# C#: eine Konkurrenz für Java?

C#: a Competitor of Java?

Jürgen F. H. Winkler, Friedrich-Schiller-Universität Jena

**Zusammenfassung** C# ist eine neue Programmiersprache, die im Wesentlichen bei Microsoft entwickelt worden ist. Erste Fassungen der Sprachdefinition wurden im Jahre 2000 veröffentlicht, im Dezember 2001 wurde bei ECMA die Norm ECMA-334 und im März 2003 bei ISO die Norm 23270 verabschiedet. C# ist inhaltlich gesehen eine Erweiterung von Java. Die Erweiterungen bestehen aus der Einführung neuer Sprachelemente wie zum Beispiel Delegate, Event und Property, und aus der Verbesserung vorhandener Elemente, wie zum Beispiel der Switch-Anweisung. Das technische Konzept der Implementierung weist ebenfalls Verbesserungen auf, und nicht zuletzt ist die Sprachdefinition noch etwas übersichtlicher als die auch recht gute Definition von Java.
Der Aufsatz gibt einerseits einen Überblick über die Geschichte von C# und Java und stellt andererseits die Neuerungen anhand von kleinen Beispielen vor. Abschließend wird auf einige Defizite hingewiesen, die vor allem auf der Verwurzelung in der C-Welt beruhen. ▶▶▶ **Summary** C# is a new programming language developed by Microsoft. It was first published in 2000. The standard ECMA-334 and the international standard ISO/IEC 23270 were accepted in December 2001 and March 2003, respectively. Conceptually, C# is an extension of Java. There are new elements, e. g., delegate, event, and property, and there are improvements of existing elements, e. g., the switch statement. The implementation has also been improved, especially with respect to interoperability. Last but not least the language specification is even more lucid than that of Java.
The paper gives an overview of the history of C# and Java. It discusses new elements and properties of C# by giving small examples. The paper closes with hints to elements which still show deficiencies.

**KEYWORDS** D.3.2, D.3.3 [Programming Languages] Language Classifications, Object-oriented languages, Language Constructs and Features, Classes and Objects, Data types and Structures; C#, Java, Property, Delegate, Event, Indexer, Arithmetic

## 1 Einleitung

C# ist eine neue Programmiersprache, die im Wesentlichen bei Microsoft entwickelt worden ist. Erste Fassungen der Sprachdefinition wurden im Jahre 2000 veröffentlicht, und im Dezember 2001 wurde bei ECMA die Norm ECMA-334 verabschiedet. Eine überarbeitete Version wurde im Dezember 2002 verabschiedet. Im März 2003 verabschiedete die ISO die Norm ISO/IEC 23270:2003.

Obwohl in der Sprachdefinition und in anderen Dokumenten C# als Fortentwicklung von C++ dargestellt wird, hat C# auch einen starken Bezug zu Java, denn es kann als Antwort von Microsoft auf die Programmiersprache Java gesehen werden, welche von der Firma Sun entwickelt worden ist. Beim technischen Vergleich wird im Wesentlichen Java als Bezugsbasis verwendet, da dies der unmittelbare Vorgänger ist.

Inhaltlich können zwei Arten von Neuerungen unterschieden werden: neue Sprachelemente, wie zum Beispiel Event und Indexer, und Verbesserungen und Modifikationen bereits vorhandener Sprachelemente wie zum Beispiel der Switch-Anweisung.

Einen groben Überblick über das Verhältnis von C# zu Java gibt die Übersicht in Tabelle 1. Daran erkennt man, dass es sich im Prinzip um eine deutliche Erweiterung von Java handelt. Dies gilt inhaltlich, da sich einige wenige Elemente syntaktisch unterscheiden, z. B. die Ableitungsklausel. Die Übersicht in Tabelle 1 gilt für das Verhältnis der beiden Kernsprachen und nicht für die jeweils dazugehörenden vordeklarierten Klassenbibliotheken. Die Neuerungen sind in drei Gruppen eingeteilt. Die erste Gruppe sind die Sprachelemente die ganz neu sind, d. h. in Java und auch den verbreiteten früheren Sprachen wie Ada, C, C++ nicht enthalten sind. Dies schließt nicht aus, dass Formen dieser Elemente bereits in früheren Sprachen aus dem Forschungsbereich enthalten sind. Die Elemente der zweiten Gruppe kommen bereits in verbreiteten früheren Sprachen vor, sind aber in Java nicht ent-

**Tabelle 1** C# versus Java.

| C# ≈ Java |
|---|
| + Property |
| + Delegate |
| + Event |
| + Indexer |
| + Attribut |
| + „new“ method |
| + Verbundtyp (Struct) |
| + Aufzählungstyp |
| + Dezimaltyp |
| + Operator-Deklaration |
| + non-virtual method |
| + neue Parameter: ref und out |
| + foreach-Schleife |
| + explizite Verweise (unsafe) |
| + conditional compilation |
| + goto |
| + Wertebereichsüberwachung |
| + Berechnungen von li nach re |
| + boxing / unboxing |

halten. Die dritte Gruppe bezieht sich nicht auf einzelne Sprachelemente, sondern auf Eigenschaften von Elementen wie zum Beispiel von arithmetischen Ausdrücken. Zu den meisten der aufgeführten Elemente werden im Rest des Aufsatzes nähere Erläuterungen gegeben. Im Detail gibt es natürlich noch weitere Unterschiede zu Java.

Die Sprachdefinition als Dokument gesehen ist noch etwas übersichtlicher strukturiert als die auch recht übersichtliche Definition von Java [1].

Eine Reihe weiterer Verbesserungen betreffen die Implementierung, in welcher Programme in verschiedenen Sprachen in eine gemeinsame Zwischensprache übersetzt werden, die ähnlich zum Code der *Java Virtual Machine* (JVM) ist. Die Ausführung dieser Programme erfolgt dann in einer einheitlichen Ablaufumgebung (CLR: *Common Language Runtime*). Vor der Ausführung erfolgt in jedem Falle eine Übersetzung in Maschinencode, d. h. es tritt keine Interpretation wie in JVM auf. Der vorliegende Aufsatz konzentriert sich auf die Sprache im engeren Sinne und geht daher auf die Implementierung nicht weiter ein.

## 2 Einordnung in die Programmiersprachenlandschaft

Die Einordnung in die Landschaft der Programmiersprachen ist in Bild 1 dargestellt. In diesem Bild ist die Entwicklung der Programmiersprachen von ihren Anfängen bis heute dargestellt. Aufgenommen sind Programmiersprachen, die praktisch in nennenswertem Maße eingesetzt werden oder wurden, oder die einen besonderen Einfluss auf die Entwicklung der Programmiersprachen hatten.

In der zweiten Dimension sind die verschiedenen Spracharten unterschieden: imperativ, funktional und logikorientiert. Die Objektorientierung wird dabei dem imperativen Paradigma zugeordnet, da das entscheidende Kriterium des imperativen Paradigmas auch auf die Objektorientierung zutrifft, nämlich die Existenz von Variablen, die im Verlaufe ihrer Lebensdauer mehrere unterschiedliche Werte annehmen können. Die imperativen Sprachen sind die umfangreichste Gruppe in Bild 1, da in der Praxis im Wesentlichen diese Sprachen eingesetzt werden.

Einflüsse zwischen Sprachen sind durch entsprechende Pfeile dargestellt, wobei hier nicht jedes Detail dargestellt ist. In der Spalte „Algol60“ wurden die späteren Sprachen jeweils durch die Vorgänger beeinflusst; hier wurde jedoch auf die entsprechenden Pfeile verzichtet. Die gestrichelten Pfeile zeigen den Weg zu C#. In der Spalte „Simula67” stehen die häufig als „rein objektorientiert” bezeichneten Programmiersprachen.

## 3 Geschichte von Java und C#

Wie bereits erwähnt worden ist, wurden die beiden Sprachen von zwei unterschiedlichen Firmen entwickelt. Java wurde etwa fünf Jahre vor C# der Öffentlichkeit vorgestellt. Viele Firmen haben Java von Sun lizenziert, darunter auch Microsoft. Die Beziehung beider Firmen bezüglich Java war aber nicht frei von Spannungen:

*23. Mai 1995*: Sun stellt Java der Öffentlichkeit vor. Dabei wird am Anfang der Einsatz als Applet hervorgehoben, da dies eine neue Idee war und gut zu dem damals rasant wachsenden WWW passte. Um die

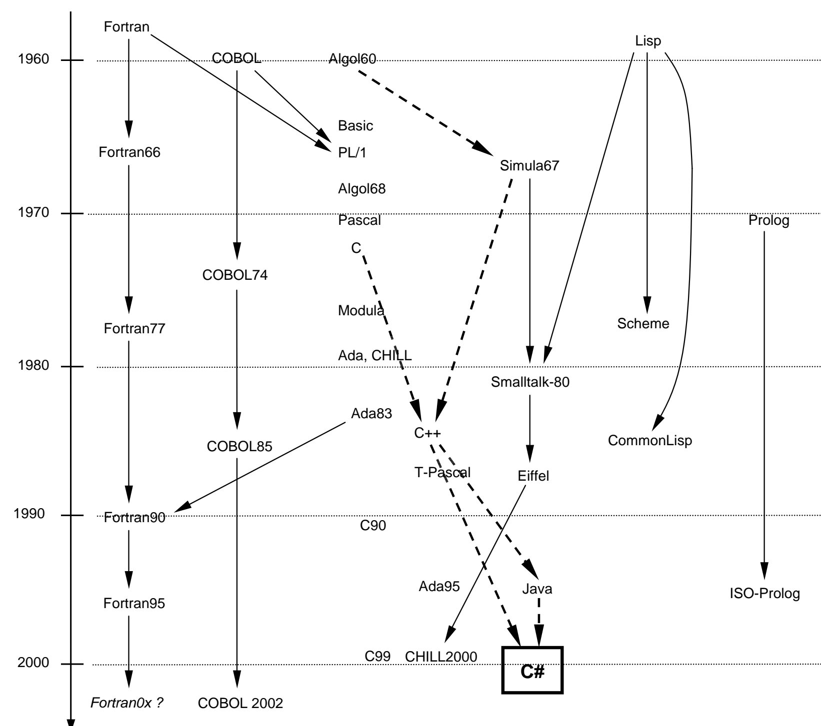



**Bild 1** Historische Zusammenhänge.

Sprache unter Kontrolle zu haben, hat sich Sun den Namen „Java“ als Warenzeichen schützen lassen.

*1996 +*: Java (JVM und Klassenbibliothek) wird in viele Browser eingebaut.

*12. März 1996*: Microsoft erhält von Sun eine Lizenz zur Benutzung von Java in seinem *Internet Explorer* (IE) Browser. Unter dem Namen *MS Visual J++* entwickelt und implementiert Microsoft eine Sprachvariante, welche weitere Elemente (zum Beispiel delegate) enthält. Microsoft möchte Java besser an Windows anpassen, da Sun im Wesentlichen in der Unix-Welt arbeitet und so kein Interesse an einer Anpassung an Windows hat. Auch im IE 4.0 implementiert Microsoft eine Erweiterung von Java.

*7. Oktober 1997*: Das Vorgehen von Microsoft ist nach den Lizenzbedingungen [2] nicht erlaubt. Daher erhebt Sun Klage gegen Microsoft. Der Streit zieht sich vor Gericht etwas hin und führt im *Januar 2001* zu einem Vergleich. Microsoft darf das Java-Logo nicht mehr verwenden und zahlt 20 Millionen USD an Sun [3]. Microsoft darf jedoch die zum Zeitpunkt des Vergleichs existierenden Java-Produkte weiter an seine Kunden liefern.

Sowohl Sun als auch Microsoft streben an, dass ihre Sprache eine möglichst weite Verbreitung erfährt, d.h. von möglichst vielen Firmen benutzt wird. Dies wird in der Regel dadurch gefördert, wenn die Sprache verbindlich festgelegt ist. Eine Normung durch internationale Normungsorganisationen hat sich zum Beispiel im Falle von COBOL, Fortran oder Ada bewährt. Wenn eine Sprache nur durch eine Firma festgelegt ist, besteht die Gefahr, dass diese Firma die Sprache einseitig ändert und zuwenig Rücksicht auf die Anwender nimmt. Beide Firmen, Microsoft und Sun, waren daher bestrebt, ihren Sprachen zu den „höheren Weihen“ einer internationalen Normung zu verhelfen.

*März 1997*: Sun reicht Java bei der ISO zur Normung ein. Nach anfänglichen Widerständen ergibt sich eine grundsätzlich positive Beschlusslage innerhalb der ISO. Die Normung von Java durch ISO scheitert aber an der Forderung von Sun, das Warenzeichen an „Java“ zu behalten: “Sun cannot and will not surrender its trademarks for the Java platform to ISO or to anyone else.” (Jim Mitchell in [4]). Diese Forderung ist mit den Regeln der ISO nicht verträglich.

*Mai 1999*: Sun zieht Java bei der ISO zurück und reicht es stattdessen bei der ECMA ein.

*März 2000*: Bei ECMA gelten bezüglich der Rechte an genormten Dingen ähnliche Regeln wie bei ISO. Da Sun an seiner Forderung festhält, führt dies dazu, dass Sun Java auch bei der ECMA zurückzieht.

*Frühjahr 2000*: Sun richtet unter dem Namen *Java Community Process* eine eigene Organisation zur Fortentwicklung von Java ein, in welcher Sun maßgeblichen Einfluss hat.

*26. Juni 2000*: Veröffentlichung der ersten Version von C#.

*Juli 2000*: Microsoft macht erstmals eine Implementierung von C# öffentlich zugänglich.

*31. Oktober 2000*: HP, Intel und Microsoft reichen C# bei der ECMA zur Normung ein.

*November 2000*: Die Entwicklung der Norm „ECMA-334 C# Language Specification“ beginnt.

*13. Dezember 2001*: Die Norm ECMA-334 wird von der ECMA angenommen.

2002: Die Programmierumgebung Visual Studio von Microsoft enthält C#.

*Dezember 2002*: Die Norm ECMA-334 (2nd edition) wird von der ECMA angenommen.

*28. März 2003*: Die Norm „ISO/IEC 23270:2003 – C# Language Specification“ wird verabschiedet.

## 4 Sprachumfang

Die Übersicht in Tabelle 1 zeigt, dass C# deutlich mehr Sprachelemente enthält als Java. Eine detaillierte Aufstellung in Form eines Sprachprofils [5] zeigt, dass in C# nur wenige der in derzeitigen Programmiersprachen vorhandenen Sprachelemente fehlen. Die wesentlichen sind:

- Generics
- benutzerdeklarierte Zahlenbereiche
- Array- und Record-Aggregate
- Trennung von Spezifikation und Rumpf
- Parallelität

## 5 Neuerungen in C#

### 5.1 Kleinere Neuerungen

**switch-Anweisung:** Jede Alternative muss mit `break` abgeschlossen werden (no fall through). Dadurch wird die ursprünglich in Algol68 und Pascal eingeführte Semantik auch in der C-Welt eingeführt. Dass `break` überflüssigerweise geschrieben werden muss, ist wohl als Konzession an die C/C++/Java-Programmierer zu sehen.

**Ausdrucksauswertung:** Die Operanden in Ausdrücken und die Ausdrücke in Ausdruckslisten (z.B. Parameterlisten) müssen von links nach rechts ausgewertet werden. Dies vermeidet mögliche Mehrdeutigkeiten, wie sie in vielen Sprachen vorkommen, in welchen diese Ausdrücke und Teilausdrücke oft in beliebiger Reihenfolge ausgewertet werden dürfen. So kann die Anweisung

```
x = readInt() - readInt();
```

zu unterschiedlichen Werten von x nach der Ausführung führen. Wenn die Eingabe die Zahlen 1 und 2 enthält, dann ergibt sich bei Auswertung von links nach rechts der Wert $-1 (= 1 - 2)$ und bei Auswertung von rechts nach links der Wert $1 (= 2 - 1)$.

Java fordert zwar auch schon die Auswertung von links nach rechts, allerdings ist die Formulierung in der Sprachdefinition in diesem Punkt etwas unbefriedigend [1: Abschn. 15.7].

**Wertebereichsüberwachung:** Die Endlichkeit der Zahlenbereiche im

Rechner ist bisher in Programmiersprachen nur unzureichend berücksichtigt [6]. In C# können Ausdrücke im *checked*- oder im *unchecked*-Modus ausgewertet werden. Im *checked*-Modus wird bei einer Wertebereichsverletzung bei Ganzzahlarithmetik eine Ausnahme ausgelöst, wie dies in Ada seit über 20 Jahren gefordert wird. Der *unchecked*-Modus entspricht der unbefriedigenden Ganzzahlarithmetik von Java. Unglücklicherweise ist als Default-Fall, d. h. wenn keine explizite Vorgabe durch den Programmersteller erfolgt, der *unchecked*-Modus festgelegt. Es ist aber wenigstens ein Schritt in die richtige Richtung.

**Definiertheit von Variablen:** Viele Variablen werden bei ihrer Erzeugung mit einem Default-Wert initialisiert. Dies gilt zum Beispiel für alle Komponenten von Objekten. Für lokale Variablen in einem Unterprogramm gilt das leider nicht. Um hier aber auch Zugriffe zu nicht initialisierten Variablen zu vermeiden, wurde das Konzept der garantierten Zuweisung (*definite assignment*) (DA) von Java übernommen. Eine Variable hat dann an einer Stelle s die Eigenschaft DA, wenn auf allen Pfaden von der Deklaration zur Stelle s mindestens eine Zuweisung erfolgt. Dies bedeutet einen beachtlichen Analyseaufwand für den Compiler, der vermieden werden könnte, wenn alle Variablen mit einem Default-Wert initialisiert würden.

**Parameter:** Gegenüber Java sind in C# drei weitere Arten von Parametern eingeführt worden: *ref*-, *out*- und *params*-Parameter. *ref*-Parameter werden per Referenz (=Alias) übergeben und müssen beim Aufruf die Eigenschaft DA haben. Im Aufruf muss der aktuelle *ref*-Parameter ebenfalls mit dem Schlüsselwort `ref` markiert werden. Dies erhöht die Lesbarkeit. *out*-Parameter werden ebenfalls per Referenz übergeben, beim Aufruf wird aber nicht die Eigenschaft DA gefordert. Sie müssen im Aufruf mit dem Schlüsselwort `out` markiert werden.

Wenn eine Methode als

```
void M(int x, ref int y,
       out int z)
```

deklariert ist, dann kann sie mit

```
M(a*b + 10, ref c, out d);
```

aufgerufen werden. Man sieht also zum Beispiel als Reviewer auch im Aufruf sofort, welche Parameter wie übergeben werden, und muss nicht in der Deklaration nachlesen.

In einer Unterprogrammdeklaration kann es höchstens einen *params*-Parameter geben, der dann der letzte Parameter sein muss. Er ist ein Wertparameter und sein Typ muss ein eindimensionaler Array-Typ sein. Im Aufruf dürfen für den *params*-Parameter null oder mehr aktuelle Parameter vom Elementtyp angegeben werden, deren Werte zu einem entsprechenden Array zusammengefasst werden.

Wenn eine Methode als

```
void M(int x,
       params int[] y)
```

deklariert ist, dann kann sie mit

```
M(a*b + 10, 10, 20, 30);
// y ist int[] {10, 20, 30}
M(17); // y ist int[] {}
```

aufgerufen werden.

**Methodeneigenschaften:** In objektorientierten Programmen kann es etwas kompliziert sein, bei einem Methodenaufruf O.M(…) festzustellen, welcher Rumpf von M bei einer bestimmten Ausführung von O.M(…) ausgeführt wird. Mehrere Mechanismen können hier zusammenwirken: die Art der Methode, Überladen, der dynamische Typ von O, die Typhierarchie, zu welcher der statische Typ von O gehört, und welche Eigenschaften die Implementierungen von M in dieser Hierarchie haben. Bei Änderungen von Typen in dieser Typhierarchie, kann es zu unerwarteten und auch unerwünschten Effekten kommen. Hier erhält der Programmentwickler mehr Kontrolle durch die Eigenschaften `new`, `override` und `virtual`.

Mit `new` wird zum Ausdruck gebracht, dass eine geerbte Methode oder auch eine andere Größe in einer abgeleiteten Klasse bewusst durch eine Deklaration mit gleicher Signatur bzw. gleichem Bezeichner verdeckt wird.

`virtual` wird wie in C++ verwendet. Ist M eine *virtual*-Methode, dann wird jeweils der zum aktuellen Wert von O gehörende Rumpf von M ausgeführt. Wenn O polymorph ist, dann können bei unterschiedlichen Werten von O unterschiedliche Rümpfe von M ausgeführt werden. Wenn M nicht *virtual* ist, dann wird stets der Rumpf von M im statischen Typ von O ausgeführt.

Eine Neuimplementierung einer *virtual*-Methode muss explizit mit `override` markiert werden. Dabei müssen Signatur und Exportstatus übereinstimmen.

**Struct:** *struct*-Typen sind Record-Typen und verhalten sich wie Wert-Typen, während in C# Klassen Referenz-Typen sind. Ein *struct*-Typ kann von Interface-Typen abgeleitet sein, nicht aber von einem anderen *struct*-Typ und auch nicht von einer Klasse.

**Boxing/Unboxing:** In Simula67, Smalltalk und Java sind fast alle Typen Referenz-Typen. In Java sind nur die primitiven Zahltypen und `boolean` Wert-Typen. In Programmen kommt es vor, dass auch diese Zahltypen als Referenztypen benötigt werden, z. B. bei einer heterogenen Liste mit Elementtyp `Object`. Dazu gibt es in Java die zu den Zahltypen korrespondierenden Referenztypen in Form von Wrapper-Klassen (`Integer` usw.). Soll nun ein `int`-Wert in solch eine Liste eingefügt werden, dann muss explizit ein `Integer`-Objekt mit `new` und geeignetem Konstruktor erzeugt werden. *boxing* und *unboxing* sind implizite Konvertierungen zwischen Wert-Typen und dem Typ `Object` oder einem Interface-Typ, der vom Wert-Typ implementiert wird.

Hat also der Listentyp die Methode

```
void add(object newElem)
```

dann kann man die Zahl 10 durch

```
L.add(10)
```

zur Liste L hinzufügen. Es wird automatisch ein Objekt erzeugt, welches 10 enthält. In Java müsste man das als

```
L.add(new Integer(10));
```

formulieren. Wenn die Methode zur Entnahme des nächsten Elementes

```
object nextElem()
```

ist, dann kann die Entnahme eines Elementes durch

```
i = (int)L.nextElem();
```

erfolgen. Dabei wird der Wert automatisch aus dem vorher implizit erzeugten Objekt entnommen. In Java kann man das mit

```
i = ((Integer)L.nextElem())
    .intValue();
```

formulieren.

**Operatordeklaration:** Nachdem Java Operatordeklarationen leider nicht enthält, sind sie in C# wieder aufgenommen worden. Deklariert werden können die vordeklarierten Operatoren und explizite und implizite Konversionen. Für die Zuweisung und die Indizierung ist die Operatordeklaration nicht erlaubt.

**Aufzählungstyp:** Ebenso wurden Aufzählungstypen wieder in C# aufgenommen.

### 5.2 Neue Sprachelemente

**Property:** Eine Property ist eine Komponente einer Klasse, welche von außen wie eine Variable (*field*) erscheint. Innerhalb der Klasse können das Lesen und Schreiben durch Anweisungsfolgen definiert werden, die beim jeweiligen Zugriff automatisch ausgeführt werden. Properties können als eine Verallgemeinerung der Programmierkonvention gesehen werden, bei welcher Variable grundsätzlich nichtöffentlich sind, und im Bedarfsfalle Zugriffsfunktionen zum Lesen und Schreiben deklariert werden (*getters* und *setters*).

a) Deklaration

```
public class Button: Control {
  private string caption;
  public string Caption {         // Def. des Property
    get { return caption; }          // Lesen
    set { if (caption != value) {    // Schreiben
            caption = value;
            Repaint();        }
        }
}
```

b) Benutzung

```
Button okButton = new Button();
okButton.Caption = "OK";  // set wird ausgeführt
          // d.h. ggf. wird auch Repaint() ausgeführt
string s = okButton.Caption;   // get wird ausgeführt
```

**Bild 2** Deklaration und Benutzung eines Property.

In der Deklaration einer Property können eine Lese- und eine Schreibfunktion festgelegt werden. Bild 2 enthält ein Beispiel für die Deklaration und Benutzung einer Property [7: 17.6.2].

Im Allgemeinen besteht keine 1 : 1-Zuordnung zwischen Properties und lokalen Größen, da in den *get*- und *set*-Blöcken beliebige Anweisungsfolgen stehen dürfen.

Properties können typspezifisch (*static*) und instanzspezifisch sein. Instanzspezifische Properties können *virtual* und auch abstrakt sein. Properties können auch in Interfaces deklariert werden.

**Delegate:** Ein Delegate ist eine geordnete Liste von Objekt-Methoden-Paaren

(O1.M1, …, On.Mn),

wobei die Methoden alle mit dem Typ des Delegate kompatibel sein müssen (d. h. gleiche Signatur und gleicher Ergebnistyp haben müssen). Mit den Operationen „+“ und „–“ kann diese Liste entsprechend manipuliert werden. Beim Aufruf eines Delegate werden die Listenelemente in der Reihenfolge der Liste nacheinander mit den gegebenen Parametern aufgerufen. Das Beispiel in Bild 3 zeigt die Arbeitsweise dieser Operationen. Ein Beispiel für die praktische Anwendung ist im folgenden Abschnitt zu finden.

**Event:** Events sind Abstraktionen von Delegates, d. h. Delegates mit beschränkten Manipulationsmög-

```
class C {
  delegate void D(int x);  // Dekl. des Delegatetyps D
  public static void M1(int i) { ... };
  public static void M2(int k) { ... };
  public void M3(int i) { ... };
       // M1, M2, M3 sind kompatibel zum Delegatetyp D
}
class Demo {
  static void Main() {
  C.D d1 = new C.D(C.M1);
        // Dekl. und Init. eines Delegate: d1 = (C.M1)
  C.D d2 = new C.D(C.M2);                    // d2 = (C.M2)
  d1 = d1+d2;          // hinzufügen:  d1 = (C.M1, C.M2)
  C t1 = new C();
  d1 += new C.D(t1.M3);      // d1 = (C.M1, C.M2, t1.M3)
  d1 += new C.D(C.M1); // d1 = (C.M1, C.M2, t1.M3, C.M1)
  d1(10);    // bewirkt die Aufrufe:
           // C.M1(10); C.M2(10); t1.M3(10); C.M1(10);
  d1 -= d2;                // d1 = (C.M1, t1.M3, C.M1)
  }
}
```

**Bild 3** Deklaration und Benutzung eines Delegate-Typs.

**Bild 4** Deklaration und Benutzung eines Events.

```
class Button: Control {
  delegate void EventH(object sender, EventArgs e);
  public event EventH Click;
             // Event Click is defined and "published"
             // Click is of "type" "event EventH"
  protected void OnClick(EventArgs e) {
    if (Click!=null) Click(this,e);
                     // Click is called like a delegate
  }
}
class LoginDialog: Form {
  Button OkButton;
  public LoginDialog() {
    OkButton = new Button(...);
    OkButton.Click += new Button.EventH(this.OkButtonClick);
              // "this" subscribes to OkButton.Click
              // the pair (this, OkButtonClick)
  }           // is added to OkButton.Click
  void OkButtonClick(object sender, EventArgs e) {
                            //  compatible to EventH
  }
}
```

lichkeiten. Sie dienen der Realisierung des Observer- bzw. des Publish-Subscribe-Pattern. Im Beispiel in Bild 4 (nach [7: 17.7]) registrieren sich Instanzen der Klasse `LoginDialog` bei Instanzen der Klasse `Button` für das Event `Click`. Wenn das Event `Click` ausgelöst wird, dann wird an der betreffenden `LoginDialog`-Instanz die Methode `OkButtonClick` aufgerufen.

Man sieht, dass bei der Verwendung von Events der Delegate-Typ dazu dient, den Typ des zugrunde liegenden Delegate festzulegen. Das Beispiel ist nicht ganz einfach zu durchschauen. Ein Grund dafür ist die unsystematische Syntax von Deklarationen. In Zeile 2 wird `EventH` als ein Delegate-Typ deklariert, während in Zeile 3 `Click` als ein Objekt vom „Typ" `event EventH` deklariert wird.

**Indexer:** Ein Indexer ist eine spezielle Property, welche denselben Bezeichner wie die zugehörige Klasse hat und welche formale Parameter hat, die in diesem Falle in eckigen Klammern notiert werden. An den Verwendungsstellen des Indexers sieht es syntaktisch so aus, als ob ein Objekt als ein Array gesehen würde. In dem Beispiel in Bild 5 wird ein Stack, der als verkettete Liste implementiert ist, auch wie ein Array manipuliert (nach [7: 8.7.7]).

**Attribut:** Mit Attributen können Eigenschaften von Größen wie zum Beispiel Klassen, Methoden oder Parametern festgelegt werden. Dies erfolgt in eckigen Klammern vor der Deklaration der betreffenden Größe:

```
[Author("Max Müller"),
 Author("Emil Meier")]
class Class1 { … }

[Help("http://hlp.comp.com/
…/Class2.htm");]
class Class2 { … }
```

Die Elemente innerhalb der eckigen Klammern sind Konstruktoraufrufe, da die Attribute als Klassen deklariert werden:

```
[AttributeUsage
  (AttributeTargets.Class,
   AllowMultiple = true)]
class AuthorAttribute:
      Attribute { … }
```

Die Bezeichner aller Attributklassen müssen mit `Attribute` enden. An der Verwendungsstelle, zum Beispiel im Konstruktoraufruf, darf dieses Suffix auch weggelassen werden.

Die Werte der Attribute einer Größe können zur Laufzeit abgefragt werden.

Die Art der Information, die in Attributen dargestellt wird, ist weitgehend beliebig. Ein Anwendungsfeld kann zum Beispiel Information für das Projektmanagement wie Autor, Version, Entwicklungsstand, Reviewer, Reviewdatum usw. sein, die dann von entsprechenden Werkzeugen ausgewertet werden können [8]. Die Attribute verbessern die Realisierung selbständigerer Komponenten und sind daher für die Praxis von großer Bedeutung.

**foreach-Schleife:** Mit einer solchen Schleife kann der Schleifenrumpf für jedes Element eines Objektes durchgeführt werden. Die Aufzählung der Elemente ist dabei durch die Iteratorschnittstelle (*IEnumerable*) gegeben. *IEnumerable* definiert im Wesentlichen die Komponenten des Iterator-Patterns [9].

```
foreach
   (ItemTy item in content)
{ do something }
```

zum Beispiel iteriert über den Elementen vom Typ `ItemTy` des Objektes `content`.

Die Steuerung mittels *First*, *IsDone* und *Next*, die beim Iterator-Pattern (und auch in Java) explizit formuliert werden muss, wird durch die `foreach`-Schleife automatisch durchgeführt.

## 6 Sprachdefizite

Anhand der Übersicht in Tabelle 1 ist leicht zu sehen, dass C# eine Verbesserung gegenüber Java darstellt, da es eine Reihe nützlicher neuer Sprachelemente enthält, und da bei bestehenden Sprachelementen Schwächen beseitigt wurden. Die Wiedereinführung von *goto* ist weniger positiv, hebt aber die Vorteile nicht auf.

Dennoch bleiben Wünsche offen, die vor allem mit der Verwurzelung von C# in der C-Welt (C, C++, Java) zu tun haben. Einige Beispiele sind:

**Syntax:** Deklarationen haben nicht die typische Struktur

Definiendum = Definiens,

wie das z. B. in Ada oder Pascal weitgehend der Fall ist. Ein Beispiel für

**Bild 5** Deklaration und Benutzung eines Indexes.

```
using System;
public class Stack {
  private Node GetNode(int index) {
    Node temp = first;
    while (index > 0) {
      temp = temp.Next;  index--;  }
    return temp;
  }                         // Deklaration des Indexers
  public object this[int index] {
    get {                             // Lese-"Methode"
      if (!ValidIndex(index))
        throw new Exception("Index out of range.");
        else return GetNode(index).Value;  }
    set {                          // Schreib-"Methode"
      if (!ValidIndex(index))
        throw new Exception("Index out of range.");
        else GetNode(index).Value = value; }
  }
}
class Test  {
  static void Main() {
    Stack s = new Stack();
    s.Push(1);  s.Push(2);  s.Push(3);
    s[0] = 33;   // Changes the top item from 3 to 33
    s[1] = 22;   // Ch. the middle item from 2 to 22
    s[2] = 11;   // Ch. the bottom item from 1 to 11
    s[0] = s[1] + s[2];
  }
}
```

die schlechte Lesbarkeit von Deklarationen in C++ ist

$$\mathtt{const\ Vector} * \&(\mathtt{vectorD})(\mathtt{int}, \mathtt{Vector}[])$$

[10]. Die Syntax der Deklarationen in C# hat zwei wesentliche Schwächen: zu sparsame Verwendung von Schlüsselworten, welche die Art der deklarierten Größe signalisieren, und unklare Struktur. Man muss häufig ein großes Anfangsstück der Deklaration lesen, um festzustellen, von welcher Art die deklarierte Größe ist. Bild 6 enthält zwei derartige Deklarationen und jeweils eine etwas besser lesbare Alternative, welche auch klar anzeigt, was deklariert wird. Auf die Schwächen der Syntax von Deklarationen in C wurde schon vor rund 20 Jahren hingewiesen [11].

Die Syntax der Ausdrücke ist sehr freizügig und kann zu verwirrenden Gebilden wie

$$-\sim+--<++<<-++!\mathtt{x}--\hat{}\sim-+++$$

führen. Der Ausdruck ist auch in Java zulässig.

Eine syntaktisch ungünstige Konstruktion innerhalb der Ausdrücke sind die Typkonversionen (*cast*). Eine Typkonversion ist inhaltlich die Anwendung einer Funktion. Daher liegt es nahe, sie in der Form $\mathtt{f(e)}$ zu schreiben, da das die Standardschreibweise für Funktionsanwendungen in C# (und C, C++. Java) ist. Aus unerfindlichen Gründen wurde aber in C dafür die Schreibweise $\mathtt{(f)e}$ erfunden, die in ihrer Form für Funktionsaufrufe einzigartig ist. Wendet man diese Schreibweise naiv an, dann ergeben sich unerwünschte Effekte: $(\mathtt{int})\mathtt{x}+\mathtt{y}$ bedeutet, dass die Typkonversion nur auf $\mathtt{x}$ angewendet wird; man muss also $(\mathtt{int})(\mathtt{x}+\mathtt{y})$ schreiben, was der Standardform $\mathtt{int}(\mathtt{x}+\mathtt{y})$ ähnlich ist und lediglich ein überflüssiges Klammerpaar enthält. Da die Typangabe maximal die Form $\mathtt{id1.id2.}\ldots\mathtt{idn}$ haben kann [7: 10.8], sind die Klammern auch wirklich überflüssig. Es gibt auch mehrdeutige Fälle wie $(\mathtt{x})-\mathtt{y}$, was syntaktisch eine Typkonversion oder ein binärer Ausdruck sein kann. Der größte Teil der Definition von *cast*-Ausdrücken in der Sprachdefinition [7: 14.6.6] beschäftigt sich mit Regelungen, wie solche syntaktisch mehrdeutigen Ausdrücke zu interpretieren sind. Es wird also an den Symptomen kuriert, während $\mathtt{x}(-\mathtt{y})$ eindeutig und leicht verständlich ist. Dies wurde schon früher beobachtet, denn in C++ ist auch diese Funktionsschreibweise erlaubt [12: 5.2.3].

**Arithmetik:** Das größte Manko liegt hier im Bereich der Gleitkommaarithmetik, da eine Reihe von nützlichen Eigenschaften der IEEE 754 nicht zur Verfügung stehen. Insbesondere fehlen die verschiedenen Rundungsarten [6]. Ein Beispiel ist die Bedingung

$$\mathtt{c}<=\mathtt{floor}(\mathtt{log}(2^{63}-1)/\mathtt{log}(2)).$$

Die in der Sprache eingebauten Rundungen ergeben für den rechten Ausdruck den Wert 63, während der korrekte Wert 62 ist. Wenn man die in IEEE 754 vorhandenen Rundungen gezielt einsetzt, dann ergibt sich das korrekte Ergebnis.

Die Lage ist hier praktisch genauso unbefriedigend wie in Java, was einer der Hauptautoren von IEEE 754 bereits vor einiger Zeit kritisiert hat [13]. Da IEEE 754

**Bild 6** Deklarationssyntax in C# und Alternative.

```
override protected sealed internal extern intern
         Extern() {
sealed internal override extern protected intern
         Extern {
```

Eine etwas lesbarere Alternative dazu ist:

```
FUNC Extern: () => intern,
     protected internal, override sealed extern {
PROP Extern: intern,
     protected internal, override sealed extern {
```

heute in praktisch allen Mikroprozessoren eingebaut ist, ergibt sich die ärgerliche Situation, dass man das, wofür man bezahlt hat, nicht direkt benutzen kann. Dies ist allerdings derzeit auch in den anderen verbreiteten Programmiersprachen der Fall und ist verwunderlich, da der auf den Definitionen von IEEE 754 basierende Intel 8087 bereits vor über 20 Jahren eingeführt wurde. Eine Ausnahme ist hier die *feset*round-Funktion in C, welche die Einstellung der Rundungsart erlaubt [20: 7.6.3.2].

**Keine Trennung von Schnittstelle und Rumpf:** Die Trennung von Schnittstelle und Rumpf ist eine der nützlichen Erfindungen im Bereich der Programmiertechnik und wurde im Zusammenhang mit der Modularisierung bereits vor 30 Jahren von Parnas vorgestellt [14]. Die syntaktische Trennung von Schnittstelle (*interface*) und Rumpf (*implementation*) ist beispielsweise in Ada, CHILL [15] und Mesa [16] realisiert. Sie kann in minderer Form in C++ praktiziert werden und ist nicht möglich in C# und Java. Wenn durch Dokumentationswerkzeuge wie *JavaDoc* die Schnittstelleninformation herausgezogen wird, dann ist das nur eine zweitbeste Lösung.

**Generizität:** Diese ist seit 1979 in Ada enthalten und wurde später unter dem Namen *templates* in C++ aufgenommen. Sie stellt im Wesentlichen die Möglichkeit dar, Größen wie Typen oder Pakete mit Typen zu parametrisieren. Ein Beispiel dafür ist ein Stacktyp, bei welchem der Elementtyp als Parameter auftritt, sodass der Code für die Stackalgorithmen nur einmal formuliert werden muss. Dies hat sich als sehr nützlich erwiesen. Sowohl bei C# als auch bei Java gibt es Bestrebungen, Generizität in die Sprache einzufügen.

## 7 Ausblick

Die weitere Entwicklung kann nicht vorausgesagt werden. Da C# besser als Visual Basic, C++, J++ und Java ist und besser angepasst an Windows und .NET ist, sind die folgenden Erwartungen naheliegend:

- C# wird innerhalb von Microsoft intensiv verwendet werden
- C# wird in zunehmendem Maße von Firmen verwendet werden, die Software für Windows und .NET entwickeln
- C# ist ein guter Kandidat für den Einsatz in der Lehre.

Betrachtet man die Entwicklung über einen längeren Zeitraum, dann stellt man fest, dass nach C++ und Java C# ein weiterer Schritt heraus aus dem „Tal der Tränen" ist, in welches die höheren Programmiersprachen durch C hineingeraten waren. In manchen Bereichen ist das Niveau von Ada [17] noch nicht erreicht. Es ist schade, dass die Entwickler von C# (und auch Java) beim Design der Sprache den Stand der Technik auf dem Gebiet der Programmiersprachen nicht in größerem Maße berücksichtigt haben.

Dennoch ist zu erwarten, dass beide Sprachen in der nächsten Zeit verbessert werden. C# und Java sind sehr ähnliche Sprachen, die Produkte konkurrierender Firmen sind. Es ist das erste Mal in der Geschichte der Programmiersprachen, dass zwei so ähnliche Programmiersprachen in solch einer Konkurrenz zueinander stehen. Wie bei anderen technischen Produkten kann eine solche Konkurrenzsituation die Konkurrenten dazu anspornen, ihr Produkt zu verbessern. Dies ist auch bereits zu beobachten: Sun hat für 2004 die Version J2SE 1.5 angekündigt, welche unter anderem folgende Neuerungen enthalten wird [18]:

- Generische Klassen und Methoden
- Aufzählungstypen
- Boxing / Unboxing
- Foreach-Schleife.

Ein Vergleich mit der Übersicht in Tabelle 1 zeigt, dass die letzten drei Neuerungen bereits in C# enthalten sind. Andererseits hat Microsoft bereits auf der OOPSLA 2002 *Generics* für C# angekündigt [19]. Der Wettlauf ist also voll im Gange!


### Danksagung

Ich danke Wolfram Amme für nützliche Hinweise zu einer früheren Fassung des Aufsatzes und Jörg Sommer für den Hinweis auf die Funktion *feset*round in C.


### Literatur


[1] J. Gosling, B. Joy, G.L. Steele, and G. Bracha: The Java Language Specification – 2nd ed., Addison-Wesley 2000. ISBN 0-201-31008-2.

[2] J. Gosling, B. Joy, and G.L. Steele: The Java Language Specification. Addison-Wesley 1996. ISBN 0-201-63451-1.

[3] silicon.de: Java-Prozess: Richter erteilt Microsoft schwere Hausaufgaben. (http://www.silicon.de/cls/Plattformen/06_120402.html)

[4] R. McMillan: Sun's plan B for Java standardization: change nothing. (www.javaworld.com/javaworld/jw-10-1997/jw-10-iso_p.html)

[5] J.F.H. Winkler: C# – Geschichte und Überblick. Friedrich-Schiller-Universität Jena, Bericht Math/Inf/2002/13, 2002.Aug.27.

[6] J.F.H. Winkler: A safe variant of the unsafe integer arithmetic of Java™. In: Software – Practice and Experience (2002). Vol. 32 No. 7. pp. 669-701.

[7] Standard ECMA-334. 2nd ed., December 2002. ECMA, Geneva, 2002.

[8] J.F.H. Winkler: Version Control in Families of Large Programs. In: 9th Int'l Conf. on Software Engineering 1987, Monterey, pp. 150-161.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design Patterns. Addison-Wesley 1995. ISBN 0-201-63361-2.

[10] B. Werther and D. Conway: A Modest Proposal: C++ Resyntaxed. In: ACM SIGPLAN Notices (1996), Vol. 31, No. 11, pp. 74-82.

[11] B. Anderson: Type Syntax in the Language "C" – an object lesson in syntactic innovation. In: SIGPLAN Notices (1980), Vol. 15, No. 3, pp. 21-27.

[12] International Standard ISO/IEC 14882:1998(E): Programming Languages – C++. First ed. 1998-09-01. ISO/IEC, Geneva 1998.

[13] W. Kahan and J.D. Darcy: How Java's Floating-Point Hurts Everyone Everywhere. (http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf).

[14] D.L. Parnas: On the criteria to be used in decomposing systems into modules. In: CACM (1972), Vol. 15, No. 12, pp. 1053-1058.

[15] J.F.H. Winkler: CHILL 2000. In: Telektronikk (2000), Vol. 96, No. 4, pp. 70-77.

[16] C.M. Geschke, J.H. Morris Jr., and E.H. Satterthwaite: Early Experience with Mesa. In: CACM (1977), Vol. 20, No. 8, pp. 540-553.

[17] Preliminary ADA Reference Manual. In: SIGPLAN Notices (1979), Vol. 14, No. 6, Part A.

[18] New Java Language Features in J2SE 1.5. (http://developer.java.sun.com/developer/community/chat/JavaLive/2003/jl0729.html)

[19] Pressemitteilung von Microsoft. (http://www.microsoft.com/presspass/press/2002/Nov02/11-08OOPSLAPR.asp).

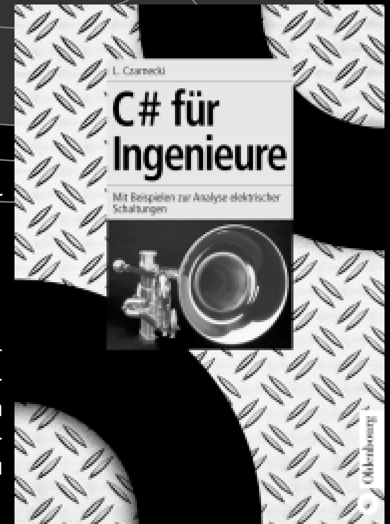
[20] International Standard ISO/IEC 9899: 1999 (E): Programming Languages – C. Second ed. 1999-12-01. ISO/IEC, Geneva 1999.

**Prof. Dr. Jürgen F. H. Winkler** ist seit 1993 Professor am Institut für Informatik der Friedrich-Schiller-Universität Jena. Seine Hauptarbeitsgebiete sind Programmverifikation und Programmiersprachen. Er erhielt das Diplom und die Promotion in Informatik von der Universität Karlsruhe.
Adresse: Friedrich-Schiller-Universität Jena, Institut für Informatik, 07740 Jena,
E-Mail: jwinkler@acm.org



**Oldenbourg! Aktuell**

Lothar Czarnecki
**C# für Ingenieure**
Mit Beispielen zur Analyse elektrischer Schaltungen
2003. 243 Seiten
Oldenbourg Lehrbücher für Ingenieure
**€ 24,80**
ISBN 3-486-27357-4




Das Buch versetzt Ingenieure in die Lage, Anwendungen aus ihrem Fachgebiet mit der Programmiersprache C# zu entwickeln. Unabdingbare Grundlagen der praktischen Informatik werden bei der Lektüre ganz nebenbei vermittelt. Das Buch eignet sich daher genauso für Ingenieure im Berufsleben wie für die Ausbildung angehender Ingenieure und Wirtschaftsingenieure im Lehrfach Informatik.

Jürgen Schröter / Helmut Seidel
**Perl**
Grundlagen und effektive Strategien
2003. 350 Seiten
**€ 24,80**
ISBN 3-486-25889-3

In lockerem Stil wird der Leser umfassend in die Programmiersprache Perl eingeführt. Mit mehr als 350 sehr ausführlich beschriebene Programm-Beispielen eignet sich das Buch ganz besonders gut für Einsteiger.

Oldenbourg Wissenschaftsverlag
Rosenheimer Straße 145
D-81671 München
Telefon 0 89 / 4 50 51-0
Fax 0 89 / 4 50 51-204
**Bestellungen:**
**www.oldenbourg-verlag.de**

Oldenbourg