

Seminar C#
Spezielle Sprachelemente in C# Properties Indexer
Arrays und Attribute

Olaf Sielemann

18. Oktober 2002

Inhaltsverzeichnis

1	Einleitung	2
2	Properties	2
2.1	Beispiel	3
2.2	Anwendungsgebiete	4
2.3	Besonderheiten	5
3	Indexer	6
3.1	Beispiel	6
3.2	Anwendungsgebiete	7
3.3	Besonderheiten	8
4	Arrays	8
4.1	Eindimensionale Arrays	9
4.2	Mehrdimensionale Arrays	9
4.3	Jagged Arrays	9
5	Attribute	11
5.1	Attribute mit Parametern	12
5.2	Anwendungsgebiete	13
6	Fazit	14

1 Einleitung

In C# gibt es einige interessante Neuerungen und spezielle Sprachelemente, die in dieser Seminararbeit näher vorgestellt werden sollen. Sie sind zum Teil aus gängigen Programmierstilen weiterentwickelt worden, siehe Properties Abschnitt 2 und in C# eingebunden. Teils sind es auch völlige Neuentwicklungen, wie im Falle der Attribute Abschnitt 5. Arrays sind zwar bekannte Sprachkonstrukte haben aber in C# einige nützliche Neuerungen erhalten siehe Arrays Abschnitt 4. Indexer sind wiederum eine erste fest in die Sprachkonstruktion C# eingebundene Anwendung des Propertykonstruktes.

2 Properties

Eigenschaftsfelder (engl. *properties*) stellen eine in C# verwirklichte neue Art der Programmierung von Klassenvariablenzugriffen dar. In diesem Abschnitt werden Properties in C# vorgestellt und ihre Verwendung an Beispielen erläutert. Properties in C# sind Felder also Members oder Elemente einer Klasse, die nach einer besonderen Vereinbarung unter Verwendung namentlich vordefinierter Methoden ein besonderes Verhalten hinsichtlich Ihrer Benutzung erhalten. Properties werden auch als "schlaue" Felder bezeichnet. Dadurch wird ausgedrückt, dass sie ausgelöst durch den Zugriffsoperationen, eigene Methoden zum Setzen oder zur Ausgabe von Daten aufrufen. Sie sind in C# eingeführt worden, weil das im

Programmieralltag häufig auftretende Problem des Eintragens bestimmter Werte in ein Datenelement, häufig gelöst durch Routinen, die Namen tragen wie `setdata`, `setdaten` oder ähnliches, eine Repräsentanz in der Programmiersprache erhalten sollten. Statt oben angeführter Klassen und Datentypen mit selbstbennannten Methoden zum Eintragen und Auslesen von Daten werden an einem Property zwei Methoden deklariert, die mit `set()` und `get()` benannt sind und den Zugriff auf andere nichtöffentliche Klassenfelder steuern können. Hieraus ergibt sich bereits eine Besonderheit eines jeden Properties, die Abschottung gegen Direktzugriffe von ausserhalb der Klasse. Also auf ein durch ein Property verwaltetes Feld darf unter keinen Umständen auf die bekannte

```
Klassenname.Klassenfeldname
```

Weise zugegriffen werden. Dies bedingt, dass die Deklaration eines Datenelementes, das durch ein Property verwaltet wird, mindestens `private` vielfach jedoch auch `protected` sein muss.

2.1 Beispiel

Den Ablauf der Benutzung eines Properties verdeutlicht ein erstes einfaches Beispiel. Die Methoden `get()` und `set()` enthalten den Code zur Zugriffsregulierung. Dies wird verdeutlicht an folgendem Code:

```
protected string bearbeiter;
public string Bearbeiter;
{
    get{...}
    set{...}
}
```

Eine Korrespondenz der Namen des eigentlichen manipulierten Klassenfeldes und des Propertyzugriffs ist nicht nötig. Es erleichtert jedoch die Verständlichkeit des Codes, wenn man sich an Form der Konvention in einem Softwareprojekt hält. An der Stelle der `...` steht dann der den Zugriff steuernde Code. Der könnte wie im folgenden Beispiel aussehen:

```
protected string bearbeiter;
public string Bearbeiter;
{
    get{
        if (sicherheitscheck_ok) return bearbeiter;
    }
    set{
        if (personalcheck_ok) bearbeiter=value;
    }
}
```

Die beiden booleschen Variablen `sicherheitscheck_ok` und `personalcheck_ok` stehen hier synonym für die Ausführung eines beliebigen selbstverfassten Codes. Jedoch für die konstruktive Benutzung eines Properties ist der wichtige Teil jeweils hinter den `if`-Bedingungen zu

sehen. In der get-Methode wird der Code untergebracht, der durch die Verwendung des Zugriffsoperators zur Ausführung kommt. Im vorgestellten Fall wird der Wert eines Klassenfeldes ausgegeben. Dies wird durch die Return Anweisung bewerkstelligt. In der set-Methode kommt der Code zur Ausführung, der über den Zugriffsoperator in der umgekehrten Richtung verwandt ausgelöst wird. In diesem Fall handelt es sich um das Setzen eines Klassenfeldes. Dies wird in der set-Methode über die Zuweisung des nicht explizit deklarierten Wertes value vorgenommen. Ausgelöst wird der Aufruf der get- und set-Methoden durch den Zugriffsoperator. Wie in der Abbildung 1 verdeutlicht ist.

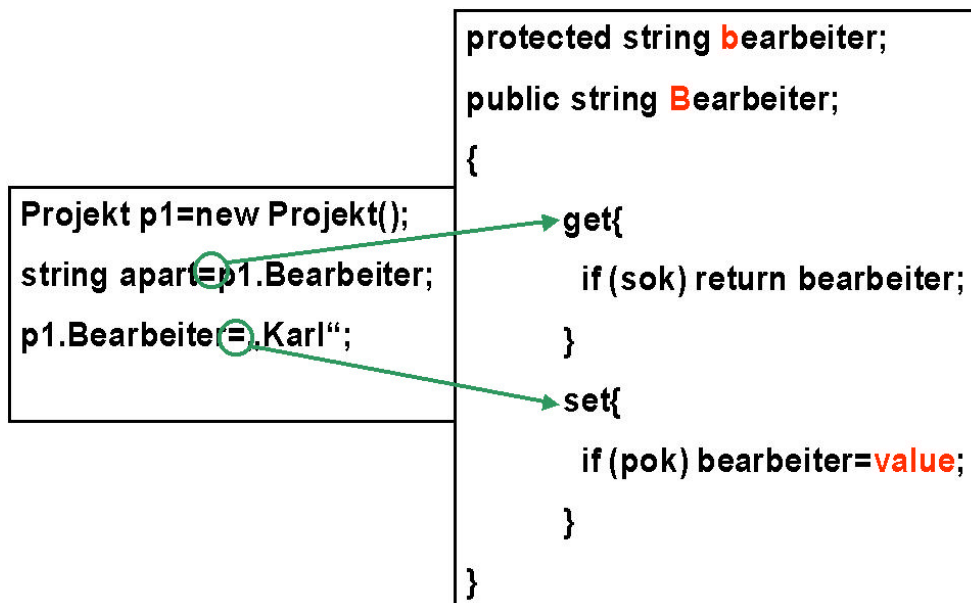


Abbildung 1: Aufruf der Methoden get und set eines Properties durch den Zugriffsoperator

Der linke Codeteil in Abbildung 1 befindet sich typischerweise im Aufrufteil, während der rechte Codeteil üblicherweise in der Klassendefinition angesiedelt ist. Es können also in aller Regel Abstände im Code zwischen den beiden hier nur schematisch dargestellten Segmenten liegen. Mit den gezeigten Beispielen ist der Aufbau der Anwendung von Properties verdeutlicht worden. Nun sollen sinnvolle Anwendungsmöglichkeiten aufgezeigt werden.

2.2 Anwendungsgebiete

Neben dem formalen Aufbau einer Propertydeklaration ist von besonderem Interesse die Andeutung möglicher Anwendungen. Die Benutzung von Properties in der im Abschnitt 2.1 vorgestellten Weise ist ein Standardanwendungsfall für in Klassen gekapselte Felder, aber Properties ermöglichen eine Vielzahl weiterer Anwendungen.

- **Statusvariablen**

Sie können dafür verwendet werden, komplexe Statusabfragen scheinbar in einer Variablen zu bündeln

- **Späte Initialisierung**

Bei der späten Initialisierung (engl. *lazy init*) wird die Eigenschaft der Properties ausgenutzt, dass ihre get- bzw. set-Methoden erst aufgerufen werden, wenn der Zugriffsoperator es auslöst. Ist es beispielsweise für das Auslesen eines bestimmten Properties notwendig, eine Verbindung zu einem entfernten Server aufzubauen, um an Online-Daten zu gelangen, so ist es möglich mittels eines Properties erst dann tatsächlich den zeit- und kostenintensiven Verbindungsaufbau zu einer serverbasierten Datenbank aufzubauen, wenn auch tatsächlich Daten übertragen werden. Die Alternative wäre bereits zum Programmstart mit einer Init-Routine die Serververbindung aufzubauen. Kosten könnten durch die nichtbenutzte aber aufgebaute Verbindung verursacht werden (Internet per Mobiltelefon o. ä.). Manche Programme starten zwar, greifen aber aus welchen Gründen auch immer, nicht auf den Server zu (Benutzerabbruch).

- **Schutz**

Unter diesem allgemein gehaltenen Punkt kann man die Zugriffskontrolle über Daten, die in einer bestimmten Form aufbereitet sein müssen, z.B. Schlüssel, fassen. Mit der Property-Methode kann leicht erreicht werden, dass nur ein exakt den Anforderungen an eine Indexschlüsselaufbau entsprechende Daten in ein durch Property-Zugriff geschütztes Feld vorgenommen werden können. Es gibt auch noch einfach die Möglichkeit durch Weglassen z.B. der set-Methode es überhaupt nicht zuzulassen, dass der Benutzer einer Klasse auf ein gewisses Datenelement zugreift.

- **Kontrolle**

Neben dem im vorherigen Punkt angesprochenen Schutz kann es ebenso sinnvoll sein zu Kontrollzwecken bei bestimmten Zugriffen beispielsweise Protokollskripte ablaufen zu lassen.

2.3 Besonderheiten

Als Besonderheiten für die Anwendung von Properties seien hier nur zwei Punkte angesprochen.

- **Vererbung**

Wird ein Property vererbt (engl. *inherited*), so erhält das ererbende Objekt das Property so wie es im Code des vererbenden Objektes deklariert wurde.

- **override**

Wird jedoch in ererbtem Code eine der beiden get- oder set-Methoden überschrieben (engl. *overridden*), so muss für den Fall, dass auch die andere Methode existiert, diese ebenfalls überschrieben werden. Diese Einschränkung wird an einem einfachen Beispiel deutlich. Eine vererbende Klasse besitzt ein Property mit get- und set-Methode. Ein ererbendes Objekt überschreibt nur die set-Methode mit dem für es selbst relevanten Code. Wenn nun von einem Zugriffsoperator die get-Methode aufgerufen wird besteht ein Zielkonflikt, nämlich dass unklar ist, ob durch die Überschreibung **nur** der set-Methode und das Weglassen der get- im ererbten Code absichtlich ein Schutz vor dem Zugriff eingebaut werden sollte, oder der Programmierer die get-Methode des Vaterobjektes benutzen wollte.

3 Indexer

In diesem Abschnitt wird das spezielle Sprachelement der Indexer vorgestellt und erläutert.

Unter einem indizierten Datenzugriff versteht man den Zugriff ein bestimmtes Datenobjekt einer Datenstruktur. Dabei identifiziert der sogenannte Index das Objekt eindeutig. Dieser Begriff stammt ursprünglich aus dem Bereich Datenbanken und bedeutet dort die Sortierung der Schlüssel der Datensätze bezüglich eines Indexfeldes. Bei gewissen Strukturen wie Listen Arrays wird mit dem Zugriff auf das i-te Elemente die Stelle festgelegt, an der sich das Datenobjekt befindet. Mit dem Sprachkonstrukt Indexer in C# kann eine solche Zugriffsmöglichkeit auch für nicht in Form von Listen oder Arrays strukturierter Daten geschaffen werden. Dies geschieht über die Benutzung eines Properties, das als Indexer verwendet wird. Die spezielle Handhabung eines Properties als Indexer erfolgt in der Überladung der [] Operatoren. Und zwar jeweils mit get()- und set()- Methode , wie schon im Abschnitt 2 vorgestellt.

3.1 Beispiel

Um einen Indexer zu deklarieren, muss man in einer Klasse selbst einen Codeabschnitt von folgender Form verwenden:

```
protected NList nodes=new NList();
public object this[int index]
{
    get{...}
    set{...}
}
```

Damit ist für die Klasse, in der sich dieser Codeteil befindet, über den this-Zeiger ein indizierter Zugriff deklariert. Dies geschieht indem der Klammeroperator [] wie ein Property deklariert wird. (Anmerkung in C++ kann ein ähnlicher Effekt durch das Überladen des Klammeroperators erreicht werden). Allgemein erlaubt die Deklaration eines Indexers es die Datenobjekte, für die ein Indexer deklariert wurde, wie ein Array benutzen zu können. Anwendung für die Konstruktion eines indizierten Zugriffs auf eine nicht sequentielle Datenstruktur kann sein, dass beispielsweise einfache Möglichkeiten bestehen für die Ausgabe aller Elemente einer Datenstruktur, die man sich wie folgt intuitiv vorstellen würde:

```
graph g=new graph();
int i=0;
do
{
    Console.WriteLine("besuche Knoten: {0}", g[i]);
    i++;
}
while(exists(g[i])==TRUE)
```

Mit diesem Code-Teil soll eine Graph unabhängig seiner Beschaffenheit in all seinen Knoten durchlaufen werden. Was hinter der Ausgabe der Knoten eines Graphen steckt, ist für den Anwender der Indizierung zunächst nicht relevant. Ihm ist nur wichtig dass nach jeder Indexerhöhung ein Knoten ausgegeben wird. Hinter der Indizierung steht in diesem Fall der Aufruf eines Properties nach einem Schema wie es in Abbildung 2

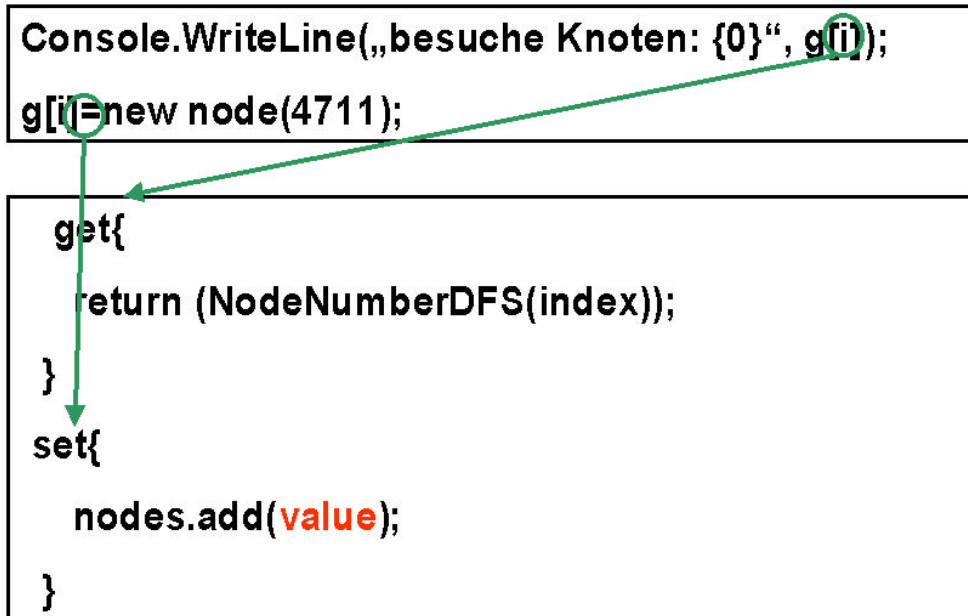


Abbildung 2: Aufruf der Methoden `get` und `set` eines Indexers durch den Klammeroperator

In der Abbildung 2 sind zwei Code-Teile dargestellt, die sich in weit voneinander entfernten Teilen des Programms befinden können. Der obere Teil befindet sich in einer die Klasse Graph verwendenden Routine und ruft somit den im unteren Code-Teil deklarierten Indexer auf. Der untere Code-Teil befindet sich in der Deklaration der Klasse Graph. Die Erzeugung des Indexwertes wird hier angedeutet durch eine Methode, die die Tiefensuchennummerierung des Graphen zugrunde legt. I

3.2 Anwendungsgebiete

Neben dem formalen Aufbau einer Indexerdekларation ist die Frage nach sinnvollen Anwendungen von besonderem Interesse. Die Benutzung von Indexern in der im Abschnitt 3.1 vorgestellten Weise ist ein formaler Anwendungsfall für die Aufzählung einer nicht in Listenform organisierten Datenstruktur. Für Indexer sind Anwendungen vor Allem auf folgenden Gebieten vorstellbar:

- **Automatische Bereichsabfragen**

Wird ein indizierter Zugriff auf Daten per Indexer verwirklicht und damit gewünscht, muss man die `get`- und `set`-Methoden des Indexers dafür benutzen nur Werte für den Index zuzulassen, die innerhalb der für die Daten definierten Bereiche liegen.

- **Sequenzen komplex strukturierter Daten**

In einer komplexen Datenstruktur kann es Bereiche geben, die wie eine Liste aufgebaut sind, diese können dann nachdem die entsprechende Stelle in der Datenstruktur lokalisiert wurde, mit Hilfe von Indexern aufgezählt werden. Beispiel hierfür sind die Intervalle zwischen zwei benachbarten Blättern eines Intervallsuchbaumes.

- **Aufzählung ohne internes Wissen**

Man benötigt, wenn Indexer verwendet werden, kein a priori Wissen darüber, wie eine Datenstruktur aufgebaut ist und welche Methoden sie beinhaltet, um Daten bereitzustellen. Man greift indiziert zu und erhält die Daten als Objekte der Reihenfolge die in der Indexerdeklaration festgelegt wurde.

3.3 Besonderheiten

Als Besonderheiten für die Anwendung von Indexern sind folgende Punkte zu erwähnen:

- **Scheinbare Sequentialität**

Programmteile, die Indexer benutzen wirken auf den ersten Blick linear, was sie aber nicht zwingend sein müssen. Für Aufwands- und Laufzeitanalysen ist hier immer der komplett durchlaufene Code zu beurteilen.

- **Laufzeiten**

Laufzeiten können nicht nur wie im vorherigen Punkt angesprochen falsch beurteilt werden, sie können sich auch erst durch die scheinbar bequeme Benutzung eines Indexers deutlich verschlechtern. Dies geschieht vor allem wenn durch die Verwendung des Indexers Programmteile, die zur Sortierung oder Suche dienen bei jedem indizierten Aufruf durchlaufen werden.

- **Ähnlichkeiten zum Java enumeration interface**

In Java existiert bereits ein enumeration interface, das eine ähnliche Arbeit wie ein Indexer leistet. Somit ist an C# neu, dass über die Property Verwendung ein Indexer als enumerierendes Element in die Sprachdefinition eingegangen ist. In Java wird ein vergleichbares Sprachkonstrukt über das Enumeration-Interface verwirklicht. Eine Klasse oder allgemein eine Datensammlung, die an das Enumeration Interface übergeben wird, kann von dem Interface mit der Methode

```
.netxElement( )
```

Element für Element aufgezählt werden. In dem Interface ist es nicht möglich, das i-te Element der Aufzählung direkt anzusprechen.

4 Arrays

In diesem Abschnitt werden die Benutzung und Besonderheiten von Arrays in C# vorgestellt. Indizierte Felder (engl. *arrays*) lassen sich in C# über die Grunddatentypen sowie deklarierte Typen definieren. Im Rahmen dieser Ausarbeitung wird beispielhaft mit Augenmerk auf die Benutzung von Arrays von Grundtypen eingegangen.

4.1 Eindimensionale Arrays

Die Benutzung von Arrays als Listen von Datentypen erfolgt wie aus den gängigen Programmiersprachen bekannt mit einem kleinen Unterschied, auf den bei der Abhandlung mehrdimensionaler Arrays eingegangen werden wird.

4.2 Mehrdimensionale Arrays

Ein mehrdimensionales Array in sogenannter Rechteckform wird unter C# folgendermassen deklariert. Im Beispiel erfolgt die Deklaration für zwei Dimensionen.

```
int [ , ] a;
```

Mit diesem Code-Teil wird ein Array namens a deklariert mit zwei Dimensionen. Sein Aussehen könnte man wie in Abbildung 3 verbildlichen. Bei der Deklaration jeglicher Arrays ist ein Unterschied zu anderen Programmiersprachen darin zu erkennen, dass die das Array deklarierenden Klammern direkt hinter dem Datentyp stehen und nicht hinter dem Bezeichner. Somit stellen sie ein Eigenschaft des Datentyps dar und nicht des Datums.

Ein vollständiges Beispiel für ein rechteckiges oder regelmässiges Array kann folgendes Aussehen haben:

```
int [ , ] a;  
a = new int[4,3];  
a[2,1] = 123;
```

Zu diesem Code-Teil dient die Abbildung 4 als Veranschaulichung.

	0	1	2
0			
1			
2			
3			

Abbildung 3: Rechteckiges zweidimensionales Array mit 3,4 Dateneinträgen

4.3 Jagged Arrays

Mit dem Begriff (engl. *jagged*) wird soviel ausgedrückt wie "gezackt" oder "unregelmässig". Dies beschreibt bildlich, wie eine veranschaulichte Darstellung eines jagged Arrays sich von der eines regelmässigen Arrays unterscheiden würde, nämlich durch die unterschiedlichen Längen der Dateneinträge. Die Deklaration eines zweidimensionalen jagged Arrays sieht im Code wie folgt aus:

```
int [ ][ ] a;
```

a	0	1	2
0			
1			
2		123	
3			

Abbildung 4: *Rechteckiges zweidimensionales Array mit 3,4 Dateneinträgen und eingetragenenem Datenelement*

	0	1	2
0			
1			
2			
3			

Abbildung 5: *jagged array mit 3,4,2 langen Einträgen in den Spalten*

Ein Beispiel für ein zweidimensionales jagged Array finden sie in Abbildung 5

Ein dynamisch erzeugtes jagged Array könnte Code von folgender Form verwenden:

```
int [ ][ ] a;
a=new int[3][ ];
for (int i=0; i<3;i++) a[i]=new int[i+1];
a[1][1]=123;
```

Der vorangegangene Code würde ein Bild wie in Abbildung 6 im Speicher erzeugen. Die Anwendung von Arrays folgt den bekannten Regeln für mehrfach indizierte Felder. Zu erwähnen sind jedoch bezogen auf die unregelmässigen Felder (engl. *jagged arrays*) einige Besonderheiten:

- **Rangabfragen zur Laufzeit**

Mit einer Rangabfrage zur Laufzeit (engl. *rank*) ist es möglich die Anzahl der Dimensionen eines Arrays herauszufinden. Das ermöglicht es einen Durchlauf eines Arrays in allen Dimensionen vorzunehmen, ohne seine Deklaration eines Arrays zu kennen.

- **Längenabfragen eines arrays**

Auf diesem Weg kann das Eintragen von Datenobjekten eines jagged Arrays abgearbeitet werden ohne seine Deklaration zu kennen. Dynamisch kann so auch auf die Veränderung der einzelnen jagged-Längen reagiert werden ohne Laufzeitfehler zu erzeugen.

a	0	1	2
0			
1		123	
2			

Abbildung 6: *jagged Array in Dreiecksform mit Dateneintrag*

5 Attribute

Attribute (engl. *attributes*) sind Zusatzinformationsfelder, die an Klassen oder Methoden angehängt werden können. Auf diese Weise sind Reaktionen auf bestimmte Attribute auch an weit entfernten Code-Stellen realisierbar. In dieser Ausarbeitung wird lediglich beispielhaft die Möglichkeit der Attributierung von Methoden vorgestellt. Die gezeigten Vorgehensweisen lassen sich ohne Abstriche auf die übrigen attributierbaren Objekte übertragen.

Allgemein wird ein Attribut durch unmittelbares Voranstellen im Code einem Objekt zugewiesen. Dies geschieht im Deklarationsteil des Objektes, das ein Attribut erhalten soll. Das Attribut selbst hat einen eigenen Attributdeklarationsteil, in dem festgelegt wird, dass es sich bei einem Objekt um ein Attribut handelt und welcher Code ausgeführt werden soll. Dieser Teil wird auch Attributkonstruktor genannt und verhält sich auch so, weil ein Attributobjekt von der Klasse `Attribute` abgeleitet ist, und somit im Wesentlichen einen Konstruktor deklariert. Dies umfasst auch wie mit Parametern zu verfahren ist. An jeder Stelle des Codes ausserhalb des Deklarationsteils des Objektes selbst kann auf das Attribut abgefragt und reagiert werden. Man kann dann je nach Attribut unterschiedliche Code-Teile zur Ausführung bringen. Zusätzlich zu den einfachen Attributen können auch noch ähnlich den Funktionen und Methoden Parameter mit den Attributen übergeben und gesetzt werden. Von besonderem Vorteil bei Parametern an Attributen ist es, dass ihre Anzahl nicht im Attributdeklarationsteil festgelegt werden muss. Somit führt die Tatsache, dass man einem Attribut einen Parameter mehr als ursprünglich geplant übergeben möchte, nicht dazu, dass man auch den Attributdeklarationsteil verändern muss. Es genügt an der gewünschten Stellen auf den zusätzlich übergebenen Parameter zu reagieren.

Ein Attribut wird bei der Benutzung in `[]` angegeben. Im vorgestellten Code-Beispiel wird das Attribut `[doc]` durch Ableitung von der Klasse `Attribute` deklariert. Dies könnte als Hinweis oder Flag auf eine Aufnahme der mit dem Attribut versehenen Methode in eine Online-Dokumentation verwendet werden. Der essentielle Code zur Zuweisung des Attributs `[doc]` könnte so aussehen:

```
public class docAttribute:Attribute
{...}
class AnyClass
{
    [doc]
    public void print(){...}
}
```

Der Bezeichner für ein Attribut befindet sich als Prefix vor dem Schlüsselwort Attribute. In unserem Fall hat der Bezeichner den Namen doc. Ein Verweis auf diesen Attributbezeichner befindet sich in der Klasse AnyClass durch die Angabe des Attributbezeichners in eckigen Klammern. In diesem Fall wird die Methode print mit dem Attribut [doc] versehen. Für einen Code-Teil, der die Klasse verwendet, in der die mit dem Attribut versehene Methode print liegt kann bei der Ausführung auf das Vorhandensein des Attributs doc reagieren. Schematisch wird eine mögliche Reaktion in Abbildung 7 dargestellt.

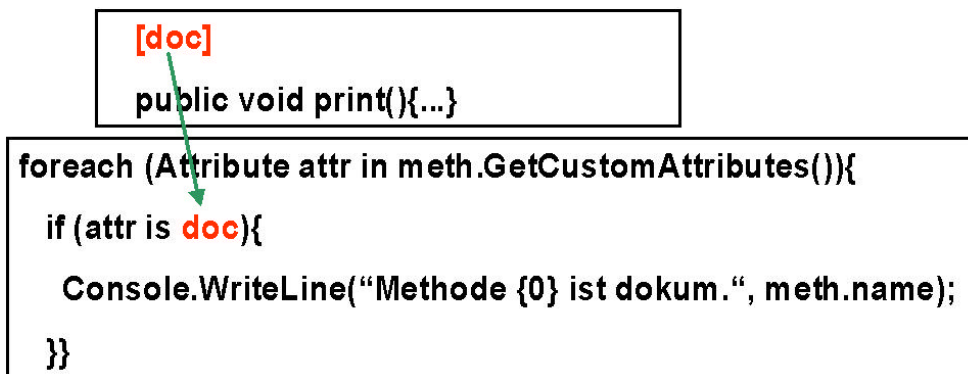


Abbildung 7: Reaktion auf das Attribut [doc] in einem die Klasse benutzenden Code-Teil

Die beiden Code-Teile befinden sich in aller Regel weit voneinander entfernt. Der obere Code-Teil befindet sich im Klassendeklarationsteil, während der untere Code-Teil in einer Anwendung angesiedelt sein könnte, die die im oberen Teil deklarierte Klasse benutzt und dann, wie dargestellt, alle Methoden dieser Klasse durchforstet und diese Methoden wiederum auf alle an ihnen haftenden Attribute untersucht. Wenn dann eine dieser Methoden das Attribut doc anhängt hat, wie im dargestellten Fall, dann ist der Anwender-Code in der Lage, wie durch die Ausgabe angedeutet darauf zu reagieren.

5.1 Attribute mit Parametern

Werden weitere Informationen an gewisse Attribute anzufügen gewünscht, so ist ein Weg dies zu realisieren, indem die zusätzlichen Informationen als Parameter an die Attribute gefügt werden. Es gibt zwei verschiedene Arten von Parametern: benannte und positionelle. Positionelle Parameter sind optionale Angaben und müssen nicht im Attributkonstruktor angegeben also "benannt" werden. Benannte Parameter sind eben solche, deren Angabe bei der Attributverwendung zwingend vorgeschrieben ist, sie werden im Attributkonstruktor benannt und ihre Verwendung programmiert. Attribute können über reine Benennung hinaus mit Parametern versehen werden. Die Parameter können ihrerseits benannt oder positionell sein. Benannte Parameter müssen in der Attributierung einer Methode, bezogen auf unseren Fall, angegeben werden, wohingegen noch eine unbestimmte Anzahl an positionellen Parametern, die beispielsweise Werte für Klassenfelder setzen mit angegeben werden können. Schematisch sieht eine solche Deklaration mit benannten und positionellen Parametern wie in Abbildung 8 dargestellt aus. Man beachte, dass die Vorschriften für die Verarbeitung von Parametern in der Attributdeklaration selbst vorgenommen werden müssen. Dieser Bereich befindet sich oberen Code-Teil. Der untere Code-Teil, auf den mit einem grünen Pfeil

verwiesen wird, ist der Attributierungsteil, der sich üblicherweise in einer Klassendeklaration befindet. Im Attributierungsteil muss auch der benannte Parameter übergeben werden. Auf einen weiteren positionellen Parameter, der in unserem Beispiel Klassenfeld heisst, wird zwar im Attributdeklarationsteil verwiesen, er ist aber positionell und fungiert daher sozusagen optional. Wenn er allerdings mit übergeben wird, dann geschieht das wie im folgenden Code-Teil dargestellt im Attributierungsteil:

```
public class docAttribute(string docpath):Attribute
{...}
class AnyClass
{
    [doc(globaldocpath, Klassenfeld=wert)]
    public void print(){...}
}
```

```
public class docAttribute(string docpath):Attribute{
    if(this.Klassenfeld==0)    this.Klassenfeld=default
    this.docpath=docpath,
}
{ [doc(globaldocpath)]
    public void print(){...}
}
```

Abbildung 8: Attribute mit benannten und positionellen Parametern

5.2 Anwendungsgebiete

Anwendungsgebiete für eine Neuerung wie Attribute sind in dieser Phase der Einführung C# noch nicht vollständig abzusehen. Aber es zeichnen sich einige geeignete Gebiete ab:

- **Dokumentation**

Attribute haben ein weites Einsatzgebiet zur Dokumentation. So können zum Beispiel per Attribut Methoden oder Klassen automatisch dokumentiert werden. Das geht dann auch noch per Attribut in verschiedenen Sprachen. Jedoch ist fraglich, ob eine Verdrängung automatischer Dokumentationssysteme in CVS-Umgebungen stattfinden wird. Attribute haben diesen Systemen mit teilweise starrer Syntax gegenüber den Vorteil erhöhter Flexibilität.

- **Markierung von Programmteilen**

Vorstellbar ist auch mit Attributen Programmcode zu markieren. Ihn mit dem Bearbeiter oder Autokommentaren zu versehen. Markierungen bergen die Möglichkeit, nur einen bestimmten Satz des gesamten Programms auszuführen zu lassen. Eine detaillierte Reaktion auf Umgebungsvariablen ist hier ebenso vorstellbar.

- **Wartung von Programmcode**

Wartung von Programmcode scheint auch ein lohnendes Einsatzgebiet für Attribute. Damit könnten Designstyles oder automatische Updates durchgeführt werden. Ein Attribut könnte steuern, wann sich eine Methode vom Installationsserver ein neues Update zu holen hätte.

6 Fazit

Eine neu auf den Markt gebrachte Programmiersprache ist immer auch auf Akzeptanz bei den Programmierern angewiesen, daher ist es wichtig den Entwicklern in einer möglichst großen Zahl an Wünschen bzgl. Vereinfachung der Programmier-Techniken entgegenzukommen. C# ist sicherlich auch der notwendige Versuch Microsofts verlorengegangenen Boden wiederzueroberen und die Lücke zur Linux-Welt zu verkleinern, die an Boden gewinnt. Neben dem wichtigen Faktor einer Plattformunabhängigkeit ist sicherlich der wichtigste Aspekt mit .Net zusammen erstmals eine integrierte Umgebung mit einer eigenen Programmiersprache C# abzurunden. Es ist seitens Microsoft auch ein erster, wenn auch zaghafter Schritt, Richtung Transparenz und Öffnung der Source-Codes

Die Vorgehensweise sich der Standardisierung eines weitestgehend unabhängigen Gremiums zu unterziehen, ist das für Microsoft wesentlich Neues. Diese Entscheidung gewährleistet aber erstmals bereits im Vorfeld Verlässlichkeit und Planbarkeit für **alle** Marktteilnehmer, die in der .Net Umgebung und mit C# arbeiten wollen.

Der Programmieralltag wird zeigen, ob C# mit den speziellen neuen Elementen ein so grosser Fortschritt ist, dass grosse Softwarepakete mit hohem Aufwand auf C# umgestellt werden, um die Neuerungen nutzen zu können. Denn plattformunabhängige Programmiersprachen gibt es schon, Indexer und Arrays auch, und ob Properties und Attribute als Neuerungen allein soviel Reiz für einen Wechsel hergeben bleibt abzuwarten. C# ist mit seinen neuen Sprachelementen eine konsequente Weiterentwicklung und Bündelung von sich abzeichnenden Veränderungen, aber ein programmiersprachlicher Quantensprung ist es nicht.

Literatur

- [1] C# Language Specification, ECMA Standard ECMA-334, Dec 2001, <http://www.ecma.ch/ecma1/STAND/ecma-334.htm>
- [2] Peter Drayton, Ben Albahari, Ted Neward: C# in a Nutshell, O'Reilly, 2002.
- [3] Jesse Liberty: Programming C#, O'Reilly, 2002
- [4] Jeffrey Richter: Microsoft .NET Framework Programmierung, Microsoft Press, 2002