Technische Universität Braunschweig
Institut für Computeranwendungen im Bauwesen
Prof. Dr.-Ing. habil. M. Krafczyk

Entwicklung eines auf der Lattice-Boltzmann-Methode basierenden interaktiven Strömungssimulators in C# mit verschiedenen Berechnungskernen

von
Jan Linxweiler

Braunschweig April 2004

Danksagung

Die vorliegende Arbeit entstand im Rahmen einer Studienarbeit am Institut für Computeranwendungen im Bauingenieurwesen der Technischen Universität Braunschweig. In diesem Zusammenhang gilt mein Dank dem Leiter des Instituts Prof. Dr. habil. Manfred Krafczyk, für seine Einführung in die Theorie der numerischen Strömungssimulation. Mein besonderer Dank gilt dem Betreuer dieser Arbeit Dipl.-Ing. Sören Freudiger, der mich während der gesamten Bearbeitungszeit unterstützt hat und stets zu konstruktiver Diskussion bereit war.

Weiterer Dank gilt Dipl.-Ing. Sebastian Geller, der mit vielen Anregungen zur Entwicklung des Simulators beigetragen hat.

Außerdem möchte ich Arne Mittelstaedt danken, der sich die Zeit nahm, die Funktionen der Anwendung in allen Details zu testen.

1 Einleitung

Das Thema dieser Arbeit ist die computerbasierte Simulation von Strömungs- und Transportprozessen. Zu diesem Zweck werden mathematische Modelle herangezogen, die diese Prozesse beschreiben. Die hier verwendete *Lattice-Boltzmann-Methode* ermöglicht die Simulation des makroskopischen Strömungsverhaltens mittels numerischer Algorithmen.

In der Forschung werden zur numerischen Simulation i. A. sequenzielle Programmcodes verwendet, die z.B. unter Verwendung der Programmiersprache *Fortran* oder *C* implementiert werden. Diese Programmiersprachen gewährleisten eine hohe Ausführungsgeschwindigkeit, die für die sehr rechenintensive numerische Simulation von besonderem Interesse ist. Zudem bieten sequenzielle *Fortran*- bzw. *C*-Programme die Möglichkeit, diese zu parallelisieren und somit auf Multiprozessorsystemen auszuführen.
Seit einigen Jahren hat mit *C++, Java* und *C#* die objektorientierte Programmierung in die Informatik Einzug erhalten. *C++* stellt eine Erweiterung der Programmiersprache *C* dar. Mit dieser wird das objektorientierte Programmieren unter Verwendung der aus *C* und *Fortran* bekannten Zeiger ermöglicht. Mit *C++* erstellte Anwendungen erzielen somit eine ähnlich hohe Ausführungsgeschwindigkeit wie sequenzieller Code auf Basis von *C* oder *Fortran*.
In den jüngsten Sprachen *Java* und *C#* wurde das Prinzip der Zeiger aufgegeben. Zudem wurde die Speicherverwaltung, die bislang Aufgabe des Programmierers war, automatisiert. Im Unterschied zu Anwendungen, die in *C++* oder in einer der sequenziellen Sprachen entwickelt wurde, werden *Java*- und *C#* - Anwendungen nicht mehr zur Entwurfszeit in Maschinensprache übersetzt. Dieser Prozess wird erst mit der Ausführung des Programms durchgeführt.

Die Programmierung unter Verwendung der modernen objektorientierten Sprachen bietet dem Entwickler einen großen Komfort im Vergleich zu früheren Programmierweisen. Allerdings führt dieser Zugewinn an Komfort teilweise zu Einbußen in der Ausführungsgeschwindigkeit.

Ziel dieser Arbeit ist es, eine Simulationsanwendung basierend auf modernsten Softwaretechnologien zu entwerfen. Es soll untersucht werden, in wie weit sich der Komfort der Anwendungsentwicklung moderner Programmiersprachen mit der Notwendigkeit hoher Ausführungsgeschwindigkeiten vereinbaren lässt.

Als Grundlage der Arbeit dient ein sequenzieller Programmcode auf Basis der *Lattice-Boltzmann-Methode*. Zur Implementierung der Anwendung wird die Programmiersprache *C#* in Verbindung mit dem *.NET Framework* verwendet. Im Mittelpunkt des Entwurfs steht die Implementierung verschiedener Berechnungskerne, die einen abschließenden Vergleich unterschiedlicher Technologien bezüglich der Ausführungsgeschwindigkeit erlauben.

2 Die Lattice-Boltzmann-Methode (LBM)

2.1 Einführung in die Strömungssimulation

In der computergestützten Strömungssimulation, im englischen auch *Computational Fluid Dynamics* (CFD) genannt, werden die makroskopischen Eigenschaften eines Fluides, wie z.B. die Dichte und Geschwindigkeit, üblicher Weise durch *Navier-Stokes* (NS) Gleichungen beschrieben. Durch die nichtlinearen advektiven Terme der Navier-Stokes-Gleichungen ist es jedoch nicht immer möglich diese Gleichungen mit hinreichender Genauigkeit numerisch zu lösen.

Anstatt also die die makroskopischen Gleichungen zu lösen, berechnet man die Lattice-Boltzmann-Gleichungen, um die Navier-Stokes-Gleichungen zu erfüllen. Da das makroskopische Verhalten des Fluides unabhängig ist von den detaillierten mikroskopischen Eigenschaften, hat man so die Möglichkeit ein vereinfachtes mikroskopisches kinetisches Model zur Simulation der Teilchenbewegung zu verwenden.

Die Lattice-Botzmann-Methode (LBM) ist ein sehr effektives Verfahren für transiente Strömungsprobleme und zudem numerisch einfacher zu lösen als die Navier-Stokes-Gleichungen.

[LWK]

2.2 Das Lattice-Boltzmann-BGK-Modell

Der grundlegende Gedanke der Lattice-Boltzmann-Methode besteht darin, die makroskopischen Eigenschaften eines Fluides durch einen mikroskopischen kinetischen Ansatz zu lösen. Diese beschreibt die Bewegung und Interaktion auf Teilchenebene.

Die Basis bildet die Boltzmann-Gleichung (BG) Gl. (2.1) die bereits 1854 von Maxwell und Boltzmann aufgestellt wurde und die kinetische Theorie eines idallen Gases beschreibt. Die Gleichung (2.1) gibt die Entwicklung der Teilchenverteilung in Abhängigkeit der Zeit wieder. Die linke Seite der Gleichung beschreibt die Advektion, während die rechte Seite den Kollisionsoperator darstellt.

(2.1) $$\partial_t f + \xi * \nabla f = \Omega$$

$f(\vec{x},\vec{\xi},t)$ ist die Wahrscheinlichkeitsdichte der Teilchen. Sie beschreibt die Wahrscheinlichkeit, eine bestimmte Zahl an Teilchen zu einem Zeitpunkt t an einem Ort $\vec{x}$

mit einer Geschwindigkeit $\vec{\xi}$ vorzufinden. Außerdem ändert sich die Teilchendichte an einem bestimmten Ort, da die Teilchen miteinander kollidieren. Letzteres wird mathematisch durch den Kollisionsoperator Ω beschrieben.
Um ein Modell zu erhalten, das die oben genannten Voraussetzungen erfüllt und dennoch einfach zu lösen ist, wird ein reguläres Gitter gewählt und die Zahl der möglichen Geschwindigkeiten auf eine diskrete Anzahl reduziert. Dies geschieht unter der Annahme, dass jedes Teilchen sich auf einer begrenzten Anzahl Richtungen bewegen kann.
Einen vereinfachten Kollisionsoperator erhält man durch die von Bhatnager, Gross und Krook [BGK] vorgeschlagene so genannte *single relaxation time approximation.* Diese besagt, dass die Funktion der Verteilungen f_i als Summe einer Gleichgewichtsfunktion f_i^{eq} und einem Nichtgleichgewichtsanteil f_i^{neq} betrachtet wird, wobei das Streben nach dem Gleichgewicht von der Relaxationszeit τ bestimmt wird. Der Index i nimmt Bezug auf die Richtung des Geschwindigkeitsvektors. Hieraus ergibt sich der Kollisionsoperator Ω_i wie folgt:

(2.2) $$\Omega_i = -\frac{1}{\tau}(f_i - f_i^{eq})$$

Das Inverse von τ wird als Relaxationsfrequenz ω bezeichnet.
Die Gleichgewichtsverteilungen werden nach Maxwell bestimmt. Eine Berechnung erfolgt durch folgende Gleichung:

(2.3) $$f_i^{eq}(\rho,\vec{u}) = w_i\rho(1+\frac{3}{2}\vec{e}_i\vec{u}+\frac{9}{2}(\vec{e}_i\vec{u})^2-\frac{3}{2}\vec{u}^2)$$

Wichtungsfaktoren: $w_0 = \frac{4}{9}$, $w_{1,2,3,4} = \frac{1}{9}$, $w_{5,6,7,8} = \frac{1}{36}$

Der numerische Ansatz zur Lösung der Lattice-Boltzmann-Gleichung erfolgt über eine Diskretisierung mittels Finiter Differenzen in Raum und Zeit. Wenn die Zeit um einen Zeitschritt erhöht wird, wandert jedes Teilchen vom Ort $\vec{x}$ zum Ort $\vec{x}+\vec{e}_i * \Delta t$. Der Effekt der Kollison wird durch die o.g. *single relaxation time approximation* erzielt, die sehr einfach zu berechnen ist. Die Gitterweite Δx und die Größe des Zeitschritts Δt erfüllen die Bedingung $\frac{\Delta x}{\Delta t} = c$, so dass sich die folgende Lattice-Boltzmann-BGK-Gleichung ergibt:

(2.4) $$f_i(\vec{x}+\vec{e}_i*\Delta t, t+\Delta t) - f_i(\vec{x},t) = \frac{\Delta t}{\tau}[f_i(\vec{x},t) - f_i^{eq})]$$

Die linke Seite der Gleichung beschreibt den physikalischen Strömungsprozess wobei die rechte Seite die Annäherung an die Gleichgewichtsverteilung modelliert. Dies bedeutet, dass das BGK Schema aus zwei numerischen Schritten besteht: dem Propagationsschritt und dem Kollisionsschritt.

(2.5) Kollision: $$f_i^{new}(\vec{x},t) - f_i(\vec{x},t) = \Omega_i$$

(2.6) Propagation: $$f_i(\vec{x}+\vec{e}_i*\Delta t, t+\Delta t) = f_i^{new}(\vec{x},t)$$

Weiterhin kann gezeigt werden, dass das Lattice-Boltzmann-Modell die Navier-Stokes Gleichungen mit einer Viskosität $\nu = \frac{2*\tau-1}{6}$ erfüllt. Die Lattice-Boltzmann-Methode ist formal ein Verfahren 2. Ordnung um inkompressible Strömungen zu berechnen. Der Ansatz erfüllt die physikalischen Gesetze der Masse-, Impuls- und Momentenerhaltung. Die makroskopischen pysikalischen Größen Dichte ρ und Geschwindigkeitsdichte $\rho\vec{u}$ der Strömung können über die Integration der Teilchenverteilungen nach e berechnet werden.

(2.7) $$\rho = \sum_i f_i \ , \ \rho\vec{u} = \sum f_i \vec{e}_i$$

Der Vorteil der LBM, der für die Anwendung in Computer gestützter Simulation spricht, ist das einfache gehaltene System an Gleichungen, das dementsprechend leicht zu implementieren ist. Der Propagationsschritt beinhaltet sehr wenig Rechenaufwand und der Kollisionsschritt findet ausschließlich lokal statt.
Während in traditionellen Simulationsverfahren der konvektive Term aus nichtlinearen Gleichungen besteht und für die Viskosität der Laplace-Operator Verwendung findet, verläuft die Konvektion in diesem Modell linear.

[BRFU]

2.3 Das D2Q9-Modell

Für die Implementierung der Lattice-Boltzmann-BGK-Methode im FlowSim 2004 .NET wurde das so genannte D2Q9-Modell verwendet. In diesem Modell erfolgt die Diskretisierung auf einem zweidimensionalen kartesischen Gitter mit einer konstanten Gitterweite $\Delta x = 1$. Jeder Knoten des Gitters ist mit seinem Nachbarn über acht Gittervektoren $\vec{e}_i$ verbunden.

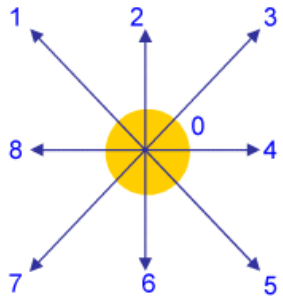

Abbildung 2.1 Knoten des D2Q9-Modells

An jedem Knoten befinden sich die Teilchenverteilungen f_i, die sich mit einer Geschwindigkeit $\vec{e}_i$ innerhalb eines Zeitschrittes $\Delta t = 1$ auf den Verbindungen zu den jeweiligen Nachbarn bewegen. Man unterscheidet drei Geschwindigkeiten: Ruhende Teilchen mit der Geschwindigkeit $\vec{e}_0 = 0$, Teilchen, die sich mit der Geschwindigkeit $\vec{e}_{1,2,3,4} = c$ entlang der horizontalen bzw. vertikalen Achse bewegen und Teilchen, die sich entlang der Diagonalen mit der Geschwindigkeit $\vec{e}_{5,6,7,8} = c * \sqrt{2}$ bewegen. Die Geschwindigkeit c ist frei wählbar und wird für die Implementierung mit 1 angesetzt. Diese Geschwindigkeit ist maßgebend für die Größe der Schallgeschwindigkeit, die sich wie folgt berechnet:

(2.8)
$$c_s = \frac{c}{\sqrt{3}}$$

Durch die Wahl der Geschwindigkeit $c = 1$ ergibt sich eine Schallgeschwindigkeit von $c_s = \frac{1}{\sqrt{3}}$. Wie bereits im Abschnitt 2.2 erwähnt wird die Belegung des Knotens durch die Verteilungsfunktionen f_i dargestellt. Der Index i = 0,…,8 ordnet dem D2Q9-Modell entsprechend sowohl die Verteilungsfunktionen f_i als auch die Gittervektoren $\vec{e}_i$ zu.

[TKS]

2.4 Die Implementierung der LBGK Methode

2.4.1 Das Speichermodell des FlowSim 2004 .NET

Zu Beginn der Berechnung muss ein für den Computer lesbares Modell der Geometrie des Strömungsgebietes erstellt werden. Dies erfolgt während des Programmablaufs in der Diskretisierung. Die vorhandene Geometrie wird in dieser Routine auf das zu Grunde liegende Gitter abgebildet. Dabei werden den Knoten des Gitters unterschiedliche Eigenschaften zugewiesen. Die Datenhaltung eines regulären Gitters geschieht im Allgemeinen in so genannten Arrays. Diese Arrays stellen ein Abbild von Matrizen im Speicher des Rechners dar. Arrays bieten die Möglichkeit große Datenmengen einfach und schnell zu verwalten. Auf die verschiedenen Arten der Arrays wird im weiteren Verlauf der Arbeit noch genauer eingegangen. Die Geometriematrix beispielsweise ist eine n-x-m-Matrix. Eine grafische Darstellung einer solchen Matrix ist exemplarisch in Abbildung 2.2 gegeben.

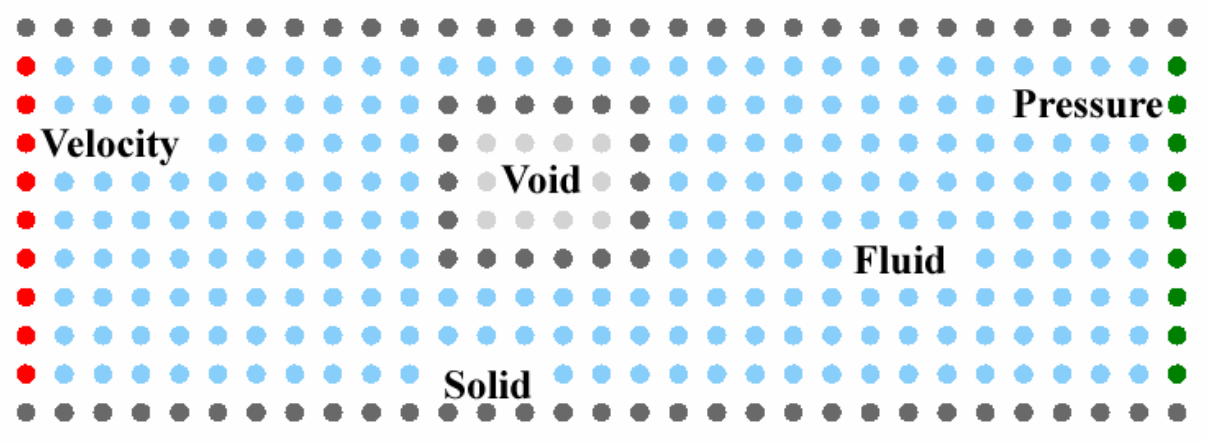

Abbildung 2.2 Matrix der Geometrie eines Strömungsfeldes

Ein Eintrag in der Geometriematrix kann im FlowSim 2004 .NET fünf verschiedene Werte annehmen. Das Strömungsgebiet wird von den „Fluid“-Knoten beschrieben. Dieses wird begrenzt von „Solid“-, „Velocity“-, oder „Pressure“-Knoten. An „Solid“-Knoten wird in der Berechnung die „Bounce-Back“-Randbedingung angewandt, wohingegen ein „Velocity“-Knoten stellvertretend für Geschwindigkeitsrandbedingung steht. Gleiches gilt für den „Pressure“-Typ, der eine Druckrandbedingung darstellt. Ein Knoten vom Typ „Void“ ist für die Berechnung ohne Bedeutung.

Ebenso wie für die Geometrie, wird innerhalb der Berechnung ein Array für die Teilchenverteilungen f_i, die Geschwindigkeiten u_x, u_y und die Dichte ρ vorgehalten.

2.4.2 Prizipieller Algorithmus der LBGK Methode

Die Implementierung des Berechnungsalgorithmus gleicht in Pseudocode etwa folgendem Schema:

```
Init Matrices geo, ux, uy, rho;
Init Distributions f;
Calc tau;
Loop                    //Berechnung
{
      collide;          //Kollision
      propagate;        //Propagation

      if (TimeStepsCalculated)     //Wenn X Berechnungsschritte
            OnTimeStepsCalculated; //Benachrichtigung der GUI
}
```

Die Berechnung beginnt damit, dass die im oberen Abschnitt beschriebenen Arrays initialisiert werden. Auf die Belegung der Geomatrix ist ebenfalls bereits eingegangen worden. Die aus dem oberen Abschnitt bekannten Knotentypen der Geomatrix bestimmen im Folgenden die Initialisierung der Matrizen für die Geschwindigkeit und die Dichte. Für Knoten des Typs „Solid“ haben sowohl die Einträge in der Matrix für die Dichte als auch die in den Geschwindigkeitsmatrizen den Wert 0. Die Einträge der Knoten des Typs „Velocity“ haben der zugrunde liegenden Geometrie entsprechende Werte in der Geschwindigkeitsmatrix zu Folge. Genau entgegengesetzt werden die Werte des Knotentyps „Pressure“ in den Matrizen eingetragen. Die Initialwerte der Verteilungsmatrix entsprechen den Gleichgewichtsverteilungen nach Maxwell.
Ein Zusammenhang der Relaxationszeit τ und der kinetischen Viskosität ν wurde bereits im Abschnitt 2.2 angesprochen. Der Parameter τ wird nach Gleichung (2.9) berechnet.

$$\nu = \frac{2\tau - 1}{6} \tag{2.9}$$

In der nun folgenden Berechnungsschleife wird die eigentliche Berechnung durchgeführt. Diese erfolgt durch die aus Abschnitt 2.2 bekannte Kollision und Propagation, die innerhalb der Schleife wiederholt durchgeführt werden.

2.4.2.1 Die Kollision

Numerisch vollzieht sich die Kollision nach Gleichung (2.5). In der Implementierung besteht die Kollision im Wesentlichen aus zwei ineinander verschachtelten Schleifen innerhalb derer über alle Felder der Matrix iteriert wird. Wiederum in einer Schleife wird dann für jeden Knoten für jede der vorhandenen neun Verteilungen f eine neue Verteilung nach der o. g. Gleichung berechnet. Dabei muss bei der Berechnung der neuen Verteilungen der Typ des Knotens berücksichtigt werden.

Für einen Knoten des Typs „Fluid“ berechnen sich die neuen Teilchenverteilungen f_i vollständig aus den vorhandenen Verteilungen. Bei einem „Velocity“ bzw. „Pressure“-Knoten fließen an dieser Stelle die gegebenen makroskopischen Randbedingungen in die Berechnung ein.

Für einen festen Knoten des Typs „Solid“ ist die so genannte „*Bounce-Back“-Randbedingung* implementiert. Sinngemäß besagt diese, dass alle Verteilungen an diesem Knoten reflektiert werden.

2.4.2.2 Die Propagation

Die Propagation erfolgt entsprechend der Gleichung (2.6). In diesem Schritt finden keinerlei Berechnungen statt. Stattdessen werden innerhalb der Propagation die Teilchenverteilungen f_i in Richtung der diskreten Geschwindigkeitsvektoren $\vec{e}_i$ an die jeweiligen Nachbarknoten verschoben. Dieses geschieht wiederum in ineinander verschachtelten Schleifen. Aus algorithmischer Sicht findet hier lediglich das Umverteilen von Variablen bzw. von Referenzen innerhalb des Speichers statt. Der Propagationsschritt ist in Abbildung 2.3 Propagation dargestellt.

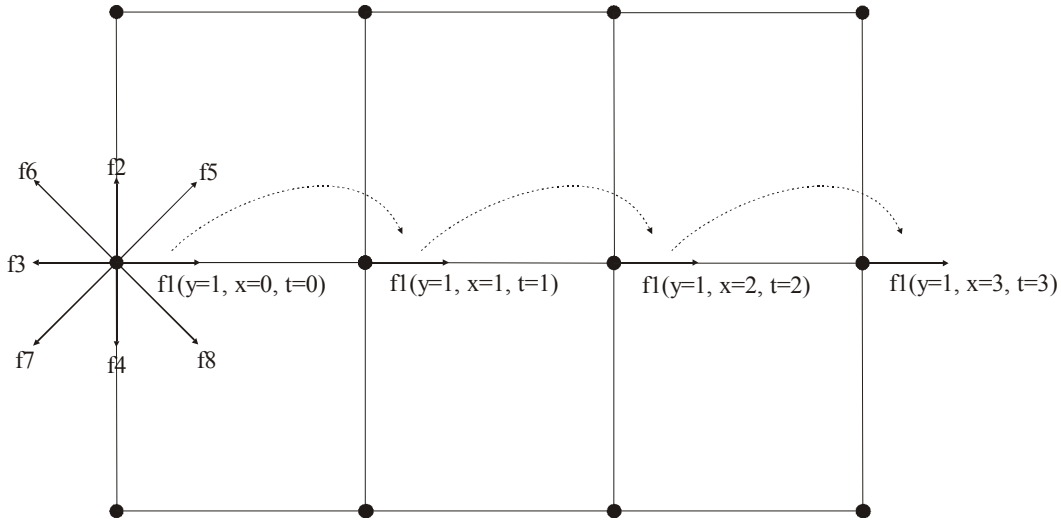

Abbildung 2.3 Propagation

3 Microsoft .NET

3.1 .NET – Was ist das?

Wenn man Microsoft Glauben schenken darf, dann vollzieht sich mit der Einführung des .NET Frameworks ein ähnlicher Evolutionsschritt wie zu damaliger Zeit mit der Einführung der grafischen Benutzeroberfläche mit Windows 3.1. Als das .NET Framework im Sommer 2000 vorgestellt wurde, hieß es noch *Next Generation Web Services* (*NGWS*). Noch im gleichen Jahr wurde es allerdings in den derzeitigen Namen umbenannt. Mit der Einführung des .NET Frameworks begann das, was allgemein als .NET Initiative bezeichnet wird. Scheinbar sollte jede neue Softwareversion aus dem Hause Microsoft das Kürzel .NET erhalten. Der Begriff .NET war jedermann geläufig, doch die wenigsten wussten ihn für sich zu definieren. .NET schien mehr als eine Marke, als als eine Technologie verstanden zu werden. Nicht zuletzt mit Einführung des neuen Windows 2003 Server, der bis dato immer als Windows 2003 .NET Server bezeichnet wurde, war die Verwirrung perfekt. Die Ergänzung des Namens um das Kürzel .NET schien allgemein zu einer Verunsicherung zu führen. Fortan konzentrierte man sich in der Informationspolitik mehr auf die Entwickler und Entscheidungsträger von Software und IT-Technologien. Für diese hält das .NET Framework zahlreiche Neuerungen bereit. Von den Neuerungen profitiert letzten Endes auch der Benutzer, doch müssen diese selbstverständlich zuerst einmal in aktuelle Softwareprodukte Einzug erhalten.

Es ist allerdings noch immer unklar, was .NET bedeutet. Der ursprüngliche Name *Next Generation Web Services* und auch das Kürzel .NET lassen auf einen direkten Zusammenhang mit dem Internet schließen. In der Tat spielen die Internetdienste oder auch *Web Services* genannt eine wesentliche Rolle in .NET. Der Name ist aber weitestgehend irreführend, denn das Framework bietet weit mehr als eben besagter Name vermuten lässt. Der Begriff .NET ist daher so irreführend, da Microsoft ihn im Zusammenhang mit verschiedensten Technologien verwendet. Die .NET Strategie besteht im Wesentlichen aus den oben genannten Web Services, den .NET Enterprise Servern und dem .NET Framework.

Obwohl für die Entwicklung des FlowSim 2004 .NET lediglich das .NET Framework von Interesse ist, wird auch kurz auf die übrigen Technologien eingegangen, um ein Gesamtbild zu erhalten.

[ElKo03],[SHB02], [MaFr02], [Kue03]

3.2 .NET Web Services

Das *World Wide Web* hat die Art und Weise, wie man auf Informationen zugreift, Produkte kauft und z.B. an Auktionen teilnimmt, in den letzten Jahren nachhaltig beeinflusst. Die maßgeblichen Anwendungen agieren mit dem Menschen direkt über eine grafische Benutzerschnittstelle (*Graphical User Interface – kurz GUI)*. Die nächste Entwicklungsstufe des Internets wird wahrscheinlich durch die Web Services geprägt sein. Ein Web Service stellt seine Dienste allerdings nicht in der üblichen Form zur Verfügung. Im Allgemeinen wird ein Web Service nicht direkt vom Benutzer angesprochen, sondern von Clientanwendungen. Dies bedeutet, dass eine Clientanwendung mit einem Web Service über das Internet nach einem vorgeschriebenen Schema kommuniziert. Das Aufrufen eines Web Services aus einer Clientanwendung heraus kann als ein Funktionsaufruf verstanden werden, der auf dem Host des Web Services ausgeführt wird. Das Ergebnis des Funktionsaufrufes wird dann von dem Web Service an die Clientanwendung zurückgeliefert. Diese Art von Funktionsaufrufen wird auch als *Remote Procedure Call (RPM)* bezeichnet. Ein Beispiel eines Web Services ist Microsoft MapPoint. Ruft man diesen Web Service mit einer Adresse oder mit Gauss-Krüger Koordinaten als Parameter auf, so liefert der Web Service als Ergebnis ein Bild mit einem entsprechenden Kartenausschnitt. Ganz ähnlich, wie es z.B. das bekannte Pendant Map24.de mit User Interface liefert. Es sei noch einmal auf den Unterschied hingewiesen: Map24.de liefert dem Benutzer den Kartenausschnitt direkt visuell auf den Bildschirm, während der Web Service das Bild in XML codierter Form als Antwort auf den Aufruf zurückliefert.

[Ca02]

3.3 .NET Enterprise Server

Bei den .NET Enterprise Servern handelt es sich um eine Familie von Software-Servern zu denen unter anderem der *Microsoft SQL Server 2000*, *Exchange Server 2000*, *BizTalk Server 2000* und der *Commerce Server 2000* zählen. Was diese Produkte als ein solches auszeichnet ist z.B die Möglichkeit ihre Funktionen aus .NET Anwendungen heraus zu nutzen. Dieses kann wiederum über die oben beschriebenen Web Services geschehen. Im .NET Framework können zudem vorhandene Technologien wie *COM* oder *COM+* oder eine neue Technologie namens *.NET Remoting* zur Integration verwendet werden.

[ElKo03], [Ca02]

3.4 .NET Framework

3.4.1 .NET Framework – Warum?

Bereits im ersten Absatz wurde auf die Bedeutung der Einführung des .NET Frameworks eingegangen. Dies soll hier noch einmal näher erläutert werden. Mit der Einführung des .NET Frameworks bricht Microsoft mehr oder weniger mit einer Vielzahl von vorhandenen Technologien. Zu nennen wären in diesem Zusammenhang u.a. *COM* und *DCOM* bzw. *COM+* und *ActiveX* sowie *ADO* und *ASP* aber auch *Visual Basic* und *Visual C++*. All diese Technologien haben sich über die Jahre erfolgreich etabliert und trotzdem hat Microsoft mit dem .NET Framework eine neue Technologie geschaffen, die die Entwicklung von Software für die Windows Plattform nachhaltig verändert.

Wenn man einige der oben genannten Technologien betrachtet, so stellt man fest, dass diese schon sehr lange existieren. Gerade der Bereich der Informationstechnologie ist jedoch sehr schnelllebig, so dass es stets erforderlich ist, die vorhandenen Sofwaretechniken zu erweitern und an neue Anforderungen anzupassen. Diese Erweiterung bedeutet meist jedoch auch, dass die Techniken dadurch oftmals komplexer und somit auch komplizierter werden. Aus der zunehmenden Komplexität resultierte sehr bald das Verlangen nach einer neuen Technologie, die die Anforderungen an moderne Software erfüllt, aber die Altlasten früherer Softwaretechnkiken hinter sich lässt.

Die Idee von .NET ist daher nicht erst vor kurzem entstanden, vielmehr ist man bereits seit fünf Jahren mit der Entwicklung beschäftigt. Eine lange Zeit, in der die Entwickler genügend Zeit hatten, von anderen Technologien zu lernen und letztlich eine solide Grundlage für die zukünftige Programmierung zu schaffen.

[ElKo03], [SHB02]

3.4.2 Die Vorteile des neuen Frameworks

Die Vorteile des .NET Frameworks, auf die im folgenden noch weiter eingegangen wird, sind folgende:

- einheitliche Klassenbibliothek
- automatische Speicherverwaltung
- Plattformunabhängigkeit
- Sprachunabhängigkeit
- Ausführungssicherheit
- Ende der „DLL-Hölle“
- Weitergabe der Anwendung

3.4.2.1 Die .NET Klassenbibliothek (FCL)

Die für Entwickler wichtigste Neuerung stellt die einheitliche Klassenbibliothek dar. Das .NET Framework ist durchgehend objektorientiert. Alles im Framework wird als Objekt betrachtet. Dazu zählen auch die nativen Datentypen der *Common Language Specification (CLS)*. Die .NET-Klassen bilden zusammen die .NET-Klassenhierarchie. Den Ausgangspunkt der Hierarchie bildet die Klasse *Object*. Jede Klasse des .NET Frameworks kann auf diese zurückgeführt werden und erbt daher deren Methoden. Dabei ist das oberste Gebot die *Typsicherheit*. Neben der statischen Typüberprüfung durch den Compiler erfolgt eine dynamische Typüberprüfung zur Laufzeit auf Basis der Laufzeittypinformationen.
Visual C++ Programmierern ist der Begriff der Klassenhierarchie von den *Microsoft Foundation Classes* (MFC) her bekannt. Allerdings wurde der objektorientierte Gedanke im Fall der MFC nicht vollständig zu Ende geführt, so dass die MFC eher ein Tool darstellen um mit der Win32-API zu programmieren, als diese zu abstrahieren.
In der .NET-Klassenhierarchie finden sich weit über 2400 Klassen. Diese sind in der Klassenbibliothek durch so genannte *Namespaces* nach Anwendungsgebieten gruppiert.
Langfristig beabsichtigt Microsoft, die *Win32-API* in der nächsten Version des Betriebssystems Windows vollständig durch *WinFX*, die nächste Generation der .NET Klassenbibliothek zu ersetzen.

[ElKo03], [SHB02], [Kue03]

3.4.2.2 Automatische Speicherverwaltung

Die Freigabe von nicht mehr benötigtem Speicher kann z.B. in C++ schnell zu einem Problem werden. Unter .NET braucht sich ein Entwickler darum nicht mehr zu kümmern, denn der im Hintergrund arbeitende Prozess des *Garbage Collectors* (GC) übernimmt diese Aufgabe vollständig. Da mehrere Algorithmen vereint wurden ist das Resultat eine enorm schnelle Speicherverwaltung, die z.B. auch mit zirkulären Verweisen zurechtkommt.

[ElKo03]

3.4.2.3 Plattformunabhängigkeit

Anders als beispielsweise bei Java stand bei der Entwicklung des .NET Frameworks die Betriebssystemunabhängigkeit nicht im Vordergrund. Da Microsoft bestrebt ist, dem Anwender von Windowsanwendungen ein einheitliches „Look and Feel“ zu ermöglichen, setzt das .NET Framework direkt auf der Win32-API auf. In Java wird für die Darstellung der Oberfläche die Grafikbibliotheken AWT und Swing verwendet, wodurch eine solche Anwendung nie das gewohnte Aussehen einer nativen Anwendung erlangen wird. Einer Portierung auf andere Betriebssysteme steht trotz alledem nichts im Wege. Mit *MONO* existieren bereits Bestrebungen .NET den Einzug in die Linux Welt zu ermöglichen.

[ElKo03], [Kue03]

3.4.2.4 Sprachunabhängikeit

Dass der Entwickler nicht an eine einzige Programmiersprache gebunden ist, ist einer der großen Vorteile die das .NET Framework bietet. Code, der im .NET Framework ausgeführt werden soll, kann in nahezu jeder Sprache entwickelt werden, solange die Sprache gewisse Konventionen einhält, die vom Framework definiert werden.
Die Sprachunabhängigkeit wird durch eine Zwischensprache erreicht, die erst zur Laufzeit in nativen Code compiliert wird. Diese Zwischensprache namens *Intermediate Language* (IL) ist die eigentliche Programmiersprache des .NET Frameworks. Alle Spachcompiler compilieren zunächst in diese Zwischensprache. Dabei wird nach Angaben Microsofts für jede Funktionalität egal, in welcher Sprache sie implementiert ist, immer derselbe IL-Code erzeugt.

[ElKo03], [SHB02], [Kue03]

3.4.2.5 Ausführungssicherheit

Ein großes Problem, wenn nicht sogar das momentan größte Problem, mit dem der Anwender konfrontiert wird, ist die Ausführungssicherheit von Softwareprodukten. Obwohl Microsoft in den aktuellen Windows Versionen ein sehr komplexes Benutzerrechtesystem integriert hat, legt Windows bei der Installation standardmäßig für den ersten Benutzer immer noch ein Benutzerkonto an, das über Administratorrechte verfügt. Dieses hat sehr häufig zur Folge, dass Code, der z.B. über das Internet auf den Rechner des Anwenders gelangt ist, ohne Einschränkungen ausgeführt wird und auf dem Rechner Schaden anrichten kann.
In das .NET Framework ist ein Sicherheitskonzept integriert, welches auf dem o.g. richtlinienbasierten Sicherheitssystem von Windows aufsetzt und eben dieses Szenario verhindert. Wurde bis dato eine Software mit den Rechten des aktuellen Benutzers ausgeführt, so werden die Rechte in .NET unter anderem von der Herkunft der Software abhängig gemacht. Eine Software, deren Herkunft das Internet ist, hat z.B. im Standardfall nicht die Möglichkeit, auf das lokale Dateisystem zuzugreifen.

[ElKo03], [SHB02]

3.4.2.6 Das Ende der „DLL-Hölle“

Das als „DLL-Hölle“ bezeichnete Problem entstand durch inkompatible DLLs im Windows Systemverzeichnis. Mit .NET wurde nun auch das Ende eben dieses Problems propagiert. Denn nun ist es möglich, verschiedene Versionen einer DLL parallel zu installieren.

3.4.2.7 Die Weitergabe der Anwendung

.NET unterstützt weitestgehend das so genannte *„XCOPY-Deployment“*. Das heißt, eine Anwendung kann zumeist einfach auf den Installationsrechner kopiert und gestartet werden. Ermöglicht wird dies, da für die meisten Anwendungen keine Einträge in der Registrierung mehr von Nöten sind. Dementsprechend einfach ist auch das Entfernen einer Anwendung von einem Rechner.

[ElKo03], [Kue03]

3.4.3 Das .NET Frameworks im Detail

Das .NET Framework stellt ein Gerüst zur Verfügung, mit dem Anwendungen erstellt, kompiliert und ausgeführt werden können. Es setzt sich aus verschiedenen Richtlinien und Komponenten zusammen. Im Wesentlichen wären folgende Komponenten zu nennen:

- *Common Language Runtime (CLR)*
- *Framework Class Library (FCL)*
- *Common Type System (CTS)*
- *Common Language Specification (CLS)*

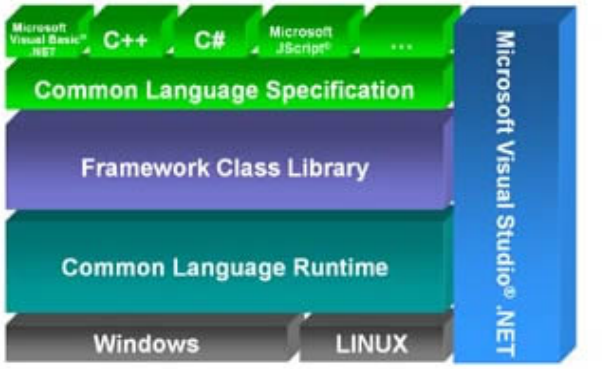

Abbildung 3.1 Microsoft .NET Framework Architektur

3.4.3.1 Die Common Language Runtime (CLR)

Die *Common Language Runtime* ist die Umgebung in der die .NET Anwendungen ausgeführt werden – gewissermaßen die allen gemeinsame Laufzeitschicht. Code, der von der *CLR* ausgeführt wird, wird allgemein als *managed Code* bezeichnet. Die *Common Language Runtime* hat einen besonderen Stellenwert, denn sie bildet den Kern von .NET. Zusätzlich zur Ausführung von Anwendungen übernimmt diese Komponente zahlreiche Aufgaben, die im Zusammenhang mit der Ausführung der Anwendungen stehen. Darunter fallen Typ-, Versions-, Speicher-, Prozessraum- und Sicherheitsmanagement.

Die *Common Language Runtime* hält folgende Dienste bereit, um diese Aufgaben zu erfüllen:

- den *Class Loader*, um Klassen in die Laufzeitumgebung zu laden
- den *Type Checker* zur Unterbindung unzulässiger Typkonvertierungen
- den *JITter*, um den IL-Code zur Laufzeit in nativen Code zu übersetzen
- den *Exception Manager*, der die Ausnahmebehandlung unterstützt
- den *Garbage Collector*, die die automatische Speicherbereinigung anstößt, wenn Objekte nicht mehr benötigt werden
- den *Code Manager*, der die Ausführung des Codes verwaltet
- die *Security Engine*, die sicherstellt, dass die Ausführung der entsprechend der Sicherheitsrichtlinie verläuft
- die *Debug Machine*, zum Debuggen der Anwendung
- den *COM Marshaller* zur Sicherstellung der Kommunikation mit *COM*-Komponenten
- einen unbenannten Dienst, der zur Laufzeit den Zugriff auf die .NET Klassenbibliothek ermöglicht

[Kue03]

3.4.3.2 Das Common Type System (CTS)

Jede Entwicklungsumgebung beinhaltet ein Typsystem, in dem es einerseits Datentypen bereitstellt und andererseits Vorschriften definiert, nach denen der Entwickler eigene Typen erstellen kann. Darüber hinaus hält es Regeln für den Zugriff auf die Typen bereit.

Das *CTS* enthält alle in .NET verfügbaren Typen und stellt somit sicher, dass alle Datentypen an zentraler Stelle definiert sind. Dadurch dass die Datentypen für alle .NET Sprachen gleich sind, bildet das *Common Type System* die Basis für die im vorherigen Abschnitt angesprochene Sprachunabhängigkeit.

Es müssen jedoch nicht alle Sprachen auch alle Typen unterstützen. *C#* kann als Systemsprache von .NET mit allen Typen umgehen. Alle anderen Sprachen müssen, um die Interoperabilität zu gewährleisten, die *Common Language Specification* unterstützen.

[ElKo03], [Kue03]

3.4.3.3 Die Common Language Specification (CLS)

Die *CLS* ist eine Untermenge des *Common Type Systems*. Diese muss von allen Sprachen unterstützt werden. Wie bereits oben angesprochen, werden von der *CLS* nicht nur Typen definiert sondern auch Richtlinien aufgestellt, um die Interoperabilität zu gewährleisten.

Die *Common Language Specification* ist ein offener Standard, so dass andere Sprachen auf .NET portiert werden können. Bislang ist dies bereits mit *Delphi*, *Cobol* und *Fortran* geschehen.

[ElKo03], [Ri02]

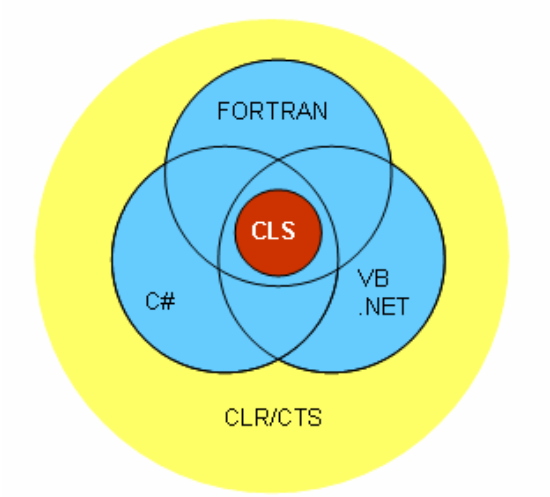

Abbildung 3.2 Common Language Specification (CLS)

[Ri02]

4 Einführung in UML

4.1 UML-Klassendiagramme

In den nächsten Kapiteln wird eingehend das FlowSim 2003 Objektmodell besprochen. Um ein Grundverständnis der verwendeten Modellierungssprache UML zu erhalten, wird hier ein Einblick in die wesentlichen Konzepte von Klassendiagrammen gegeben.
Ein Klassendiagramm beschreibt die Typen von Objekten in Systemen und die verschiedenen Arten von statischen Beziehungen zwischen diesen.

Es gibt zwei grundsätzliche Arten statischer Beziehungen:

- *Assoziation* („ein Auto hat vier Räder“)
- *Untertypen* („ein Cabrio ist ein Auto“)

Klassendiagramme zeigen aber auch die Attribute und Operationen einer Klasse sowie die Einschränkungen bei der Verbindung ihrer Objekte.

4.2 Assoziationen

Assoziationen repräsentieren Beziehungen zwischen Instanzen von Klassen. Jede Assoziation besitzt zwei Assoziationsenden. Jedes Ende ist mit einer der Klassen der Assoziation verbunden. Ein Ende kann explizit mit einer Beschriftung benannt werden. Diese Beschriftung wird Rollenname genannt. Wenn keine Beschriftung vorhanden ist, benennt man das Ende nach der Zielklasse.
Ein Assoziationsende besitzt auch eine *Multiplizität*, die besagt, wie viele Objekte an der Beziehung beteiligt sein können. Eine 1 deutet auf genau ein Objekt hin. Ein * repräsentiert 0 bis unendlich viele Objekte. 2..4 würde auf 2 bis 4 Objekte hindeuten.

Abbildung 4.1 UML-Beispiel: Assoziation

Die *Navigierbarkeit* wird durch Pfeile an den Assoziationsenden verdeutlicht. Existiert eine Navigierbarkeit nur in einer Richtung, nennt man die Assoziation *gerichtete (unidirektionale) Assoziation*. Eine *ungerichtete (bidirektionale) Assoziation* enthält Navigierbarkeiten in beide Richtungen. In UML bedeuten Assoziationen ohne Pfeile, wie in Abbildung 4.1 dargestellt, dass die Navigierbarkeit unbekannt oder die Assoziation ungerichtet ist.
Eine Assoziation repräsentiert eine dauerhafte Verbindung zwischen zwei Objekten. Das heißt, die Verbindung existiert während der gesamten Lebensdauer der Objekte, obwohl die miteinander verbundenen Instanzen sich während dessen ändern.

4.3 Attribute

Attribute sind Assoziationen sehr ähnlich. Im objektorientierten Modell würde man ein Attribut als Feld bezeichnen. Je nachdem wie detailliert ein Diagramm ist, kann die Notation für ein Attribut den Attributnamen, den Typ und den voreingestellten Wert zeigen.

Die UML Syntax ist: *Sichtbarkeit Name: Typ = voreingestellter Wert*.

Was ist der Unterschied zwischen einem Attribut und einer Assoziation?
Aus konzeptioneller Sicht gibt es keinen Unterschied. Ein Attribut bietet lediglich eine andere Sichtweise, die man benutzt, wenn sie einem geeignet erscheint. Attribute weisen gewöhnlich einen einzelnen Wert auf.
Für die Implementierung zeigt sich dennoch ein Unterschied, denn Attribute implizieren eine Navigierbarkeit ausschließlich vom Typ zum Attribut. Weiterhin ergibt sich, dass der Typ seine eigene Kopie des Attributobjekts für sich enthält. Jeder als Attribut benutzte Typ besitzt also Wert- und nicht Referenzsemantik. Zum Verständnis sei erwähnt, dass Werttypen ihren tatsächlichen Wert in den für sie reservierten Speicherbereich ablegen, während der Speicherbereich eines Referenztypen lediglich einen Verweis auf das tatsächliche Objekt enthält.

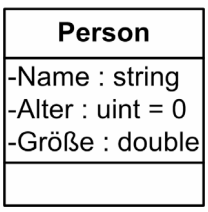

Abbildung 4.2 UML-Beispiel: Attribute

4.4 Operationen

Operationen sind die Prozesse, die ein Typ ausführen kann. Sie korrespondieren mit den öffentlichen Methoden einer Klasse. Normalerweise zeigt man einfache Operationen zur Manipulation von Attributen nicht an, da man einfach auf sie schließen kann.

Die UML-Syntax für Operationen ist:
Sichbarkeit Name (Parameterliste) : Rückgabetypausdruck {Eigenschaften}

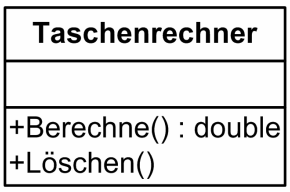

Abbildung 4.3 UML-Beispiel: Operationen

4.5 Generalisierung

Generalisierung bedeutet, dass die Schnittstelle des Untertyps alle Elemente des Obertyps enthält. Die Schnittstelle des Untertyps bezeichnet man somit als konform zur Schnittstelle des Obertyps.
Für die Implementierung ist Generalisierung gleichzusetzen mit Vererbung. Die Unterklasse erbt alle Methoden und Felder von der Oberklasse und kann geerbte Methoden überschreiben.
[FoSc02]

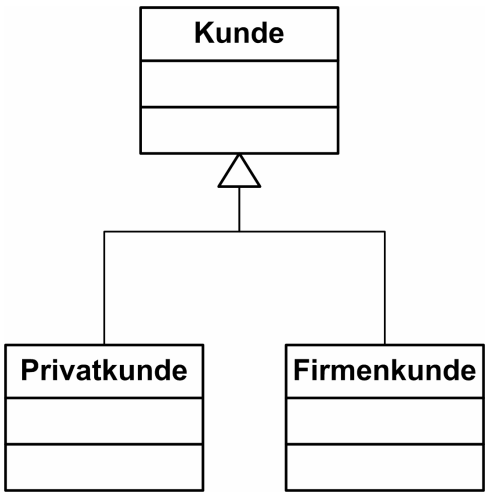

Abbildung 4.4 UML-Beispiel: Operationen

5 Namespaces im FlowSim 2004 .NET

5.1 Übersicht vorhandener Namspaces

Im FlowSim 2004 .NET Objektmodell stehen eine Vielzahl von Klassen zur Verfügung. Diese sind je nach Verwendung in Namespaces organisiert. Konvention im FlowSim Objektmodell ist, dass alle Klassen mit den Anfangsbuchstaben des zugehörigen Namespace beginnen. So ist beispielsweise allen Klassen im Namespace Geometries das Kürzel „Geo“ vorangestellt.

Die Struktur der Namespaces ist wie folgt:

FlowSim
FlowSim.Core
FlowSim.Dialogs
FlowSim.GUI
FlowSim.Document
FlowSim.Document.Editing
FlowSim.LB
FlowSim.LB.Core
FlowSim.LB.MC
FlowSim.LB.Qt
FlowSim.LB.Remoting
FlowSim.Geometries
FlowSim.Geometries.GeoData
FlowSim.Geometries.GeoVisual

5.2 Die Typen des Namespace FlowSim.Geometries

5.2.1 Die elementaren Datentypen Geo2D und GeoWindowCoords

Der elementarste Datentyp des Namespace ist der Typ *Geo2D*. Er besitzt lediglich zwei Attribute vom Typ *double*. Außerdem sind für diesen Datentyp die gängigen Operatoren wie +, -, *, ==, != überladen. Der Typ *Geo2D* ist bewusst nicht als eine Klasse, sondern als Struktur implementiert. Dazu ist zu wissen, dass Strukturen im .NET Framework Wertsemantik aufweisen, d.h. Instanzen des Typs werden, sofern sie lokal verwendet werden, nicht auf dem Heap sondern auf dem Stack angelegt. Da das Reservieren von Speicher auf dem Stack wesentlich schneller verläuft als für Speicher auf dem Heap, ist die Implementierung des Typs *Geo2D* als Struktur zu bevorzugen.
Verwendung findet dieser Datentyp hauptsächlich für die Repräsentation von 2D Koordinaten.

Der Datentyp *GeoWindowCoords* ist ebenso wie der Typ Geo2D als Struktur definiert. Folglich gelten für ihn ebenfalls die vorausgegangenen Anmerkungen bzgl. der Allokierung von Speicher.
Die Struktur *GeoWindowCoords* besitzt bis zu vier Attribute und repräsentiert im Allgemeinen einen bestimmten Bereich innerhalb eines 2D-Koordinatensystems. Dabei definiert sie diesen Bereich über zwei Eckpunkte (links unten und rechts oben).

5.2.2 Der Typ GeoTransformMatrix

Der FlowSim 2004 .NET berechnet zweidimensionale Strömungsprobleme und verwendet daher ausschließlich zweidimensionale kartesische Koordinatensysteme. In diesem Zusammenhang sind des öfteren Transformation zwischen verschienen Koordinatensystemen vorzunehmen. Beispielsweise findet eine Transformation bei der Darstellung der geometrischen Objekte statt. Hierfür werden die Weltkoordinaten der geometrischen Objekte zu Bildschirmkoordinaten umgerechnet und umgekehrt. Eine weitere Transformation, ähnlich der erst genannten, ist notwendig, wenn die Objekte auf die bereits erwähnte Geometriematrix abgebildet werden.

Mathematisch erfolgen diese Transformationen durch *affine Abbildungen*. Dazu gehören beispielsweise Translationen und Skalierungen. In homogenen Koordinatensystemen können diese affinen Abbildungen einzeln oder auch in Kombination als eine 3x3-Matrix dargestellt werden. Eine Einführung in die Thematik affiner Abbildungen ist unter [Fe92] zu finden.

Im FlowSim 2004 .NET Objektmodell übernimmt die Klasse *GeoTransformMatrix* die Funktion einer solchen Transformationsmatrix. Für diese Klasse ist der Multiplikationsoperator überladen, so dass dieser auf Objekte des Datentyps *Geo2D* angewendet werden kann.
Der Typ *GeoTransformMatrix* bildet somit die Grundlage für Transformationen zwischen verschiedenen Koordinatensystemen.

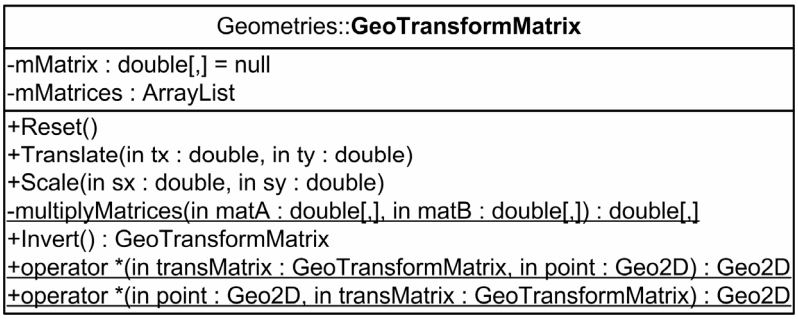

Abbildung 5.1 UML-Notation: GeoTransformMatrix

5.2.3 Die GeoVisual- und GeoDataTypen

Ein zentrales Element im FlowSim 2004 .NET ist das Erstellen und Modifizieren von Geometrien für die Strömungsberechnung. Zu diesem Zweck stehen verschiedene geometrische Objekte zur Verfügung. Zu ihnen zählen Kreise, Rechtecke und Polygone. Einen Sonderfall bildet das so genannte Boundarypolygon, das mit besonderen Eigenschaften bzw. Randbedingungen versehen werden kann. Der Namespace *Geometries* stellt für die Repräsentation dieser Geometrien im Objektmodell zahlreiche Typen bereit. Aus Gründen der Modularität, auf die im Verlauf der Abhandlung noch genauer eingegangen wird, wird an dieser Stelle zwischen Typen, die zur Darstellung dienen und solchen, die die Daten und Logik implementieren unterschieden.
Erstgenannte sind im Namespace *GeoVisual* untergebracht, wohingegen der Namespace *GeoData* die Typen zur Datenhaltung aufnimmt.

Alle Typen bzw. Klassen im Namespace *GeoData* implementieren ein spezielles Interface, um ihre Daten und Logik für die Darstellungstypen zur Verfügung zu stellen.
Zudem besitzen alle Datentypen die zur Visualisierung dienen über das Interface *IVisual* eine Methode namens *Paint()*, die letztlich zur Darstellung auf dem Bildschirm dient. Ähnliches gilt für die Data-Typen und das Interface *IDataBase*. Mittels diesem können die Data-Typen über den Aufruf der Methode *Discretize()* auf die Geometriematrix abgebildet werden.

Beispielhaft für die Implementierung der Geometrieobjekte sind in Abbildung 5.2 die Typen *VisCircle* und *DataCircle* in einem Klassendiagramm dargestellt.

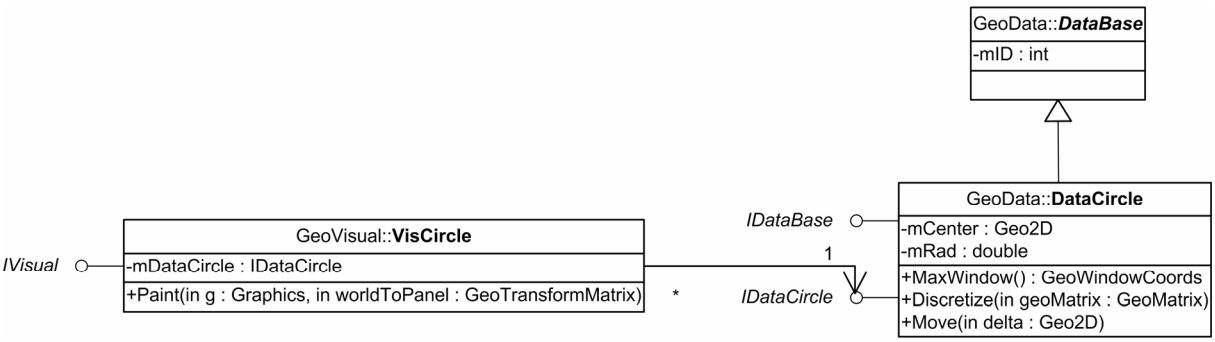

Abbildung 5.2 UML-Klassendiagramm: Relation VisCircle – DataCircle

Der Vorteil einer solchen Modularität ist, dass ein Darstellungsobjekt keine besondere Kenntnis von der Implementierung des zu Grunde liegenden Datenobjekts haben muss. D.h. die Datenhaltung und genaue Details der Logik bleiben dem *VisualTyp* verborgen und können somit beliebig modifiziert werden.
Folglich ist ein *VisualTyp* nicht an einen bestimmten *DataTyp* gebunden und kann somit auf beliebigen *DataTypen* operieren sofern diese das geforderte Interface implementieren.

5.2.4 Collections – Listen in .NET

Der Namespace *Geometries* enthält zwei Typen die die Datenhaltung der Geometrieobjekte übernehmen. Der Typ *DataGeoList* kann Objekte vom Typ *IDataBase* aufnehmen, während der Typ *VisGeoList* entsprechend *IVisual* Objekte aufnehmen kann.

Beide Klassen sind von der abstrakten Basisklasse *CollectionBase* abgeleitet. Wie sich aus dem Namen schließen lässt, erhalten sie von dieser Basisklasse die Funktionalität einer *Collection*. Collections sind in etwa vergleichbar mit den *STL-Listen* in C++.
Die Funktionalität der beiden Listenklassen ist so implementiert, dass sie dem Standardverhalten der .NET eigenen Collections entspricht.
Eine ausführliche Beschreibung der im .NET Framework enthaltenen Collections und der Basisklasse *CollectionBase* ist in [ElKo03] zu finden.

Die Listenklassen aus dem Namespace *Geometries* können über den so genannten *Indexer* angesprochen werden, oder aber über einen *Enumerator*. Letzteres ermöglicht es, die Listen mit einer *foreach-Anweisung* zu durchlaufen, was die Handhabung sehr vereinfacht.

Die Liste vom Typ *DataGeoList* stellt außerdem eine Methode namens *Discretize()* zur Verfügung, die es ermöglicht alle enthaltenen *DataObjekte* auf ein Objekt vom Typ *GeoMatrix* abzubilden.
Die Klasse *VisGeoList* besitzt ebenfalls eine *Paint()*-Methode, die zur Darstellung der enthaltenen *VisObjekte* dient.

5.2.5 Der Typ GeoMatrix – ein gerastertes Abbild

In Abschnitt 2.4.1 wurde bereits das Speichermodell des FlowSim 2004 .NET und die Abbildung der Geometrieobjekte auf Matrizen besprochen. Zur vereinfachten Vorstellung einer solchen Matrix gilt, dass eine Geomatrix ein gerastertes Abbild des Strömungsfeldes darstellt. Für die visuelle Darstellung sei hier noch einmal auf Abbildung 2.2 verwiesen.
Für diese Form der Darstellung hält der Namespace *Geometries* unter anderem den Datentyp *GeoMatrix* bereit. Als Attribute besitzt diese Klasse zum einen ein Array zur Speicherung der Daten der Gitterknoten, und zum anderen Attribute die Informationen über die Größe der Matrix und deren Gitterweite enthalten.
Außerdem enthält der Typ *GeoMatrix* eine Referenz auf eine Instanz der Klasse *WorldToGeoMatrixTransformer*. Der Typ *WorldToGeoMatrixTransformer* kann ein Objekt vom Datentyp *GeoTransformMatrix* liefern und ermöglicht so eine Transformation der Geometrieobjekte in das Koordinatensystem der *GeoMatrix* und umgekehrt.
Wie für die meisten Datentypen im Namespace *Geometries* existiert auch für den Typ *GeoMatrix* eine Klasse, die eine visuelle Darstellung ermöglicht. In diesem Fall ist dies die Klasse *GeoVisMatrix*. Ein Objekt dieses Typs erhält bei seiner Instantiierung eine Referenz auf ein Objekt des Typs *GeoMatrix*. Daraufhin ist über den Aufruf der Methode *Paint()* eine Darstellung des Objekts möglich.

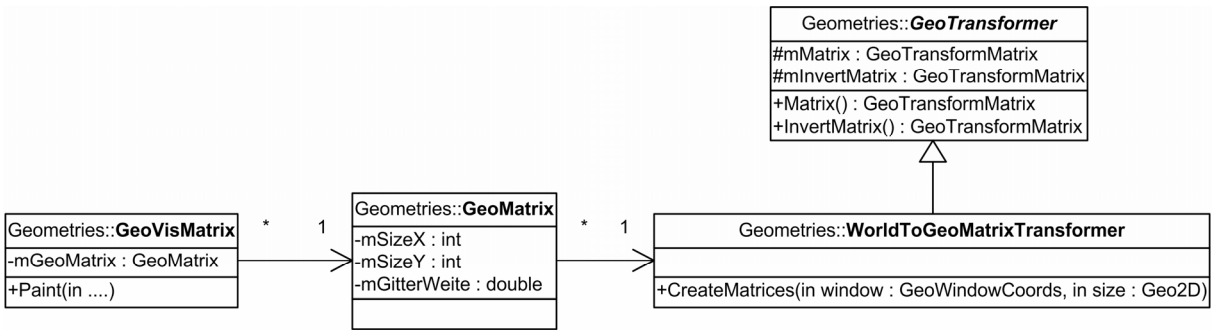

Abbildung 5.3 UML-Klassendiagramm: GeoMatrix – GeoVisMatrix

5.3 Der Namespace FlowSim.Core

Im Namespace *FlowSim.Core* sind die elementaren Datentypen enthalten, die teilweise in mehreren Komponenten der Anwendung verwendet werden. Im Wesentlichen ist hier nur die Klasse *FlowMessenge*r von Interesse. Innerhalb der Anwendung übernimmt ein Objekt dieses Typs die Funktion eines „Nachrichtenübermittlers“. Die meisten Objekte haben während des Programmablaufs die Möglichkeit auf eine Instanz des *FlowMessengers* zuzugreifen und mittels diesem Nachrichten in der Statusleiste auszugeben.
Hierbei handelt es sich allerdings lediglich um Nachrichten, die zur Information des Benutzers dienen. Diese Nachrichten sind nicht zu verwechseln mit *Events*, die Einfluss auf den Programmablauf haben.

5.4 Der Namespace FlowSim.Dialogs

Der Namespace enthält ausschließlich Dialoge die der Benutzerinteraktion dienen. Zudem sind in diesem Namespace einige abgeleitete Steuerelemente enthalten, die für die Verwendung im FlowSim 2004 .NET Objektmodell angepasst wurden.

5.5 Weitere Namespaces

Die übrigen Namespaces werden im nächsten Kapitel ausführlich betrachtet und werden hier nicht näher erläutert. Dennoch soll erwähnt werden, wofür die enthaltenen Klassen hautsächlich Verwendung finden.
Der Namespace *FlowSim.GUI* enthält die Typen, die eine Visualisierung und Benutzerinteraktion ermöglichen.
Im Namspace *FlowSim.Document* sind die Typen enthalten, die für die Datenhaltung und Modifikation zuständig sind.
Die Klassen des Namespace *FlowSim.LB* stellen die Logik zur eigentlichen Simulation bereit.

6 Das Objektmodell des FlowSim 2004 .NET

6.1 Die Modularität der Anwendung

Der Entwurf der Anwendung ist dem Leitbild des .NET Frameworks folgend vollständig *objektorientiert*. Zudem sind einzelne Programmteile in Komponenten gekapselt. Derartige Komponenten, die eine grafische Benutzerschnittstelle besitzen, werden in .NET als *Steuerelemente* bezeichnet. Als solche .NET-Steuerelemente sind beispielsweise das *FlowPanel* und der *FlowController* implementiert.

Die Programmierung mittels Steuerelementen wird oftmals auch als *Komponentenorientierte Programmierung* bezeichnet. Sie soll den nächsten Evolutionsschritt nach der Objektorientierung darstellen.

Der Aufbau der Anwendung ist weitestgehend modular. Dies bedeutet, dass sie in unabhängig voneinander agierende Programmteile, hier Module bezeichnet, unterteilt ist. Der modulare Aufbau des FlowSim 2004 .NET wird durch die Objektorientierung ermöglicht. Im objektorientierten Modell werden die Module durch Klassen bzw. Steuerelemente repräsentiert.

An Stellen an denen eine Kommunikation bzw. Interaktion der einzelnen Komponenten erfolgt, findet diese über definierte Schnittstellen, so genannte *Interfaces*, oder durch *abstrakte Basisklassen* statt.

Für die Interaktion der einzelnen Programmteile ist folglich keine Kenntnis genauer Details der Implementierung nötig. Dies ermöglicht, dass einzelne Programmteile sehr einfach gegen andere ausgetauscht oder erweitert werden können, ohne den Entwurf der Anwendung zu beeinflussen.

Zudem können die einzelnen Module unabhängig voneinander entwickelt werden. Somit ist es möglich, dass verschiedene Entwickler auf einfache Art zusammen an einem Projekt arbeiten.

Die Objektorientierung hat bereits zur Entwurfszeit den Vorteil, dass dieser auf einem sehr hohen Abstraktionsniveau stattfinden kann. Für den Anwendungsentwurf sind genaue Implementierungsdetails genau so wenig von Interesse wie sprachspezifische Eigenschaften.

Folglich ist der Entwurf sehr flexibel und einfach zu erweitern und zu modifizieren.

6.2 FlowSim 2004 .NET - eine dreischichtige Anwendung

Der Aufbau des FlowSim lässt sich in drei Schichten unterteilen:

- GUI - Darstellung und Benutzerinteraktion
- Data - Datenhaltung und -modifikation
- Simulation - Berechnungskerne

Die Benutzerinteraktion und die Darstellung der Daten wird hauptsächlich von den Klassen *FlowForm* und *FlowPanel* übernommen, wobei die Klasse *FlowForm* zusätzlich das Hauptformular der Anwendung ist. Es enthält den Einstiegspunkt für die Anwendung und stellt das Bindeglied zwischen den einzelnen Programmteilen dar.
Zur grafischen Darstellung und benutzergesteuerten Erstellung neuer Geometriebjekte wird die Klasse *FlowPanel* verwendet.
Die Datenhaltung und deren Modifikation findet im Data-Layer statt. Die zentrale Klasse dieser Schicht ist die Klasse *FlowDocument*. Diese Klasse bedient sich letztlich der Typen *GeoEditor* und *DataGeoList* um diese Aufgaben zu erfüllen.
Die dritte Schicht, die Simulation, wird von der Klasse *FlowSimulation* und mehreren verschiedenen Typen gebildet die die Interfaces *ILBComputation* oder *ILBVisComputation* implementieren. In dieser Schicht finden die eigentlichen Simulationsberechnungen statt.
Die wesentlichen Klassen der einzelnen Schichten und deren Zusammenspiel sind in Abbildung 6.1 in einem Klassendiagramm dargestellt.

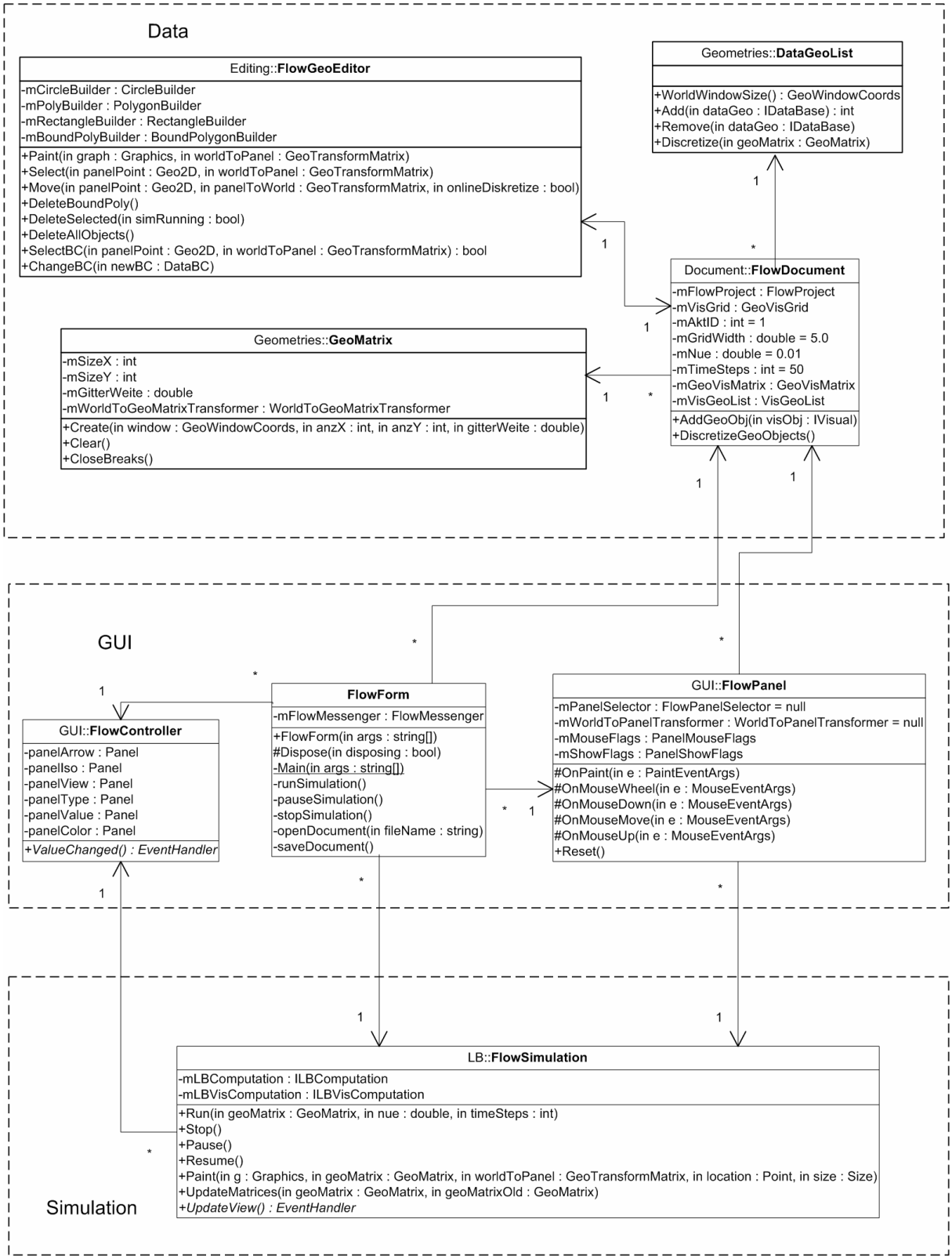

Abbildung 6.1 UML-Klassendiagramm: FlowSim 2004 .NET Objektmodell

6.3 GUI - Darstellung und Benutzerinteraktion

Die Klassen für die Darstellung und Benutzerinteraktion befinden sich im Namespace *FlowSim.GUI*.

In Abbildung 6.2 ist die Anwendungsoberfläche des FlowSim 2004 .NET dargestellt. Innerhalb dieser Darstellung sind die einzelnen Komponenten der Oberfläche gekennzeichnet. Das *FlowForm* bildet, wie bereits erwähnt, das Hauptformular. Es enthält am oberen Rand eine Menüleiste sowie eine so genannte Schnellstartleiste. Diese Elemente sind Standardelemente des .NET Frameworks und werden hier nicht näher erläutert.
Außerdem sind dem *FlowForm* Elemente vom Typ *FlowPanel* und *FlowController* hinzugefügt. Der *FlowController* ist am unteren Rand orientiert, während die übrige Fläche des Hauptformulars vom *FlowPanel* eingenommen wird.

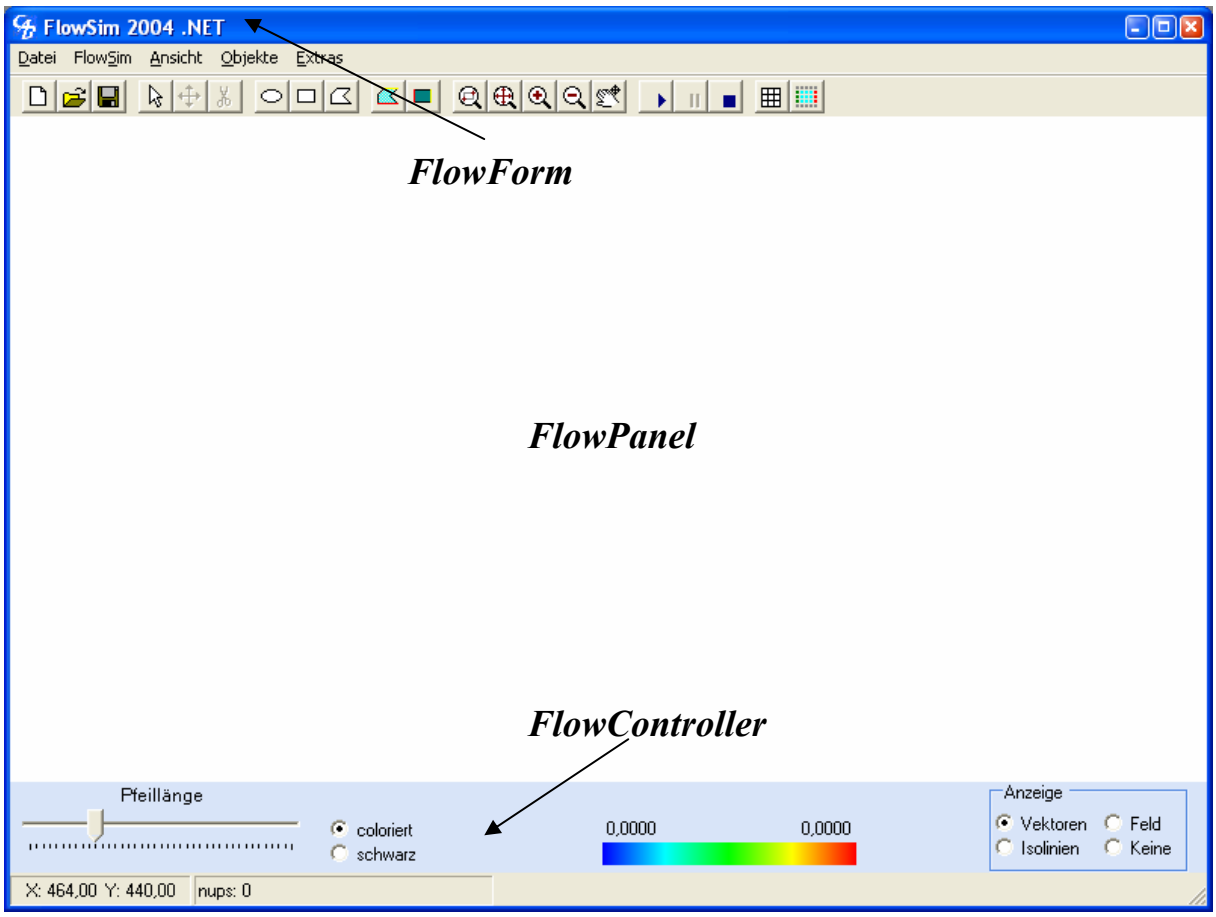

Abbildung 6.2 Anwendungsfenster des FlowSim 2004 .NET

6.3.1 Die Darstellung auf dem FlowPanel

Die Aufgabe der Darstellung der Objekte wird in der Anwendung von der Klasse *FlowPanel* übernommen. Die Klasse *FlowPanel* ist ein Steuerelement, das von dem Typ *System.Windows.Forms.Panel* abgeleitet ist. Auf dem *FlowPanel* werden sowohl die geometrischen Objekte des Strömungsgebietes als auch das Strömungsfeld der Simulation an sich angezeigt.

Um eine maßstabsgerechte Ansicht der Objekte zu ermöglichen, verwendet das *FlowPanel* ein Objekt des Typs *WorldToPanelTransformer*. Der *WorldToPanelTransformer* bietet ähnlich wie der *WorldToGeoMatrixTransformer* die Möglichkeit, ein Objekt vom Typ *GeoTransformMatrix* zu liefern. Unter Verwendung dieses Objektes kann somit eine Transformation der Objekte in das Koordinatensystem der Anzeige erfolgen, ähnlich zu der bereits in Abschnitt 5.2.5 erwähnten Abbildung von Objekten auf die *GeoMatrix*.

Die Typen *WorldToGeoMatrixTransforme*r und *WorldToPanelTransformer* haben dieselbe Basisklasse *GeoTransformer*.

Der Typ *WorldToPanelTransformer* bietet allerdings weitaus mehr Funktionalität als der Typ *WorldToGeoMatrixTransformer*. Beispielsweise ist es mittels dieser Klasse möglich, die Ansicht der Objekte in beliebiger Weise zu skalieren. In der Anwendung findet sich dies in der Zoom-Funktionalität wieder. Außerdem bietet der *WorldToPanelTransformer* die Möglichkeit die Ansicht zu verschieben oder auf einen bestimmten Bereich festzulegen.

Der Zusammenhang zwischen dem *FlowPanel* und dem *WorldToPanelTransformer* ist in Abbildung 6.3 dargestellt.

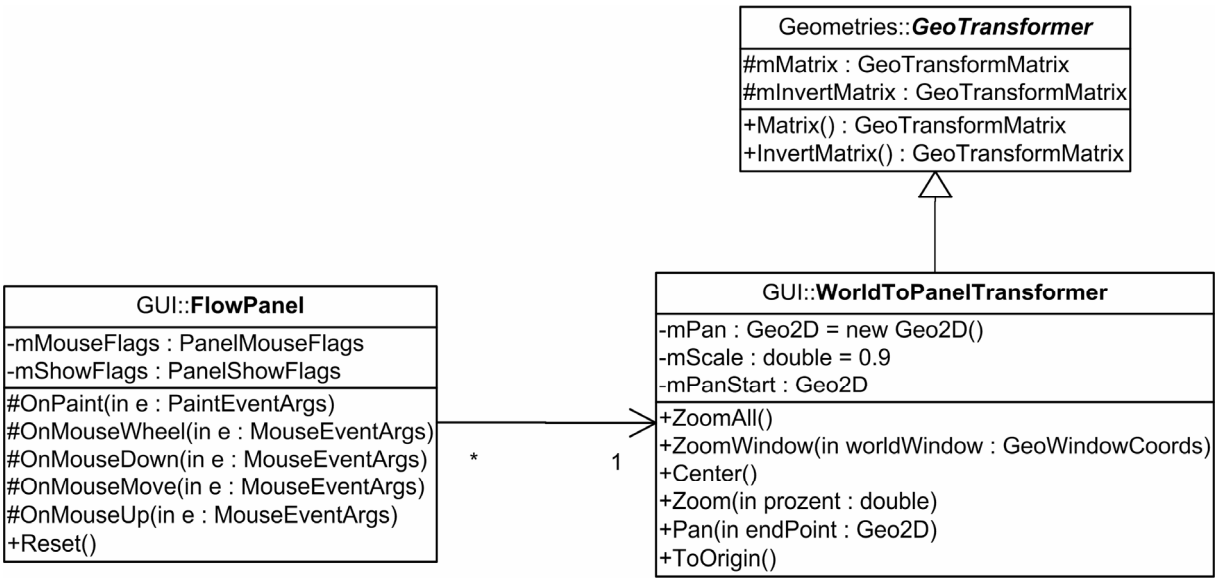

Abbildung 6.3 UML-Klassendiagramm: FlowPanel - WorldToPanelTransformer

6.3.2 Möglichkeiten der Benutzerinteraktion

6.3.2.1 Die Menüleiste

Die Benutzerinteraktion kann in der Anwendung auf vielfältige Art und Weise geschehen. Zum einen bietet das *FlowForm* als Hauptformular der Anwendung eine Menüleiste. Über diese wird dem Anwender Zugang zu den Funktionen des FlowSim 2004 .NET ermöglicht. Einige Befehle sind mit der Ausführung eines Dialogfensters aus dem bereits erwähnten Namespace *FlowSim.Dialogs* verbunden.

6.3.2.2 Die Schnellstartleiste

Die Schnellstartleiste ermöglicht dem Benutzer die Hauptfunktionen der Anwendung auf einfachere Art zu erreichen. Jedoch enthält die Schnellstartleiste nur Funktionen, die auch über die Menüleiste erreichbar sind.

6.3.2.3 Der FlowController

Der *FlowController* ist lediglich während einer laufenden Simulation am unteren Rand des Anwendungsfensters sichtbar. Ebenso wie das *FlowPanel* ist der *FlowController* ein abgeleitetes Steuerelement. Er enthält zahlreiche Elemente um die Ausführung der Simulation zu steuern. Unter anderem kann über den *FlowController* die Darstellungsart gewählt werden. Zur Auswahl stehen beispielsweise Vektorpfeile oder Isolinien.
Der für die Simulation verantwortliche Programmteil, in diesem Fall eine Instanz der Klasse *FlowSimulation*, wird zur Laufzeit des Programms mittels eines *Events* von der Benutzerauswahl benachrichtigt.

6.3.2.4 Interaktion über das FlowPanel

Das *FlowPanel* ist das zentrale Element der Visualisierung. Seine Darstellungsfunktionen wurden in Abschritt 6.3.1 bereits ausführlich besprochen. Eine weitere wichtige Funktion des *FlowPanels* ist die Benutzerinteraktion.
Diese findet innerhalb der Klasse *FlowPanel* über die Maus statt. Jede Aktion die der Benutzer mit der Maus ausführt löst ein *Event* innerhalb der Anwendung aus. Diese Events werden zusammen mit *Flags* innerhalb des *FlowPanel* dazu verwendet, das Verhalten der Anwendung zu steuern.

Eingangs wurde die Möglichkeit zur Manipulation der Darstellung im Zusammenspiel mit der Klasse *WorldToPanelTransformer* angesprochen. Diese Funktionalität ist auch dem Benutzer zugänglich. So kann dieser Beispielsweise über die Maus einen beliebigen Bildausschnitt markieren um die Ansicht auf diesen Ausschnitt zu beschränken. Wiederum über die Maus kann der Benutzer diesen Ausschnitt beliebig verschieben. Zuletzt steht dem Anwender auch eine Zoomfunktionalität zur Verfügung. Zugänglich ist diese u.a. über die Verwendung des Mausrades.

Alle bislang erwähnten Funktionen des *FlowPanels* beschränkten sich mehr oder weniger auf die Darstellung. Eine Benutzerinteraktion ist aber auch für die Erstellung neuer Geometrieobjekte notwendig. So bietet die Klasse *FlowPanel* zusammen mit der Klasse *FlowGeoEditor* aus dem Namespace *FlowSim.Document.Editing* die Möglichkeit zur Erzeugung neuer Objekte. Der Benutzer kann diese neuen Objekte zur Laufzeit mittels Maus auf dem *FlowPanel* zeichnen. Letztlich werden die Geometrieobjekte von speziellen Klassen aus dem *FlowSim.Document.Editing* erstellt. Diese Klassen werden nachfolgend genauer besprochen.

Eine weitere interaktive Funktionalität, die das *FlowPanel* in Kombination mit dem *FlowGeoEditor* anbietet, ist das Markieren bereits existierender Geometrieobjekte. Dabei wird unterschieden, ob ein Objekt mit der rechten oder der linken Maustaste markiert wird. Je nach Maustaste wird ein dynamisches PopUp-Menü generiert, das Funktionen auflistet, die auf die markierten Objekte angewendet werden können. Die Klasse *FlowPanel* besitzt zudem einen Verweis auf ein Objekt vom Typ *FlowPanelSelector,* der es ermöglicht mehrere Objekte gleichzeitig mittels eines Auswahlrechtecks zu markieren.

Markierte Objekte können wiederum durch den *FlowGeoEditor* manipuliert werden. Die Möglichkeiten die der *FlowGeoEditor* zur Datenmanipulation anbietet werden weiter unter besprochen.

Abbildung 6.4 zeigt die Beziehung zwischen den Klassen *FlowPanel* und *FlowGeoEditor*.

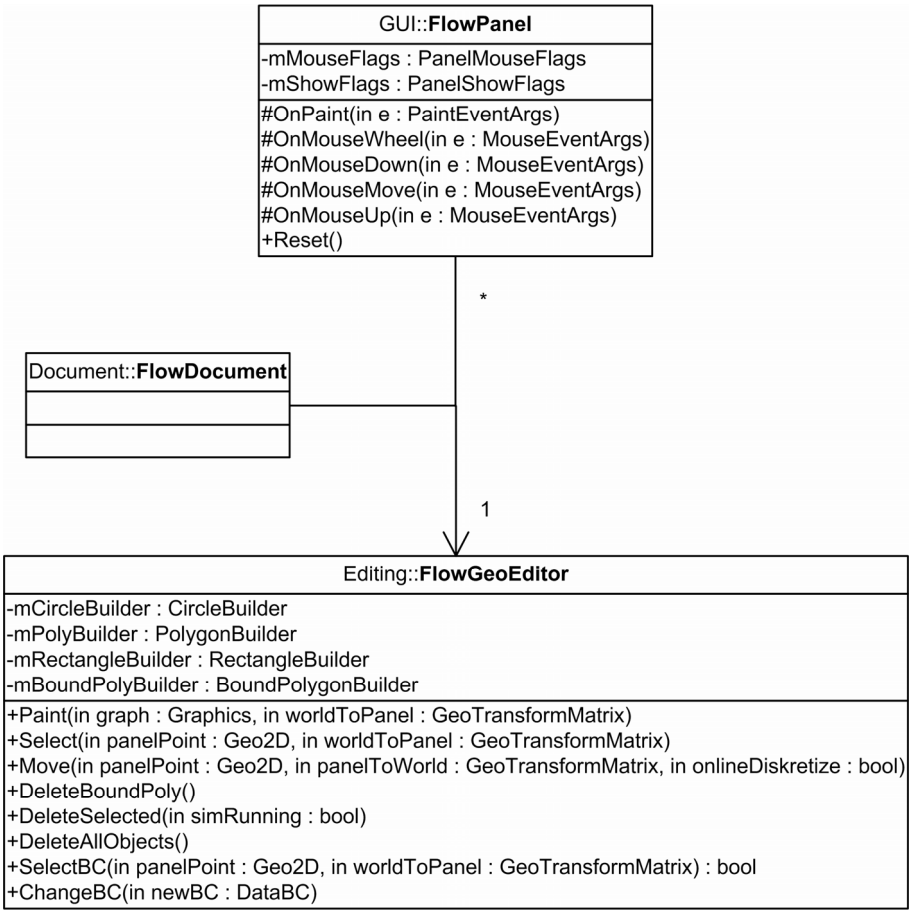

Abbildung 6.4 UML-Klassendiagramm: FlowPanel - FlowGeoEditor

6.4 Datenhaltung und Modifikation

In diesem Abschnitt wird die Datenschicht der Anwendung besprochen. Die hier behandelten Klassen dienen der Datenhaltung bzw. der Modifikation von Daten. Die zugehörigen Klassen befinden sich unterhalb des Namespace *FlowSim.Document*.

6.4.1 Der Datentyp FlowDocument

Der zentrale Datentyp dieser Schicht ist die Klasse *FlowDocument*. Der Typ *FlowDocument* stellt die Daten und die Funktionalität der Schicht für andere Klassen zur Verfügung. Er ist das Bindeglied zu anderen Schichten.
Um der Anwendung ein großes Maß an Funktionalität zu bieten, enthält der Namespace *FlowSim.Document* zahlreiche Klassen, derer sich das *FlowDocument* bedient.
Das *FlowDocument* liefert beispielsweise eine Referenz auf ein Objekt des bereits angesprochenen Datentyps *FlowGeoEditor*, das zur Erstellung und Modifikation von Geometrieobjekten dient.
Ein weiteres Element der Klasse ist ein Objekt des Typs *FlowProject,* ebenfalls aus dem Namespace *FlowSim.Document*. Dieses Objekt ist ein Abbild des aktuellen Projekts.
Das *FlowDocument* übernimmt die Aufgabe der Datenhaltung. Dazu verwendet es die bereits im Abschnitt 5.2.4 besprochene Collection *DataGeoList* aus dem Namespace *FlowSim.Geometries*. Außerdem enthält es ebenfalls eine Instanz des zugehörigen Typs *VisGeoList*, um eine Darstellung der Geometrieobjekte zu ermöglichen.
Ebenfalls bereits besprochen wurden die Typen *GeoMatrix* und *VisGeoMatrix*. Zur Laufzeit können über das *FlowDocument* Referenzen auf die Instanzen dieser Typen bezogen werden.

Die bislang ausgeführten Eigenschaften des *FlowDocuments* beschränken sich auf das Veröffentlichen von externen Funktionalitäten. Dies verdeutlicht noch einmal den zentralen Charakter der Klasse.
Die Klasse *FlowDocument* besitzt zudem Methoden, die für die Anwendung von hoher Bedeutung sind. Eine davon ist die Methode *AddGeoObj()*. Neue Geometrieobjekte können nur mit dieser Methode den Listen *DataGeoList* und *VisGeoList* hinzugefügt werden.
Eine weitere Methode der Klasse *FlowDocument* ist *DiscretizeGeoObjects()*. Diese erstellt die notwendigen Daten, die für die Abbildung der geometrischen Objekte auf die GeoMatrix benötigt werden.

Das Zusammenspiel der oben genannten Klassen ist nachfolgend in Abbildung 6.5 veranschaulicht.

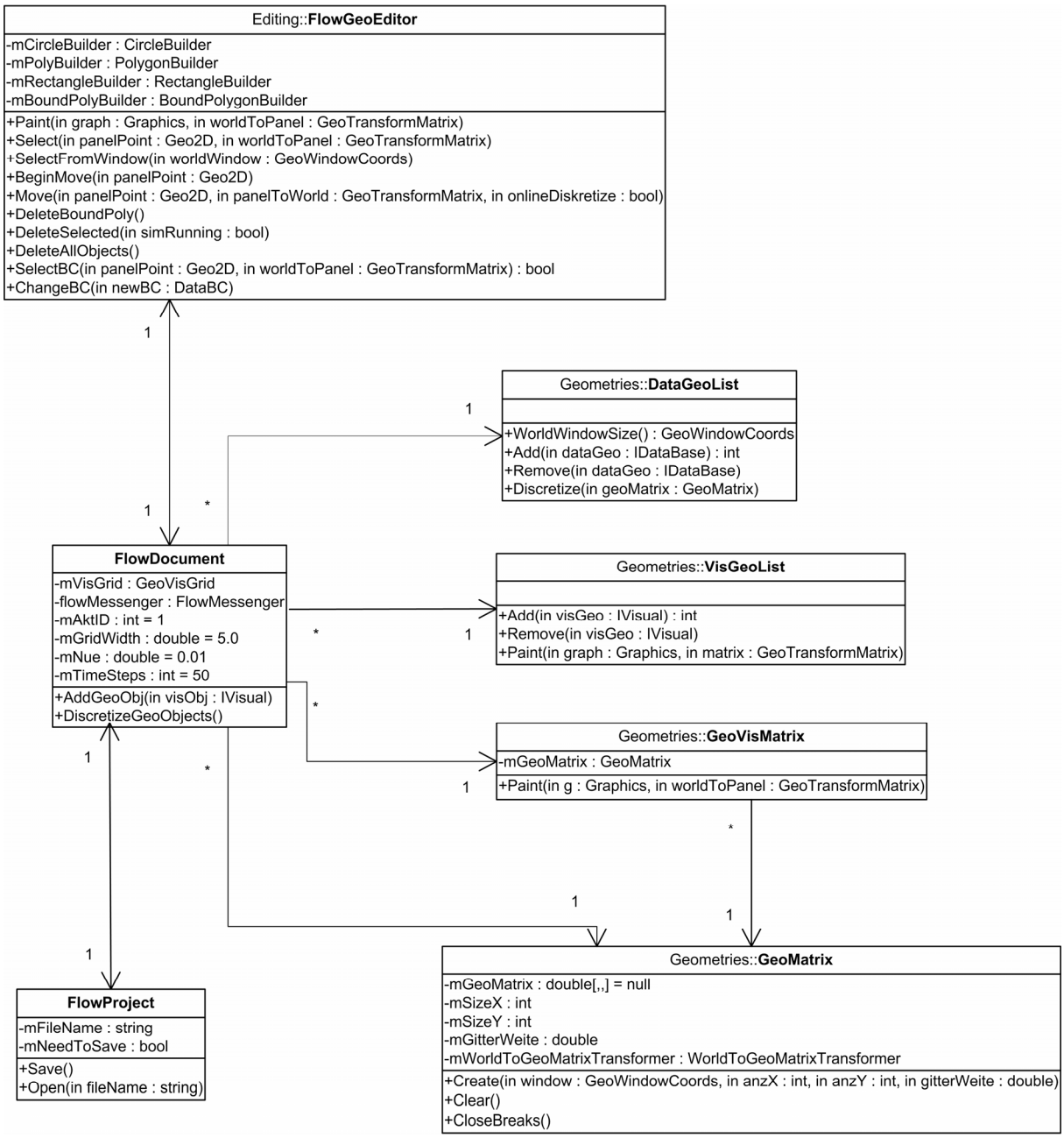

Abbildung 6.5 UML-Klassendiagramm: FlowDocument

6.4.2 Der Datentyp FlowProject

Mit der Klasse *FlowProject* werden die aktuellen Projektdaten des FlowSim 2004 .NET dauerhaft in einer Datei gespeichert oder ein bestehendes Projekt geöffnet. Bei letzterem werden die Projektdaten aus einer Datei eingelesen. Projektdateien des FlowSim 2004 .NET haben das Suffix *.flow*.
Beim Speichervorgang werden über die Klasse *FlowProject* alle für das Projekt relevanten Daten in eine *.flow-Datei* geschrieben. Zu diesen Daten zählen unter anderem die Listen vom Typ *DataGeoList* und *VisGeoList*, die die geometrischen Objekte enthalten. Um diese und alle enthaltenen Typen in Dateien speichern zu können, müssen sie durch ein spezielles Attribut als serialisierbar gekennzeichnet werden. Die meisten .NET Datentypen sind per Definition serialisierbar.

6.4.3 Der Datentyp FlowGeoEditor

Der Typ *FlowGeoEditor* ermöglicht in Kombination mit dem *FlowPanel* das interaktive Erstellen und Modifizieren von Geometrieobjekten. Der Typ bietet zahlreiche Funktion zum Zweck der Bearbeitung von Geometrieobjekten. Im Detail sind dies Methoden zum Markieren von Objekten oder einzelnen Objektelementen, wie z.B. Kanten von Polygonzügen. Darüber hinaus bietet die Klasse die Möglichkeit markierte Objekte zu löschen oder zu verschieben. Weiterhin können mit Hilfe des *FlowGeoEditors* die Randbedingungen des Umrandungspolygons geändert werden.
Zusätzlich bietet die Klasse Zugriff auf Instanzen verschiedener Objekte, welche die Programmlogik zum interaktiven Erstellen von Geometrieobjekten enthalten. Diese Objekte befinden sich, ebenso wie der Typ *FlowGeoEditor*, im Namespace *FlowSim.Document.Editing* und implementieren alle das Interface *IGeoBuilder*. Im Einzelnen sind diese Objekte und deren Vererbungshierarchie in Abbildung 6.6 dargestellt.

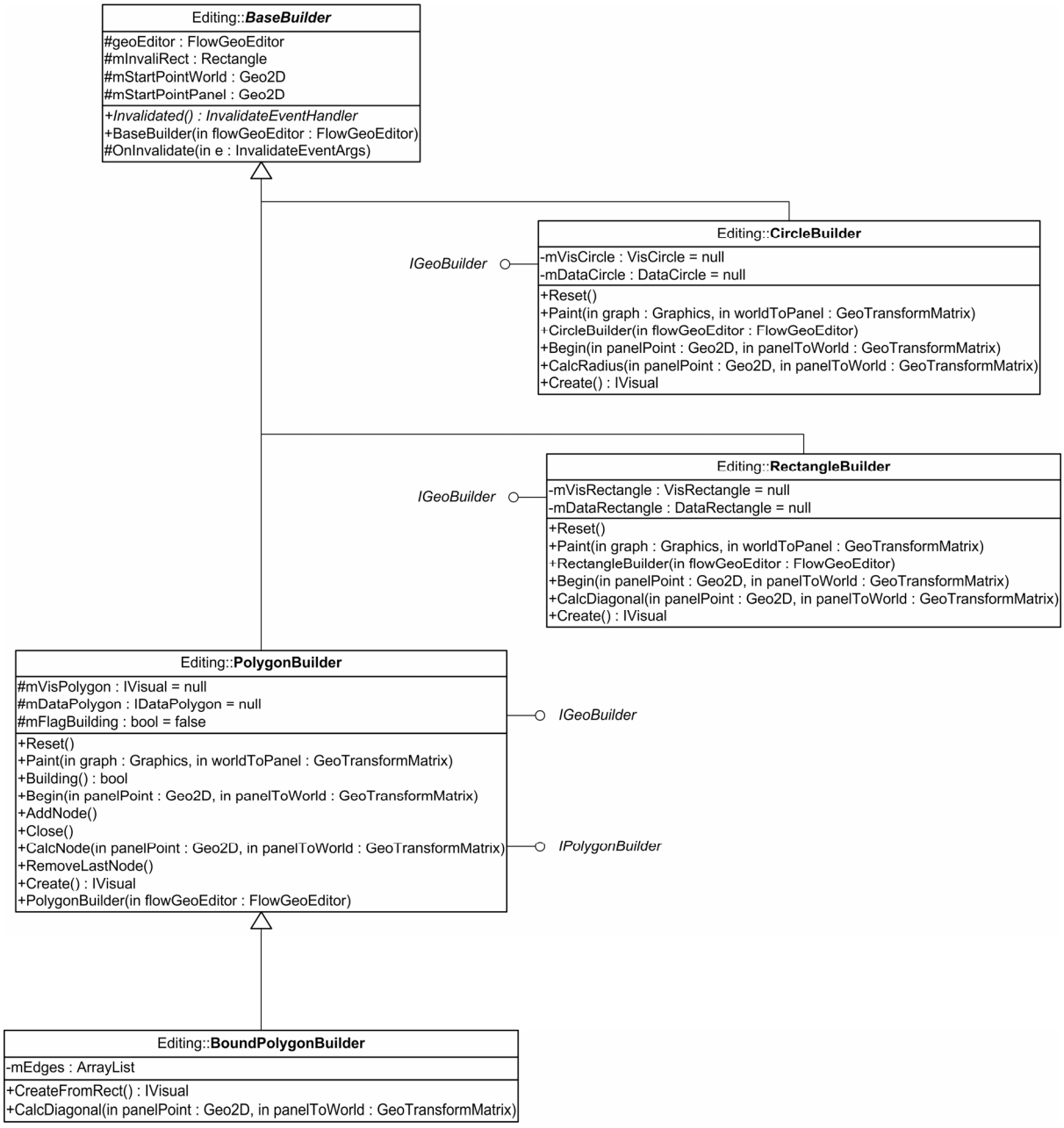

Abbildung 6.6 UML-Klassendiagramm: Vererbungshierarchie GeoBuilder

6.5 Simulation

In der Simulationsschicht findet die eigentliche Berechnung auf Basis der in Kapitel 2 beschriebenen Lattice-Boltzmann-Methode statt.
Die Klassen, die für diese Berechnung verwendet werden, befinden sich unterhalb des Namespace *FlowSim.LB*.
Prinzipiell enthält dieser Namespace drei verschiedene Arten von Klassen.

6.5.1 Der Typ FlowSimulation

Die zentrale Klasse des Namespace ist die Klasse *FlowSimulation.* Sie hat eine ähnliche Aufgabe wie die Klasse *FlowDocument* in der Datenschicht. Der Typ *FlowSimulation* stellt Methoden und Attribute zur Verfügung, um die Berechnung und deren Darstellung zu steuern.
Zur eigentlichen Berechnung stehen weitere Klassen im Namespace bereit. Diese Klassen haben gemein, dass sie das Interface *ILBComputation* implementieren über das sie von der Klasse *FlowSimulation* angesprochen werden. Weiterhin steht zur Darstellung jeder dieser Berechnungsklassen eine weitere Klasse bereit, die im Kontext der Klasse *FlowSimulation* über das Interface *ILBVisComputation* angesprochen wird.

Beispielhaft für alle Berechnungsklassen und die entsprechenden Darstellungsklassen ist ihre Verwendung innerhalb der Anwendung in Abbildung 6.7 veranschaulicht.

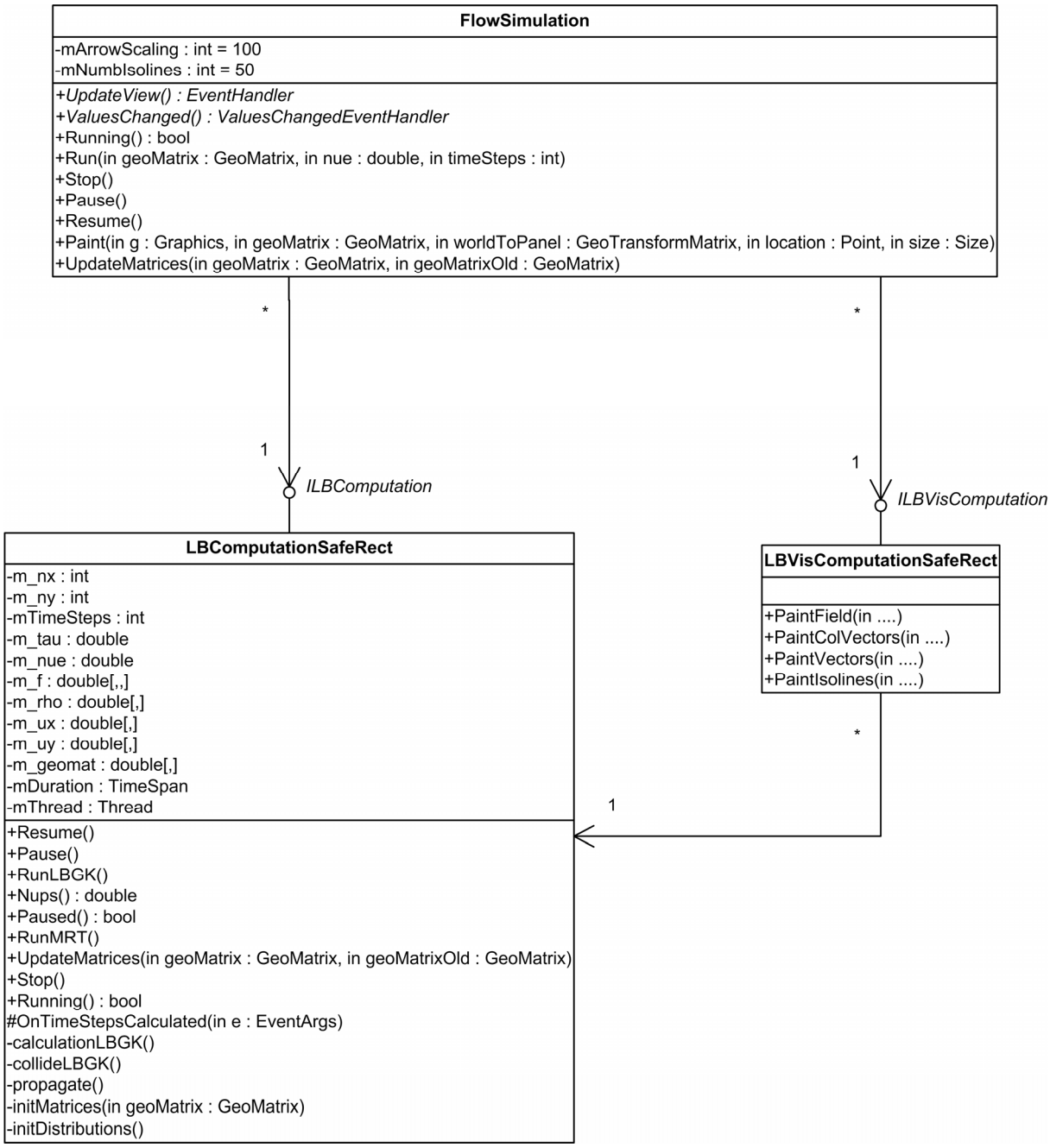

Abbildung 6.7 UML-Klassendiagramm: FlowSimulation

6.5.2 Die LBComputation Typen

Die *LBComputation-Klassen* des Namespace *FlowSim.LB* enthalten die Berechnungskerne. Da die Berechnung innerhalb der Anwendung auf verschiedene Arten ausgeführt werden kann, sind mehrere Klassen mit unterschiedlichen Berechnungsfunktionen vorhanden. Diese implementieren, wie im vorherigen Abschnitt beschrieben, das Interface *ILBComputation*, um von der Klasse *FlowSimulation* angesprochen werden zu können.
Auf die detaillierten Unterschiede bezüglich der Implementierung wird im nächsten Kapitel eingegangen. An dieser Stelle wird lediglich der Aufbau der Klassen und deren Funktionsweise besprochen, die allen Berechnungsklassen gemein ist. Diese Gemeinsamkeit ist besonderes hervorzuheben, denn sie ist die Grundlage für den abschließenden Performancevergleich.

6.5.2.1 Instantiierung

Ein Objekt der *LBComputation-Klassen* bekommt zur Instantiierung die Geometrie des Strömungsfeldes in Form der *GeoMatrix* mitgeteilt. Des Weiteren sind zusätzlich Parameter für die kinetische Viskosität und die Anzahl der Zeitschritte vorhanden. Die Zahl der Zeitschritte gibt an, nach wie vielen Berechnungsschritten die Darstellung des Strömungsfeldes aktualisiert werden soll. Dies geschieht wiederum mittels eines *Events*. Im Konstruktor der Klassen werden daraufhin individuelle Variablen und Matrizen für Geometrie, Geschwindigkeit, Dichte und Teilchenverteilung initialisiert. Zu diesem Zweck besitzt jede Klasse u.a. die privaten Methoden *initMatrices()* und *initDistributions()*.

6.5.2.2 Starten der Berechnung – ein neuer Thread

Nach dem Aufruf des Konstruktors kann die Berechnung über den Aufruf einer der beiden *Run()*-Methoden erfolgen. Die beiden Methoden initiieren unterschiedliche Berechnungsmethoden. Ein Aufruf der Methode *RunLBGK()* bewirkt, dass die Berechnung mittels der im Kapitel 2 beschriebenen *Lattice-Boltzmann-Methode* erfolgt. Die zweite Variante hat eine Berechnung mittels der Momentenmethode zur Folge. Auf das Momentenmodell wird an dieser Stelle nicht näher eingegangen. Detaillierte Informationen hierzu können [LaLu] entnommen werden. Die relevanten Unterschiede dieser beiden Methoden werden im Verlauf der Abhandlung noch verdeutlicht.

Innerhalb der Methode *Run()* wird die Berechnung über den Aufruf der entsprechenden Methode *calculation()* gestartet. Damit die Berechnung während der Visualisierung nicht unterbrochen werden muss, wird die Methode *calculation()* innerhalb eines neuen *Threads* aufgerufen. Dies hat zur Folge, dass die Berechnung unabhängig von der übrigen Anwendung abläuft.
Die Entkopplung der Berechnung von der übrigen Anwendung bietet zudem den Vorteil, dass die Anwendung während der rechenintensiven Simulation weiterhin uneingeschränkt bedienbar ist.

6.5.2.3 Die Berechnungschleife

Prinzipiell gleicht die Methode *calculation()* im Aufbau dem in Abschnitt 2.4.2 dargestellten Pseudocode. In der Berechnungsschleife werden sukzessive die Methoden *collide()* und *propagate()* nacheinander aufgerufen. Nach der vorgegebenen Anzahl von Zeitschritten wird innerhalb der Schleife das Event *TimeStepsCalculated* ausgelöst. Dieses Event veranlasst die Aktualisierung der Darstellung.
Da der prinzipielle Algorithmus bereits in dem oben genannten Abschnitt besprochen wurde, bleibt eine weitere Erläuterung an dieser Stelle aus.

6.5.3 Die LBVisComputation Typen

Die Klassen des Typs *LBVisComputation* sind die Darstellungsklassen des Namespace *FlowSim.LB*. Ebenso wie die Berechnungsklassen implementieren alle Klassen ein Interface über das sie der Klasse *FlowSimulation* zugänglich gemacht werden. Für jede Berechnungsklasse wird eine Darstellungsklasse vorgehalten, die auf die speziellen Eigenheiten der Berechnungsklasse zugeschnitten ist. Prinzipiell sind sich aber auch die *LBVisComputation* Typen sehr ähnlich.

Alle Klassen dieses Typs bieten differenzierte Methoden zur Darstellung des Strömungsfeldes. So kann zum Beispiel eine Darstellung mit Vektorpfeilen oder Isolinien erfolgen. Der Methodenaufruf bietet zahlreiche Parameter um die Darstellung individuell anzupassen.

7 Die Performance des FlowSim 2004 .NET

7.1 Bedeutung des Berechnungskerns

Es ist bereits im Verlauf der Abhandlung erwähnt worden, dass die numerische Strömungssimulation sehr rechenintensiv ist. Die Anzahl der erforderlichen Knoten des Strömungsfeldes ist in etwa proportional zu Re^3. Folglich ist bei der Entwicklung eines Simulators wie dem FlowSim 2004 .NET besonderer Wert auf performanten Code innerhalb des Berechnungskerns zu legen. Die Ausführungsgeschwindigkeit des Rechenteils steht besonders bei interaktiven Strömungssimulatoren im Mittelpunkt, da diese sich unmittelbar auf Handhabbarkeit der Anwendung auswirkt.

7.2 Das Ziel: Performance unter .NET ?

Eine Frage, der in dieser Arbeit nachgegangen werden soll ist die, ob der Wunsch nach Rechengeschwindigkeit mit der Programmierung unter dem .NET Framework zu vereinbaren ist. Motiviert wird diese Frage dadurch, dass im Allgemeinen angenommen wird, der so genannte *managed Code* könne durch den zusätzlichen Verwaltungsaufwand der *CLR* die Ausführungsgeschwindigkeit eines plattformabhängig compilierten Codes nicht erreichen.
Das .NET Framework bietet jedoch die Möglichkeit, den entstehenden Verwaltungsaufwand zu minimieren. Des weitern besteht die Möglichkeit plattformspezifischen Code innerhalb einer .NET-Anwendung zu verwenden.
Welche Resultate sich daraus ergeben und unter welchem Aufwand dies geschieht wird im Folgenden erläutert.

7.3 Die verschiedenen LB-Berechnungskerne des FlowSim 2004 .NET

Um der Frage aus dem vorangegangenen Abschnitt nachzugehen, steht im FlowSim 2004 .NET eine Auswahl verschiedener Berechnungskerne zur Verfügung. Damit ein abschließender Geschwindigkeitsvergleich erfolgen kann, müssen alle Berechnungskerne denselben strukturellen Aufbau aufweisen und die gleiche Funktionalität bieten. Sie unterscheiden sich allerdings hinsichtlich Implementierungsdetails.

Insgesamt stehen im FlowSim 2004 .NET fünf verschiedene Berechnungsklassen zur Auswahl. In erster Hinsicht werden die Klassen dahingehend unterschieden, ob sie ausschließlich verwalteten Code enthalten, oder anteilig bzw. ausschließlich aus unverwaltetem Code bestehen. Im Fall von verwaltetem, so genanntem *managed Code*, wird weiterhin eine Unterscheidung zwischen sicherem und unsicherem Code vorgenommen. In der Fachsprache werden diese als *unsafe* bzw. *safe* bezeichnet. Weiterhin wird unter Einsatz von sicherem Code die Auswirkung der Verwendung unterschiedlicher Arraytypen genauer betrachtet. Innerhalb des unverwalteten Teils wird ein weiteres mal unterschieden in Klassen, die vollständig aus unverwaltetem Code bestehen und denen, deren Code verwaltet ist, aber auf unverwalteten Daten agiert.

Die

Abbildung 7.1 veranschaulicht die vorausgehend beschriebene Hierarchie der LB-Berechnungskerne.

Die Unterschiede in den Implementierungsdetails und die in diesem Zusammenhang verwendeten Begriffe werden nachfolgend näher erläutert.

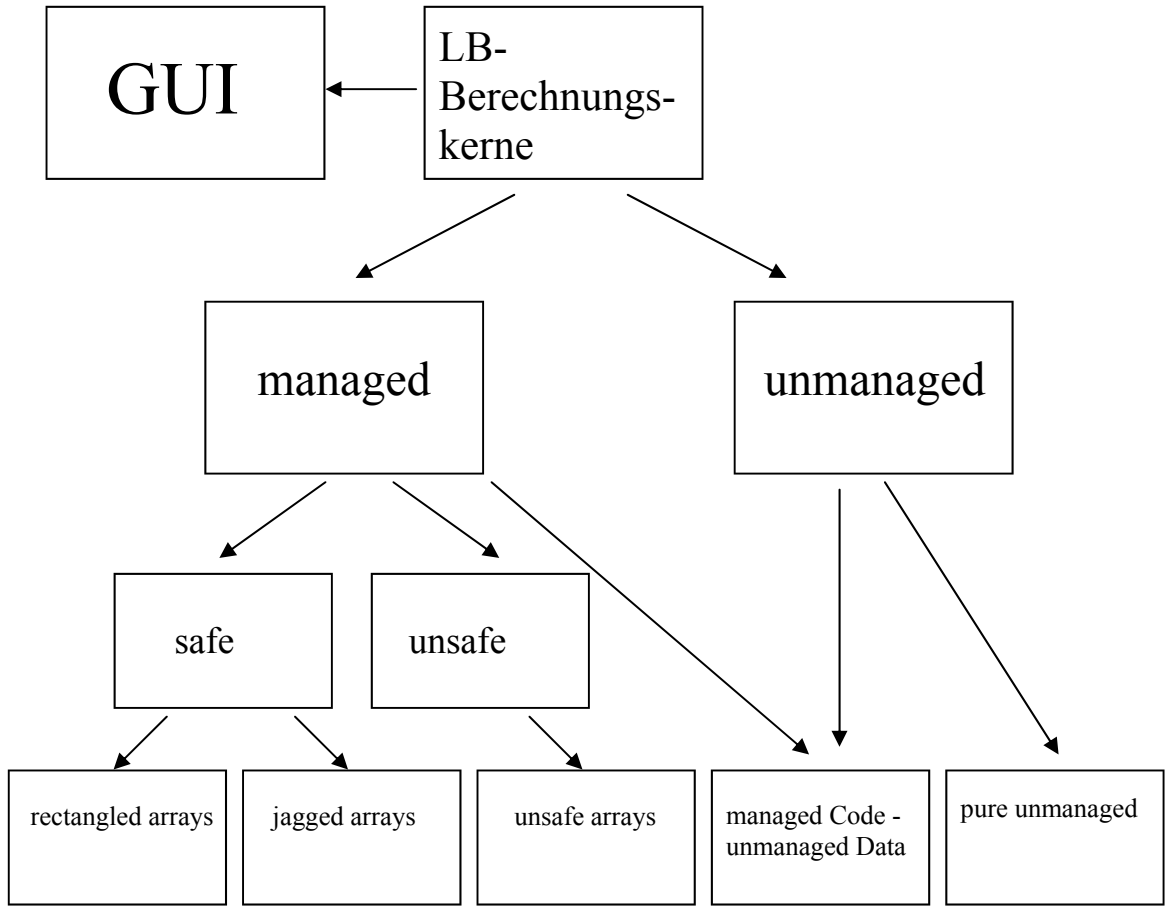

Abbildung 7.1 Hierarchie der LB-Berechnungskerne

7.3.1 managed vs. unmanaged Code

Der Begriff des verwalteten bzw. *managed Code* ist bereits im Kapitel 3 in der Abhandlung über das .NET Framework definiert worden. Es sei noch einmal erwähnt, dass *managed Code* in einer Zwischensprache genauer der *Microsoft Intermediate Language (IL)* vorliegt. Dieser Code wird erst zur Laufzeit von einer Komponente der *CLR* der so genannten *Just-In-Time Compiler* oder auch *JIT-Compiler* in systemeigenen Code umgewandelt. Hieraus resultieren zwei für die Ausführungsgeschwindigkeit relevante Umstände. Zum einen ist es einleuchtend, dass die *Just-in-Time-Compilierung* zusätzlichen Aufwand bedeutet, zum anderen wird vom *JIT-Compiler* Code erzeugt, der auf den jeweiligen Prozessor zugeschnitten ist. D.h. die *JIT-Compilierung* an sich benötigt zusätzliche Rechenzeit, erzeugt allerdings Systemcode, der von dem jeweiligen Prozessor optimal ausgeführt werden kann.

Unmanaged Code hingegen wird schon auf dem Entwurfsrechner in systemeigenen Code compiliert. Im Fall des FlowSim 2004 .NET ist dieser Code vom Compiler zwar auf Ausführungsgeschwindigkeit optimiert, jedoch nicht auf einen speziellen Prozessor zugeschnitten.

Dies sind allerdings nicht die einzigen die Unterschiede, die Einfluss auf die Ausführungsgeschwindigkeit haben. *Managed Code* bietet in Kombination mit der .NET-Laufzeitumgebung einige Besonderheiten, die für die Stabilität der ausgeführten Anwendungen sowie das Entwickeln von verwalteten Anwendungen von besonderer Bedeutung sind. Diese Besonderheiten gereichen der Anwendung durch den zusätzlich entstehenden Aufwand in Bezug auf die Ausführungsgeschwindigkeit zum Nachteil.

In diesem Zusammenhang bietet die .NET Laufzeitumgebung im Gegensatz zu unverwaltetem Code z.B. eine ständige *Typsicherheit*. Diese verhindert beispielsweise, dass während der Ausführung der Anwendung Speicherzugriffe erfolgen, die nicht dem an der Adresse befindlichem Datentyp entsprechen, da ein derartiger Zugriff die Stabilität der Anwendung im Extremfall sogar des gesamten Systems beeinträchtigen könnte.
Ermöglicht wird diese Typsicherheit durch so genannte *Metadaten*, die für jedes Objekt innerhalb eine verwalteten Anwendung existieren.

Das .NET Framework bietet mit dem Schlüsselwort *unsafe* allerdings eine Möglichkeit, die Typsicherheit zu umgehen und einen direkten Zugriff auf den Speicher zu erhalten.

7.3.2 Arrays in .NET

Auf die Verwendung von Arrays wurde an verschiedenen Stellen bereits eingegangen. Hier erfolgt ein detaillierter Blick auf die zwei verwendeten .NET-Arraytypen. Zum einen bietet das Framework die Verwendung so genannter *sequenzieller* oder auch *rectangled* Arrays an. Die Daten sequenzieller Arrays werden nach einem definierten Schema in jeweils zusammenhängenden Blöcken im Speicher abgelegt. Dabei werden sequenzielle Arrays die lediglich lokal definiert und verwendet werden auf dem Stack angelegt.

Der zweite Arraytyp, der im FlowSim Verwendung findet, wird als so genannter *verschachtelter* oder *jagged* Typ bezeichnet. Verschachtelte Arrays werden immer auf dem *verwalteten Heap* angelegt. Der von jagged Arrays belegte Speicherplatz ist dabei nicht zusammenhängend. Vielmehr werden im verwalteten Heap abgelegte Objekte ggf. von der Laufzeitumgebung bewegt, um eine optimierte Ausführungsgeschwindigkeit zu erzielen.

Die Besonderheiten von lokalen Variablen auf dem Stack bzw. Heap wurden bereits Abschnitt 5.2.1 diskutiert. Noch einmal erwähnt sei an dieser Stelle, dass für die Bereitstellung von Speicher auf dem Stack weniger Zeit beansprucht wird, als für Objekte, die auf dem Heap angelegt werden. Andererseits erfolgt der Zugriff auf Objekte auf dem Heap schneller als auf Variablen auf dem Stack. Für die lokale Verwendung von Arrays ist also sinnvoll abzuwägen, welcher Typ an der jeweiligen Stelle zu bevorzugen ist.

Mit dem zusätzlichem Verwaltungsaufwand durch die Typsicherheit geht die Bereichsüberprüfung, auch Range Checking genannt, einher. Das Range Checking gewährleistet, dass bei einem Zugriff auf ein Array dieser nur innerhalb des vom Array eingenommenen Speicherbereichs geschieht. Gerade bei der Verwendung des Indexers innerhalb einer Schleife können sehr schnell derartige Fehler unterlaufen.
Diese aufzuspüren kann unter Umständen sehr zeitaufwändig sein. Unter .NET wird im Fall einer Bereichsüberschreitung eine so genannte *Exception* geworfen, die den Anwender explizit auf den Fehler aufmerksam macht.

Ebenso wie für die Typsicherheit gilt aber auch hier, dass unter Verwendung des Schlüsselwortes *unsafe* in C# die Möglichkeit besteht, das Range Checking zu umgehen.

Ein Auswahlkriterium eines Arraytyps im sicheren Kontext ist die Tatsache, dass die Laufzeitumgebung in der Lage ist, das Range Checking beim Durchlaufen von Arrays mittels *for-Schleifen* zu optimieren. D.h. unter Umständen wird dies unterdrückt. Diese Option gilt allerdings nur für verschachtelte Arrays und nicht für sequenzielle. Hinweise zu geschwindigkeitsoptimierten Verwendung von Arrays im .NET Framework finden sich unter [Wi04].

7.3.3 Unsafe Code

Aufgrund der Speicherverwaltung des .NET Frameworks, in Form der *Garbage Collection* und der Verwendung von Referenzen, sind Zeiger innerhalb des Frameworks nicht mehr notwendig. Dennoch wird auch weiterhin die Verwendung von Zeigern ermöglicht. Deren Einsatz ist zum einen zur Interaktion mit unverwaltetem Code notwendig. Des Weiteren sind Zeiger noch immer die performanteste Lösung, auf den Speicher zuzugreifen.

In C# wird die Verwendung von Zeigern durch das Schlüsselwort *unsafe* möglich. Dabei gewährleistet .NET den Zugriff über Zeiger nicht nur auf unverwaltete, sondern auch auf verwaltete Typen. Es besteht jedoch die Einschränkung, dass diese Wertsemantik besitzen müssen. Die Zeigerarithmetik entspricht in diesem Fall weitestgehend der von C++. Dennoch oder eben deswegen ist der Umgang mit Zeigern in C# für einen C++ Programmierer nicht besonders intuitiv.
Unter Umständen leidet zusätzlich die Lesbarkeit des Quelltextes unter der Verwendung von Zeigern in C#. Detaillierte Informationen zur Verwendung von Zeigern in C# finden sich unter [EC02] und [OBEC02].

Im Abschnitt über Arrays wurde bereits angesprochen, dass die *CLR* den verwalteten Heap durch Umschichtungen in seiner Ausführungsgeschwindigkeit optimiert. Aus diesem Grund muss beim Zugriff auf Objekte, die sich auf dem verwalteten Heap befinden, darauf geachtet werden, dass diese als *fixed* bzw. nicht verschieblich markiert werden. Es ist an dieser Stelle anzumerken, dass der verwaltete Heap unter Umständen in seiner Funktion zur Optimierung beeinträchtigt wird.

Prinzipiell sei an dieser Stelle auch darauf hingewiesen, dass unsicherer Code nicht gleichbedeutend ist mit unverwaltetem Code, wie oftmals angenommen. Unsicher bedeutet in diesem Kontext lediglich, dass die Mechanismen zur Typüberprüfung oder ein eventuelles Range Checking nicht angewendet werden. Der unsichere Code unterliegt dennoch der Verwaltung der *Common Language Runtime* und ist somit *managed Code*.

7.3.4 Unmanaged Code – C++ Managed Extensions

Im FlowSim 2004 .NET werden unverwaltete Programmteile oder solche mit nicht verwalteten Anteilen unter Verwendung der Programmiersprache *C++ Managed Extensions* erstellt. Dabei handelt es sich um die bekannte Programmiersprache C++ mit zusätzlichen Erweiterungen, die es ermöglichen Anwendungen unter Verwendung der .NET Klassenbibliothek zu schreiben. Bei *C++ Managed Extensions* handelt es sich lediglich um eine Erweiterung der bestehenden Sprache. Mit dem aktuellen Compiler können auch weiterhin konventionelle unverwaltete Projekte erstellt werden. Hinweise zur Programmierung mit den *C++ Managed Extensions* finden sich unter [Gr03].

Der Grund, diese Sprache für die Entwicklung von Teilen des FlowSim 2004 .NET einzusetzen ist der, dass es sich bei *C++ Managed Extensions* bei Weitem um die flexibelste Sprache zum Erstellen von verwalteten Anwendungen handelt. So lassen sich in *Managed C++* auf einfache Art verwaltete Anwendungen erstellen, die unverwaltete Anteile haben. D.h. es ist beispielsweise möglich, bestehende unverwaltete Klassen bzw. Bibliotheken in verwalteten Projekten weiter zu verwenden.
Ermöglicht wird dies durch eine Technologie die als *IJW* bzw. *„It just works“* bezeichnet wird.
Bei der gleichzeitigen Verwendung von verwalteten und unverwalteten Typen ist eine besondere Aufmerksamkeit auf die Schnittstellen zwischen diesen Programmteilen zu legen. An diesen Stellen findet unter Umständen ein so genanntes *Marshaling* statt. Dieses ermöglicht eine Umwandlung verwalteter Typen in unverwaltete und umgekehrt.

7.4 Implementierungsdetails der Berechnungskerne

Aus Abschnitt 7.3 ist bereits bekannt, dass im FlowSim 2004 .NET die Auswahlmöglichkeit zwischen fünf unterschiedlichen Berechnungskernen besteht. An dieser Stelle sollen die maßgeblichen Unterschiede der einzelnen Klassen aufgezeigt werden, die sich aus den vorausgehend erläuterten Technologien ergeben.

Die Klassen *LBComputationSafeRect*, *LBComputationSafeJagged* und *LBComputationUnsafeRect* sind in C# implementiert und liegen in compilierter Form als *managed Code* vor.
Die beiden erstgenannten Klassen unterscheiden sich dabei lediglich hinsichtlich der verwendeten Arrays zu Datenhaltung für die notwendigen Matrizen. Im ersten Fall finden sequenzielle Arrays Verwendung, während die Klassen *LBComputationSafeJagged* verschachtelte Arrays verwendet.
Innerhalb der Klassen wird regulär auf die verwendeten Arrays zugegriffen. Die Laufzeitumgebung unterliegt also keiner Einschränkung bezüglich der Typüberprüfung oder der Bereichsüberprüfung in Bezug auf den Arrayzugriff. Der Code beider Klassen wird auch als sicherer Code bezeichnet.

Anders ist es bei der Klasse *LBComputationUnsafeRect*. Innerhalb dieser Klasse wird für den Zugriff auf die Arrays Verwendung der *unsafe* Option gemacht. Wie aus den vorangegangenen Abschnitten bekannt ist, wird so die Laufzeitumgebung hinsichtlich einiger Sicherheitsoptionen umgangen. Der Zugriff auf die Daten der Arrays geschieht innerhalb der Berechnungsklasse mittels Pointerarithmetik. Der Code dieser Klasse wird als „unsicher“ bezeichnet.

Die Klassen *LBComputationMCUnsafe* und *LBComputationMCNative* sind in *Managed C++* geschrieben. Die erstgenannte Klasse ist unter Verwendung von nativen Arrays implementiert. Der Programmcode ist aber, ebenso wie bei den vorangegangenen Klassen, verwalteter Code. Allerdings unterliegt der Zugriff auf die Arrays nicht den üblichen Sicherheitsüberprüfungen der *CLR*, da diese native bzw. unverwaltete Typen sind.
Die Klasse *LBComputationMCNative* ist eine so genannte *Wrapperklasse*. Sie kapselt die unverwalteten Klassen *LBComputationQtLBGK* und *LBComputationQtMRT* in denen die Berechnung durchgeführt wird. Die Wrapperklasse ist lediglich für die Integration der

unverwalteten Klassen in die verwaltete Anwendung zuständing. Die Berechnung erfolgt ausschließlich innerhalb der unverwalteten Klassen unter Verwendung von unverwalteten Datentypen.

7.5 Performancevergleiche

7.5.1 Das Momentenmodell

Neben dem in dieser Abhandlung beschriebenen LBGK-Model wurde in die Berechnungsklassen auch das so genannte Momentenmodell implementiert. Das Momentenmodell unterscheidet sich von der LBGK-Methode durch die Definition der Kollision. Diese berechnet sich nach dem Momentenmodell durch Einsatz eines höheren Rechenaufwandes. Allerdings bietet das Momentenmodell im Allgemeinen eine höhere Stabilität.
Für den Vergleich der Ausführungsgeschwindigkeiten ist an dieser Stelle lediglich der größere Berechnungsaufwand von Interesse.

7.5.2 Durchführung der Tests

Zum Abschluß der Arbeit wurde ein Benchmark der verschiedenen Berechnungskerne durchgeführt. Zu diesem Zweck wurde ein Gebiet mit 30.000 Knoten definiert, das auf verschiedenen Testrechner berechnet wurde. Auf allen Testrechnern wurde jede Variation des Berechnungskerns sowohl unter Verwendung der LBGK-Methode als auch des Momentenmodells bewertet.

Als relevantes Kriterium werden die so genanten *nups* (nodal updates per second) herangezogen. Dieser Wert steht für die Anzahl der in einer Sekunde bearbeiteten berechnungsrelevanten Knoten.

Die Ergebnisse sind in den nachfolgenden Tabellen und Grafiken dargestellt:

Computer	managed Code rectangled Arrays	managed Code jagged Arrays	managed Code unsafe	managed Code unmanaged Data	unmanaged Code unmanaged Data
(CPU / Ram)	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]
Mobile P4 1.6 GHz 768 MB	722	1.055	1.168	1.246	**1.668**
s.o.	721	1.050	1.169	1.245	**1.678**
Athlon XP 2400+ 2.0 Ghz 1,5 GB	1.084	1.467	1.616	1.505	**1.766**
P4 1.7 GHz 1,5 GB DDR	803	1.165	1.286	1.417	**1.955**
Athlon XP 2500+ 1.84 GHz 1,5 GB	914	1.220	1.392	1.276	**1.425**
P4 2.4 GHz 1,0 GB	924	1.100	1.350	1.310	**1.500**
P4 mobile 1.6 GHz 500 MB	680	963	1.077	1.155	**1.530**
Opteron 2.0 Dual 2x 2,75 GB DDR	1.385	2.194	2.227	1.966	**2.699**
P4 2.4 GHz 1,0 GB	929	1.200	1.340	1.315	**1.509**
Athlon XP 1800+ 1.53 GHz 1,5 GB	698	845	980	910	**1.014**

Tabelle 7.1 Benchmark unter Verwendung der LBGK-Methode

Computer	managed Code rectangled Arrays	managed Code jagged Arrays	managed Code unsafe	managed Code unmanaged Data	unmanaged Code unmanaged Data
(CPU / Ram)	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]	x1000 [nups]
Mobile P4 1.6 GHz 768 MB	629	**1.207**	1.005	916	1.159
Athlon XP 2400+ 2.0 Ghz 1,5 GB	967	1.474	1.515	1.389	**1.559**
P4 1.7 GHz 1,5 GB DDR	700	**1.402**	1.145	1.031	1.315
Athlon XP 2500+ 1.84 GHz 1,5 GB	827	1.235	1.375	1.176	**1.325**
P4 2.4 GHz 1,0 GB	825	1.215	1.250	1.061	**1.270**
P4 mobile 1.6 GHz 500 MB	595	**1.159**	960	835	1.065
Opteron 2.0 Dual 2x 2,75 GB	1.267	**2.480**	2.360	2.250	2.340
P4 2.4 GHz 1,0 GB	843	**1.328**	1.240	1.065	1.260
Athlon XP 1800+ 1.53 GHz 1,5 GB	640	850	**935**	840	920

Tabelle 7.2 Benchmark unter Verwendung des Momentenmodells

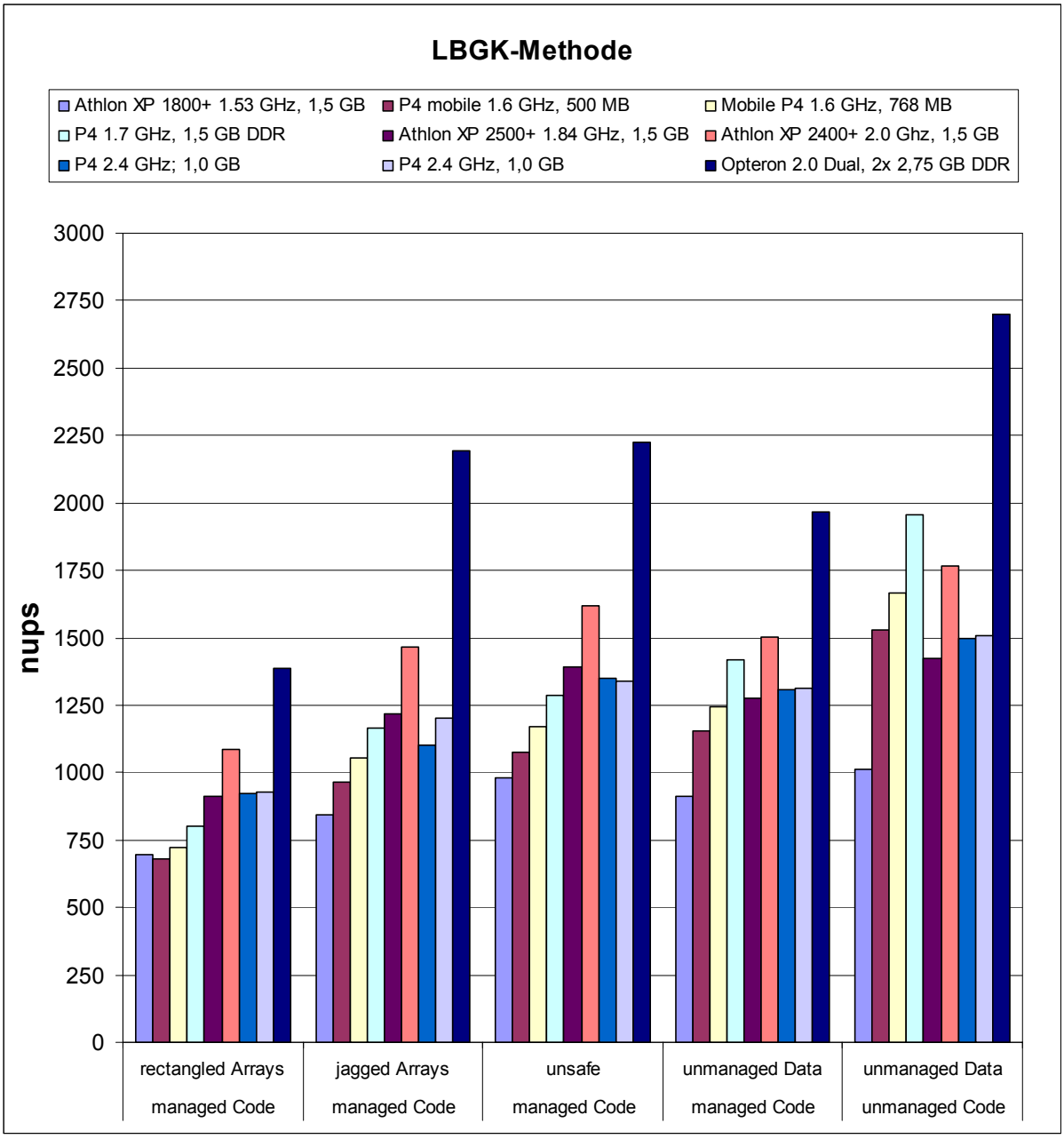

Abbildung 7.2 Benchmark unter Verwendung der LBGK-Methode (Balkendiagramm)

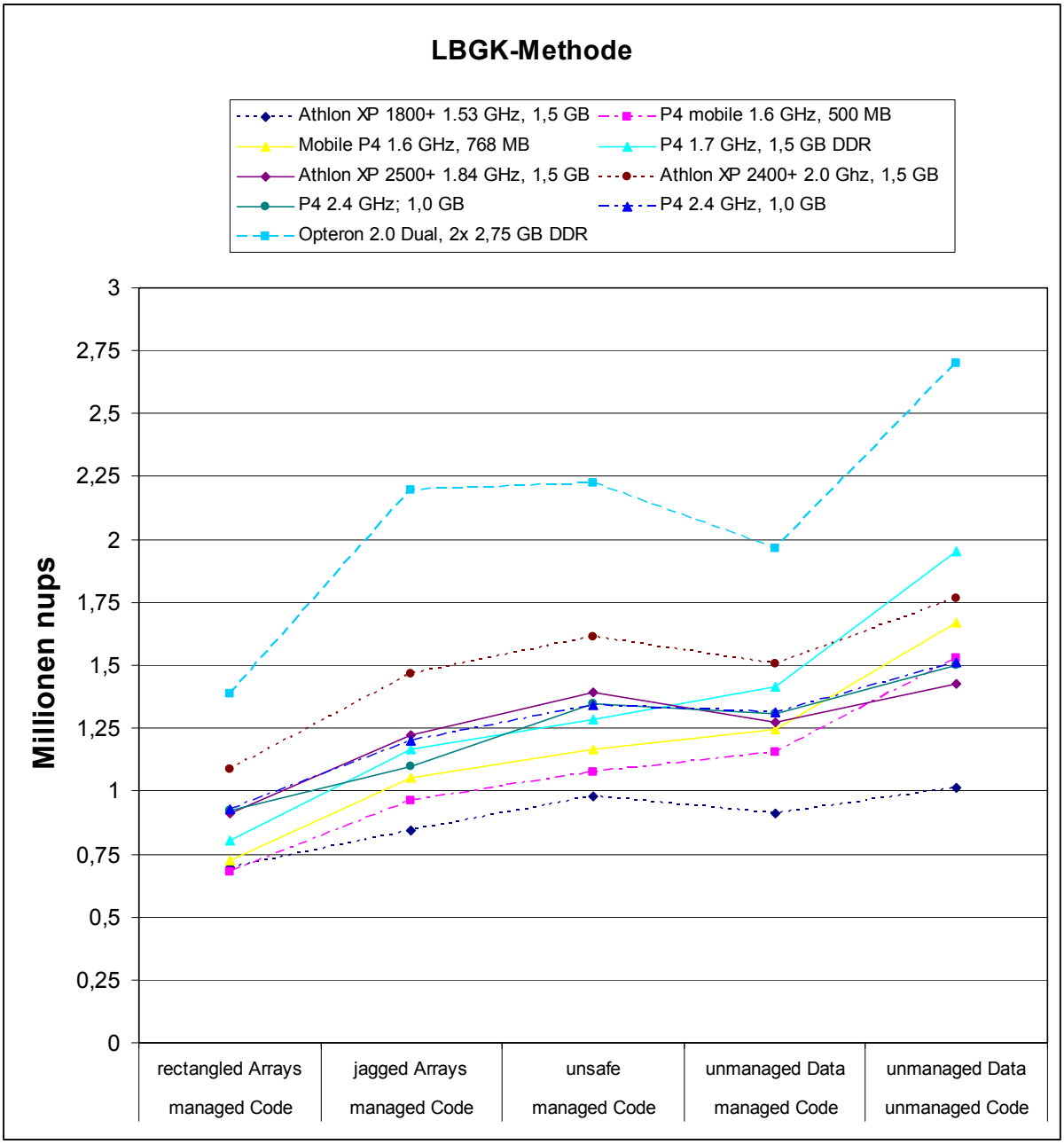

Abbildung 7.3 Benchmark unter Verwendung der LBGK-Methode (Liniendiagramm)

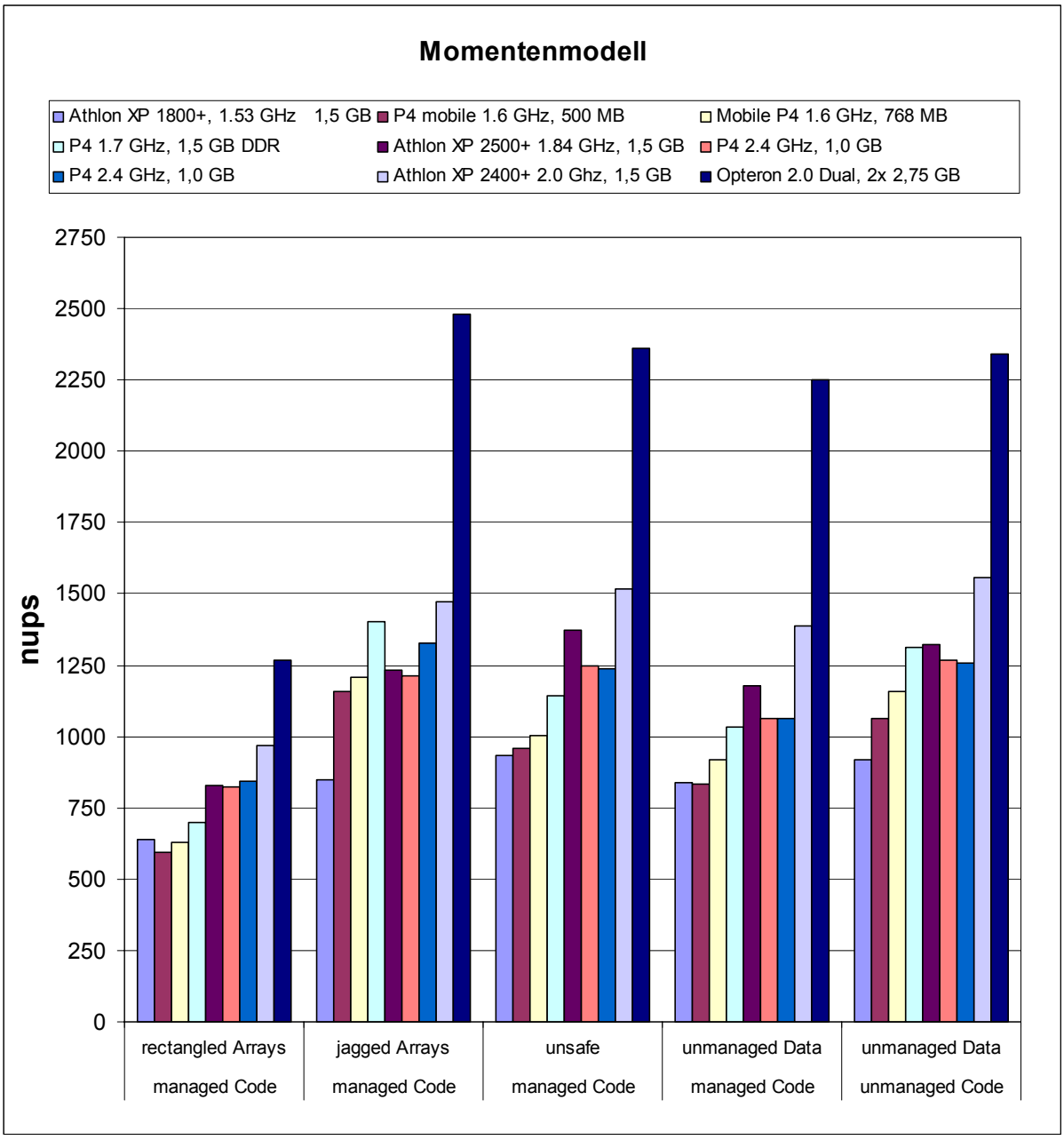

Abbildung 7.4 Benchmark unter Verwendung des Momentenmodells (Balkendiagramm)

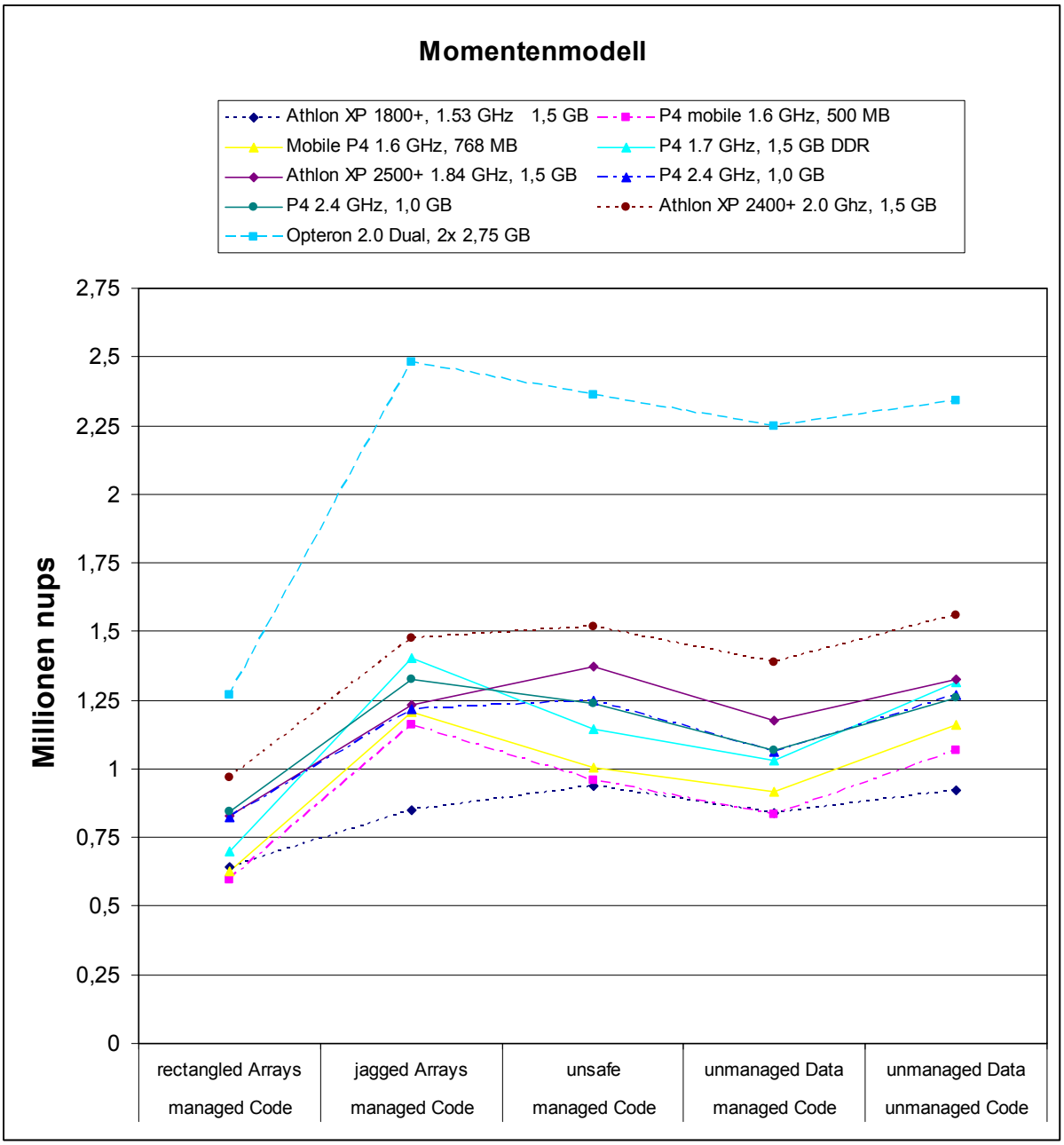

Abbildung 7.5 Benchmark unter Verwendung des Momentenmodells (Liniendiagramm)

7.6 Testergebnisse

7.6.1 LBGK-Methode

Die in Tabelle 7.1 dargestellten Ergebnisse wurden unter Verwendung der LBGK-Methode erzielt. Generell ist aus den Ergebnissen hinsichtlich der Ausführungsgeschwindigkeit zu entnehmen, dass die vollständig unverwalteten Programmteile die besten Ergebnisse erzielen. Die geringste Performance wird bei der LBGK-Methode durch den Einsatz von sicherem, verwaltetem Code erzielt. Dabei ist weiterhin festzustellen, dass Klassen, die Gebrauch von sequenziellen Arrays machen, deutlich ineffizienter sind, als diejenigen, die verschachtelte Arrays verwenden. Die Differenz liegt im Durchschnitt bei über 25%. Im mittleren Bereich liegen die erzielten Ergebnisse der verwalteten Klassen mit unsicherem Code bzw. unverwalteten Daten sehr dicht beieinander. In diesem Fall ist die darunter liegende Rechnerarchitektur vermutlich ursächlich dafür, welcher Code schneller ausgeführt wird. Es ist teilweise von Bedeutung welcher Prozessortyp oder welche Art von Arbeitsspeicher in dem jeweiligen System vorhanden ist.

Deutlicher fällt allerdings der Unterschied in der Ausführungsgeschwindigkeit zwischen verwalteten und rein unverwaltetem Code aus. Hier liegt der Geschwindigkeitsvorteil des *unmanaged* Berechnungskerns gegenüber dem schnellsten *managed* Kern bei ca. 19% im Mittel.

7.6.2 Momentenmodell

Die erzielten Ergebnisse für Kalkulationen mit dem Momentenmodell sind in Tabelle 7.2 aufgelistet. Im direkten Vergleich zu den Ergebnissen aus Tabelle 7.1 ist hier, in der überwiegenden Zahl der Testfälle die erzielte Ausführungsgeschwindigkeit des Momentenmodells geringer als die der LBGK-Methode. Eine Ausnahme bildet der sichere, verwaltete Code mit Verwendung von verschachtelten Arrays.
Desweitern fällt auf, dass der unverwaltete Code nicht mehr uneingeschränkt der performanteste ist. Vielmehr ist von Bedeutung, dass auch hier der eben hervorgehobene sichere, verwaltete Code annähernd die gleichen Ergebnisse erzielt wie der unverwaltete Code. Letzten Endes scheint auch in diesem Fall die zugrunde liegende Architektur den Ausschlag dafür zu geben, welcher Code auf der jeweiligen Maschine schneller ausgeführt wird.

Am unteren Ende der Skala findet sich wiederholt der verwaltete Code mit Verwendung von sequenziellen Arrays, wobei die Differenz zu den verschachtelten Arrays mit durchschnittlich ca. 40% noch deutlicher ausfällt.
Weiterhin sehr dicht beieinander liegen der unsichere verwaltete Kern und der verwaltete Kern mit unverwalteten Anteilen. Allerdings erzielt in allen Testfällen erstgenannter unsicherer Berechnungskern minimal bessere Ergebnisse.

7.7 Diskussion der Ergebnisse

Ein auffälliges Merkmal beider Tests ist der Geschwindigkeitsunterschied, der einzig aus der Wahl des Arraytyps resultiert. Wie bereits erwähnt, liegt zwischen den verwalteten Klassen die sequenzielle Arrays verwenden und denen die verschachtelte verwenden in einigen Fällen ein Performanceunterschied von ca. 40%. Dieser ist mit der in Abschnitt 7.3.2 erläuterten Optimierung der Laufzeitumgebung zu begründen. Wie angesprochen verfügt die *CLR* über interne Optimierungsmöglichkeiten, die im Fall von verschachtelten Arrays die Option bieten, das Range Checking unter bestimmten Gegebenheiten zu unterlassen.

Weiterhin ist aus beiden Tests ersichtlich, dass die unsichere Berechnungsklasse, die mittels Pointerarithmetik auf die Arrays zugreift nahezu die gleiche Ausführungsgeschwindigkeit aufweist, wie die verwaltete *Managed C++* Klasse, die auf unverwalteten Arrays agiert. Zur Erklärung lässt sich sagen, dass dieses Ergebnis vorhersehbar war, denn beide Klassen arbeiten auf ähnliche Weise. Es werden in beiden Klassen Pointer verwendet um die Objekte im Speicher zu adressieren. Dabei wird beim Zugriff auf den Speicher in beiden Fällen weder ein Range Checking noch eine Typüberprüfung vorgenommen. Außerdem ist in beiden Fällen der Programmcode verwalteter Code. Der einzige Unterschied ist der, dass im Fall der C# Klasse auf Daten im verwalteten Heap zugegriffen wird, während die C++ Klasse auf Daten des unverwalteten Heaps zugreift. Hierbei kann es unter Umständen zu Marshalingvorgängen kommen, die zu Geschwindigkeitseinbußen führen.

Wie bereits erwähnt, erfordert die Simulation mit der LBGK-Methode im Vergleich zum Momentenmodell etwas weniger Rechenaufwand innerhalb der Kollision, woraus der allgemeine Unterschied in den Testergebnissen beider Methoden resultiert.

Deutlich wird außerdem, dass die Ausführungsgeschwindigkeit von nativem bzw. unverwaltetem Code im Allgemeinen die von verwaltetem Code übertrifft. Diese Aussage wird zumindest von den Testergebnissen unter Verwendung der LBGK-Methode bestätigt.

Aus den Testergebnissen der Berechnung mittels Momentenmodell wird jedoch deutlich, dass auch verwalteter Code zu sehr performantem Ergebnissen führen kann, der den unverwalteten Code wie in diesem Fall in der Ausführungsgeschwindigkeit sogar übertrifft. Zurückzuführen ist dieses Ergebnis darauf, dass die .NET Laufzeitumgebung, wie bereits erörtert, Programmcode generiert, der auf den jeweiligen Prozessor zugeschnitten ist. Zusätzlich bietet die *CLR* interne Optimierungsmöglichkeiten die zur Laufzeit stattfinden.

Die Simulation unter Verwendung des Momentenmodell ist deutlich von der Durchführung von Rechenoperationen geprägt, während im Programmcode der LBGK-Methode Speicherbewegungen im Vordergrund stehen. In diesem Fall bietet das Momentenmodell durch das Mehr an Rechenoperationen eine größere Möglichkeit zur Optimierung, die wiederum in einer gesteigerten Ausführungsgeschwindigkeit resultiert. Eingriffe des Programmierers in die Funktionsweise der *CLR*, beispielsweise durch das Fixieren von Speicher im Heap in Verbindung mit einem unsicheren Kontext, können diese Optimierung allerdings verhindern und schränken unter Umständen die Ausführungsgeschwindigkeit ein.
Aus diesen Sachverhalten erschließt sich, die höhere Performance des sicheren, verwalteten Berechnungskerns im Vergleich zu den übrigen verwalteten Berechnungsklassen.

8 Zusammenfassung und Ausblick

8.1 Fazit

Aus den vorangegangenen Kapiteln ist ersichtlich, dass ein Einsatz des .NET Framework auch im Bereich des *High Performance Computing* möglich ist. Das .NET Framework stellt dem Entwickler zahlreiche Möglichkeiten zur Verfügung, um die Ausführungsgeschwindigkeit der Anwendung zu optimieren. An dieser Stelle kann jedoch keine allgemein gültige Aussage darüber getroffen werden, welche Optimierungen die besten Ausführungsgeschwindigkeiten zur Folge haben. Aus den Ergebnissen ist jedoch ersichtlich, dass bereits regulärer C# -Code ohne besonderen Optimierungsaufwand in bestimmten Fällen sehr hohe Ausführungsgeschwindigkeiten erzielen kann. Es geht weiterhin hervor, dass die Berechnungsklassen, die unsicheren verwalteten Code verwenden, die gleichen Ergebnisse erzielen wie die *Managed C++* Klassen, die verwalteten Code und unverwaltete Daten verwenden. Aus der Tatsache, dass die Programmierung von unsicheren, verwalteten Klassen mittels C# und Pointerarithmetik sehr aufwendig und unübersichtlich ist, ist daher von einem derartigen Vorgehen abzuraten. Vielmehr haben die Erfahrungen gezeigt, dass stattdessen der Einsatz von *Managed C++* zu sehr flexiblen und ebenso performanten Lösungen führen kann. Die verwalteten Erweiterungen der Programmiersprache C++ bieten nicht zuletzt durch die große Flexibilität die besten Voraussetzungen um *High Performance Computing* unter .NET zu betreiben. Mit den *Managed Extensions* ist es möglich unverwaltete Klassen unter .NET zu verwenden. Somit können perfomancerelevante Programmteile unverwaltet implementiert werden, um so die Ausführungsgeschwindigkeit von nativen Applikationen zu erzielen. Weiterhin bietet dies die Möglichkeit vorhandene C++ Lösungen auf einfache Art in neue Anwendungen zu integrieren.

Aus den Erfahrungen, die während der Entwicklung des FlowSim 2004 .NET gesammelt werden konnten lässt sich im Allgemeinen sagen, dass das .NET Framework zusammen mit dem *Microsoft Visual Studio .NET 2003* eine sehr komfortable Entwicklungsplattform bildet. Mit der besprochenen .NET Klassenbibliothek steht dem Entwickler für den Anwendungsentwurf eine große Zahl an Komponenten zur Verfügung. Durch den vollständig objektorientierten Aufbau der Klassenbibliothek wird ein einheitlicher Entwurf ermöglicht.

Das Visual Studio bietet in diesem Zusammenhang mit zahlreichen Funktionen wie zum Beispiel dem *Rapid Application Development (RAD)* oder dem *Intellisense* einen hohen Entwicklungskomfort.

8.2 Die Zunkunft des FlowSim 2004 .NET

Die Funktionalität des FlowSim 2004 .NET ist auf Grund des zeitlich begrenzten Rahmens einer Studienarbeit auf die wesentlichsten Funktionen beschränkt. Durch den Modularen Aufbau der Anwendung ist eine einfache Erweiterbarkeit allerdings gewährleistet. Die vorhandene Anwendung bietet somit eine Basis für weitere Projekte.
Wie bereits in Kapitel 3 besprochen bietet das .NET Framework eine Vielzahl neuer Technologien, die in der bestehenden Anwendung nur anteilig eingesetzt werden.

8.2.1 .NET Remoting

Es wurden bereits die Einsatzmöglichkeiten von .NET im Rahmen von verteilten Anwendungen erwähnt. Mit *.NET Remoting* ist es möglich, Prozessraumgrenzen zu überschreiten, dies kann auch über Rechnergrenzen hinweg geschehen.

In Abbildung 8.1 ist die Funktionsweise des *.NET Remoting* für einen bestimmten Anwendungsfall dargestellt. Die mit dem .NET Framework eingeführte *Application Domain* kann mit einem logischen Prozess verglichen werden. Dabei ist es möglich, dass ein Win32-Prozess mehrere *Application Domains* enthält. Der Zugriff auf Objekte über die Grenzen von *Application Domains* hinweg wird von der *.NET Remoting Infrastructure* ermöglicht. Der clientseitige Zugriff auf Remoteobjekte erfolgt in diesem Fall über zwei Proxyobjekte. Namentlich sind dies der so genannte *Transparent Proxy* sowie der *Real Proxy*. Dabei kann auf das *TransparentProxy*-Objekt direkt von dem Client zugegriffen werden. Dieser Zugriff unterscheidet sich aus Sicht des Clients nicht von einem Zugriff auf ein lokales Objekt. Wenn ein clientseitiger Zugriff auf das *TransparentProxy*-Objekt z.B. in Form eines Methodenaufrufs stattfindet, so wird dieser Aufruf in eine Nachricht verpackt und an das *RealProxy*-Objekt weitergereicht. Unter Verwendung *.NET Remoting Infastructure* wird die Nachricht letztlich von dem *Real Proxy* an das tatsächliche Remoteobjekt übermittelt. Detaillierte Informationen zur Programmierung mit *.NET Remoting* finden sich unter [Ra02].

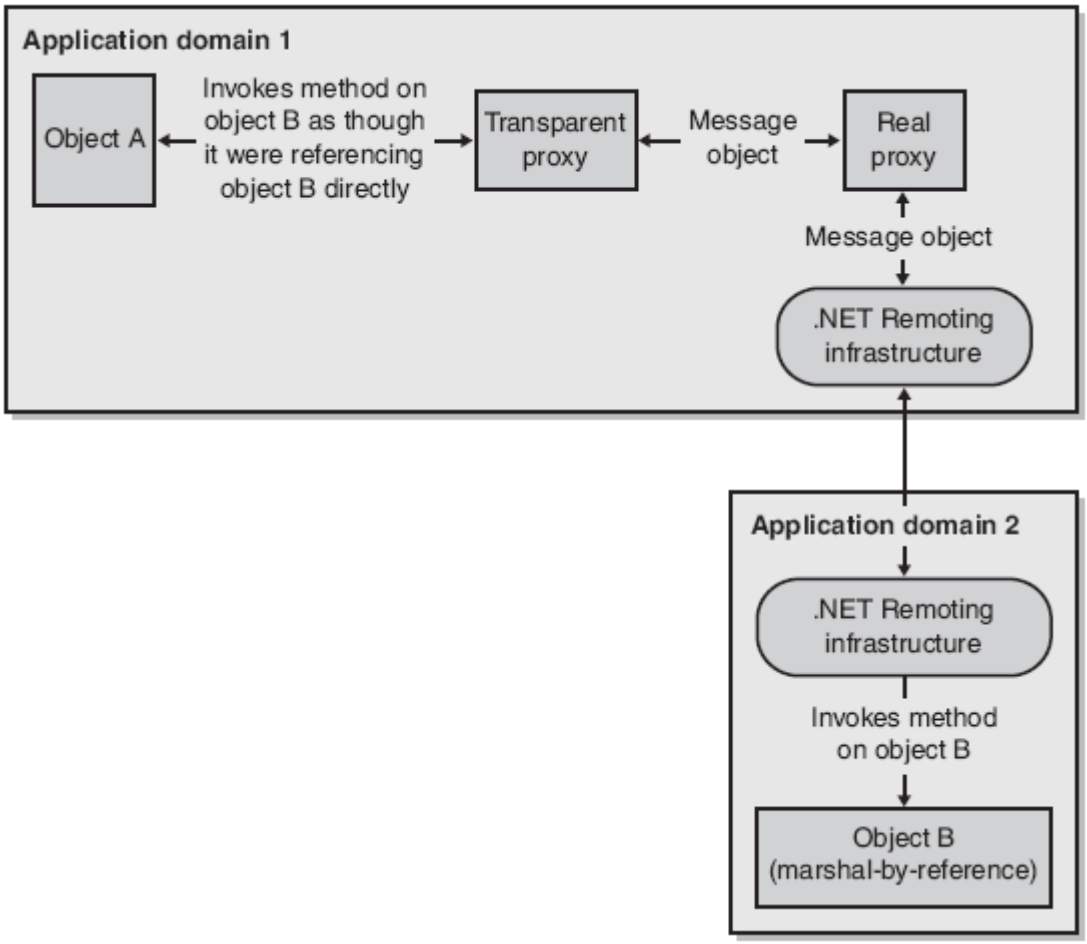

Abbildung 8.1 Funktionsweise von .NET Remoting

Die derzeitige Version des FlowSim 2004 .NET enthält bereits einen Ansatz die Anwendung verteilt ausführen zu können. Dabei wurde die Berechnungsschicht in einen eigenen Prozess ausgelagert und kann somit auch auf einem separaten Rechner ausgeführt werden. Zu diesem Zweck wurde der FlowServer 2004 .NET implementiert, der die Berechnungsfunktionalität anderen Anwendungen zur Verfügung stellt.

Das FlowSim Objektmodell wurde um den Namespace *FlowSim.LB.Remoting* erweitert. Dieser Namespace enthält zusätzliche Berechnungsklassen, die Gebrauch von *.NET Remoting* machen. Wenn der FlowSim 2004 .NET als verteilte Anwendung ausgeführt wird, so wird beim Starten der Berechnung von der Klasse *LBComputationRemoteFacade* eine auf TCP basierende Verbindung zu einem FlowServer aufgebaut. Daraufhin wird von der Klasse *LBComputationRemoteFacade* eine Instanz der Klasse *LBComputationRemote* erzeugt. Diese wird allerdings im Prozessraum des FlowServers instantiiert. Innerhalb des FlowSim wird von

der Laufzeitumgebung ein Proxyobjekt erstellt, über das Zugriffe auf das Remoteobjekt erfolgen können.
Derzeit ist lediglich die verteilte Berechnung unter Verwendung von rechteckigen Arrays möglich. Es kann sowohl eine Berechnung mittels LBGK-Methode als auch mit dem Momentenmodell erfolgen.
Über die erzielten Ergebnisse kann derzeit noch keine qualitative Aussage gemacht werden. Erste Tests haben allerdings gezeigt, dass die zu erwartenden Ergebnisse auf den Remotesystemen bislang etwa 10% unter den Ausführungsgeschwindigkeiten liegen, die aus lokalen Berechnungen resultieren.

8.2.2 Managed DirectX 9

Bislang erfolgt die Darstellung im FlowSim 2004 .NET ausschließlich zweidimensional. Das .NET Framework bietet allerdings zusammen mit *Managed DirectX 9* die Möglichkeit zur 3D-Darstellung. Die Darstellung erfolgt in der derzeitigen Version auf dem bekannten *FlowPanel*. Die Modularität des FlowSim 2004 .NET ermöglicht es, dieses Steuerelement um dreidimensionale Funktionalität zu erweitern.

8.2.3 OpenMP 2.0 Support

Die nächste Version des .NET Frameworks soll die Unterstützung des OpenMP 2.0 Standards für Managed C++ beinhalten. Somit wird es möglich verwaltete Anwendungen für Shared-Memory-Multiprozessorsysteme zu entwickeln.

8.2.4 Das FlowSim 2004 .NET Objektmodell

Einen Ansatz den FlowSim 2004 .NET in einem folgenden Projekt zu erweitern bzw. zu modifizieren bietet das FlowSim 2004 .NET Objektmodell. Das bestehende Objektmodell ist unter subjektiven Ansichten des Autors entworfen worden und bietet an vielen Stellen Möglichkeiten zur Erweiterung und Optimierung.

9 Anhang

9.1 Inhalt der CD-ROM

Auf der beiliegenden CD-ROM befinden sich die Installationsdateien des FlowSim 2004 .NET. Zudem ist auf der CD die Redistributable Version des Microsoft .NET Framewok 1.1 enthalten.

9.2 Informationen im Internet

Auf der Webseite *http://www.bau-ings.de/flowsim/* sind aktuelle Informationen zum FlowSim 2004 .NET zu finden.

[BGK] P. L. Bhatnagar, E. P. Gross and M. Krook, *Physical Review*, 94:511, 1954

[LWK] W. Li, X. Wei, A. Kaufman, *Implementing Lattice Boltzmann Computation on Graphics Hardware*

[BRFU] G. Bella, N. Rossi, *Using OpenMP on a Hydrodynamic Lattice-Boltzmann Code*

[TKS] C. van Treeck, M. Krafczyk, M. Schulz, *Lattice-Boltzmann Verfahren im Bauwesen – Tutorial eines grafischen-interaktiven Strömungssimulators in 2D*

[LaLu] P. Lallemand, L.-S. Luo, *Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invarianc, and stability,* 2000

[Fe92] W. D. Fellner, *Computergrafik*, 142-149, 1992

[Ca02] David Chappel, *.NET verstehen – Einführung und Analyse*, 2002

[MaFr02] A. Maslo, J. M. Freiberger, *.NET Framework Developers's Guide*, 2002

[ElKo03] F. Eller, M. Kofler, *Visual C#, Grundlagen, Programmiertechniken, Windows-Programmierung*, 2003

[SHB02] A. Schäpers, R. Huttary, D. Bremes, *C# Kompendium*, 2002

[Kue03] A. Kühnel, *Visual C#*, 2003

[Ri02] J. Richter, *Microsoft .NET Framework Programmierung*, 2002

[FoSc00] M. Fowler, K. Scott, *UML konzentriert*, 2000

[EC02] Standard ECMA-334, *C# Language Specification,* 2002

[OBEC02] L. O´Brien, B. Eckel, *Thinking in C#, 2002*

[Gr03] R. Grimes, *Programming with Managed Extensions for Visual C++ .NET*, 2003

[Wi04] N. Wienholt, *Maximizing .NET Performance*, 2004

[Ra02] I. Rammer, *Advanced .NET Remoting*, 2002