee tabbellen tonen de resultaten van de CBO metriek van de serviceklassen. De eerste kolom bevat de namen van de serviceklassen. De 2de kolom en 3de geven per serviceklasse respectievelijk hoeveel adapter- en domeinklassen door de serviceklassen gebruikt worden. Op die manier maakt de `ConsoleUserInterface` klasse gebruikt van 3 adapterklassen en 0 domeinklassen. De 4de kolom bevat totale CBO waarde per serviceklasse.

Adapterapplicatie

Serviceklassen	CBO – Adapter	CBO – Domein	Totaal
ExceptionHandler	-	-	-
ConsoleUserInterface	3	-	3
ReportGenerator	4	-	4
AccountPersistence	3	-	3
BankPersistence	3	-	3
ClientPersistence	1	-	1
DataBaseManager	-	**-**	-
Security	-	**-**	-
User	-	**-**	-
Formatter	-	**-**	-
TraceMethod	-	**-**	-
TransactionValidator	1	**-**	1
Util	1	**-**	1
ClientValidator	1	**-**	1
Totaal	**18**	**-**	**18**

AOP Applicatie

Serviceklassen	CBO – Aspect	CBO – Domein	Totaal
ExceptionHandler	-	-	-
ConsoleUserInterface	-	2	2
ReportGenerator	-	4	4
AccountPersistence	-	5	5
BankPersistence	-	5	5
ClientPersistence	-	1	1
DataBaseManager	-	**-**	-
Security	-	**-**	-
User	-	**-**	-
Formatter	-	**-**	-
TraceMethod	-	**-**	-
TransactionValidator	-	2	1
BankWorld	-	1	1
ClientValidator	-	1	1
	0	**21**	**21**

1.4. Response For a Class

1.4.1 Tussenlaagklassen

De volgende twee tabellen tonen de RFC waarden van de klassen uit de tussenlagen. De 2de kolom geeft het aantal methoden(inclusief constructors, getters en setters) van deze klassen. De 3de kolom bevat het aantal van de methoden uit de serviceklassen die door de tussenlaagklassen gebruikt worden, (Number of Remote Methods). Bijvoorbeeld, de `ClientAdapter` klasse maakt gebruik van 2 methoden uit de serviceklassen.

De 4de kolom bevat het aantal van de methoden uit de domeinklassen die door de tussenlaagklassen gebruikt worden. De 5de kolom bevat de RFC waarde van de tussenlaagklassen.

Adapterapplicatie

Adapterklassen	NLM	NRM – Service	NRM – Domein	RFC
ClientAdapter	20	2	13	35
AccountAdapter	13	11	9	33
BankAdapter	10	13	5	28
TransactionAdapter	4	2	2	8
Totaal	**47**	**28**	**29**	**99**

AOP applicatie

Aspectklassen	NLM	NRM – Service	NRM – Domein	RFC
AuthAspect	1	2	-	3
CreatNewAspect	4	6	2	12
ExceptionAspect	2	2	-	4
LoadDatAspect	1	4	-	5
TraceAspect	2	2	-	4
TransactionAspect	2	5	-	7
ValidationAspect	3	4	-	7
Util	2	-	-	2
Totaal	**17**	**25**	**2**	**44**

1.4.2 Serviceklassen

In de onderstaande tabbellen worden de RFC waarden van de serviceklassen weergegeven. De 2de kolom geeft het aantal methoden van de klassen weer, (Number of Local Methods) . De 3de kolom bevat het aantal van de methoden uit de adapterklassen die door de serviceklassen gebruikt worden, (Number of Remote Methods). De 4de kolom bevat het aantal van de methoden uit de domeinklassen die door de serviceklassen gebruikt worden. De 5de kolom bevat de RFC waarde van de serviceklassen.

Adapterapplicatie

Serviceklassen	NLM	NRM – Adapter	NRM – Domein	RFC
ExceptionHandler	3	-	-	4
ConsoleUserInterface	13	10	-	23
ReportGenerator	3	6	-	10
AccountPersistence	16	9	-	24
BankPersistence	4	8	-	12
ClientPersistence	7	7	-	14
DataBaseManager	5	-	-	6
Security	7	-	-	7
User	3	-	-	7
Formatter	1	-	-	4
TraceMethod	3	-	-	5
TransactionValidator	6	1	-	8
BankWorld	4	1	-	5
ClientValidator	5	5	-	11
	80	**47**	**0**	**127**

AOP applicatie

Serviceklassen	NLM	NRM – Aspect	NRM – Domein	RFC
ExceptionHandler	3	-	-	3
ConsoleUserInterface	13	-	6	19
ReportGenerator	3	-	6	9
AccountPersistence	16	-	7	23
BankPersistence	4	-	6	10
ClientPersistence	7	-	7	14
DataBaseManager	5	-	-	5
Security	7	-	-	7
User	3	-	-	3
Formatter	1	-	-	1
TraceMethod	3	-	-	3
TransactionValidator	6	-	2	8
BankWorld	4	-	-	4
ClientValidator	5	-	5	10
	80	**0**	**39**	**119**

Bijlage B

Source code

2.1 Aspectklasse –TransactionAspect.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

using PostSharp.Laos;

using Domain.Bank;
using Domain.Client;
using Domain.Account;
using Service.PersistenceService;

namespace AopBank.Aspect
{
    [Serializable, AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
    public sealed class TransactionAspect : OnMethodBoundaryAspect
    {
        private TransactionType transactionType;

        public TransactionAspect(string transactionType)
        {
            this.transactionType = AccountPersistence.ConvertToTransactionType(transactionType);
        }

        public override void OnExit(MethodExecutionEventArgs eventArgs)
        {
            // De benodigde arugumenten uitlezen
            IAccount account = (IAccount)eventArgs.Instance;
            int issuerNumber = (int)Util.GetParameterValue(eventArgs, "issuerNumber");
            double amount = (double)Util.GetParameterValue(eventArgs,"amount");

            AccountPersistence ap = new AccountPersistence(account);

            switch (transactionType)
            {
                case TransactionType.Credit:
                    {
                        if (!ap.Deposit(amount, issuerNumber))
                            throw new Exception("Deposit transaction has failed!");
                        break;
                    }
                case TransactionType.Debit:
                    {
                        if (!ap.WithDraw(amount, issuerNumber))
                            throw new Exception("Withdraw transaction has failed!");
                        break;
                    }
                case TransactionType.Transfer:
                    {
                        IAccount targetAccount = (IAccount)Util.GetParameterValue(eventArgs, "targetAccount");
                        if (!ap.Transfer(amount, targetAccount, issuerNumber))
                            throw new Exception("Transfer transaction has failed!");

                        break;
                    }
                default:
                    break;
            }
        }

    }
}
```

2.2 Adapterklasse – AccountAdapter.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

using Domain.Account;
using Domain.Client;
using Service.TraceService;
using Service.ValidationService;
using Service.PersistenceService;

namespace AdapterBank.Adapter
{
    public class AccountAdapter
    {
        private IAccount account;
        private AccountPersistence ap;
        private TransactionValidator tv;

        public AccountAdapter()
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public AccountAdapter(ClientAdapter owner)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            this.account = new Account(owner.Client);
            ap = new AccountPersistence(this);
            ap.AddAccountTransactions();

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public AccountAdapter(IAccount account)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            this.account = account;
            ap = new AccountPersistence(this);
            foreach (TransactionAdapter transaction in ap.GetTransactions())
                this.account.Transactions.Add(transaction.Transaction);

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public AccountAdapter(int accountNumber, ClientAdapter owner, double balance)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            this.account = new Account(accountNumber, owner.Client, balance);
            ap = new AccountPersistence(this);
            foreach (TransactionAdapter transaction in ap.GetTransactions())
                this.account.Transactions.Add(transaction.Transaction);

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public void Deposit(double amount, int issuerClientNumber)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            tv = new TransactionValidator(this, amount, issuerClientNumber);
            if (!tv.Validate())
                throw new Exception("Deposit validation has failed!");
            this.account.Deposit(amount, issuerClientNumber);
            if(!ap.Deposit(amount,issuerClientNumber))
                throw new Exception("Deposit persistence has failed");

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public void Withdraw(double amount, int issueClientNumber)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());
```

```
            tv = new TransactionValidator(this, amount, issueClientNumber);
            if(!tv.Validate())
                throw new Exception("Withdraw validation has failed!");
            this.account.Withdraw(amount, issueClientNumber);
            if(!ap.Withdraw(amount, issueClientNumber))
                throw new Exception("Withdraw persistence has failed");

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public void MoneyTransfer(double amount, AccountAdapter targetAccount, int issueClientNumber)
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            this.account.MoneyTransfer(amount, targetAccount.Account, issueClientNumber);
            ap.Transfer(amount, targetAccount, issueClientNumber);

            TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), String.Empty);
        }

        public override string ToString()
        {
            TraceMethod.MethodEntry(MethodBase.GetCurrentMethod());

            string accountInfo = this.account.ToString();

            return TraceMethod.MethodExit<String>(MethodBase.GetCurrentMethod(), accountInfo);
        }


        internal IAccount Account
        {
            get { return this.account; }
        }

        public ClientAdapter Owner
        {
            get { return new ClientAdapter(this.account.Owner); }
        }

        public int AccountNumber
        {
            get { return this.account.AccountNumber; }
        }

        public double Balance
        {
            get { return this.account.Balance; }
        }

        public List<ITransaction> Transactions
        {
            get { return this.account.Transactions; }
        }

    }
}
```