Bachelorarbeit

Bewertung der Einsatzfähigkeit von Aspektorientierter Programmierung in einem kommerziellen Software Projekt

Studienrichtung Telematik
Fakultät für Informatik
Sommersemester 2009

Genewein Tim

Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Wotawa Franz
Institut: Institut für Softwaretechnologie
Datum: 27. September 2009

Firma: IVM Technical Consultants
Betreuer: Andreas Hafellner

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am 27. September 2009 _____

(Unterschrift)

Vorwort

Die vorliegende Arbeit basiert auf Erfahrungen und Erkenntnissen, die ich als Mitglied eines Projektteams in der Grazer Niederlassung der Firma **IVM Technical Consultants** sammeln konnte. Mein besonderer Dank gilt Herrn Dipl.-Ing. Stasny Georg, der mir die Mitarbeit an einem kommerziellen Projekt ermöglicht hat. Vor allem das Mitwirken an Diskussionen und das Kennenlernen der Abläufe, die hinter einem Projekt stehen, bietet einen Aspekt der Ausbildung, der nur durch persönliche Erfahrung vermittelt werden kann.

Weiters möchte ich mich bei Herrn Hafellner Andreas bedanken, der das Projektteam geführt hat und als Ansprechpartner für fachliche, organisatorische aber auch menschliche Fragen immer zur Verfügung stand. Dieser Dank gilt auch dem gesamten Projektteam, welches mir die Mitarbeit am Projekt in vielerlei Hinsicht erleichtert hat.

Von Seiten der Universität wurde ich von Herrn Univ.Prof. Dipl.-Ing. Dr. Franz Wotawa betreut. Bei ihm und dem Institut für Softwaretechnologie möchte mich vor allem für die stets pragmatische und unbürokratische Betreuung bedanken.

Genewein Tim

Abstract

The aim of this Bachelor's thesis is to evaluate the utilizability of aspect-oriented programming (AOP) for a commercial software project. One weakness of object-oriented concepts is the handling of so called *cross-cutting concerns*. Contrary to concerns which can be encapsulated within a single object, *cross-cutting concerns* affect several objects and the according code can be found throughout the whole project.

The AOP is very capable of handling *cross-cutting concerns* and can lead to a better *separation of concerns*, which increases the modularity and maintainability of source code. However, many developers do not know the concept of the AOP and its application is not really widespread. The main goal of this thesis is the evaluation of the applicability of AOP in a small team of software developers with little to no experience in aspect-oriented programming.

The setting for the thesis is a commercial software development project at the company „IVM Technical Consultants". In the course of this project, the utilizability of aspect-oriented concepts for **logging** and **exception-handling** (EH) shal be evaluated. If the results are positive, the according solutions shall be implemented.

The project was being developed in C# (.NET 2.0) and the chosen AOP-Framework was *PostSharp*. Logging concerns turned out to be easily separable and well-suited for an aspect-oriented implementation. However, the analysis of aspect-oriented exception-handling concerns soon led to the conclusion, that this task was much more complex than initially expected. Mainly, this is due to the fact, that exception-handling code is very context-sensitive in most cases and cannot be seperated from the remaining code easily.

Resulting from the above findings, only the logging-task has been implemented. From the analysis of AOP-exception-handling some general strengths and weaknesses of the AOP could be derived.

The overall result of the thesis is that aspect-oriented concepts may be very suitable and utilizable for several aspects of projects, but on the other hand may also lead to unclear and ill-structured code for other aspects. Even more hindering is the fact that fully developed AOP (and EH) design patterns are not yet available. The evaluation of the applicability of the AOP has to be made on a sub-project level and strongly depends on the corresponding requirements and the skills of the developer-team.

Inhaltsverzeichnis

1

1 Einleitung

Die Objektorientierte Programmierung (kurz: OOP) hat sich als sehr mächtiges und dennoch flexibles Konzept erwiesen. Die zugrunde liegende Idee ist es, zusammengehörige Teilaufgaben in einem Objekt zu kapseln, sodass die Komplexität zur Lösung einer Teilaufgabe im Objekt liegt. Nach „*Außen*“ bieten Objekte möglichst einfache Schnittstellen zur Interaktion an. Die OOP hilft damit bereits im Designprozess und ist wesentlich mehr als nur ein Sprachkonzept.

In manchen Fällen treten jedoch Anforderungen auf, die sich nicht mit einem einzelnen Objekt kapseln lassen. So kommt es bei fast jedem Softwareprojekt zu sogenannten „*Crosscutting Concerns*“ (zu deutsch: querschnittliche Belange). Sie betreffen das gesamte Projekt und deren Behandlung findet sich bei einem reinen OOP-Ansatz in vielen Codeteilen (von unterschiedlichen Objekten) wieder. Ein Beispiel für *Crosscutting Concerns* wäre die Anforderung alle Methodenaufrufe zu loggen (das heißt zu protokollieren) um im Fehlerfall eher die Möglichkeit zu haben, den fehlerhaften Zustand reproduzieren zu können. Bei einer objektorientierten Umsetzung wäre es nötig, einen Log-Aufruf innerhalb von jeder Methode zu haben.

Derartiger Code hätte einen Großteil seiner Modularität eingebüßt. Beispielsweise wäre es nur mehr mit einigem Aufwand möglich, das Logging für bestimmte Klassen zu „deaktivieren“, oder den selben Code mit einem anderen Logger-Objekt (das vermutlich nicht die selbe Parameteranzahl und -Reihenfolge beim Log-Aufruf aufweist) zu betreiben. Es wären hier Änderungen an vielen Stellen im Code nötig. Darüber hinaus wird die Nachvollziehbarkeit des Codes beeinträchtigt, da immer wieder Statements auftreten, die mit der unmittelbaren Logik des Objektes nichts zu tun haben.

Ein Konzept das mit *Crosscutting Concerns* sehr gut umgehen kann ist die Aspektorientierte Programmierung (kurz: AOP). Dort werden querschnittlichen Belange als *Aspekte* angesehen. Jeder *Aspekt* besteht aus einem sogenannten *Advice* und einem oder mehreren *Pointcuts*. Der *Advice* beschreibt was passieren soll, wenn der *Aspekt* schlagend wird - im obigen Beispiel wäre der *Advice* ein Codeteil, welcher die Log-Methode aufruft. Die *Pointcuts* beschreiben an welchen Stellen im Code *Advices* angewandt werden. Im Beispiel wären die *Pointcuts* also alle Methodeneintritte.

Ein *Pointcut* kann aber nur an einen sogenannten *Joinpoint* „andocken“. Mögliche *Joinpoints* sind Methodenein- und -austritt, Lese- oder Schreibzugriff auf Properties oder das Auftreten von Exceptions. Der genaue Umfang hängt von der jeweiligen Implementierung des AOP-Frameworks ab.

Die AOP bietet also die Möglichkeit bestimmten Code beim Eintreten oder Verlassen einer Methode bzw. beim Auftreten einer Exception auszuführen. Darüber hinaus kann der selbe Code an mehreren Stellen im Projekt angewandt werden. Trotzdem lassen sich *Advices* bzw. *Aspekte* im Code zentralisieren und sind somit sehr modular und einfach wartbar.

Es gibt zwar Sprachen, die AOP bereits nativ unterstützen - in C++, Java oder C# bzw. .NET ist dies nicht der Fall. Allerdings sind für die genannten Sprachen AOP-Frameworks verfügbar. Dabei ist prinzipiell zwischen zwei Arten der Umsetzung zu unterscheiden:

- *AOP durch Proxy-Objekte:*
  Hier wird das Objekt auf das ein Aspekt angewandt werden soll durch ein Proxy-Objekt vertreten. Alle Methodenaufrufe und Schreib-/Lesezugriffe erfolgen über das Proxy-Objekt. Daher bietet diese Art der Umsetzung eine größtmögliche Flexibilität zur Laufzeit, hat aber den Nachteil, dass durch die Verwendung der Proxy-Objekte Performance-Verluste entstehen.
- *AOP durch Static Weaver:*
  Hier wird der aspektorientierte Code zur Build-Zeit in den bereits compilierten Code „*eingewoben*“ (vgl. engl.: *weaving*). Dadurch entstehen kaum Performance-Verluste zur Laufzeit, allerdings können Werte, die nur zur Laufzeit feststehen nur bedingt abgefragt und modifiziert werden.

Die Verbreitung und der Einsatz der AOP halten sich nach wie vor in Grenzen. Es lässt sich zwar eine Vielzahl von einführenden Texten und Codebeispielen finden - der Großteil davon setzt sich jedoch nur auf einer sehr praktischen Ebene mit der AOP auseinander. Interessant wäre die Frage, wie schnell und mit wie viel Aufwand sich die Methoden der AOP in bestehende Softwareentwicklungs-Teams und -Prozesse integrieren lassen. Außerdem stellt sich die Frage, ob die AOP für den Einsatz in beliebigen Projekten geeignet ist, oder ob konkrete Faktoren aus dem Projekt die Einsetzbarkeit stark beeinflussen.

2 Problemstellung

Im Rahmen der Bachelorarbeit soll die Einsetzbarkeit der AOP für ein konkretes Softwareprojekt bewertet werden. Der Umfang der Arbeit erstreckt sich neben einer Einlern- und Recherchephase auch auf die praktische Umsetzung und Mitwirkung im Projekt. Es handelt sich dabei um ein kommerzielles Softwareprojekt in Zusammenarbeit mit der Firma "IVM Technical Consultants". Der Einsatz erfolgt als lose gebundenes Mitglied eines fünfköpfigen Teams.

Die Bewertung der Einsetzbarkeit soll sich vor allem auf das Projekt beziehen und wird von zwei Arten von Faktoren getragen:
Einerseits gibt es allgemeine Aspekte, welche unabhängig vom konkreten Projekt sind und vor allem Stärken und Schwächen der AOP und Probleme bei deren Umsetzung beschreiben.
Andererseits muss auch das konkrete Umfeld im Team für die Bewertung in Betracht gezogen werden. Hier sind unter anderem die Kenntnisse und Fähigkeiten der Teammitglieder, der zeitliche Spielraum und die Akzeptanz von alternativen Ansätzen mit einzubeziehen.

Da der Fokus der Arbeit auf der Bewertung (und Umsetzung) für das konkrete Projekt liegt, können zwar einige allgemeine Gesichtspunkte - Stärken und Schwächen - gefunden werden; die allgemeine Bewertung der Einsetzbarkeit von AOP unabhängig von einem konkreten Projekt würde jedoch einen anderen Ansatz erfordern und den Fokus der Arbeit verschieben. Im Folgenden werden daher die Ergebnisse für das beschriebene Projekt und Projektteam präsentiert - es ist durchaus denkbar, dass gewisse Aspekte im Bezug auf andere Projekte oder andere Teams völlig anders zu bewerten sind.

2.1 Umfeld

Bei dem Projekt handelt es sich um eine kommerzielle Auftragsarbeit. Die Entwicklung erfolgt in *C#* bzw. im *.NET* Framework, als Vorgehensmodell kommt *SCRUM* zum Einsatz. Der zeitliche Rahmen ist mit fünf Monaten festgelegt und deckt sich mit dem Sommersemester 2009. Daher bietet sich die wertvolle Möglichkeit im Rahmen der Bachelorarbeit den Lebenszyklus eines kommerziellen Softwareprojektes von der Anforderungsphase bis zur Auslieferung mitzuerleben und so die erlernte Theorie praktisch zu „erfahren" und anzuwenden.

Teil der Anforderungen ist es die Software so zu gestalten, dass sie auch von einem Laien bedienbar ist. Daraus ergibt sich die Forderung nach einem möglichst umfassenden Logging, um im Fehlerfall aus den Log-Dateien und (Call-)Stackdumps auf die Ursache des Fehlers schließen zu können.

Der Benutzer kann als Laie nur sehr eingeschränkt zur Fehlerbehandlung herangezogen werden. Daher ergibt sich eine weitere Forderung nach einem möglichst umfassenden Exception-Handling, das den Benutzer so wenig wie möglich miteinbezieht. Trotzdem muss das Programm nach Möglichkeit in einem datenkonsistenten Zustand gehalten werden.

Darüber hinaus muss die Software mit anderen Applikationen von Drittherstellern eng zusammenarbeiten. Da das geplante Zielsystem nur über eingeschränkte Hardware-Ressourcen verfügt und die Software der Dritthersteller bereits einen Großteil dieser Ressourcen beansprucht, muss auf eine ökonomische Ausnutzung von Rechenzeit und Speicherverbrauch geachtet werden.

2.2 Einsatz im Projekt

Die erste Phase der Bachelorarbeit hat das Ziel, die Konzepte der Aspektorientierten Programmierung kennen zu lernen und die grundlegenden Ideen dahinter zu verstehen. Anschließend sollen verschiedene AOP-Frameworks für .NET bewertet und gegebenenfalls evaluiert werden. In Konkurrenz zu einer möglichst umfassenden Recherche-Phase steht die Anforderung der praktischen Umsetzung; durch SCRUM kann zwar agil reagiert werden, da aber ein Teil des Gesamtdesigns und der Aufgabenaufteilung vom gewählten AOP-Framework und dessen Umsetzungsmöglichkeiten abhängt, muss relativ rasch die Festlegung auf ein AOP-Framework erfolgen.

Für die aspektorientierte Umsetzung bieten sich vor allem das Logging und Exception-Handling an. Beide Anforderungen sind *Crosscutting Concerns* die sich über alle Teile des Projektes erstrecken.

Wie bereits erwähnt, soll das Logging möglichst umfassend sein. So sollen unter anderem Methodenaufrufe, Werte von Übergabeparametern, Rückgabewerte und Werte von bestimmten Properties protokolliert werden. Andererseits muss bei der Umsetzung auf eine möglichst performante Implementierung und eine ökonomische Ausnutzung des Speicherbedarfs der Log-Dateien geachtet werden. Da keine Möglichkeit besteht, die Log-Dateien in regelmäßigen Abständen zu sichern, muss der Log-Betrieb auch über mehrere Wochen funktionieren, ohne den vorhandenen Speicherplatz zu überfüllen.

Die Behandlung von Exceptions soll möglichst ohne Benutzer-Eingriff oder -Benachrichtigung erfolgen. In jedem Fall muss versucht werden die Konsistenz der Programmdaten aufrecht zu erhalten. Als letzte Möglichkeit bleibt nur mehr das kontrollierte Schließen der Applikation.

Für beide Anforderungen sollen aspektorientierte Lösungen gefunden und bewertet werden. Im Falle einer positiven Bewertung sollen die entsprechenden Lösungen umgesetzt werden.

3 Realisierung

Die Umsetzung des Projektes erfolgt in C# unter Microsofts Visual Studio 2005 auf Basis des .NET-Frameworks in der Version 2.0. Zur Vereinfachung des Datenbankzugriffs wird das NHibernate-Framework verwendet. NHibernate verwendet intern *Log4Net* als Logging-Framework.

3.1 Logging-Framework

Log4Net ist ein einfach zu verwendendes Logging-Framework, das bereits in einem sehr ausgereiften Entwicklungsstadium ist und vielfach eingesetzt wird. Das bekannte Java-Pendant heißt Log4J. Das Framework bietet die Möglichkeit Nachrichten in unterschiedlichen Formaten in eine Vielzahl von Datensenken zu schreiben. Darüber hinaus kann jede Nachricht mit einem Log-Level priorisiert werden. Diese Level sind hierarchisch angeordnet und es wird die Möglichkeit geboten, nur Nachrichten mit einem gewissen Mindestlevel und darüber zu loggen. Außerdem können parallel mehrere Logger verwendet werden, wobei jede Nachricht durch alle Logger durchgereicht wird und sich auch diese Logger hierarchisch gruppieren lassen. Die Konfiguration des Frameworks erfolgt über eine XML-Datei. Eine vollständige Übersicht aller Features, sowie Links zur kompletten Dokumentation finden sich unter [2].

Im Projekt gibt es die Anforderung Nachrichten mit unterschiedlichen Log-Levels in einem XML-Datenformat abzulegen. Darüber hinaus soll für die Log-Datei(en) eine Maximalgröße angegeben werden; wird diese überschritten, so soll beim Eintreffen neuer Nachrichten jeweils die älteste Nachricht aus der Datei entfernt werden - dieses Verhalten wird als „rollen“ bezeichnet. *Log4Net* erfüllt bereits alle diese Anforderungen und bietet als Datensenke einen sogenannten *RollingFileAppender* an. Bei der Implementierung von *Log4Net* wurde auf eine gute Laufzeit-Performance, auch bei massivem Logging-Betrieb geachtet. Da *Log4Net* bereits implizit durch NHibernate im Projekt ist und alle Anforderungen erfüllt, wird auf eine alternatives Logging-Framework oder auf eine Eigenimplementierung verzichtet.

3.2 AOP-Framework

Für .NET sind bereits mehrere AOP-Frameworks verfügbar. Aus Zeitgründen werden jedoch nur zwei davon evaluiert. Beide sind bereits in einem relativ ausgereiften Zustand und haben eine große und aktive Community. Es handelt sich dabei um *Spring .NET*, das aus dem bekannten Spring-Framework für Java hervorgegangen ist und um *PostSharp*. Beide Frameworks sind frei verfügbar und dürfen in kommerziellen Software-Projekten eingesetzt werden.

Spring .NET
Das gesamte Spring-Framework umfasst weit mehr als nur die aspektorientierte Programmierung (unter anderem: Dependency Injection, Testing, Messaging, Scheduling, Data Binding, Threading). Da aber für die Bewertung der Einsatzfähigkeit für das Projekt hauptsächlich der AOP-Teil des Frameworks relevant ist, wird im Folgenden nur auf den AOP-Teil von *Spring .NET* eingegangen.

Zur Bereitstellung von AOP-Funktionalität verwendet *Spring .NET* sogenannte „Dynamic Proxy Objekte“. Das Proxy-Objekt wirkt in diesem Fall als Vertreter für das Objekt auf das ein Aspekt angewandt werden soll und verfügt über das selbe Interface (d.h. über die selben Methoden und Properties). Somit lässt sich das ursprünglich Objekt einfach durch das Proxy-Objekt „ersetzen“. Zur Verdeutlichung folgt der beispielhafte Code eines beliebigen Objektes, sowie der Code für das zugehörige Proxy-Objekt. Der Einfachheit halber soll hier nur das Grundprinzip des Proxy-Objektes demonstriert werden - die Funktionalität zur dynamischen Modifikation während der Laufzeit wird hier nicht gezeigt.

```
public class MyObject
{
  public void AMethod()
  {
    //method body
  }
}
```

List. 3.1: Pseudocode für ein beliebiges Objekt

Der Pseudocode für das zugehörige Proxy-Objekt sieht folgendermaßen aus:

```
public class MyObjectProxy : public MyObject
{
  public override void AMethod()
  {
    //1.) code to be executed before original method call
    Before();

    try
    {
      //original method call
      base.AMethod();
    }
    catch(Exception e)
    {
```

```
            //2.) code for exception handling
            ExceptionHandler();
        }
        finally
        {
            //3.) code to be executed after original method call (even in case
                of an exception)
            After();
        }
    }

    //override this methods for custom aspects
    public void Before() {}
    public void ExceptionHandler() {}
    public void After() {}
}
```

List. 3.2: Zugehöriges Proxy-Objekt

Wird nun jede Instanz von *MyObject* durch *MyObjectProxy* ersetzt, so können an den Punkten 1.), 2.) und 3.) im Code zusätzliche Befehle ausgeführt werden ohne den restlichen Code modifizieren zu müssen. Im einfachsten Fall könnte von *MyObjectProxy* abgeleitet werden und die Methoden *Before*, *ExceptionHandler* und *After* könnten überschrieben werden. Um diesen zusätzlichen Code zur Laufzeit verändern zu können muss das Proxy-Objekt natürlich etwas komplexer aufgebaut sein (siehe Design Pattern: Dynamic Proxy) - bei Spring .NET kommt hier zur Laufzeit generierter IL-Code zum Einsatz. Somit wird das Rahmenwerk von Spring .NET bereits zur Verfügung gestellt.

Die „Konfiguration“ des gesamten Frameworks - und somit auch der AOP-Funktionalität - erfolgt vorwiegend über XML-Dateien. Die beispielhafte Konfiguration eines Pointcuts würde folgendermaßen aussehen:

```
<object id="settersAndAbsquatulatePointcut"
    type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut,
        Spring.Aop">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</object>
```

List. 3.3: Konfiguration eines Pointcuts über Regular Expressions

Der dargestellte Pointcut würde alle Methoden betreffen, die „set“ im Namen enthalten oder mit „absquatulate“ enden. Jeder Aspekt braucht neben einem Pointcut auch einen Advice. *Spring .NET* stellt dazu mehrere Advice-Basisklassen zur Verfügung, sodass für einen neuen Advice lediglich von der richtigen Basisklasse abgeleitet und die jeweilige Methode überschrieben werden muss. Es folgt der Pseudocode für einen „Before-Advice“ - der entsprechende Code wird also vor der betroffenen Methode ausgeführt.

```
public class CountingBeforeAdvice : IMethodBeforeAdvice {
```

```
    private int count;

    public void Before(MethodInfo method, object[] args, object target) {
        ++count;
    }

    public int Count  {
        get { return count; }
    }
}
```

List. 3.4: Konfiguration eines Before-Advices

Im Beispiel würde die Variable *count* vor jedem betroffenen Methodenaufruf inkrementiert werden. Das Verknüpfen von Pointcuts und Advices erfolgt in ähnlichem Stil entweder über die XML-Konfigurationsdateien oder direkt im Code.

AOP durch Proxy-Objekte bietet eine größtmögliche Flexibilität zur Laufzeit, hat allerdings (vor allem im Bezug auf das Projekt) den Nachteil, dass spürbare Performance-Verluste entstehen. Die gesamte Referenz-Dokumentation des Frameworks sowie die oben angeführten Beispiele mit ausführlichen Kommentaren sind unter [5] zu finden.

PostSharp
Im Gegensatz zu *Spring .NET* arbeitet *PostSharp* mit einem *Static Weaver*, wobei nach dem ersten Kompilier-Schritt der PostSharp-Weaver den aspektorientierten Code in einem zweiten Kompilier-Schritt in den ursprünglichen Code „einwebt“. Dadurch entstehen kaum Performance-Verluste zur Laufzeit - allerdings ist die *Code-Manipulation* nach dem Kompilieren bei einigen Entwicklern sehr verpönt.

Die Konfiguration/Programmierung erfolgt direkt im Quellcode mit den aus dem Sprachumfang von C# bekannten Attributen. Darüber hinaus werden (überschreibbare) Aspekt-Klassen zur Verfügung gestellt. Es folgt der (Pseudo-)Code für einen *OnMethodBoundaryAspect*. Diese Aspektklasse erlaubt das Ausführen von Code vor und nach dem eigentlichen Methodenaufruf.

```
[Serializable]
public sealed class TraceAttribute : OnMethodBoundaryAspect
{
    private String category;

    public TraceAttribute(String Category)
    {
      category = Category;
    }

    public override void OnEntry( MethodExecutionEventArgs eventArgs )
    {
        Trace.WriteLine(
            string.Format( "Entering {0}.{1}.",
                           eventArgs.Method.DeclaringType.Name,
                           eventArgs.Method.Name ),
            this.category );
    }

    public override void OnExit( MethodExecutionEventArgs eventArgs )
    {
```

```
            Trace.WriteLine(
                string.Format( "Leaving {0}.{1}.",
                               eventArgs.Method.DeclaringType.Name,
                               eventArgs.Method.Name ),
                this.category );
        }
```

List. 3.5: Ableitung eines Aspektes von OnMethodBoundary

Für jede Methode auf die der Aspekt angewandt wird, wird beim Eintritt und beim Verlassen eine Nachricht geloggt, die das Ereignis, den deklarierenden Typ und den Methodenname enthält. Um den Aspekt verwenden zu können, müssen die entsprechenden Methoden lediglich mit dem gleichnamigen Attribut dekoriert werden (Attribute in C# stehen in eckigen Klammern vor der Methodendeklaration):

```
internal static class Program
{
    private static void Main()
    {
        Trace.Listeners.Add(new TextWriterTraceListener( Console.Out));

        SayHello();
        SayGoodBye();
    }

    [TraceAttribute( "MyCategory" )]
    private static void SayHello()
    {
        Console.WriteLine("Hello, world." );
    }

    [TraceAttribute("MyCategory")]
    private static void SayGoodBye()
    {
        Console.WriteLine("Good bye, world.");
    }
}
```

List. 3.6: Beispielprogramm zur Anwendung von Aspekten unter PostSharp

Das Beispielprogramm ruft zwei Methoden auf - *SayHello* und *SayGoodBye*. Beim Ausführen des Programms wird vor dem String „Hello, world.“ der String „MyCategory: Entering Programm.SayHello.“ und nachher die Zeichenkette „MyCategory: Leaving Programm.SayHello.“ ausgegeben. Der gleiche Ablauf wiederholt sich beim zweiten Methodenaufruf.

Als weitere Möglichkeit bietet *PostSharp* sogenannte Multicast-Aspects an. Mit einem Multicast kann eine Vielzahl von Methoden mit dem selben Aspekt belegt werden - es ist nicht erforderlich jede Methode mit einem Attribut zu dekorieren. Für einen Multicast-Aspect, der den oben dargestellten TraceAspect auf alle Methoden im Namespace „My.BusinessLayer“ anwendet, muss folgender Code in der zugehörigen *AssemblyInfo.cs* abgelegt werden:

```
[assembly: Trace( "MyCategory", AttributeTargetTypes =
    "My.BusinessLayer.*")]
```

List. 3.7: Beispielprogramm zur Anwendung von Aspekten unter PostSharp

Da PostSharp als Static Weaver arbeitet ist es möglich, das erstellte Executable einem Reverse Engineering zu unterziehen und so den eingewobenen Code sichtbar zu machen.

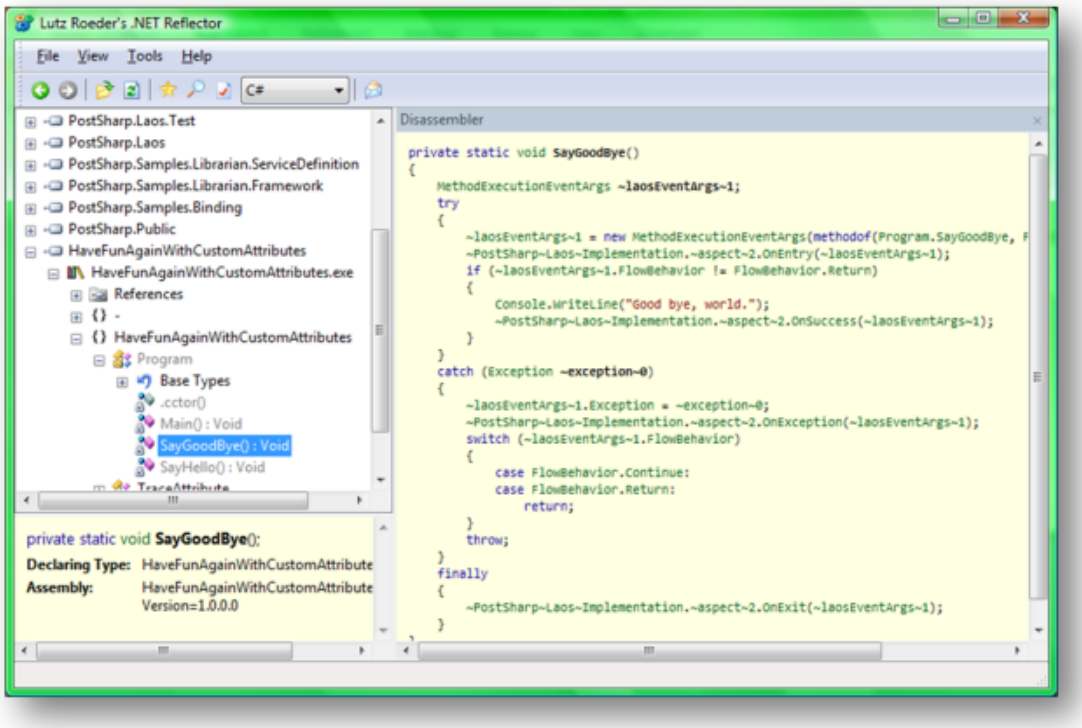

Abb. 3.1: Code nach dem Kompilier-Schritt mit dem Static Weaver

In Abb. 3.1 zu sehen ist ein Teil der Methode *SayGoodBye* mit dem von PostSharp eingewobenen Code.

Zur Zusammenarbeit von *PostSharp* mit *Log4Net* gibt es ein Plugin Namens *Log4PostSharp*. Wie auch *Spring .NET*, bietet *PostSharp* die Möglichkeit von Multicast-Aspects - die Angabe der *Targets* erfolgt dabei über Ausdrücke mit Wildcards (*) oder Regular Expressions.

Das vollständige Beispiel zur Verwendung von *PostSharp* kann unter [1] abgerufen werden. Ein Überblick über alle Features und die vollständige Dokumentation von *PostSharp* können unter [4] gefunden werden.

Enterprise Library
Als dritte Alternative ist die *Enterprise Library* von Microsoft zu nennen. Sie wurde von der Patterns & Practices Group entwickelt und zur Verfügung gestellt. Die *Enterprise*

Library ist in sogenannte „Application-Blocks“ unterteilt und umfasst unter anderem: Caching, Kryptographie, Data Access, Logging, Exception Handling und Policy Injection. Der Policy Injection Block bietet ein einfaches Rahmenwerk zur aspektorientierten Programmierung. Allerdings sind dessen Möglichkeiten stark eingeschränkt; so ist beispielsweise das Auslesen von Werten von Übergabeparametern nicht möglich. Darüber hinaus ist die „Policy Injection“ (die *Injection* einer Policy entspricht in etwa dem Anwenden eines Aspektes) nur für öffentlich sichtbare Methoden und Eigenschaften verwendbar. Außerdem werden die Matching-Regeln (für das Angeben von mehreren Targets) nur gegen statische Klasseninformation geprüft - Laufzeitinformation, wie z.B. Parameterwerte können daher nicht für das Matching verwendet werden. Es folgt ein simples Beispiel zur Verdeutlichung der Anwendung des Policy Injection Blocks.

```
public interface IOrder
{
    void Return(string reason);
}


<policies>
    <add name="Logging">
        <matchingRules>
            <add
                type="...TypeMatchingRule..."
                name="Type Matching Rule"
                match="IOrder"
                ignoreCase="false" />
            <add
                type="....MemberNameMatchingRule..."
                name="Member Matching Rule"
                match="Return"
                ignoreCase="false" />
        </matchingRules>
        <handlers>
            <add
                name="Logging Handler"
                type="...LogCallHandler..."
                logBehavior="Before"
                beforeMessage="Logging Return..."
                includeParameterValues="true"
                includeCallTime="true"
                includeCallStack="false"
                severity="Information">
                    <categories>
                        <add name="General" />
                    </categories>
            </add>
        </handlers>
    </add>
</policies>
```

List. 3.8: Policy Injection Application Block - Microsoft Enterprise Library

Das Beispiel zeigt die Definition der Policy: „Logging“. Zur Definition wird zunächst ein Interface deklariert. Danach wird die Policy Injection über das XML-Konfigurationsfile aufgesetzt. Dargestellt ist die Deklaration des Interfaces und ein Auszug aus der XML-Datei. In den „machtingRules“ wird angegeben, dass jeder Typ der das Interface *IOrder* implementiert, gematcht wird. Der Methodenaufruf der mit einem Aspekt belegt werden

soll trägt den Namen *Return*. Im zweiten Teil des XML-Ausschnitts wird der zugehörige „Handler“ festgelegt, der in diesem Fall Aufrufdaten mitprotokollieren soll.

Die *Enterprise Library* bietet auch einen Application Block für das Logging und einen für das Exception Handling an. Wie alle Blöcke der *Enterprise Library* werden auch diese über XML-Files konfiguriert. Allerdings wird mit der Library ein Tool mit grafischer Oberfläche ausgeliefert, das die Konfiguration erleichtern und beschleunigen soll. Der Logging Block bietet im Vergleich mit *Log4Net* mehr oder weniger die selbe Funktionalität an. Besonders zu erwähnen ist jedoch der Exception Handling Block, der es erlaubt relativ einfach eine komplexe Exception Handling Policy umzusetzen. Aus Zeitgründen muss auf eine genauere Evaluierung der *Enterprise Library* verzichtet werden, wobei hier weniger die Möglichkeiten zur AOP, sondern mehr die Einsatzfähigkeit des Exception Handling Blocks interessant wäre. Die vollständige Dokumentation der *Enterprise Library* kann unter [3] abgerufen werden.

Auswahl des Frameworks
Die Zielplattform für das Projekt verfügt nur über eingeschränkte Hardware-Ressourcen und diese werden durch Drittsoftware bereits zu einem Großteil belegt. Daher spielt die Laufzeit-Performance bei der Auswahl des AOP-Frameworks die größte Rolle. Aufgrund der Umsetzung mit einem *Static Weaver* und der dadurch besseren Laufzeit-Performance wird *PostSharp* als AOP-Framework verwendet. *Spring .NET* arbeitet mit Proxy-Objekten zur AOP und kann daher nicht mit der Laufzeit-Performance von *PostSharp* mithalten. Konkret kommt die Version 1.0 des *PostSharp*-Frameworks zum Einsatz, da diese Version im Gegensatz zu *PostSharp 1.5* unter .NET 2.0 lauffähig ist.

Neben der geringeren Auslastung der vorhandenen Ressourcen scheint das *PostSharp*-Framework einen einfacheren und intuitiven Einstieg zu bieten. *Spring .NET* weist einen deutlich höheren Funktionsumfang auf, ist aber in seiner Bedienung und Verwendung deutlich komplexer. Vor allem die Einlernphase dürfte somit beim Einsatz von *PostSharp* deutlich kürzer ausfallen. Da der volle Funktionsumfang von *Spring .NET* im Projekt nicht erforderlich ist und auch die Einlerndauer eine signifikante Rolle spielt, fällt auch in diesem Punkt die Entscheidung auf den Einsatz von *PostSharp*.

Die *Enterprise Library* bietet ebenfalls weit mehr als nur die AOP an. Im Gegenteil: Der Policy Injection Application Block stellt nur eine simple Möglichkeit zur AOP zur Verfügung. In der ersten Phase des Projektes war noch unklar, ob die AOP-Möglichkeiten der Enterprise Library ausreichen würden und für eine genauere Evaluierung der anderen Blöcke (insbesondere des Exception Handling Application Blocks) fehlten die zeitlichen Ressourcen. Daher fiel die Entscheidung gegen den Einsatz der Enterprise Library - ausschlaggebend dafür waren die eingeschränkten AOP-Möglichkeiten des Policy Injection Application Blocks.

4 Ergebnisse

4.1 Allgemein

Im Verlauf der Bachelorarbeit hat sich gezeigt, dass die *Aspektorientierte Programmierung* durchaus ein sehr mächtiges Konzept ist und in einer Vielzahl von Szenarien Vorteile im Vergleich zur objektorientierten Implementierung bringen kann. Oft ist jedoch ein hybrider Ansatz, der OOP mit AOP (bereits im Design) vermischt, die beste Wahl - in vielen Fällen kann dadurch die Lesbarkeit und Modularität des Codes deutlich gesteigert werden.

Die Einsetzbarkeit der AOP hängt somit Großteils von Faktoren ab, die sich auf die Fähigkeiten und Anforderungen im Projektteam beziehen. Es hat sich gezeigt, dass sich das Logging als reiner *Crosscutting Concern* sehr gut vom restlichen Code entkoppeln und getrennt umsetzen lässt. In diesem Fall ist es daher nicht nötig, dass alle Teammitglieder mit der AOP vertraut und in das *PostSharp*-Framework eingearbeitet sind. Es ist ausreichend, wenn sich ein Teammitglied einarbeitet und um die Umsetzung kümmert. Vom restlichen Team muss lediglich angegeben werden, welche Methoden / Parameter geloggt werden sollen.

Zur Verdeutlichung folgt ein Beispiel - zunächst wird die klassische Implementierung dargestellt; anschließend die aspektorientierte Umsetzung.

```
public class AClass
{
  public void AMethod()
  {
    Logger.log("`Entering AClass.AMethod"');

    //do something

    Logger.log("`Leaving AClass.AMethod"');
  }
}

public class AnotherClass
{
  public void AnotherMethod()
  {
    Logger.log("`Entering AnotherClass.AnotherMethod"');

    //do something

    Logger.log("`Leaving AnotherClass.AnotherMethod"');
  }
```

```
}
```

List. 4.1: Klassisches Logging

Zu sehen sind zwei Klassen die jeweils eine Methode enthalten. Das Eintreten und Verlassen dieser Methoden soll jeweils geloggt werden (die Variable *Logger* sei eine statische Instanz einer Logger-Klasse). Bei der klassischen Implementierung ist es also nötig, die Log-Statements am Beginn und Ende jeder Methode einzufügen. Im Gegensatz dazu lässt sich diese Aufgabe mit Hilfe der AOP zentralisieren.

```
public class AClass
{
  public void AMethod()
  {
    //do something
  }
}

public class AnotherClass
{
  public void AnotherMethod()
  {
    //do something
  }
}

[Serializable]
public sealed class LogAspect : OnMethodBoundaryAspect
{
  public override void OnEntry( MethodExecutionEventArgs eventArgs )
  {
    String msg = "`Entering "` + eventArgs.Method.DeclaringType.Name +
        "`."' + eventArgs.Method.Name;
    Logger.log(msg);
  }

  public override void OnExit( MethodExecutionEventArgs eventArgs )
  {
    String msg = "`Leaving "` + eventArgs.Method.DeclaringType.Name +
        "`."' + eventArgs.Method.Name;
    Logger.log(msg);
  }
}


//this code is placed in AssemblyInfo.cs
[assembly:LogAspect(AttributeTargetTypes ="*.*Method" )]
```

List. 4.2: Aspektorientiertes Logging

Anders als bei der klassischen Implementierung kann der Logging-Code nun aus den ursprünglichen Methoden herausgenommen werden - die Methoden werden schlanker, die Lesbarkeit des Codes wird verbessert. Die Klasse LogAspect implementiert nun das (generische) Logging beim Eintritt und Verlassen einer Methode. Mit dem Code in der letzten Zeile (das Assembly-Attribut) wird der LogAspect auf alle Methoden im Namespace angewandt, die in ihrem Namen den String „Method“ enthalten. Prinzipiell lassen sich damit also beliebig viele Methoden loggen. Zu erwähnen ist, dass üblicherweise alle Klassen in

verschiedenen Codedateien liegen - d.h. beim Lesen des Codes der Klasse *AClass* oder *AnotherClass* findet sich kein Hinweis mehr auf das vorhandene Logging. Für das prinzipielle Verständnis des Codes der jeweiligen Klassen sollte dies jedoch nicht relevant sein; es ist ausreichend zu wissen dass (aspektorientiert) geloggt wird.

Im Gegensatz zu den ersten Erwartungen zeigt sich jedoch für das aspektorientierte Exception-Handling (kurz: EH) ein anderes Ergebnis. Das Problem hat mehrere Ursachen - die Hauptgründe sind jedoch ein zu grobes EH-Design und eine „Verschmelzung der Belange“ (*Concern Diffusion*). Letzteres bezeichnet das Problem, dass zur Behandlung einer Exception in vielen Fällen kontextsensitive Information nötig ist. Der EH-Code lässt sich daher nur schwer vom restlichen Quellcode lösen - eine saubere Trennung („*Separation of Concerns*“) ist oft nicht möglich (siehe auch: 5.1).

Die Problematik soll anhand eines einfachen Beispiels demonstriert werden:

```
public class TheClass
{
  public float TheMethod(float a, float b)
  {
    double result = 0;

    try
    {
      result = a / b;
    }
    catch(Exception e)
    {
      if(e is DivideByZeroException)
      {
        if(a > 0)
          return a / 125;  //return result using default value for b
        else
          return FLTMAX;  //return maximum float value
      }
      else
        throw;  //something else went wrong - rethrow exception
    }

    return result;
  }
}
```

List. 4.3: Klassisches Exception Handling

Das Listing zeigt eine Klasse mit einer Methode, welche die zwei Übergabeparameter durchdividiert und das Ergebnis zurückgibt. Im Falle einer Division durch Null, wird in Abhängigkeit vom Wert von a jeweils ein verschiedenes Ergebnis zurückgegeben. Anzumerken ist, dass in diesem Fall die Division durch Null durch eine einfache If-Abfrage vor der Division zu vermeiden wäre; Zweck des Beispiels ist es jedoch den Unterschied zur aspektorientierten Umsetzung zu zeigen.

```
public class TheClass
{
```

```
  public float TheMethod(float a, float b)
  {
    double result = 0;
    result = a / b;
    return result;
  }
}

[Serializable]
public sealed class ZeroDivisionAspect : OnMethodBoundaryAspect
{
  public override void OnException(MethodExecutionEventArgs eventArgs)
  {
      float result = 0, a = 0;
      if(eventArgs.Exception is DivideByZeroException)
      {
        a = eventArgs.Method.Parameters[0];
        if(a > 0)
          result = a / 125;  //use default value for b
        else
          result = FLTMAX;   //set return value to maximum float value

        eventArgs.Method.ReturnValue.Set(result); //set return value of
            method accordingly
        eventArgs.Exception.handeled = true;  //mark exception as handeled

        //cause method to return immediatley
        eventArgs.Method.FlowBehaviour.Flow = FlowBehaviour.Return;
      }
      else
      {
        eventArgs.Exception.handeled = false; //mark exception as
            unhandeled --> causing a rethrow
      }
  }
}

//this code is placed in AssemblyInfo.cs
[assembly:ZeroDivisionAspect(AttributeTargetTypes ="*.*Method" )]
```

List. 4.4: Aspektorientiertes Exception Handling

Zu sehen ist die Aspektorientierte Umsetzung, die den selben Programmfluss nachbildet. Zunächst fällt auf, dass die ursprüngliche Methode (*TheMethod*) nun deutlich schlanker ist. Auf den ersten Blick sieht es so aus, als hätte sich die Lesbarkeit der Methode verbessert. Allerdings sei noch einmal darauf hingewiesen, dass die Methode und der *ZeroDivisionAspect* sowie das Assembly-Attribut (Multicast) in verschiedenen Code-Dateien liegen. Beim Lesen des Codes von *TheMethod* gibt es also keinen Hinweis, dass hier ein Exception-Handling zur Anwendung kommt.

Darüber hinaus enthält der Exception-Handler kontextsensitive Information. Während der Behandlung muss der Wert des Übergabeparameters „a“ abgefragt werden. Diese Abfrage ist jedoch Teil der „Business-Logic“ von *TheClass* und zum Verständnis des Codes von *TheMethod* müsste dieser Code dort ersichtlich sein. Durch die AOP wird dieser Teil der Business-Logic aus der Methode heraus in den Aspekt gezogen. Es ist nicht zu erwarten, dass dieser Handler für eine Vielzahl von anderen Methoden wiederverwendbar ist. Durch die Extraktion eines Teils der Logik von *TheClass* hat der Handler-Aspekt seine Generik

verloren. Es besteht also eine *Concern Diffusion* zwischen *TheClass.TheMethod* und dem *ZeroDivisionAspect*.

Der Exception-Handling Code kann zwar in eine eigene Datei ausgelagert werden - die logische Trennung zwischen Methode und Exception-Handler ist in diesem Fall jedoch nicht möglich. Der Handler hat somit jegliche Modularität und Generik verloren. Unweigerlich drängt sich die Frage auf, ob die AOP-Umsetzung gegenüber der klassischen Implementierung einen Mehrwert mit sich bringt. Nachdem der AOP-Handler auch nicht generischer als der klassische Handler ist und die Nachvollziehbarkeit des Codes bei der AOP-Lösung in Mitleidneschaft gezogen wird, bleibt lediglich die leicht verbesserte Modularität durch den AOP-Handler. Ob diese Modularität die beschriebenen Nachteile überwiegt ist im Einzelfall zu entscheiden.

Abgesehen von einfachen Fällen (wie z.B. Log-and-Ignore: Protokollieren und „Verwerfen“ der Exception) ist es daher schwierig einen generischen Handler zu schreiben. Das Wissen zur Behandlung der Exception hat der Entwickler, der den zugehörigen Code programmiert hat. Es ist daher nur schwer möglich ein einzelnes Teammitglied mit der Behandlung aller Exceptions zu beauftragen - vielmehr müsste jeder Entwickler in der Lage sein, die von ihm benötigten Handler selbst zu implementieren; und dazu müssten alle Mitglieder des Teams die AOP beherrschen und mit dem verwendeten AOP-Framework vertraut sein. Durch ein sehr detailliertes EH-Design könnte der beschriebenen Problematik teilweise entgegengewirkt werden.

Die AOP hat sich für das Projekt als bedingt einsatzfähig erwiesen. In Bereichen, die sich gut vom restlichen Projekt entkoppeln lassen, kann der Einsatz der AOP deutliche Vorteile erzielen. Allerdings ist die AOP kein „Allheilmittel“ um *Crosscutting Concerns* zu behandeln. Vor allem wenn generische Ansätze aufgrund von kontextsensitiven Anforderungen scheitern, birgt die AOP-Umsetzung oft einen deutlichen Mehraufwand ohne die Nachvollziehbarkeit oder Modularität des Codes deutlich zu verbessern.

4.2 Logging

Eine der Anforderungen im Projekt war es, ein möglichst umfassendes Logging zur Protokollierung im Fehlerfall bereitzustellen. Für diese Anforderung eignet sich die AOP mit Multicast-Aspekten hervorragend. Mit einer einzigen Deklaration könnten beispielsweise alle Methodenaufrufe innerhalb einer Assembly geloggt werden.

Die ersten Versuche möglichst umfassend zu loggen, führten zu starken Performance-Einbrüchen der Applikation. Allein das Starten der Anwendung erzeugte über 16.000 Log-Nachrichten und dauerte über 30 Sekunden (anstelle von ursprünglich ca. 4) - ein Großteil davon betraf Konstruktoren, Getter und Setter und Lese-Methoden.

Nach einer Analyse der produzierten Log-Messages konnten viele irrelevante Quellen bestimmt werden. Ein schrittweises „Herausnehmen“ dieser Quellen mit anschließenden Perfomance-Tests führte dann zu einer zufriedenstellenden Performance bei ausreichender Informationsdichte. Um auch den zur Verfügung stehenden Speicherplatz effizient auszunutzen wurde ein dreistufiger Logger eingeführt, wobei jede Stufe in eine eigene Log-Datei mit festlegbarer Maximalgröße abgelegt wird:

- Detailed - protokolliert alle Nachrichten; füllt schnell den vorhandenen Speicherplatz aus und gibt im Fehlerfall detaillierte Information über die unmittelbar zurückliegenden Ereignisse.
- Medium - protokolliert keine Nachrichten auf dem untersten Level; gibt im „Volllast-Betrieb“ Auskunft über die letzten Stunden.
- Long-Term - protokolliert nur Nachrichten ab dem Level WARNING und darüber (ERROR und FATAL); kann im Normalbetrieb mehrwöchig betrieben werden, ohne den vorhandenen Speicherplatz vollständig auszufüllen.

Dadurch kann sichergestellt werden, dass die Informationsdichte im Fehlerfall hoch ist und im Langzeitbetrieb keine wichtigen Nachrichten (Level WARNING und darüber) verloren gehen.

4.3 Exception-Handling

Da nicht alle Teammitglieder mit den Konzepten der AOP und mit dem *PostSharp*-Framework vertraut waren, war es nicht möglich, dass jeder Entwickler die von ihm benötigten Exception-Handler selbst implementiert. Andererseits scheint es wenig sinnvoll, alle Handler von einem einzelnen Teammitglied entwickeln zu lassen, da dies einen erheblichen Mehraufwand an Kommunikation und Entwicklungszeit bedeuten würde. Darüber hinaus bietet die AOP in diesem Fall kaum Mehrwert - überzogen gesprochen kann lediglich der Code für die Exception-Handler in eine eigene Datei ausgelagert werden. Im Gegensatz zum Logging wird durch das Zentralisieren des Codes der restliche Quelltext nicht lesbarer; im Gegenteil: die Behandlung von Exceptions wäre nur mehr schwer nachvollziehbar. Eine Alternative wäre der Einsatz eines EH-Frameworks, das auf aspektorientierter Basis arbeitet, gewesen. Damit hätte jeder Entwickler die von ihm benötigten Handler selbst implementieren können, ohne mit der AOP oder *PostSharp* vertraut sein zu müssen. Die Umsetzung eines derartigen „AOP-EH-Frameworks“ hätte den Rahmen dieser Bachelorarbeit gesprengt und konnte aufgrund der kurzen Projektdauer nicht durchgeführt werden.

Aus diesen Gründen fiel die Entscheidung, das Exception-Handling zunächst klassisch zu implementieren und nach Möglichkeit ganz oder teilweise einem Refactoring in Richtung AOP-Exception-Handling zu unterziehen. Um ein späteres Refactoring zu erleichtern ist es nötig, bestimmte Konstrukte, die sich nachher nur mit deutlichem Aufwand adaptieren

lassen, zu vermeiden. Die wichtigsten Regeln für ein späteres Refactoring des Exception-Handling-Codes wurden in einer Guideline zusammengefasst (siehe: A.1). Die größten Probleme für ein AOP-Refactoring des Exception-Handlings sind der Zugriff auf nicht sichtbare Variablen und ein schwer nachzubildender Programmfluss. Zur Verdeutlichung ein einfaches Beispiel:

```
public class Dummy
{
  public void method()
  {
    int x = 0;
    try
    {
      x = n();   //n might throw an exception
    }
    catch(Exception e)
    {
      x = p();  //write access to local variable
    }

    q(x);
  }
}
```

List. 4.5: Zugriff auf lokale Variablen im Exception-Handler

Das Problem bei der aspektorientierten Umsetzung liegt hier beim Schreibzugriff auf die lokale Variable - es gibt keine Möglichkeit von einem Aspekt direkt auf eine lokale Variable zu schreiben.

```
public class Dummy
{
  [EHAspect]
  public void method()
  {
    int x = 0;
    x = n();   //n might throw an exception
    q(x);
  }
}

[Serializable]
public sealed class EHAspect : OnMethodBoundaryAspect
{
  public override void OnException(MethodExecutionEventArgs eventArgs)
  {
    //calculate new value for x
    int value = n();

    //write new value to x
    ???
  }
}
```

List. 4.6: Zugriff auf lokale Variablen im AOP-Exception-Handler

Um den Programmfluss nachzubilden wäre ein Refactoring nötig. Beispielsweise könnte die Variable x als privates Feld der Klasse *Dummy* deklariert werden. Eine andere Möglichkeit wäre es, die Methode *method* in zwei Methoden zu unterteilen; in der einen Methode den Wert von x zurückzugeben und diesen Wert der anderen Methode mitzugeben. Beide Lösungen sind aber im Sinne eines sauberen Designs möglicherweise unbefriedigend.

Zur Behandlung von Exceptions ist es oft erforderlich die Exception in Abhängigkeit von bestimmten Parametern verschieden zu behandeln (siehe auch 4.1). Diese Parameter hängen vom Kontext der Exception ab und können beispielsweise Werte von lokalen Variablen, Werte von privaten Variablen eines Objektes, etc. sein. Der Zugriff auf diese Parameter gestaltet sich mit der AOP schwierig und ist in vielen Fällen mit einem Refactoring verbunden, das die Kapselung des zugehörigen Objektes aufweicht oder lokale Variablen einer Methode als Attribute eines Objektes offen legt.

Ein weiteres Problem besteht darin, dass mit der AOP nicht jeder beliebige Programmfluss genau nachgebildet werden kann. Können z.B. im Körper einer Schleife Exceptions auftreten, und soll beim Eintreffen einer Exception nur der aktuelle Schleifendurchlauf abgebrochen werden (*Loop-Iteration-Handler*), so ist es unter Umständen nötig, ein massives Refactoring durchzuführen, um den selben Programmfluss aspektorientiert nachbilden zu können.

```
public class Dummy
{
  public void method()
  {
    int x,y,z;
    for(int i=0; i<10; i++)
    {
      try
      {
        //any of the methods below might throw an exception
        x = a();
        y = b();
        z = c(x,y);
        n(z);
      }
      catch(Exception)
      {
        continue;
      }
    }
  }
}
```

List. 4.7: Loop-Iteration-Handler

Die Problematik liegt hier in der Art wie ein AOP-EH-Aspekt um eine Methode gelegt wird. Im AOP-Fall wird um die ganze Methode (*method*) ein try-Block gelegt. Beim Auftreten einer Exception kann folglich die Schleife nicht fortgesetzt werden, da dazu der try-catch-Block innerhalb des Schleifenkörpers liegen müsste. Eine Alternative wäre es, den EH-Aspekt nicht auf *method* anzuwenden, sondern auf die Methoden innerhalb des Schleifenkörpers (a,b,c,n). Dazu wäre es im schlechtesten Fall nötig vier eigene Handler-Aspekte

(einer pro Methode) zu implementieren. Alternativ dazu kann auch der gesamte Schleifenkörper in eine eigene Methode ausgelagert werden, auf die dann der Aspekt angewandt wird. Würden im catch-Block noch Schreib-/Lesezugriffe auf lokale Variablen erfolgen oder nach dem catch-Block noch weitere Statements folgen, so verschärft sich die Problematik weiter und die zugehörigen AOP-Refactoring-Maßnahmen würden noch umfassender ausfallen. Und erneut stellt sich dann die Frage nach dem Mehrwert der AOP-Lösung - wenn weder Modularität noch Generik der Handler deutlich verbessert werden können, warum dann ein derartiges Refactoring durchführen, wenn dadurch die Nachvollziehbarkeit des Codes in Mitleidenschaft gezogen wird?

Die Problematik kann zumindest teilweise durch ein detailliertes EH-Design im Vorfeld abgeschwächt werden. So könnte der Schleifenkörper von vorn herein in eine eigene Methode ausgelagert werden. Oder die Methoden *a,b,c,n* könnten nach Möglichkeit so gestaltet werden, dass ein einziger (generischer) Handler für alle vier Methoden ausreicht. Genauso können Schreib-/Lesezugriffe auf lokale Variablen eventuell auch bereits im Vorfeld umgangen werden.

Im weiteren Verlauf der Bachelorarbeit wurden weitere Gründe gefunden, die gegen ein Refactoring bzw. eine aspektorientierte Umsetzung des EH sprechen (siehe: 5.3 und [8]). Somit wurde die klassische Implementierung des Exception-Handlings beibehalten.

Aspektorientiertes Exception-Handling wäre für dieses Projekt nur mit deutlichem Mehraufwand und kaum Mehrwert (es gab kein Verlangen nach einem hochgradig modularen EH) durchzuführen. Daher wird die Einsatzfähigkeit der AOP für das Exception-Handling dieses konkreten Projektes als unzureichend bewertet.

5 Diskussion

5.1 Diffusion of Concerns

Um die Problematik besser illustrieren zu können, wird folgendes Beispiel herangezogen: In einem beliebigen System gibt es eine Methode, die eine Datenbankabfrage durchführt und die Resultate der Abfrage zurück liefert. Der zugehörige Ablauf wird im Folgenden als *Kernlogik* bezeichnet bzw. wäre der Ablauf ein Teil der *Business-Logic* des zugehörigen Objektes. Zusätzlich soll jeder Aufruf einer Methode, die auf die Datenbank zugreift und das erhaltene Resultat geloggt werden und aus Performance-Gründen soll ein simples Caching zum Einsatz kommen (das heißt, dass die Resultate der Datenbank-Abfragen zwischengespeichert werden, um die Performance bei mehrfachen Lesezugriffen auf das selbe Resultat zu erhöhen).

Diese beiden Anforderungen oder Belange (engl.: *Concerns*) betreffen zwar auch die oben genannte Methode, gehören aber nicht zur *Kernlogik* bzw. zur *Business-Logic* des zugehörigen Objektes. Trotzdem wird bei einer klassischen (objektorientierten) Implementierung der Code zur Behandlung aller drei *Concerns* im Körper der Methode zu finden sein. Die *Kernlogik* wird dadurch schwerer nachvollziehbar - die Lesbarkeit und Modularität des Codes wird beeinträchtigt.

Laut den Autoren von [11] gibt es seit ca. 10 Jahren Bestrebungen eine sogenannte *Separation of Concerns* herbeizuführen bzw. deren Umsetzung zu begünstigen. Durch diese Trennung (der Codeteile) soll die Komplexität von Quellcode reduziert und seine Wiederverwendbarkeit erhöht werden. Als prominentesten Ansatz das Problem zu lösen, geben die Autoren die aspektorientierte Programmierung an.

Die AOP erlaubt es, den Code für das Logging und Caching im vorherigen Beispiel aus dem Körper der Methode herauszunehmen und ihn stattdessen in Form von Aspekten „auf die Methode anzuwenden“. Das Logging und Caching würde im zugehörigen Projekt vermutlich noch weitere Methoden betreffen - es handelt sich also um *Crosscutting Concerns* (das heißt der Code, der das Logging und Caching betrifft zieht sich durch das gesamte Projekt). Mit Hilfe der AOP lässt sich dieser „querschnittliche“ Code zentralisieren. Dadurch werden sowohl das Logging, als auch das Caching wesentlich modularer und könnten somit einfach gewartet oder ausgetauscht werden. Darüber hinaus verbessert sich die Lesbarkeit des gesamten Codes, da in den jeweiligen Methodenkörpern nur mehr die Logik zur Behandlung eines *Concerns* enthalten ist.

Letzteres trifft jedoch nicht auf alle *Crosscutting Concerns* zu. Angenommen beim vorherigen Beispiel könnten Exceptions beim Zugriff auf die Datenbank auftreten. Es wäre ein entsprechender Handler in der zugehörigen Methode zu implementieren. Die AOP würde es nun erlauben, diesen Handler „aus der Methode heraus zu nehmen“ und als Aspekt anzuwenden. Dadurch würde zwar die Lesbarkeit des Codes im fehlerfreien Fall verbessert werden; aber die Nachvollziehbarkeit im Fehlerfall wäre äußerst schlecht, da in der Methode selbst kein Hinweis auf den Exception-Handler mehr ersichtlich wäre. Obwohl auch das Exception-Handling ein *Crosscutting Concern* ist, der sich in allen Teilen des Projektes wiederfindet, ist hier eine Zentralisierung nicht immer ratsam. Im Falle von Logging reicht es vollkommen aus zu wissen, dass die jeweilige Methode geloggt wird; wo der zugehörige Code steht und wie genau geloggt wird, ist jedoch für das Nachvollziehen der jeweiligen *Kernlogik* irrelevant.

In Fällen wie z.B. dem Exception-Handling spricht man von einer sogenannten *Concern Diffusion*. Die *Concerns* sind so eng aneinander gekoppelt, dass eine *Separation of Concerns* zwar technisch möglich ist, allerdings die Nachvollziehbarkeit des Codes stark verschlechtert wird. Für eine ausführlichere Diskussion sei auf die Artikel [7] und [8] verwiesen.

5.2 Nachvollziehbarkeit des aspektorientierten Codes

Wie aus dem vorhergehenden Kapitel zu sehen ist, kann die AOP die Lesbarkeit von Code erhöhen - in ungünstigen Fällen aber auch deutlich verschlechtern. Unabhängig vom verwendeten AOP-Framework entsteht außerdem das Problem, dass das „Auftrennen“ der Codeteile (*Separation of Concerns*) die Nachvollziehbarkeit des Codes stark beeinträchtigt. Wie im Beispiel des AOP-Exception-Handlings schon gesehen (siehe: 5.1), befindet sich im Methodenkörper keine Zeile Code die auf das Abfangen und Behandeln der Exception hindeuten würde. Auf den ersten Blick ist es daher nicht ersichtlich, ob nun ein Handler zur Anwendung kommt oder nicht. Vor allem beim Einsatz von Multicast-Aspekten (mit Multicasts kann ein Advice auf eine Vielzahl von Methoden angewandt werden) wäre es nötig händisch zu überprüfen, welcher Multicast die entsprechende Methode mit einschließt. Es ist also nicht (einfach) ersichtlich, an welchen Stellen im Code Aspekte zur Anwendung kommen.

Umgekehrt ergibt sich eine ähnliche Problematik: Bei der Deklaration eines Multicast-Aspektes können die *Targets* über Ausdrücke mit Wildcards oder über Regular Expressions definiert werden. Allerdings gibt es keinerlei Rückmeldung, wie viele und welche Ziele vom Multicast tatsächlich betroffen sind - ein einfacher Tippfehler reicht aus, um eine Vielzahl von Aspekten unwirksam zu machen. Wenn nicht explizit danach getestet wird, kann nicht mit völliger Sicherheit angegeben werden, ob wirklich alle beabsichtigten Methoden vom Multicast „erwischt“ wurden. Noch ungünstiger, aber durchaus praxisrelevant wäre folgender Fall: eine große Anzahl von Methoden (30 bis 40 als realistische Größenordnung) wurde nach einem bestimmten Namensschema benannt, um nachher möglichst

einfach über einen Multicast mit einem Aspekt belegt zu werden. Wenn nun eine Methode durch eine Unachtsamkeit, einen Tippfehler, oder Ähnliches falsch benannt wurde, so wird der Fehler vermutlich auch nicht durch stichprobenartiges Testen mit irdendeiner Methode gefunden. Eine saubere Implementierung würde für jedes Ziel jedes Aspektes einen zugehörigen Test erfordern.

Beide Probleme sind substantiell für die AOP und ließen sich durch eine saubere Dokumentation im Code und äußerst sorgfältiges Arbeiten vermeiden. In der kommerziellen Praxis ist es jedoch undenkbar sich ausschließlich auf diese beiden Faktoren zu verlassen. Was für .NET fehlt ist die Integration der AOP in die Entwicklungsumgebung. Das Problem, dass unklar ist, ob auf den aktuellen Code / die aktuelle Methode Aspekte angewandt werden, ließe sich durch eine entsprechende Anzeige in der IDE relativ einfach lösen. Beispielsweise wäre es denkbar, alle Aspekte, die auf eine Methode angewandt werden, als interaktiven Kommentar vor die jeweilige Methode zu stellen; ein Klick darauf öffnet das Code-Fenster des zugehörigen Aspektes bzw. seiner Deklaration. Damit würde auch das zweite Problem zumindest teilweise entschärft. Nach der Deklaration eines Aspektes bzw. eines Multicasts mit wenigen Zielen könnte eine Sichtprüfung der entsprechenden Methoden ausreichen, um festzustellen, ob alle beabsichtigten Ziele mit dem Aspekt belegt wurden. Bereits im Jahr 2000 stellten die Autoren von [10] eine ähnliche Anforderung für das Java-Framework *AspectJ*.

Momentan ist für .NET noch kein AOP-Framework verfügbar, das derart in eine IDE integriert wurde und die angesprochenen Features anbietet. Die Entwickler von *PostSharp* geben jedoch in ihrem Forum an, dass derartige Features in Arbeit sind und mit dem nächsten Release (Version 2.0) in der Professional Edition erhältlich sein sollen (siehe: [6]). Wie die Implementierung genau aussieht und wann mit dem Release zu rechnen ist, wird aber nicht bekanntgegeben. Für die massentaugliche kommerzielle Verwendung wäre eine derartige Funktionalität zumindest ein Grundstein.

5.3 Aspektorientiertes Exception Handling

Der Titel des Artikels [7] lautet besonders treffend: „*Exceptions and Aspects: The Devil is in the Details*“. Im gleichen Artikel wird die Problematik auf den Punkt gebracht: „*Even though introductory texts [14, 16]* (Quellenangaben beziehen sich auf das Original) *often cite exception handling as an example of the (potential) usefulness of AOP, only a few works attempt to evaluate the suitability of this new paradigm to modularize exception handling code.*“ (Filho et. al. in [7], Kapitel 5. Related Work)

Es lässt sich tatsächlich eine Vielzahl von einführenden AOP-Texten und (Code-)Beispielen finden, welche das AOP-Exception-Handling und seine Vorteile demonstrieren sollen. Vor allem Logging, Caching und Exception-Handling finden sich immer wieder als „perfekt für eine AOP Umsetzung geeignet“. Allerdings handelt es sich dabei meist um Minimal-Beispiele, bei denen anhand von ein, zwei Exception-Handlern die AOP-Implementierung

demonstriert wird. Die angesprochenen Probleme werden allerdings erst bei Umsetzungen in größerem Umfang schlagend - die vorgeschlagenen Lösungen skalieren nicht ausreichend.

Die Autoren von [10] haben sich mit dem Refactoring des Exception-Handlings eines Java-Frameworks beschäftigt. Die erzielten Resultate wurden als durchaus positiv bewertet, allerdings konnten auch einige Schwachstellen gefunden werden. Unter anderem wurde die mangelhafte Nachvollziehbarkeit des AOP-Codes angesprochen, da sich AOP-Code und der Code, wo der jeweilige Aspekt zur Anwendung kommt, in verschiedenen Dateien befinden und keine offensichtlichen Verweise auf das jeweilige Gegenstück vorhanden sind (siehe: 5.2). Als mögliche Abhilfe sprechen die Autoren (bereits im Jahr 2000, allerdings für Java) eine entsprechende IDE bzw. IDE-Integration an.

Zum Thema „Refactoring des EH mit Hilfe der AOP“ lassen sich einige wissenschaftliche Artikel finden. So kommen die Autoren von [10] wie bereits erwähnt zu positiven Resultaten. Allerdings sind dabei zwei Details zu beachten:
Zur Bewertung der Umsetzung wurde von den Autoren hauptsächlich die Anzahl der „Lines of Code“ (LOC) herangezogen. Allerdings sind LOC allein kein aussagekräftiges Maß für die Qualität der Umsetzung (die selbe Kritik wurde auch von den im Artikel [7] geäußert).
Außerdem konnten die Autoren mit der bestehenden Implementierung Messungen durchführen und so feststellen, welche Exceptions mit welcher Häufigkeit auftreten und wie viele unterschiedliche Behandlungspfade pro Exception existieren. Anschließend wurden nur die plausibelsten Exception-Handler aspektorientiert umgesetzt. Der Großteil dieser Handler war wenig komplex und somit stießen die Autoren auf wenige Probleme bei der Umsetzung.

Im Vergleich dazu beschäftigten sich die Autoren von [7] mit mehreren Projekten und zogen zur Bewertung der Qualität der Umsetzung eine Vielzahl von Faktoren in Betracht (unter anderem: Concern Diffusion unter drei verschiedenen Gesichtspunkten, Kopplung zwischen Komponenten, Tiefe der Vererbungshierarchie, LOC, Anzahl von Attributen, Anzahl von Operationen, ...). Die Autoren kamen zum Schluss, dass es weitaus schwieriger sei, als in gängiger Literatur behauptet, Exception-Handler wiederverwendbar zu gestalten. Aus ihren Resultaten kamen sie zum Schluss, dass das Refactoring des Exception-Handlings nur in gewissen Fällen sinnvoll sei und dass andere, problematische Fälle besser in der Original-Implementierung belassen werden sollten. Die jeweilige Bewertung hänge aber von einer Vielzahl von Faktoren ab. Generell aber raten die Autoren dazu, das AOP-EH von Beginn an in Betracht zu ziehen und ins Design mit aufzunehmen, um problematische Fälle von vorn herein vermeiden zu können.

Ein Teil der selben Autoren verfasste etwas später den Artikel [8] in dem eine Guideline bzw. ein „*Cookbook*“ für das AOP-Refactoring des Exception-Handlings bereitgestellt werden soll. Nach einer Einführung werden die problematischen Fälle detailliert beschreiben und kategorisiert. Anschließend folgt eine „Anleitung“ für das Refactoring der einzelnen Fälle. Auch hier geben die Autoren zu bedenken, dass die AOP ein schwaches EH-Design

nicht „ausbügeln“ kann; allerdings kann die AOP bei einem vorhandenen starken EH-Design weitere Verbesserungen mit sich bringen.

5.4 Exception-Handling Framework

Anstelle einer aspektorientierten Umsetzung des Exception-Handlings, wäre für das Projekt der Einsatz eines EH-Frameworks wesentlich passender gewesen. Ob dieses Framework im Hintergrund aspektorientiert arbeitet oder nicht, ist hierbei zweitrangig. Eine umfassende Recherche nach EH-Frameworks war im Rahmen der Arbeit nicht vorgesehen und aufgrund des engen Zeitplans im Projekt nicht möglich. Trotzdem konnte zumindest der *Exception-Handling Application Block* (EHAB) von Microsofts *Enterprise Library* kurz unter die Lupe genommen werden.

Die wesentliche Stärke des EHAB liegt in der impliziten Strukturierung des Exception-Handlings. Es wird die Möglichkeit geboten, beliebig viele *Policies* zu definieren. Zu jeder Policy können beliebig viele Exception-Klassen angegeben werden, die von der Policy behandelt werden. Die selbe Exception-Klasse kann auch in mehreren Policies unterschiedlich behandelt werden. Außerdem kann für jede Exception (innerhalb jeder Policy) eine Mehrzahl von Handlern angegeben werden.

Diese dreistufige Strukturierung (Policy - Exception-Klasse - Handler) bringt mehrere Vorteile mit sich:
Durch die Möglichkeit mehrere Handler pro Exception zu vergeben, wird die Tendenz dazu wiederverwendbare, generische Handler zu schreiben, erhöht. Außerdem bietet die Library bereits einige Standard-Handler (Log-and-Ignore, Cast-and-Rethrow, ...) an.
Durch die eingeführten Policies wird auch der EH-Designprozess unterstützt - oft besteht die Anforderung auf die selbe Exception an verschiedenen Stellen im Code unterschiedlich zu reagieren (z.B. bei einer Null-Reference-Exception). Mit Hilfe von Policies lassen sich diese Anforderungen gut strukturieren und sauber umsetzen.

Aus diesen Überlegungen konnten einige Anforderungen an ein EH-Framework für das Projekt gestellt werden:

- Die Einlernphase soll möglichst gering sein.
- Der Code soll übersichtlich und einfach nachvollziehbar sein (vor allem im Hinblick auf die mögliche Verwendung von Aspekten).
- Die Modularität und Wiederverwendbarkeit von Handlern soll durch das Framework begünstigt werden. Darüber hinaus soll eine Reihe von Standard-Handlern verfügbar sein.
- Es soll die Möglichkeit geben, das Exception-Handling logisch bzw. hierarchisch gruppieren zu können. Zumindest eine dreistufige Struktur (Policy - Exception-Klasse - Handler) soll einfach umgesetzt werden können.

- Der Einsatz des Frameworks darf keinen erheblichen (zeitlichen) Mehraufwand bedeuten. Außerdem soll durch die Verwendung des Frameworks ein Mehrwert gegenüber klassischem EH entstehen.
- Es darf kein signifikanter Performance-Verlust zur Laufzeit entstehen.
- Das Framework muss weitgehend fehlerfrei sein (bezieht sich vor allem auf eine Eigenimplementierung).

Es wäre durchaus möglich gewesen ein EH-Framework in kurzer Zeit selbst zu implementieren. Allerdings hätte dies einerseits den Rahmen der Bachelorarbeit gesprengt und andererseits wäre es wohl kaum möglich gewesen, alle oben genannten Anforderungen in zufriedenstellendem Maß zu erfüllen. Ein schnell implementiertes, simples EH-Framework wäre hier wohl mehr Schaden als Nutzen - und die Entwicklung eines ausgereiften, umfassenden EH-Frameworks würde vermutlich mehr Ressourcen benötigen, als im gesamten Projekt zur Verfügung standen.

Es ist nicht außer Acht zu lassen, dass beim Einsatz alternativer EH-Konzepte und -Technologien (Frameworks, AOP, ...) vor allem das EH-Design eine sehr große Rolle spielt. Es gibt aber Projekte, bei denen die Anforderungen an das Exception-Handling eher simpel sind. Dort macht es weniger Sinn viele Ressourcen ins EH-Design zu stecken. Vor allem erfahrene Entwickler können diese Aufgabe während der Implementierung umsetzen; ein Grobdesign ist ausreichend. Wenn allerdings die verwendete EH-Technologie (wie z.B. die AOP) ein umfassendes EH-Design erfordert, so ist die Einsetzbarkeit für derartige Projekte auf jeden Fall zu hinterfragen. Im gegenteiligen Fall - wenn hohe Anforderungen an das EH gestellt werden, oder die Anforderungen sehr komplex sind - muss ein detailliertes EH-Design erstellt werden. Und genau in solchen Fällen stößt das klassische Exception-Handling oft an seine Grenzen und der Einsatz alternativer EH-Technologien bietet sich an.

6 Schluss

Das Potential und die theoretische Einsetzbarkeit des Konzeptes „*Aspektorientierte Programmierung*“ stehen außer Frage. Vor allem als Ergänzung zur OOP, kann die AOP oft genau die Schwächen der OOP kompensieren und zu sehr effizienten, modularen und eleganten Umsetzungen führen. Allerdings ist die AOP kein „Allheilmittel“ und ihre Verwendbarkeit hängt stark vom konkreten Umfeld und den gestellten Anforderungen ab. Für bestimmte Aspekte eines Projektes können die Methoden der AOP sehr gut geeignet sein, während sich andere Teilaufgaben besser mit anderen Konzepten umsetzen lassen.

Die AOP bringt in bestimmten Szenarien deutliche Vorteile und kann die Modularität und Wartbarkeit von Code signifikant steigern. Allerdings gilt dies nur für sehr spezielle Fälle - in anderen Fällen bringt die Verwendung der AOP keine Verbesserung mit sich; in manchen Fällen stellt eine aspektorientierte Umsetzung gar eine Verschlechterung zur klassischen Implementierung dar. Vor allem Fälle, bei denen die AOP hervorragend funktioniert sind relativ weit verbreitet und verleiten dazu, die AOP als Universallösung zur Handhabung von *Cross-Cutting-Concerns* anzusehen. Dies führt dann meist zu einer viel zu breiten Anwendung der AOP und in weiterer Folge oft zu suboptimalen Lösungen. Daher muss die Bewertung der Einsetzbarkeit der AOP immer für den jeweiligen Task geprüft werden und kann nicht pauschal für ein ganzes Projekt bestimmt werden. Dieser Prozess kann mühsam und zeitintensiv sein und stellt Entwickler vor eine nicht-triviale Aufgabe. Große Softwareschmieden sind fast schon dazu gezwungen neue Paradigmen in die Praxis umzusetzen aber gerade in kleinen Softwareteams fehlt oft die Zeit für derartige Evaluierungen, wodurch der Einsatz der AOP oft a-priori abgelehnt wird. Prägnante, ausgereifte *AOP-Design-Patterns* könnten (vor allem „kleinen“ Softwareentwicklern) den Einsatz der AOP erleichtern.

Es gibt zwar durchaus Vorschläge für AOP-Patterns und -Antipatterns. Viele davon sind jedoch nicht Ergebnisse von wissenschaftlichen Betrachtungen der Thematik. Beispielhaft sei das Exception-Handling erwähnt, das oft als passend für eine AOP-Umsetzung angesehen wird - genauere wissenschaftliche Untersuchungen haben jedoch gezeigt, dass dies nur für sehr bestimmte Fälle uneingeschränkt gilt. Das Hauptproblem besteht darin, dass es kaum möglich ist, Design-Patterns „im Vorhinein“ zu definieren; vielmehr werden sie aus immer wiederkehrenden Mustern in einer Vielzahl von Projekten abgeleitet.

Vielversprechend wäre auch das AOP-Refactoring von bestehenden Design-Patterns. Für bekannte Entwursmuster wäre zu prüfen, ob eine AOP-Umsetzung Vorteile gegenüber der klassischen Implementierung bietet und falls ja, wie diese Umsetzung konkret aussieht.

Auch hier gibt es bereits Ansätze auf wissenschaftlicher Basis, aber der Mehrwert, der durch diese neuen Patterns entsteht, hält sich in Grenzen.

Ähnliches trifft auch für das Exception-Handling zu. Auch dafür gibt es nur wenige prägnante Design-Patterns. Das Problem liegt hier jedoch nicht an mangelnden Implementierungen von denen Patterns abgeleitet werden könnten. Vielmehr wird das EH vielfach als relativ simpel und trivial angesehen - oft werden Exception-Handler während der Implementierung ad-hoc entworfen und umgesetzt. In vielen Projekten, die relativ geringe Anforderungen an das EH-Konzept und seine Komplexität haben, reicht ein derartiger Ansatz aus. Doch auch dort, könnte ein strukturierter Ansatz, der das EH bereits von Anfang an ins Design mit aufnimmt, zu qualitativ höherwertiger Software führen. Auch im Bereich des EH gibt es Ansätze und Vorschläge für Design-Patterns, was fehlt ist die Auswahl und Zusammenfassung der aussagekräftigsten und prägnantesten Patterns (auf der Basis von wissenschaftlichen Betrachtungen). Eine ausführlichere Diskussion zur angesprochenen Problematik findet sich in den Artikeln [12], [11] und [9].

A Kapitel im Anhang

A.1 AOP-Refactoring Guidelines (Auszug)

Es folgt ein Auszug aus den Guidelines für ein mögliches AOP-Refactoring des Exception-Handlings, welche nach Abschluss der Recherche-Phase für die Verwendung im Projekt erstellt wurden. Diese Guidelines basieren auf den Artikeln [8] und [7]. Dort finden sich genauere Details zur allgemeinen Problematik aber auch ausführliche Begründungen für die beschriebenen Restriktionen und Workarounds.

A.1.1 PostSharp Implementierung

Zur Implementierung von EH-Aspekten bietet PostSharp zwei Aspektklassen an: *OnExceptionAspect* und *OnMethodBoundaryAspect*. Nach Möglichkeit ist letzterer zu verwenden bzw. beinhaltet der BoundaryAspect unter anderem die selbe Funktionalität wie der ExceptionAspect.

Das Anwenden des *OnMethodBoundaryAspect* auf eine Methode führt nach dem Kompilieren und „Weaven“ zu folgendem Ergebnis (in Pseudocode):

```
int MyMethod(int arg0)
{
  OnEntry();
  try
  {
    //Original method body

    OnSuccess();
    return returnValue;
  }
  catch(Exception e)
  {
    OnException();
  }
  finally
  {
    OnExit();
  }
}
```

List. A.1: Code Manipulation durch PostSharp

Die vier Methoden (OnEntry, OnSuccess, OnException und OnExit) lassen sich in der jeweiligen Ableitung von *OnMethodBoundaryAspect* überschreiben. Der originale Methoden-Körper wird nun von einem try-catch-finally Block umschlossen. Würde nur ein *OnExceptionAspect* verwendet werden, so stünde lediglich die OnException-Methode zur Verfügung - eine genaue Nachbildung eines try-catch-finally Blocks wäre damit nicht möglich.

A.1.2 Try-Block

Wenn auf den try-catch(-finally)-Block noch Statements folgen, so darf im try-Block höchstens ein Methodenaufruf stehen, der eine Exception erzeugen könnte. Wenn mehrere Aufrufe unvermeidbar sind, so sollten diese in eine gemeinsame Methode zusammengefasst werden.

```
void function()
{
  try
  {
     N();  //might throw E
     P();  //might throw E
  }
  catch(E e) {...}
  Q();
}
```

List. A.2: Try-Block Problematik

Würde man in diesem Fall einen Aspekt auf „*function*“ Anwenden, so würde um den gesamten Methodenkörper ein try-catch-Block gelegt. Beim Auftreten einer Exception in *N()* oder *P()* würde *Q()* nicht mehr ausgeführt, da sich *Q()* ebenfalls im selben try-Block befindet. Würde man nun sowohl auf *N()* als auch auf *P()* einen eigenen EH-Aspekt anwenden, so würde beim Auftreten einer Exception in *N()* die Methode *P()* trotzdem ausgeführt, da sie ja nicht im selben try-Block liegt. Der hier dargestellte Programmfluss lässt sich nur mit einem Refactoring aspektorientiert umsetzen.

```
void exceptional()
{
  N();  //might throw E
  P();  //might throw E
}

void function()
{
  try
  {
    exceptional();  //might throw E
  }
  catch(E e) {...}
  Q();
}
```

List. A.3: Lösung durch Auslagern in eigene Methode

Durch das Auslagern des gesamten try-Blocks in eine eigene Methode kann der gewünschte Programmfluss nachgebildet werden. In diesem Fall ist es möglich einen EH-Aspekt auf die Methode „*exceptional*“ anzuwenden. Beim Auftreten einer Exception in *N()* wird *P()* nicht mehr ausgeführt - bei einer Exception in *N()* oder *P()* wird *Q()* trotzdem aufgerufen.

A.1.3 Catch-Block

Der Zugriff auf lokale Variablen im catch-Block, sowohl lesend als auch schreibend, ist zu vermeiden. Dies gilt auch für Methodenaufrufe mit lokalen Variablen als Parameter bzw. das Schreiben eines Rückgabewertes auf eine lokale Variable. Oft lässt sich das Problem durch geeignetes Design umgehen.
Falls dies nicht möglich sein sollte, gibt es mehrere Workarounds:

- Eigene Exception-Klasse ableiten, die es erlaubt, die Variable mitzugeben
- Verwendung der Variable in eine eigene Methode auslagern und Variable als Parameter übergeben
- Lokale Variable in eine Feld (Attribut mit Getter) umwandeln
- Letzte Zuweisung der Variable mit einem Aspekt „abfangen“ (d.h. letzte Zuweisung muss durch den Rückgabewert einer Methode erfolgen)

Die ersten beiden Lösungen sind in den meisten Fällen ausreichend und umsetzbar. Im Gegensatz dazu sollten die letzten beiden Ansätze nur dann verwendet werden, wenn es ausreichende Gründe gibt, die gegen die beiden ersten Varianten sprechen.

Falls möglich und sinnvoll, sollten Statements im catch-Block in eine eigene Methode ausgelagert werden, sodass im Block lediglich der Methodenaufruf bestehen bleibt. Eventuell wird dadurch das Design etwas komplexer - andererseits werden so leichter Exception-Handler-Methoden gefunden, die sich an mehreren Stellen im Code verwenden lassen.

A.1.4 Programmfluss nach dem catch-Block

Auf den Catch-Block folgt in vielen Fällen ein finally-Block. Statements in diesem Block sollten möglichst nicht auf lokale Variablen zugreifen und wenn es sich sinnvoll umsetzen lässt, sollen diese Statements in eine eigene Methode ausgelagert werden.

Problematischer sind Anweisungen nach einem catch-Block, die nicht im zugehörigen finally-Block stehen. Wenn es sich einfach umsetzen lässt, so sind derartige Statements zu vermeiden. Ansonsten sollen diese Statements (sofern möglich und sinnvoll) in eine eigene Methode gekapselt werden, sodass nach dem catch-Block höchstens noch ein Methodenaufruf folgt. In diesem Fall, muss der gesamte try-Block in eine eigene Methode ausgelagert werden. Andernfalls wird eine aspektorientierte Umsetzung mit demselben Verhalten deutlich erschwert.

Literaturverzeichnis

[1] Fraiteur, G. (Hrsg.): *Getting Started With PostSharp*. http://www.postsharp.org/about/getting-started, Abruf: 24.09.09

[2] Apache Software Foundation (Hrsg.): *log4net Features*. http://logging.apache.org/log4net/release/features.html, Abruf: 31.08.2009

[3] patterns & practices Developer Center (Hrsg.): *Microsoft Enterprise Library*. http://msdn.microsoft.com/en-us/library/cc467894.aspx, Abruf: 31.08.2009

[4] Fraiteur, G. (Hrsg.): *PostSharp Documentation*. http://doc.postsharp.org/1.0/, Abruf: 31.08.2009

[5] Pollack, M. et. al. (Hrsg.): *The Spring.NET Framework Reference Documentation*. http://www.springframework.net/docs/1.3.0-RC1/reference/html/index.html, Abruf: 31.08.2009

[6] Fraiteur, G. (Hrsg.): *Visualization of Aspects - PostSharp Forum*. http://www.postsharp.org/forum/integration/visualization-aspects-t749.html, Abruf: 31.08.2009

[7] Filho, F.C. ; Cacho, N. ; Maranhao, R. ; Figueiredo, E. ; Garcia, A. ; Rubira, C. M. F.: Exceptions and Aspects: The Devil is in the Details. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), S. 152 – 162

[8] Filho, F.C. ; Garcia, A. ; Rubira, C. M. F.: Extracting Error Handling to Aspects: A Cookbook. In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* (2007), S. 134–143

[9] Guerra, P. A. C. ; Filho, F. C. ; Pagano, V. A. ; Rubira, C. M. F.: Structuring Exception Handling for Dependable Component-Based Software Systems. In: *Proceedings of the 30th EUROMICRO Conference* (2004), S. 575 – 582

[10] Lippert, M. ; Lopes, C. V.: A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In: *Proceedings of the 22nd international conference on Software engineering* (2000), S. 418 – 427

[11] Viega, J. ; Vuas, J.: Can aspect-oriented programming lead to more reliable software? In: *Software, IEEE* 17 (2000), S. 19–21

[12] Wirfs-Brock, R. J.: Toward Exception-Handling Best Practices and Patterns. In: *Software, IEEE* 23 (2006), S. 11–13