



## C# - Microsoft schlägt neue Töne an

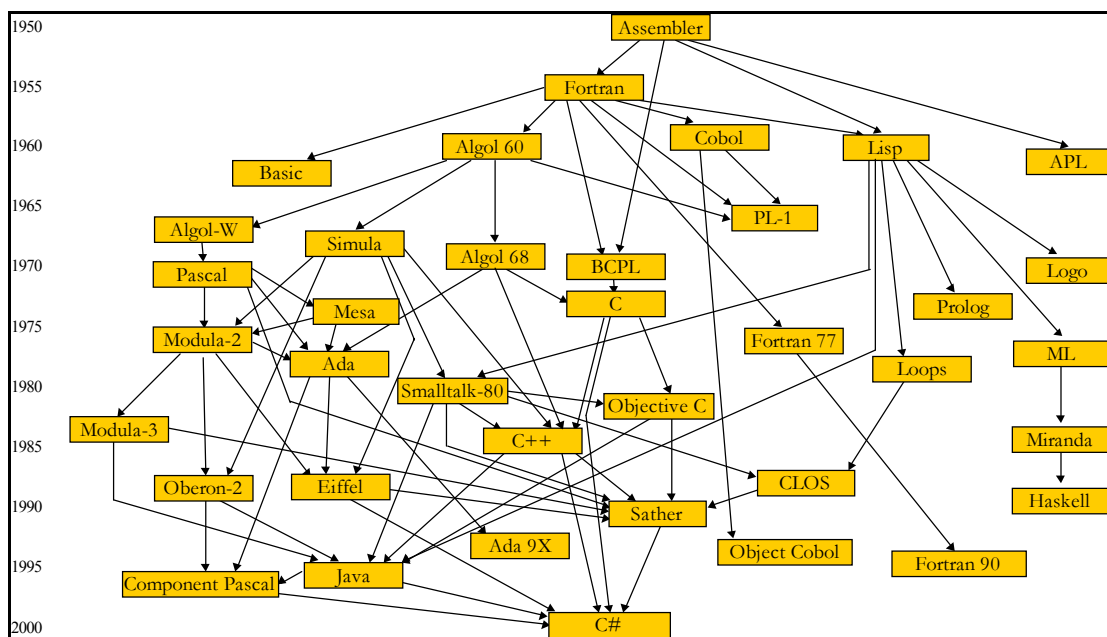
Dominik Gruntz, FH Aargau, [d.gruntz@fh-aargau.ch](mailto:d.gruntz@fh-aargau.ch)

Microsoft hat am 26. Juni im Zusammenhang mit ihrer .NET Initiative die neue Programmiersprache C# ("C sharp" – oder auf Deutsch "cis") vorgestellt. C# wird als einfache, moderne, objektorientierte und typsichere Sprache präsentiert, welche die Einfachheit von Visual Basic und die Mächtigkeit und Effizienz von C++ in sich vereinigen soll. Dieser Artikel vermittelt einen ersten Überblick über C#.

Eine neue Programmiersprache greift immer die Konzepte und Erfahrungen existierender Programmiersprachen auf. Microsoft deklariert C# als Nachfolger von C und C++, doch es ist offensichtlich, dass insbesondere auch Java die Programmiersprache C# massgeblich beeinflusst hat, aber auch andere Sprachen wie Eiffel, Sather und Component Pascal (siehe Figur 1: Entwicklung der Programmiersprachen).

Hauptdesigner von C# ist Anders Hejlsberg [Hej00], der vor seinem Wechsel zu Microsoft 15 Jahre lang bei der Firma Borland gearbeitet hat und dort an der Entwicklung von Turbo Pascal und Delphi wesentlich beteiligt war. Bei Microsoft hat er in der Visual-J++-Gruppe gearbeitet und ist für gewisse Spracherweiterungen (wie z.B. die J++ Delegates) im Java-Dialekt von Microsoft verantwortlich.

Bevor nun ein genauerer Blick auf C# geworfen wird, soll aufgezeigt werden, was für eine Bedeutung C# innerhalb der Dot-Net (.NET) Strategie von Microsoft hat, die alle Produkte zu "Web-Software" umfunktionieren will. Anschliessend wird C# im Kontext von Java und von C++ vorgestellt.



Figur 1: Entwicklung der Programmiersprachen

### C# im Kontext von .NET

C# ist eine der Programmiersprachen, die verwendet werden können, um Dienste für das .NET Framework zu programmieren. Das .NET Framework soll es ermöglichen, Komponenten und web-basierte Applikationen schneller und einfacher als mit herkömmlichen Ansätzen herzustellen. Kern des .NET Frameworks ist das *Common Language Runtime (CLR)*, das Komponenten laden und ausführen kann und eine Grundinfrastruktur mit Speicherverwaltung, Threading- und Sicherheitsunterstützung bereitstellt. Die Ideen des CLR gehen auf das ursprüngliche COM+ zurück, so wie es im September 1997 als Erweiterung des Component Object Models (COM) vorgestellt wurde [Cha97], der Name

COM+ wurde dann allerdings für die Technologien rund um den Microsoft Transaction Server (MTS) verwendet.

Das CLR lädt Komponenten, die in der *Microsoft Intermediate Language (MSIL)* vorliegen, und übersetzt diese zur Installationszeit, zur Ladezeit oder bei der ersten Verwendung in Maschinencode der jeweiligen Zielpattform. MSIL ist vergleichbar mit Java-Byte-Code, ist also auch ein mit Typinformation angereicherter Assembler Code.

Das .NET-Framework stellt zudem Bibliotheken zur Verfügung die von allen Komponenten verwendet werden können, auch über Sprachgrenzen hinweg. Die *Base Class Library* enthält Klassen für Multithreading, verteiltes Rechnen, Sicherheit, Input-Output und Kollektionen (analog zu den Java Paketen *java.lang*, *java.io* und *java.util*). Die *Win Forms* und *Web Forms* Bibliotheken erlauben es, grafische Benutzungsoberflächen im .NET Framework zu bauen, wobei *Win Forms* die Win32 Schnittstellen kapselt und *Web Forms* die Definition von Webschnittstellen unterstützt. Mit *Web Forms* kann man browserbasierte Benutzerschnittstellen auf die gleiche Art erstellen, wie bis anhin grafische Benutzungsschnittstellen mit Visual Basic, d. h. durch Drag & Drop von Controls in einem Formulareditor.

Das .NET Framework ist bezüglich Programmiersprachen offen und definiert mit der *Common Language Infrastructure (CLI)* eine für alle .NET-fähigen Programmiersprachen gültige Basis. Diese enthält die Spezifikationen der MSIL, der Grunddatentypen, einer XML-basierten deklarativen Syntax, um Typinformationen zu definieren, sowie eine kleine Menge von Basisklassen. Die CLI kann von verschiedenen Programmiersprachen aus genutzt werden. An der diesjährigen *Programmers Developers Conference (PDC)* sind neben Visual Basic, Managed C++ (ANSI konforme CLI Erweiterung von C++) und C# auch bereits weitere Sprachen vorgestellt worden, die an das .NET Framework angepasst wurden. Es sind dies neben anderen COBOL, Eiffel#, Lightning Oberon, Perl, Haskell und Smalltalk. Die .NET Plattform ermöglicht somit sprachübergreifende Softwareentwicklung und überwindet die Babylonische Verwirrung der Programmiersprachen so wie diese in Figur 1 dargestellt sind (wobei diese Grafik noch lange nicht vollständig ist!).

### C# im Kontext von Java

C# ist wie Java eine sichere Sprache. C# ist typsicher, d. h. man kann keine ungesicherten Konvertierungen ausführen und Variantenrecords (Unions) sind ebenfalls aus der Sprache verbannt. Sicherheit bezüglich Speicherverwaltung garantiert ein Garbage-Collector und der Programmierer kann nicht mehr mit Adressen arbeiten (ausser in speziell als *unsafe* markierten Bereichen). Die Sprache stellt zudem sicher, dass alle Variablen initialisiert sind; bei Feldern durch Initialisierung mit Null-Werten und bei lokalen Variablen durch "definite assignments", d. h. alle lokalen Variablen müssen vor der ersten Verwendung initialisiert werden. Zur Laufzeit werden zudem Zugriffe über Null-Referenzen und Zugriffe auf Felder mit ungültigen Indexwerten abgefangen.

Wie Java unterstützt C# nur noch Einfachvererbung und erlaubt ebenfalls, dass eine Klasse beliebig viele Interfaces implementieren kann. C#-Interfaces enthalten wie in Java nur Signaturen und unterstützen Mehrfachvererbung.

C# kennt wie Java keine globalen Prozeduren und Variablen. Diese müssen als statische Methoden bzw. als statische Felder in Klassen deklariert werden. Eine statische *Main* Methode definiert den Startpunkt für Applikationen. Folgendes Programm zeigt, wie mit C# "Hello World" auf die Konsole geschrieben werden kann.

```
class Hello {
    static void Main(){
        System.Console.WriteLine("Hello World");
    }
}
```

Wie in Java werden Typinformationen als Metainformation abgelegt, auf die, im Gegensatz zu Java, auch von anderen Programmiersprachen aus zugegriffen werden kann. Deklarationen von Schnittstellen in Headerdateien gehören damit der Vergangenheit an. Auf die Typinformation kann auch in C# mit Hilfe von Reflection zugegriffen werden.

## C# im Kontext von C/C++

Obwohl C# offensichtlich sehr viele Konzepte von Java übernommen hat, bleiben seine C++ Wurzeln doch unverkennbar. C# erlaubt es z. B. weiterhin, im Programmcode Präprozessordirektiven anzugeben für bedingte Übersetzung, Ausgabe von Warnungen, Definition von Symbolen, etc. (`#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, `#warning`, `#error`). Die Definition von Makros wird hingegen nicht mehr unterstützt und auch Templates kennt C# nicht mehr.

Aus C++ übernommen wurde aber die Möglichkeit, statische Strukturen zu definieren, also Typen, die nicht auf dem Heap, sondern statisch auf dem Programmstack oder "in line" in Objekten angelegt werden. Im Vergleich zu Objekten wird dabei der benötigte Speicherplatz nicht von der Speicherverwaltung bereitgestellt und Strukturen brauchen auch nicht vom Garbage-Collector weggeräumt werden, was zu viel effizienterem Code führen kann.

Auch bei mehrdimensionalen Feldern hat man in C# die Möglichkeit, diese effizient als zusammenhängenden Speicherblock zu erzeugen und muss nicht – wie in Java – das Feld aus mehreren eindimensionalen Feldern zusammensetzen.

Ein weiteres Feature von C#, das viele C++ Programmierer lieb gewonnen haben, ist die Möglichkeit Operatoren zu überladen. C# schränkt dieses Sprachmerkmal so ein, dass subtile C++-Operator-Programmierfallen eliminiert werden, ohne aber die Mächtigkeit einzuschränken.

## C# Kompakt

Im folgenden werden die Sprachkonzepte von C# genauer vorgestellt. Zur Verdeutlichung dieser Konzepte wird in Listing 1 eine Klasse für die Datenstruktur Stack und in Listing 2 eine Struktur für komplexe Zahlen vorgestellt.

### *Einfache Datentypen*

Wie in Java sind die Größen der Grunddatentypen in der Sprache definiert, was die Portabilität von Programmen erleichtert. Wie in C/C++ wird bei den Ganzzahltypen zwischen vorzeichenlosen und vorzeichenbehafteten Typen unterschieden. Die Grunddatentypen sind *bool*, *char* (16 Bit, Unicode), *sbyte* und *byte* (8 Bit), *short* und *ushort* (16 Bit), *int* und *uint* (32 Bit), *long* und *ulong* (64 Bit), *float* und *double* (IEEE754) sowie *decimal* (28 Dezimalstellen). Zwischen dem Datentyp *bool* und anderen Datentypen existieren keine Standardkonvertierungen.

### *Klassen*

C#-Klassen enthalten Felder und Methoden und werden analog wie Klassen in C++ und Java definiert. Klassen können als nicht instanzierbar (*abstract*) oder nicht erweiterbar (*sealed*) deklariert werden. Felder und Methoden können als statisch (*static*) deklariert und so der Klasse zugeordnet werden.

Felder können zudem als *readonly* deklariert werden, was der Deklaration von *final* Feldern in Java (seit Version 1.1) entspricht, d. h. sie können nur in der Deklaration oder in einem Konstruktor initialisiert und danach nur noch gelesen werden. C# kennt neben den read-only Feldern auch noch echte Konstanten, die bereits vom Compiler aufgelöst werden. Diese sind allerdings auf die Grunddatentypen beschränkt.

Methoden müssen in C# (leider) als virtuell deklariert werden, wenn man erzwingen will, dass die Methodenaufrufe dynamisch gebunden werden. Ohne *virtual* Deklaration werden Methoden statisch gebunden. Methoden können überladen und überschrieben werden. Wenn eine Methode aus einer Basisklasse überschrieben wird, muss die überschreibende Methode explizit mit dem Schlüsselwort *override* markiert werden.

Neben den Methoden können in einer Klasse auch Konstruktoren definiert werden. Diese werden verwendet, um Instanzen bei der Erzeugung oder Klassen beim Laden zu initialisieren. Konstruktoren sind Methoden ohne Resultattyp, die denselben Namen haben wie die Klasse. Der Klassenkonstruktor wird als statisch deklariert und hat keine Parameter. Der Instanzkonstruktor kann mit beliebigen Signaturen überladen werden. Aus einem Instanzkonstruktor heraus kann ein anderer Konstruktor

derselben Klasse oder der Basisklasse aufgerufen werden. C# bedient sich dabei der Syntax von C++. Neben den Konstruktoren kann auch ein Destruktor definiert werden, der aufgerufen wird, wenn das Objekt vom Garbage-Collector rezykliert wird. Destrukturen können nicht überschrieben werden und werden automatisch aufgerufen. Destrukturen sind per Definition immer virtuell (im Gegensatz zu den Destrukturen in C++). Im folgenden Beispiel wird eine Klasse *Rectangle* als Erweiterung einer Klasse *Figure* mit zwei Konstruktoren und einem Destruktor definiert.

```
class Rectangle : Figure {
    int w, h;
    static int nOfRects = 0;

    Rectangle(int x, int y, int w, int h) : base(x, y) {this.w = w; this.h = h; nOfRects ++;}
    Rectangle(int w, int h) : this(0, 0, w, h) { };
    ~Rectangle() { nOfRects--; }
    ...
}
```

In einer C#-Klasse können auch lokale Klassen deklariert werden. Dabei handelt es sich jedoch lediglich um eine Einbettung einer Klassendefinition in den Gültigkeitsbereich einer anderen Klasse (analog zu C++) und nicht um Instanzklassen mit einer impliziten Referenz auf die umgebende Klasse wie in Java. Die geschachtelten C#-Klassen entsprechen damit den als statisch deklarierten inneren Klassen in Java.

Als Beispiel einer Klassendefinition wird in Listing 1 ein Stack mit den Methoden *Push*, *Pop* und *Empty* definiert. Konstruktoren und Destrukturen werden nicht benötigt. Ohne explizite Angabe eines Konstruktors wird ein Defaultkonstruktor generiert. Der Stack wird mit Hilfe einer linearen Liste implementiert, wobei für die einzelnen Elemente der linearen Liste eine lokale Klasse *Node* mit Konstruktor definiert wird.

Listing 1: *Definition einer Klasse Stack*

```
class Stack {
    private Node root = null;

    // Methoden
    public void Push(object x){
        root = new Node(x, root);
    }
    public object Pop(){
        if(Empty()) throw new Exception("Empty stack.");
        object top = root.value;
        root = root.next;
        return top;
    }
    public bool Empty(){
        return root == null;
    }

    // Indexer
    public object this[int index]{
        get {
            Node t = root;
            while( index > 0 ){t = t.next; index--;}
            return t.value;
        }
    }

    // local class
    private class Node {
        internal Node next;
        internal object value;
        internal Node(object value, Node next){this.value = value; this.next = next;}
    }
}
```

### Interfaces

In C# wurde wie in Java das Konzept der Interfaces in die Sprache aufgenommen. Ein Interface definiert einen Vertrag und enthält nur Signaturen und keine Implementierungen, insbesondere auch keine Deklarationen von Feldern. Ein Interface kann beliebig viele andere Interfaces erweitern und Interfaces können in Klassen implementiert werden. Der Typ einer Variablen kann ebenfalls ein Interface sein. Eine solche Variable kann auf eine Klasse verweisen, die das entsprechende Interface (oder eine Erweiterung davon) implementiert.

Im Unterschied zu Java ist C# in einem wesentlichen Punkt erweitert worden. Falls man in einer Klasse zwei Interfaces implementieren möchte, die beide eine Methode mit demselben Namen und derselben Signatur enthalten, so kann man in Java nur eine Implementierung bereitstellen, die beiden Methoden gerecht werden muss. Bereits in der Sprache Eiffel ist dieses Problem erkannt und gelöst worden. In C# können nun Methoden, die aus verschiedenen Interfaces geerbt werden, unterschiedlich implementiert werden. Dazu muss bei der Methodendeklaration auch der Name des Interfaces angegeben werden. Diese Methoden können allerdings nur aufgerufen werden, wenn die Klasse in den entsprechenden Interfacetyp konvertiert wird, z.B. durch Zuweisung an eine entsprechende Interfacevariable oder mit einer expliziten Typkonvertierung.

```
using System;
interface Figure {
    void Draw();           // draw figure
    void Move(int dx, int dy); // move figure
}
interface Cowboy {
    void Draw();           // draw pistol
    void Shoot();          // shoot
}
class LuckyLuke : Figure, Cowboy { // a figure in a cowboy simulation game
    int x, y;
    void Cowboy.Draw(){Console.WriteLine("Lucky Luke draws pistol");}
    void Cowboy.Shoot(){Console.WriteLine("peng");}
    void Figure.Draw(){ Console.WriteLine("draws Lucky Luke at (x,y)"); }
    void Move(int dx, int dy){x += dx; y += dy; ((Figure)this).Draw(); }
}
```

### Strukturen

In C# hat man die Möglichkeit, neben Klassen, deren Instanzen immer auf dem Heap erzeugt werden, auch C++-Strukturen (Structs) zu definieren. Im Gegensatz zu C++ unterstützen C#-Strukturen keine Vererbung; sie besitzen jedoch Konstruktoren und können Interfaces implementieren. Als kleines Beispiel ist in Listing 2 die Definition einer Struktur für komplexe Zahlen angegeben.

Neue Instanzen von Strukturen werden mit dem Operator *new* initialisiert. Das Schlüsselwort *new* mag verwirren, denn es wird bei dieser Definition kein Speicher erzeugt, sondern nur der Konstruktor aufgerufen, der das Objekt initialisiert.

### Aufzählungstypen

C# kennt Aufzählungstypen. Diese werden wie Klassen definiert, die nur Ganzzahlkonstanten enthalten. Die Werte für die Elemente des Aufzählungstyps können explizit oder implizit zugeordnet werden und optional kann auch der Grunddatentyp der Elemente spezifiziert werden. Im folgenden Beispiel hat die Farbe *Red* den Wert 0, *Green* den Wert 1 und *Blue* als auch *Max* den Wert 2.

```
enum Color: short {
    Red, Green, Blue,
    Max = Blue
}
```

Aufzählungstypen sind eigene Typen. Um eine Konstante eines Aufzählungstyps als Ganzzahlausdruck zu verwenden, muss eine explizite Typkonvertierung eingefügt werden.

### Operatoren

C# erlaubt das Überladen von Operatoren. Überladen werden können die unären Operatoren +, -, !, ~, ++ und -- (wobei bei letzteren nicht zwischen pre- und post-Versionen unterschieden wird), die binären Operatoren +, -, \*, /, %, &, |, ^, << und >> und die Vergleichsoperatoren. Es ist jedoch nicht

möglich, den Zuweisungsoperator zu überladen, und anstelle eines überladbaren Index-Operators stellt C# sogenannte Indexer zur Verfügung. Beim Überladen eines binären Operators wird implizit der entsprechende Zuweisungsoperator definiert (+=, -=, etc.) und gewisse Operatoren dürfen nur paarweise definiert werden: == und !=, < und > sowie <= und >=. Damit wird in C# versucht, unsinnige und verwirrende Operatordefinitionen zu reduzieren. In Listing 2 werden in der Klasse Complex die Operatoren +, - und \* sowie die Vergleichsoperatoren == und != überschrieben.

Neben diesen Operatoren können in einer Klasse oder Struktur auch Konversionsoperatoren definiert werden, wobei explizit spezifiziert werden kann, ob diese implizit anwendbar sind oder nur explizit in entsprechenden Konvertierungsoperationen.

Listing 2: Definition einer Struktur für komplexe Zahlen

```

struct Complex {
    private double re, im;

    // Konstruktoren
    public Complex(double re, double im){this.re = re; this.im = im;}
    public Complex(double re) : this(re, 0.0) {}

    // Properties
    public double Re {
        get { return re; }
        set { re = value; }
    }
    public double Im {
        get { return im; }
        set { im = value; }
    }

    public double Abs {
        get { return System.Math.Sqrt(re*re + im*im); }
    }

    // Operatoren
    public static Complex operator+ (Complex x, Complex y){
        return new Complex(x.re + y.re, x.im + y.im);
    }
    public static Complex operator- (Complex x, Complex y){
        return new Complex(x.re - y.re, x.im - y.im);
    }
    public static Complex operator* (Complex x, Complex y){
        return new Complex(x.re*y.re - x.im*y.im, x.re*y.im + x.im*y.re);
    }
    public static bool operator == (Complex x, Complex y){
        return x.re == y.re && x.im == y.im;
    }
    public static bool operator != (Complex x, Complex y){
        return !(x == y);
    }

    // Konversionsoperator double -> Complex
    public static implicit operator Complex(double x){
        return new Complex(x);
    }

    // Redefinition der aus object geerbten Methode ToString
    public override string ToString(){
        return re + " + " + im + "i";
    }
}

```

### Unifiziertes Typsystem

Das Typsystem in C# ist unifiziert, d. h. die Klasse *object* ist die Basisklasse aller Objekte und alle Objekte können als Instanzen der Klasse *object* betrachtet werden – das gilt insbesondere auch für Werttypen, also für alle Grunddatentypen und Strukturen. Falls eine Instanz eines Werttyps einer

Referenz von Typ *object* zugewiesen wird (z.B. wenn ein Objekt in einer Sammlungsklasse abgelegt wird), so wird automatisch ein Wrapper erzeugt, der das Objekt enthält ("boxing").

```
class Test {
    static void Main(){
        Stack s = new Stack();           // reference object
        Complex c1 = new Complex(2, 3);   // struct object
        s.Push(c1);                       // implicit boxing
        Complex c2 = (Complex) (s.Pop()); // implicit unboxing
        Console.WriteLine("c = {0}", c2); // implicit boxing
    }
}
```

In obigem Beispiel wird sowohl beim Ablegen des Elementes auf dem Stack (Listing 1) als auch bei der Ausgabe automatisch ein Objektwrapper erzeugt, der eine Instanz der Struktur *Complex* (Listing 2) enthält. Bei der Ausgabe wird dann über den Wrapper die Methode *ToString* aufgerufen, um die komplexe Zahl in einen String umzuwandeln.

### Parameterdeklaration

Neben Wertparametern unterstützt C# auch Referenz- (*ref*) und Out-Parameter (*out*). Bei Referenz- und Out-Parametern wird keine Kopie, sondern nur eine Referenz auf den aktuellen Parameter übergeben. Out-Parameter werden in einer Methode als uninitialized betrachtet und müssen zwingend definiert werden.

Da Methoden überladen werden können und da der Parameterübergabemechanismus mit zur Signatur gehört, muss beim Aufruf einer Methode angegeben werden, ob ein Parameter per Wert oder per Referenz übergeben wird. Dies hat auch den Vorteil, dass man beim Lesen von Programmcode erkennt, ob ein Parameter per Referenz oder nur per Wert übergeben wird.

```
class Test {
    static void Swap(ref int x, ref int y){
        int t = x; x = y; y = t;
    }

    Static void Main(){
        int i = 1, j = 2;
        Swap(ref i, ref j); // Angabe des Übergabemechanismus beim Aufruf
        Console.WriteLine("i = {0}, j = {1}", i, j); // erzeugt Ausgabe i =2, j = 1
    }
}
```

Es ist auch möglich, einen formalen Parameter zu definieren, der mit beliebig vielen aktuellen Parametern aufgerufen werden kann (*params* Parameter). Die Methode *Console.WriteLine* in obigem Beispiel enthält so einen Parameter. Die aktuellen Parameter sind in der Methode über ein Array zugreifbar.

### Namespaces

Um Namen aus verschiedenen Bibliotheken unterscheiden zu können, bietet C# wie C++ Namensbereiche (*namespaces*) an. Hierbei handelt es sich um eigene Gültigkeitsbereiche, in denen man Definitionen platzieren kann. Namen aus einem Gültigkeitsbereich sind in anderen Gültigkeitsbereichen nicht sichtbar, können aber mit expliziter Qualifikation zugegriffen werden. Namensbereiche können geschachtelt werden und sie können auch jederzeit durch neue Definitionen erweitert werden. Mit der *using*-Deklaration können einzelne oder alle in einem Namensbereich definierten Objekte in einem anderen Namensbereich verfügbar gemacht werden. Es ist dabei auch möglich, Namen in einer *using* Deklaration umzudefinieren (alias Definitionen). Im Gegensatz zu den Java Packages haben C# Namensräume keinerlei Zugriffskontrollfunktion.

### Assembly

Ein Assembly ist eine Menge von Definitionen (Klassen, Interfaces etc.), aus denen eine Komponente ("unit of deployment") produziert wird (konkret eine DLL- oder eine EXE-Datei ). Diese Komponenten können in anderen Projekten verwendet werden. C#-Assemblies sind vergleichbar mit Modulen aus Modula oder Packages aus Ada, denn auf der Stufe Assembly werden spezielle Zugriffsrechte definiert.

### Zugriffsrechte

C# unterscheidet (wie auch Java in der Version 0.9) folgende fünf Zugriffsrechte:

public	Zugriff von überall her möglich
protected	Zugriff nur innerhalb der Klasse und in Ableitungen der Klasse, nicht jedoch ausserhalb der Klassenhierarchie möglich
internal	Zugriff nur innerhalb derselben Komponente (Assembly) möglich
protected internal	Zugriff sowohl innerhalb der Komponente als auch in Unterklassen möglich
private	Zugriff nur innerhalb der Klasse möglich

Typen, die in einem C#-Assembly definiert werden, können als *public* oder *internal* deklariert werden; standardmässig sind sie als *internal* deklariert. Einem Element, das innerhalb einer Klasse oder einer Struktur definiert wird, kann eines der fünf Zugriffsrechte (bei Strukturen ohne *protected*) zugewiesen werden. Standardmässig ist das *private*! Elemente, die innerhalb eines Interfaces oder Aufzählungstyps definiert werden, sind immer *public*.

### Exceptions

Auch die Ausnahmebehandlung ist in der Sprache C# definiert, d. h. es gibt eine try-catch-finally Anweisung und die Möglichkeit, Ausnahmen zu definieren und zu werfen. Im Gegensatz zu Java gehören die Ausnahmen jedoch nicht zur Signatur einer Methode, d. h. man sieht einer Methode nicht an, welche Ausnahmen geworfen werden, und der Compiler kann nicht sicherstellen, ob geworfene Ausnahmen von den Klienten auch abgefangen werden.

### Anweisungen

C# unterstützt alle von C/C++ und Java her bekannten Anweisungen. Interessant ist hingegen, dass auch hier die Sprache so definiert wurde, dass die häufigsten „Fehler“ nicht mehr auftreten können. Bei einer switch-Anweisung ist z.B. kein „fall-through“ mehr möglich, d. h. dass jeder switch-Zweig mit einer break-, einer return- oder einer goto-Anweisung abgeschlossen werden muss (man kann dabei auch explizit zum nächsten switch-Zweig springen). Bei der goto-Anweisung sind keine Sprünge in Blöcke hinein erlaubt, und ein finally-Block darf keine return-Anweisung enthalten (die eine return-Anweisung im try-Block überdecken würde).

Neu ist hingegen die *foreach*-Schleife: *foreach(T x in e){ ... }*, mit der leicht über eine Sammlungsklasse iteriert werden kann. Eine Sammlungsklasse ist definiert als eine Klasse, die eine Methode mit der Signatur *E GetEnumerator()* anbietet, wobei *E* eine Methode *bool MoveNext()* und eine Eigenschaft *Current* vom Typ *T* enthalten muss. Dies ist eine der wenigen Stellen in C#, an der eine Namenskonvention definiert wird. Das ist allerdings nicht problematisch, da die Einhaltung dieser Konvention vom Compiler sichergestellt wird.

### Kontrollierte Ganzzahloperationen

Ganzzahloperationen können in C# kontrolliert (*checked*) oder unkontrolliert (*unchecked*) ausgeführt werden. Bei einer kontrollierten Ausführung werden bei Überläufen Ausnahmen geworfen, bei einer unkontrollierten keine. Die Anweisung *checked(x \* y)* mit  $x = y = 1000000$  ergibt z.B. einen Laufzeitfehler und nicht  $-727379968$ .

### Multithreading

C# definiert eine *lock*-Anweisung, mit der gegenseitiger Ausschluss realisiert werden kann. Als Argument muss eine Referenz auf ein Objekt übergeben werden, bezüglich dessen synchronisiert wird. Weitergehende Unterstützung von Multithreading ist in den .NET Bibliotheken zu finden.

### Delegates

Das Konzept der Delegates entspricht jenem der Prozedurvariablen in C++ und anderen Sprachen. Im Gegensatz zu diesem kann ein C#-Delegate jedoch sowohl auf statische Methoden als auch auf Instanzmethoden verweisen. Dabei wird nicht nur eine Referenz auf die Methode, sondern auch eine Referenz auf das Objekt abgelegt, über das die Methode aufzurufen ist. Zur Deklaration eines Delegates gehört nur die Angabe einer Methodensignatur. Instanziiert wird ein Delegate mit einer Referenz auf eine Methode, die der deklarierten Delegatesignatur entspricht. Verwendet werden Delegates hauptsächlich bei der Definition von call-back Methoden welche in Objekten der Benutzungsoberfläche registriert werden. Ein Beispiel folgt im Abschnitt über Events.

Das Konzept der Delegates wurde aus Visual J++ übernommen. C#-Delegates mit Resultattyp *void* entsprechen den J++-Multicast-Delegates (mit dem Additionsoperator können C#-Delegates zu Multicast-Delegates verknüpft werden). Sun hatte sich auch überlegt, Delegates in Java einzuführen, sich dann aber für das Konzept der Elementklassen entschieden. Das Konzept der Delegates von C# ist jedoch viel leichtgewichtiger.

### Properties

Neben Feldern können in C#-Klassen und Strukturen auch sog. Eigenschaften (Properties) definiert werden. Diese werden wie normale Felder benutzt, der Zugriff wird jedoch auf *set*- und *get*-Methodenblöcke abgebildet. Der Programmierer hat damit freie Wahl, wie er eine Eigenschaft realisieren will. Er kann den Eigenschaftswert in einem Feld ablegen oder aber diesen bei jeder Abfrage neu berechnen. Zudem können beim Setzen einer Eigenschaft weitere Aktionen ausgelöst werden (wie z.B. die Aktualisierung einer grafischen Darstellung). Der wichtigste Vorteil ist jedoch, dass Eigenschaften auch im Zusammenhang mit dem Decorator-Entwurfsmuster (vgl. [Gam95]) verwendet werden können, z.B. um über Rechengrenzen hinweg auf Eigenschaften zuzugreifen.

In Listing 2 wird der Zugriff auf Real- und Imaginärteil der komplexen Zahl mit Eigenschaften realisiert. Dazu muss ein *set*- und ein *get*-Block implementiert werden. Im *set*-Block steht der neue Wert im impliziten Parameter mit dem Namen *value* zur Verfügung. Wird der *set*- oder *get*-Block weggelassen, so erhält man eine nur lesbare oder nur schreibbare Eigenschaft. Die Eigenschaft *Abs* der Klasse *Complex* ist ein Beispiel einer nur lesbaren Eigenschaft. Eigenschaften können auch in Schnittstellen vorgegeben werden. Auf Real- und Imaginärteil der Struktur *Complex* kann nun zugegriffen werden, wie wenn es normale Felder wären:

```
Complex c = new Complex(1, 2);
c.Re = 2.3;      // Aufruf der set-Methode von Re
c.Im += 1.0;    // Aufruf der get- und set-Methoden von Im
```

Properties gibt es bereits in COM und wurden mit der JavaBeans Spezifikation so richtig populär. In JavaBeans beruht die Definition von Properties jedoch vollständig auf Namenskonventionen – alle Methoden, die mit *set*, *get* oder *is* starten, werden von entsprechenden Werkzeugen automatisch als Properties erkannt – auch eine Methode *geteilt* (Eigenschaft *eilt*) oder *getrennt* (Eigenschaft *rennt*). Sprachunterstützung wird hier Klarheit schaffen.

### Indexers

Indexer erlauben es, Objekte wie Felder zu indizieren. Ein Indexer wird wie eine Eigenschaft deklariert, mit dem Unterschied, dass als Name der Eigenschaft das Schlüsselwort *this* verwendet wird (es wird ja das Objekt selbst indiziert) und die Parameter in eckigen Klammern spezifiziert werden. Ein Indexer kann mit verschiedenen Index-Parameter-Signaturen überladen werden. Die Implementierung eines Indexers enthält wie die Eigenschaften einen *get*- und einen *set*-Block. In Listing 1 ist ein Indexer definiert, der den direkten (nur lesenden) Zugriff auf die Elemente eines Stacks erlaubt.

### Events

Ebenfalls in die Sprache aufgenommen wurde die Definition von Events. Wenn eine Klasse einen Event definiert, so zeigt sie damit an, dass sie Ereignismeldungen verschicken kann. Klienten können bei einem Event-Handler Methoden registrieren, welche beim Auftreten eines Ereignisses aufgerufen werden. Man kann sich die Deklaration eines Events so vorstellen, als ob damit implizit zwei Methoden *add\_X* und *remove\_X* definiert würden. Der Typ eines Events ist immer ein Delegate. Registriert werden neue Event-Handler mit dem Operator *+=*, und ein installierter Event-Handler kann mit dem Operator *-=* entfernt werden. Dies sind die einzigen Zugriffsmöglichkeiten, welche ein Klient auf einen Event hat. Das folgende Beispiel definiert eine Klasse *Button* mit einem *Click*-Event. In der Klasse *Form* wird dann der Event-Handler *OnClick* registriert.

```

public delegate void EventHandler (object sender, Event e);

public class Button {
    public event EventHandler Click;
    protected void NotifyClick(Event e){
        if (Click != null) Click(this, e);
    }
}

public class Form {
    public Form(){
        Button b = new Button();
        b.Click += new EventHandler(OnClick);
    }
    private void OnClick(object sender, Event e){
        Console.WriteLine("Button clicked " + e);
    }
}

```

### *Attribute*

Neu in C# ist die Möglichkeit, Definitionen mit benutzerdefinierten Attributen zu versehen. Interessant ist dabei, dass auf diese Attribute auch über den Reflection Mechanismus von C# zugegriffen werden kann. Damit wird auch Meta-Information erweiterbar. Verwendet werden die Attribute von Microsoft um die Interoperabilität mit COM zu ermöglichen. Eine Klasse kann z.B. mit einem COM-GUID Attribut versehen werden.

### *Refactoring-Support*

Ein wichtiger Aspekt von C# ist, dass Code Refactoring unterstützt wird (im Sprachreport unter der Bezeichnung "Versioning"). Unter Refactoring versteht man Änderungen im Design einer Klassenbibliothek wie z.B. Umbenennen, Hinzufügen oder Entfernen von Methoden. Bei diesem Vorgang kann es passieren, dass man z.B. eine Methode in einer Basisabstraktion entfernt oder umbenennet, diese Änderung aber nicht in allen Erweiterungen nachführt. Dies kann zu schwer zu findenden Fehlern führen. Aus diesem Grund wird in bestehenden Klassenbibliotheken leider viel zu wenig aufgeräumt. C# unterstützt nun diesen Vorgang explizit bereits in der Sprache mit den *new* und *override* Modifiern und orientiert sich dabei an Erfahrungen die mit den Programmiersprachen Eiffel [Mey92] und Component Pascal [Omi97] und teilweise mit Clascal und Object Pascal [Tes85] gemacht wurden.

Wenn in einer Klasse eine Methode aus der Basisklasse überschrieben wird, muss dies explizit mit dem Schlüsselwort *overrides* markiert werden. Wenn nun die Methode aus der Basisklasse entfernt wird und die abgeleitete Klasse neu übersetzt wird, so erkennt der Compiler, dass die überschriebene Methode nicht mehr vorhanden ist und offensichtlich ein Fehler vorliegt. In Java würde die Methode in der abgeleiteten Klasse einfach als neue Methode interpretiert werden. Wenn in einer Basisklasse eine neue Methode eingeführt wird, deren Signatur bereits in einer abgeleiteten Klasse verwendet wird, wird dieses Problem beim Übersetzen der abgeleiteten Klasse ebenfalls erkannt.

Ein ähnliches Problem entsteht, wenn man in einer Klasse eine neue Methode definiert, deren Signatur bereits in der Basisklasse definiert ist. In diesem Fall gibt C# eine Warnung aus, die eliminiert werden kann, indem die Methode explizit mit dem Modifier *override* (die Methode in der Basisklasse soll überschrieben werden) oder mit dem Modifier *new* (die Methode ist eine neue Methode) markiert wird.

Was in C# jedoch fehlt ist die Möglichkeit, eine Methode als nicht überschreibbar zu deklarieren, denn der *sealed* Modifier darf nur auf Klassen, nicht jedoch auf Methoden angewendet werden. Es ist zu hoffen, dass dieser Fehler korrigiert wird, denn der Sprachreport liegt ja erst in Version 0.17b vor.

## Fazit

Microsoft hat mit C# eine sehr interessante Sprache vorgestellt, welche viele Schwachpunkte von C++ und von Java behebt. Als Beispiel zwei weitere Stolperfallen aus C++/Java die dem Autor auch schon einige Nerven gekostet haben und in C# nun korrigiert sind: Methodennamen dürfen in C# nicht mit dem Klassennamen übereinstimmen und es ist daher nicht mehr möglich, aus Versehen (nur weil man *void* vor den Konstruktor schreibt) eine Methode anstelle eines Konstruktors zu definieren. Auch die Initialisierungssemantik wurde gegenüber Java so geändert dass die Variableninitialisierer einer Klasse vor dem Basisklassenkonstruktor aufgerufen werden. Dies verhindert, dass über eine virtuelle Methode auf noch nicht von Variableninitialisierern initialisierte Werte zugegriffen wird (vgl. Praxis 68 in [Hag00]). Man spürt, dass in diese Sprache sehr viele Erfahrungen eingeflossen sind, und es wäre zu wünschen, wenn diese Sprache eine tragende Rolle in den kommenden Jahren spielen würde.

Auch wenn man sich über den Sinn einer neuen Sprache streiten kann, so hat C# zumindest klar gemacht, dass die Entwicklung von Programmiersprachen weitergeht und dass Java nicht der Weisheit letzter Schluss ist. Wer dies geglaubt hat, der hat nichts aus der Geschichte gelernt. Dass die neue Sprache allerdings bereits zum heutigen Zeitpunkt kommt, mag doch etwas überraschen.

Vielleicht hat Microsoft diese Sprache definiert, weil es sich mit J++ in eine Sackgasse manövriert hat, denn die Weiterentwicklung von Java à la Microsoft wurde mit der Klage von Sun microsystems 1997 gestoppt. Microsoft hat inzwischen die Entwicklung von J++ eingestellt und das Produkt der Firma Rational übergeben. Mit C# hat Microsoft nun eine eigene Variante von Java. C# wird in Visual Studio Release 7 (Visual Studio .NET) verfügbar sein, ob auch J++ noch enthalten sein wird ist nicht klar. Eigentlich kann Java nicht in die .NET Infrastruktur integriert werden, weil ein Compiler dazu ja MSIL und nicht Java Byte-Code erzeugen müsste, und das werden die Lizenzverträge von Sun wohl kaum erlauben.

Ein weiteres pikantes Detail ist, dass Microsoft die C# Sprachdefinition bei der ECMA zur Standardisierung eingereicht hat [ECMA00]. Damit soll eine breite Unterstützung durch die Industrie sichergestellt werden. Es sieht aber so aus, als wolle Microsoft mit diesem Schritt Sun microsystems ein weiteres Schnippchen schlagen, denn Sun hat sich bis anhin geweigert, die Kontrolle über Java einem Standardisierungsgremium abzutreten.

Ob und wie weit die Sprache Bestand haben wird ist schwer zu sagen. Man muss jedoch auch bedenken, dass mit der Realisierung der .NET Plattform und damit der Realisierung von Komponentensoftware die Frage nach der "idealen" Programmiersprache an Bedeutung verliert. Jeder kann für seine Dienste jene Programmiersprache verwenden, die am besten auf das Problem zugeschnitten ist, sei dies nun Java, C# oder eine exotische Sprache wie z.B. ML. Doch vielleicht ist es gerade deshalb plausibel und reizvoll, sich mit der neuen Sprache C# auseinanderzusetzen, ohne sich auf eine weitere Sprachinsel setzen zu müssen.

## Literatur und Links

- [Cha97] David Chappell,  
*COM+: The Next Generation*,  
BYTE Magazine, December 1997, p. 99-102.
- [ECMA00] ECMA - Standardizing Information and Communication Systems,  
*Two new projects for ECMA TC39*,  
<http://www.ecma.ch/ecma1/news/news.htm>
- [Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides,  
*Design Patterns*,  
Addison-Wesley, 1995.
- [Hag00] Peter Hagggar,  
*Practical Java*,  
Addison -Wesley, 2000.
- [Hej00] Anders Hejlsberg & Scott Wiltamuth,  
*C# Language Reference*, Microsoft Corp, 2000,  
<http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp>
- [Mey92] Bertrand Meyer,  
*Eiffel: The Language*,  
Prentice Hall, New York, 1992.
- [Omi97] Oberon microsystems Inc.,  
Component Pascal Language Report, 1997,  
[http://www.oberon.ch/resources/component\\_pascal/language\\_report.html](http://www.oberon.ch/resources/component_pascal/language_report.html)
- [Tes85] Larry Tesler,  
*Object Pascal Report*,  
Structured Language World, Vol 9, No 3, 1985.  
<http://www.mactech.com/articles/mactech/Vol.02/02.12/ObjectPascal/>
- [Wil00] Christoph Wille,  
*Presenting C#*,  
SAMS, Jul 2000, ISBN 0-67232-037-1.

Dominik Gruntz ist Professor im Studiengang Informatik an der Fachhochschule Aargau in Brugg-Windisch in der Schweiz. Seine Schwerpunkte sind Komponentensoftware und Frameworkdesign.  
E-mail: [d.gruntz@fh-aargau.ch](mailto:d.gruntz@fh-aargau.ch)

