ASPEKTORIENTIERTE PROGRAMMIERUNG MIT ASPECTC++ FÜR INGENIEURANWENDUNGEN

Sebastian Geller [1], Olaf Spinczyk [2]

[1] Institut für Computeranwendungen im Bauingenieurwesen, TU Braunschweig,
geller@cab.bau.tu-bs.de

[2] Verteilte Systeme und Betriebssysteme, Uni Erlangen-Nürnberg,
os@cs.fau.de

Kurzfassung: *Durch die immens gestiegene Komplexität der Ingenieuranwendungen ist selbst in der besten Softwarearchitekur das Problem des „Code-tangling“ nicht vermeidbar. Verschiedene Sichtweisen der Fachingenieure werden durch hochgradige Objekthierarchien abgebildet. Eine Durchmischung verschiedener Belange lässt sich somit nicht verhindern. Im folgenden Beitrag wird die aspekt-orientierte Programmierung und die C++ Erweiterung AspectC++ vorgestellt. Diese bieten dem Entwickler eine Alternative um die erstgenannten Probleme zu umgehen. Erste Ansätze umgesetzt im Strömungslöser Virtual Fluids zeigen den Nutzen dieser Programmiertechnik.*

1 Einleitung

Computeranwendungen im Bauwesen sind heutzutage hochgradig komplex. Entwickler kämpfen vor allem mit Konfigurierung, Wiederverwendbarkeit, Durchmischung verschiedener Belange, z.B. Code zur Entwicklungsunterstützung und der eigentlichen Berechnung sowie Codeduplikation. Solche Systeme erfordern spezielle Entwicklungsmethoden, die immer noch im Bereich der Informatik erforscht und verbessert werden.

Die aspektorientierte Programmierung (AOP) [3] ist eine von mehreren neuen Programmiertechniken, die zum Ziel haben, die Ausdruckskraft der Programme, verglichen mit herkömmlichem, beispielsweise rein objektorientiertem oder prozeduralem Code, zu

steigern. Beispiele für solche Ansätze sind die generative und generische Programmierung, domänenspezifische Sprachen, Reflektion und Metaprogrammierung[1].

Der folgende Abschnitt beschreibt zunächst die Grundidee der aspektorientierten Programmierung. Dies wird in Abschnitt 3 mit konkreten Beispielen, in der Sprache AspectC++ vertieft. Abschnitt 4 skizziert dann einen sinnvollen Anwendungsfall in einer Ingenieuranwendung. Der Abschluss des Artikels bildet eine Diskussion des Ansatzes und ein Ausblick auf geplante zukünftige Arbeiten.

2 Aspektorientierte Programmierung

Aspektorientierte Programmiersprachen oder -werkzeuge erreichen die zuvor erwähnte gesteigerte Ausdruckskraft durch die Eigenschaft, dass ein einzelner separater Teil eines Programms Auswirkungen auf viele andere Teile des Programmsystems haben kann, und dass die Programmteile, auf die ein solcher separater Programmteil einwirkt, nicht speziell dafür präpariert werden müssen. Der Entwickler muss also keine Ankerpunkte vorsehen. Man kann dieses Verhalten auch als implizite Ausführung bezeichnen [2]. So ist es zum Beispiel möglich zu verlangen, dass eine festgelegte Aktion ausgelöst wird, immer bevor eine bestimmte Funktion ausgeführt wird, ohne dass dies dem Programmcode des Aufrufers oder der aufgerufenen Funktion anzusehen ist.

Durch diese Eigenschaft wird es möglich, die so genannten querschneidenen Belange (*Crosscutting Concerns)* in modularer Weise zu implementieren. Dies sind Belange in der Softwareentwicklung, die zu Programmcode führen, der nicht in das Modularisierungsschema des restlichen Programms passt, und so über weite Teile des Programms verstreut wird. Statt einen logisch zusammenhängenden Programmcode in Fragmente zu zerlegen und an verschiedenen Stellen eines Programms einzusetzen, wird die implizite Ausführung genutzt. Die Module, die dies leisten, werden *Aspekte* genannt und bestehen aus den auszuführenden Codefragmenten selbst und einer Beschreibung, von welchen Punkten aus die Ausführung stattfinden soll. Ein solcher Punkt wird als *Join Point* bezeichnet.

Aspekte können zum Beispiel benutzt werden, um quasi von außen Code zur Überprüfung von Vor- und Nachbedingungen von Funktionen in ein System einzubringen. Häufig sind solche Bedingungen für eine große Zahl von Funktionen identisch, so dass eine modulare, aspektorientierte Implementierung weniger Programmtext bedeutet. Gleichzeitig erhöht sich die Übersichtlichkeit der einzelnen Funktionen und am Ende der Entwicklungsphase kann der Aspekt sehr leicht entfernt werden. Aspekte können aber auch sinnvoll im produktiven Betrieb eingesetzt werden. Ein häufig genanntes Beispiel

[1] Eine gute Übersicht über all diese Ansätze liefern K. Czarnecki und U. Eisenecker in ihrem Buch [1].

sind hier Aspekte, die für die Synchronisation innerhalb einer Komponente sorgen, wenn diese in einer vielfädigen Umgebung eingesetzt werden.

Den technischen Vorgang, der für die implizite Ausführung des Aspektcodes an den *Join Points* sorgt, nennt man *Aspektweben*. Der *Aspektweber* kann zum Beispiel ein Codetransformationssystem sein, das Funktionsaufrufe generiert, ein Compiler, der entsprechende Aufrufe als Maschineninstruktionen einfügt oder ein Laufzeitsystem, das in einem Interpreter oder eine virtuelle Maschine eingebunden ist und dort Verbindungspunkte zur Laufzeit erkennt und mit Aspektcode verbindet.

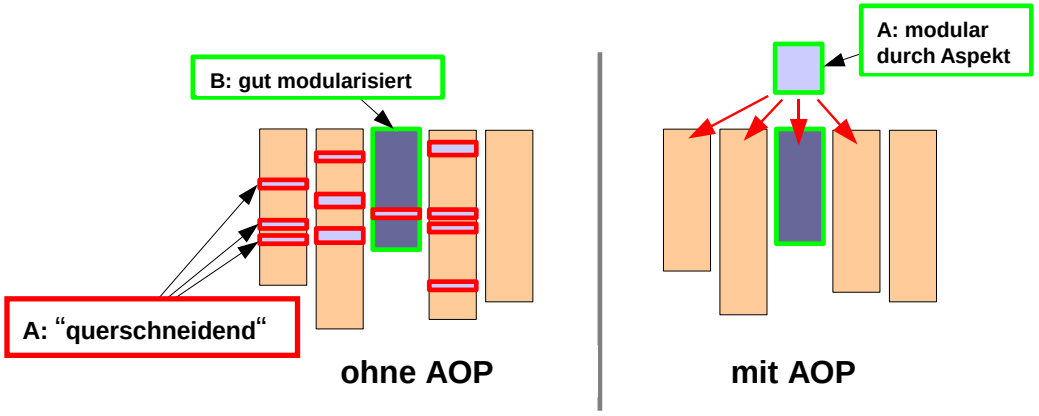

Abb. 1: Modulare Implementierung eines querschneidenen Belanges

Abbildung 1 zeigt links die Implementierung eines querschneidenen Belanges ohne und rechts mit Hilfe von Aspekten. Die Rechtecke symbolisieren die verschiedenen Quelltexteinheiten, die zu dem Programmsystem gehören. Der Code, der für die Implementierung des querschneidenen Belanges "A" benötigt wird, ist hervorgehoben. Wie zu erkennen ist, führt die Lösung mit Aspekten auf der rechten Seite zu einer verbesserten Modularität. Auf diese Weise wird durch die Anwendung von Aspekten die Lesbarkeit des Codes erhöht und damit seine Wartbarkeit verbessert. Zudem wird die zu implementierende Codemenge reduziert, da redundanter Code vermieden wird, und so die Produktivität der Entwickler gesteigert.

3 AOP mit AspectC++

AspectC++ ist eine aspektorientierte Erweiterung der Sprache C++, die spezielle Sprachelemente zur modularen Implementierung von querschneidenen Belangen im Sinne von AOP enthält [4][5]. Die wichtigsten dieser Sprachelemente sind *Pointcuts*, *Advice, Slices* und Aspekte. Sie werden im Folgenden kurz erläutert.

3.1 Pointcuts

Unter einem *Pointcut* wird in AspectC++ eine Menge von *Join Points* verstanden. *Pointcuts* werden benötigt, um zu beschreiben, wo ein Aspekt auf andere Module einwirken soll. Diese Beschreibung erfolgt deklarativ mit Hilfe einer *Pointcut*-Beschreibungssprache, zum Beispiel:

pointcut p1() = "A" || "B" || "C"; // die Menge der Klassen "A", "B" und "C"
pointcut p2() = derived("D" || "E"); // "D" und "E" sowie alle davon abgeleiteten Klassen
pointcut p3() = "F%"; // alle Klassen, deren Name mit "F" beginnt.

Wie man sieht, beschreiben alle drei Pointcuts Mengen von Klassen. Sie werden in AspectC++ als Ziel für die Einfügung der weiter unten erläuterten *Slices* benötigt. Neben *Pointcuts*, die Elemente der statischen Programmstruktur (wie Klassen) beschreiben, gibt es in AspectC++ auch *Pointcuts*, die Ereignisse im dynamischen Programmablauf beschreiben, zum Beispiel:

pointcut p4() = execution("void f(int)"); // die Ausführung der Funktion "f"
pointcut p5() = call("% A::%(...)"); // Aufruf einer beliebigen Member-Funktion von "A"
pointcut p6() = construction(p1()); // Erzeugung einer Instanz von "A", "B" oder "C"

3.2 Advice

Beide beschriebenen Arten von *Pointcuts* (Mengen von Elementen der statischen Programmstruktur und Mengen von Ereignissen im dynamischen Programmablauf) können das Ziel einer Einflussnahme durch Aspekte werden. Dies erfolgt in Form des sogenannten *Advice*. Hier wird zunächst gezeigt, wie mit Hilfe einer *Advice*-Definition auf ein bestimmtes Laufzeitereignis reagiert werden kann:

```
advice p5() : before ()
{
  cout << "gleich wird " << tjp->signature() << " aufgerufen!" << endl;
}
```

In diesem Beispiel wird dafür gesorgt, dass vor ("before") jedem Aufruf einer Member-Funktion der Klasse "A" (das war die Bedeutung von "p5") der in geschweiften Klammern stehende Code ausgeführt wird, der wiederum eine Ausgabe macht. Neben dem "before" *Advice* bietet die Sprache auch "after" und "around" Advice, um Code *nach* oder *anstatt* eines Laufzeitereignisses auszuführen.

Bemerkenswert an dem *Advice*-Code ist noch der Aufruf "tjp->signature()". Er liefert den Namen der *jeweiligen* Zielfunktion des abgefangenen Funktionsaufrufs, d.h. eine Kontextinformation über den *Join Point*, von dem aus der *Advice* implizit aufgerufen wurde.

Für den Zugriff auf diese Kontextinformation benutzt der Programmierer immer die implizit in jedem Advice-Code definierte Variable "tjp"[2]. Mit diesem Mechanismus kann man nicht nur Parameter und Rückgabewerte von aufgerufenen oder ausgeführten Funktionen, sondern sogar statische Typinformationen ermitteln, mit deren Hilfe man in AspectC++ auch hochgradig generischen *Advice* realisieren kann [6].

3.3 Slices

Für das Einwirken auf Elemente der statischen Programmstruktur werden in AspectC++ die sogenannten *Slices* verwendet. Eine Slice ist ein Fragment eines solchen Strukturelements. Derzeit unterstützt AspectC++ nur "class" und "struct" Fragmente, "enum" Fragmente werden in naher Zukunft folgen. Beispielhaft zeigt der folgende Programmausschnitt eine Klassen-*Slice* mit einem Attribut und zwei Zugriffsfunktionen:

```
#include <string>
slice class Named {
  string name_; // eine Name für das Objekt als Zeichenkette
public:
  string get_name () const { return name_; }
  void set_name (const string &n) { name_ = n; }
};
```

Dieses Klassenfragment hat für sich allein im Programm keinerlei Wirkung und es ist auch nicht erlaubt, Instanzen davon zu erzeugen. Erst im Zusammenhang mit einem *Advice* für Klassen entfaltet die *Slice* ihre Wirkung:

```
advice "A"||"B" : slice Named;
```

In diesem Beispiel wird die Slice "Named" in die Klassen "A" und "B" eingefügt. Das Ergebnis ist, dass beide Klassen nun ein Attribute "name_" und entsprechende Zugriffunktionen bekommen.

3.4 Aspekte

Aspekte in AspectC++ bilden ganz im Sinne von AOP eine modulare Einheit zur Kapselung der Implementierung eines querschneidenen Belanges. Rein syntaktisch entspricht eine Aspektdefinition einer C++ Klassendefinition, nur dass statt des "class" Schlüsselwortes "aspect" zu verwenden ist. Genau wie eine Klasse kann ein Aspekt auch herkömmliche Member-Variablen und Member-Funktionen haben. *Advice* ist dagegen nur in Aspekten, aber nicht in Klassen erlaubt. Der *Advice*-Code verhält sich abgesehen

[2] "tjp" steht für *this join point*.

von seinem impliziten Aufruf wie eine Member-Funktion des Aspekts und kann so z.B. direkt auf seinen Member-Variablen zugreifen. Dies erlaubt den Transport von Zustandsinformationen zwischen *Advice*-Ausführungen unabhängig vom eigentlich Programmfluss.

Neben der Kapselung dienen Aspekte auch als Einheit der Wiederverwendung. Dazu dient das Konzept der Vererbung von Aspekten, auf welches hier aus Platzgründen nicht näher eingegangen wird.

3.5 Implementierung

Derzeit existiert eine Implementierung von AspectC++ in Form eines Codetransformationssystems, das AspectC++ Code in C++ Code umsetzt. Es handelt sich um ein Open Source Projekt[3]. Messungen im Bereich Betriebssysteme und eingebettete Systeme haben bereits gezeigt, dass mit AspectC++ das Prinzip der Trennung der Belange auch ohne zusätzlichen Ressourcenverbrauch angewendet werden kann, da der vom statischen Aspektweber eingefügte Code keine nennenswerten Nachteile gegenüber einer "Handimplementierung" hat.

Neben dem AspectC++ Übersetzer existieren Integrationen in die Visual Studio IDE und in Eclipse.

4 AOP in einer Ingenieuranwendung

Die Erfahrungen aus der Implementierung von Aspekten in Betriebssystemen motivierten zur Untersuchung deren Anwendbarkeit in dem mittlerweile sehr umfangreichen Softwarepaket „*Virtual Fluids“.* Virtual Fluids ist der Strömungslöser, der am Institut für Computeranwendungen im Bauingenieurwesen an der TU Braunschweig entwickelt wird. Auf die Softwarearchitektur soll in diesem Zusammenhang auf folgende Veröffentlichungen verwiesen werden: [7] [9].

Der Löser basiert auf der Lattice-Boltzmann-Methode [8]. Das Rechengebiet wird mit adaptiven, baumstrukturierten Gittern beschrieben. Momentan existieren acht verschiedene Unterpakete (2D: Mehrphasen, Flachwasser, Fluid-Struktur-Wechselwirkung, Sedimenttransport / 3D: Ein-, Mehrphasen, freie Oberflächen, Temperatur), die sich hinsichtlich der physikalischen Beschreibung und somit auch durch verschiedene Rechenkerne unterscheiden. Das Grundprinzip des numerischen Verfahrens für Einphasensimulation lässt sich im Wesentlichen durch zwei Methoden beschreiben: Kollision und Propagation [8]. Für die Mehrphasensimulation werden zusätzliche Terme

[3] Homepage des AspectC++ Projekts: www.aspectc.org

(u.a. Oberflächenspannung, Entmischung) berechnet und zusätzlich das Phasenfeld transportiert. Für die Fluid-Struktur-Interaktion kommen Funktionen zur Berechnung der Bewegung der Struktur und der daraus resultierenden Rechengitteraktualisierung hinzu. Dies soll andeuten, dass die Numerik und somit auch die Systemarchitektur der einzelnen Rechenkerne prinzipiell ähnlich ist, was redundanten Code zur Folge hat. Dies ist genau die Art der Software, von der in der Einleitung die Rede war.

Abbildung 2 zeigt examplarisch die Zeitmessung zur Geschwindigkeitsoptimierung der einzelnen Funktionen zweier numerischer Rechenkerne. Bedingte Übersetzung und Code, der mit der eigentlichen Berechnung nichts zu tun hat, macht das Verstehen der Abläufe schwierig. Das ist ein typisches Problem, das AOP zu lösen vermag. Es wird als „Code-tangling“ Phänomen bezeichnet.

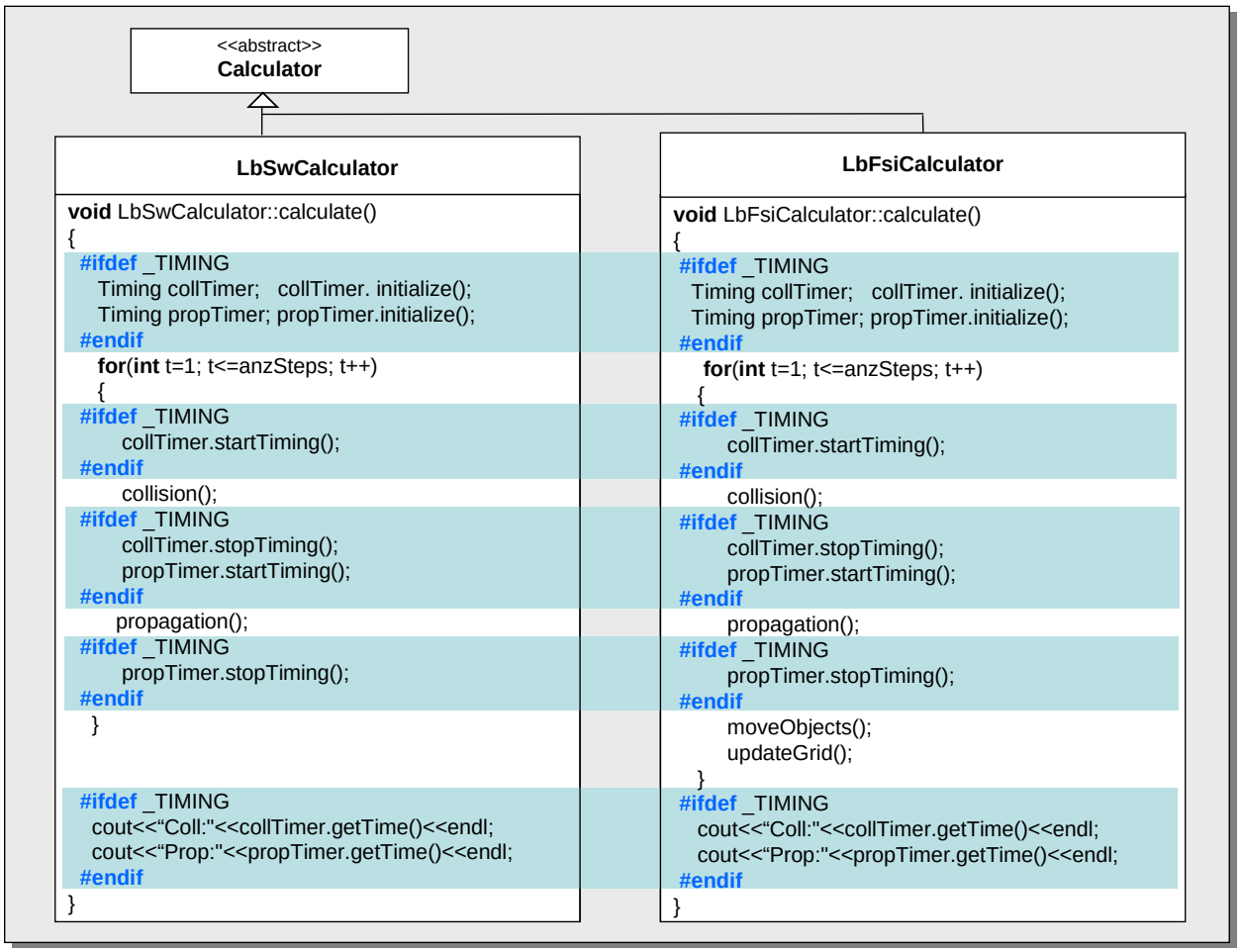

Abb. 2: Zeitmessung im nicht AOP-Code

Abbildung 3 zeigt dieselben Funktionen der Berechnungsklassen unter Verwendung eines Aspekts. Hier bleibt lediglich der wesentliche Kernbestandteil bestehen. Der zusätzliche Code für den Aspekt *TimeMeasurement* ist auf der linken Seite der Abbildung dargestellt und mit Kenntnis der wenigen Grundelemente von AspectC++ einfach nach-

vollziehbar. Wie man sehen kann, befindet sich bei dieser Implementierungsvariante kein Code zur Zeitmessung mehr in den Klassen *LbSwCalculator* und *LbFsiCalculator*. Dieser ist nun einzig im Aspekt *TimeMeasurement* zu finden, was die Gesamtlänge des Codes reduziert, die Duplikation vermeidet und letztendlich auch die Absicht beim Entwurf des Systems besser zum Ausdruck bringt. Darüber hinaus wurde die Zahl der #ifdef Direktiven im Code drastisch von 10 auf 1 verringert. All dies fördert nicht nur die Lesbarkeit sondern auch die Austauschbarkeit der Module. Der Code den ein Mitarbeiter in seinem Forschungszweig (assoziert mit einem numerischen Kern) erstellt hat, wird an den entsprechenden Stellen eingewebt und erspart insbesondere Codeumbauzeit. Oftmals hat man den Effekt, dass neue Module in einem Forschungscode fehlerfrei arbeiten, in anderen Projekten jedoch nicht. Der dann notwendige Um- und Rückbau des Codes ist normalerweise sehr zeitintensiv. Das Einweben neuer Module spart hier viel Entwicklungszeit.

TimeMeasurement

```
#ifdef _TIMING
aspect TimeMeasurement
{
  Timing collTimer;
  Timing propTimer;
  pointcut pc1() = execution("% %::collision(...)");
  advice pc1() : around()
  {
    collTimer.startTiming();
    tjp->proceed();
    collTimer.stopTiming();
  }
  pointcut pc2() = execution("% %::propagation(...)");
  advice pc2() : around()
  {
    propTimer.startTiming();
    tjp->proceed();
    propTimer.endTiming();
  }
  advice execution("% %::calculate (...)") : before()
  {
    collTimer. initialize();
    propTimer.initialize();
  }
  advice execution("% %::calculate (...)") : after()
  {
    cout<<“Coll:"<<collTimer.getTime()<<endl;
    cout<<“Prop:"<<propTimer.getTime()<<endl;
  }
};
#endif
```

LbSwCalculator

```
void LbSwCalculator::calculate()
{
  for(int t=1; t<=anzSteps; t++)
  {
     collision();
     propagation();
  }
}
```

LbFsiCalculator

```
void LbFsiCalculator::calculate()
{
   for(int t=1; t<=anzSteps; t++)
   {
      collision();
      propagation();
      moveObjects();
      updateGrid();
   }
}
```

Abb. 3: Zeitmessung mit AOP-Code

5 Zusammenfassung

Analog zur Zeitmessung hat vor allem auch das Tracing die bisherige Entwicklung der Anwendung beschleunigt. Die verschiedenen Parallelisierungs- und Visualisierungsstrategien, die in der momentanen Entwicklung getestet werden, bieten das Potential für eine übersichtliche Implementierung mit AOP.

Wie am Beispiel des Strömungslösers angedeutet, finden sich in vielen Ingenieuranwendungen querschneidene Belange, die nach Extrahierung zu einer Verbesserung der Übersichtlichkeit führen und andererseits auch eng an der wirklichkeitsnahen Beschreibung sind. Verschiedene Sichtweisen werden momentan in den Ingenieurapplikationen über hochgradige Objekthierarchien abgebildet. Die Implementierung via *Slices* (Abschnitt 3.3) bietet ebenfalls eine Möglichkeit diese zu entzerren.

Durch die bereits bestehenden Plug-Ins für diverse Entwicklungsumgebungen gelingt der Einstieg in die aspektorientierte Entwicklung relativ schnell und einfach. Beispiel-Code wie beim Visual Studio AspectC++ AddIn [10], fördern das Verständnis der genutzten Syntax.

Die Autoren sind überzeugt, dass bei konsistenter Programmierung mit AOP das Codeverständnis gerade für auszubildene Ingenieure vereinfacht wird.

Literatur

[1] K. Czarnecki und U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Publishing, 2000.

[2] T. Elrad, R. E. Filman und A. Bader. *Aspect-oriented programming*. Communications of the ACM, Seiten 29–32, Okt. 2001.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier und J. Irwin. *Aspect-oriented programming*. In M. Aksit and S. Matsuoka, editors, 11th Eur. Conf. on OOP (ECOOP ’97), volume 1241 of LNCS, Seiten 220–242. Springer, Juni 1997.

[4] O. Spinczyk, A. Gal und W. Schröder-Preikschat. *AspectC++: An aspect-oriented extension to C++*. In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific ’02), Seiten 53–60, Sydney, Australien, Feb. 2002.

[5] O. Spinczyk, D. Lohmann und M. Urban. *The design and implementation of AspectC++*. In Journal on Knowledge-Based Systems, Special Issue on Creative Software Design. Elsevier, 2006. (to appear).

[6] D. Lohmann, G. Blaschke und O. Spinczyk. *Generic advice: On the combination of AOP with generative programming in AspectC++*. In G. Karsai and E. Visser, editors, Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE ’04) , volume 3286 of Lecture Notes in Computer Science, Seiten 55–74. Springer-Verlag, Okt. 2004.

[7] S. Geller, S. Freudiger, J. Zimmermann, J. Tölke und M. Krafczyk. A Computional Steering Environment for adaptive Lattice Boltzmann Multi-Physics flows, (in Vorbereitung), ICMMES-Konferenz, Hampton, 2006.

[8] M. Krafczyk: Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung, Habilitationsschrift, Lehrstuhl für Bauinformatik, TU-München, 2001.

[9] M. Brüggemann: Informationsmodellierung im Bauwesen, Habilitation, BTU Cottbus, 2006

[10] AspectC++ AddIn für Visual Studio: http://www.pure-systems.com/