



Universität Paderborn  
Fachbereich Mathematik–Informatik

---

## Seminar C# - Microsofts Antwort auf Java

### Seminararbeit Ereignisbehandlung - Events und Delegates

Ingo Höckenschnieder

#### Abstract

Dieser Bericht präsentiert, nach einer kurzen Einführung in die heutige Windowsprogrammierung und die Visual-Studio.NET Umgebung, die Themen *Delegates* und *Events* in der Programmiersprache C#. Bei einem *Delegate* handelt es sich um eine Art von objektorientiertem und typsicheren Funktionszeiger, der für synchrone und asynchrone Funktionsaufrufe genutzt werden kann. *Events* werden unter C# mit Hilfe von *Delegates* behandelt. Bei beiden handelt es sich um Sprachkonstrukte, die zusätzlich eine Klasse darstellen. Diese bis dahin recht ungewöhnliche Implementierung, erlaubt dem Programmierer eine relativ flexible und nicht sehr aufwendige Programmierung.

---

# 1 C# und Studio .Net

Die herkömmliche Entwicklung von Windows-Anwendungen wurde von einer ganzen Reihe von Problemen begleitet. In den letzten Jahren wurden vor allen Dingen vier verschiedene Sprachen mit sehr unterschiedlichen Möglichkeiten und Vorgehensweisen in der Programmierung eingesetzt. Viele Windowsanwendungen basieren auf der relativ alten Programmiersprache C und der Windows-API<sup>1</sup>; eine komplizierte Vorgehensweise, da C-Entwickler sich mit der manuellen Speicherverwaltung, einer komplizierten Zeigerarithmetik und diffizilen Syntaxkonstrukten herumschlagen müssen [1].

C++ ist zwar eine wesentliche Verbesserung von C, da über die alte Programmiersprache eine objektorientierte Schicht aufgesetzt wurde, trotzdem blieben wesentliche Nachteile von C erhalten.

Bei Visual Basic dagegen handelt es sich um eine einfache Möglichkeit Windows-Anwendungen zu erstellen, Programmierer konnten aber viele Dinge nur schwer mit Visual Basic realisieren, da auch diese Sprache nicht wirklich objektorientiert, sondern lediglich objektfähig ist. Größere Anwendungen z.B. mit Grafik oder Netzwerkkommunikation sind aus diesem Grund mit Visual Basic nur schwer zu realisieren.

Java bot eine komplett neue Programmieralternative, da es sich um eine vollständig objektorientierte Sprache handelt, deren Syntax an C++ angelehnt ist. Darüber hinaus ist Java plattformunabhängig. Java Programme werden in Bytecode übersetzt und dieser läuft dann auf der *Java Virtual Machine*, die für alle gängigen Computersysteme angeboten wird.

C# unter der Umgebung von Visual-Studio.Net ist eine neue Alternative, die Vorteile aus den vier oben aufgeführten Programmiermodellen bieten soll und eine wesentliche Vereinfachung der Programmierung darstellt.

C# ist eine unter Studio .Net neueingeführte Sprache, die im wesentlichen eine Weiterentwicklung von Java ist und an C angelehnt wurde. Sowohl C# wie auch Studio .Net selbst wurden neu entwickelt und sind komplett anders aufgebaut, als ihre Vorgänger.

## 1.1 Visual-Studio .NET

Bei der Visual-Studio.NET Umgebung handelt es sich um ein vollkommen neues Modell, bei dem fast alles anders funktioniert als bei der herkömmlichen Visual-Studio 6.0 Umgebung, so wird aus allen .NET fähigen Sprachen nicht plattform-

---

<sup>1</sup>Application Programming Interface, die von Windows angebotene Programmierschnittstelle

pezifischer ausführbarer Code, sondern die Zwischensprache *MSIL*<sup>2</sup> erzeugt, die ihrerseits plattformunabhängig ist.

Alle .NET fähigen Programmiersprachen erzeugen also einen gemeinsamen Code und benutzen die gemeinsame Laufzeitmaschine von .NET. Ein besonderer Aspekt dieser Laufzeitmaschine ist die wohldefinierte Typenmenge [2], die sprachübergreifende Vererbung, Ausnahmebehandlung und Fehlersuche.

Alle mit .NET erstellten Programme haben die gleichen Endungen (.EXE und .DLL) wie die klassischen COM-Binärdateien, die bisher unter Windows verwendet wurden, intern haben die .NET-Binärdateien allerdings keine Ähnlichkeit mit ihren Vorgängern [2].

## 1.2 C#

Bei C# handelt es sich um eine Programmiersprache die ganz neu für die .NET Umgebung entwickelt wurde. C# ist eine Kombination aus verschiedenen Sprachen, so wurde die Syntax teilweise von Java übernommen (so wie Java die Syntax teilweise von C++ übernahm). Genau wie in Java, ist eine C#-Klassendefinition in einer einzelnen Quellcodedatei (\*.cs) enthalten, im Gegensatz zu C++ wo eine Klassendefinition in eine einzelne Headerdatei (\*.h) und Implementierungsdateien (\*.cpp) aufgeteilt ist [3].

C# ist, wie Java, vollständig objektorientiert. Wie in Java gibt es keine mehrfache Vererbung mehr, stattdessen kann man nun *Interfaces* einbinden. Das Konstrukt *switch* wurde aus Visual Basic übernommen und kann nun ganze Strings und beliebige Objekte vergleichen.

C# greift auf Syntaxkonstrukte aus C++ zurück und bietet darüberhinaus einige neue Programmiermodelle, so daß C# syntaktisch so sauber wie Java, nicht ganz so einfach wie Visual Basic (aber deutlich einfacher als C++) und nahezu so leistungsfähig und flexibel wie C++ ist.

C# zeichnet sich dadurch aus, daß

- die ungeprüfte Zeigerarithmetik entfallen ist,
- es eine automatische Speicherverwaltung gibt,
- man im Gegensatz zu Java Operatoren für benutzerdefinierte Typen überschreiben kann,
- es echte Objektverweise gibt, die Programm- und Adressraum übergreifend übergeben und verarbeitet werden können.

---

<sup>2</sup>Microsoft Intermediate Language

---

Wie auch unter Java üblich, gibt es in C# nicht mehr die Möglichkeit globale Methoden oder Variablen zu definieren. Programme bestehen nur noch aus Klassen, nicht mehr aus globalen Methoden.

## 2 Delegates

Ein Delegate ist ein neues Sprachkonstrukt, welches in VisualStudio .NET eingeführt wurde (und nicht nur in C#, sondern unter anderem auch in der neuen Visual Basic Version zur Verfügung steht). Ein *Delegate* (also Delegat oder *Bevollmächtigter*) ist technisch gesehen eine normale Klasse, die von `System.MulticastDelegate` abgeleitet wird, allerdings wurde die Klasse als Konstrukt in die Sprache eingebaut, so gibt es neben der Klasse *Delegate* auch das Schlüsselwort **delegate**. Ein Delegate-Objekt kann auf ein (oder mehrere) bestimmte Methode(n) verweisen. Es enthält also den Verweis auf eine Methode eines bestimmten Objektes. Durch dieses Konstrukt wird der klassische Funktionszeiger ersetzt, den es z.B. unter C++ und einigen anderen Programmiersprachen gibt.

Der Vorteil eines Delegates ist, daß es sich um eine vollkommen objektorientierte und typsichere Verzeigerung handelt.[4]. Durch den Einsatz von Delegates, kann es dadurch zur Laufzeit keinen Typkonflikt geben, da in ein Delegate nur Methoden gepackt werden können, welche die gleichen Übergabeparameter haben, wie das Delegate selbst.

Für Delegates gibt es zwei wichtige Einsatzbereiche: Rückrufe (callbacks) und die Verarbeitung von Ereignissen [5].

### 2.1 Delegates als Funktionszeiger

Rückrufe spielen in der Programmierung eine große Rolle. Sie können z.B. zur Realisierung einer Programmschnittstelle oder von Ereignissen verwendet werden. Das Aufrufende Programm übergibt einen Funktionszeiger an das entsprechende Programm, das den Rückruf ausführen soll. Die wichtigste Eigenschaft der Delegates ist die Verknüpfung unabhängig entwickelter Programmteile. Durch die Übergabe eines Delegates kann einer Methode mitgeteilt werden, welche Funktionen sie aufrufen soll - die Funktionen, die im übergebenen Delegateobjekt verpackt sind. Dabei sind bei der Benutzung von Delegates zwei Varianten denkbar:

- **Asynchrone Aufrufe** eignen sich immer dann, wenn die Verarbeitung der aufgerufenen Methode eine größere Zeitspanne in Anspruch nimmt, währenddessen aber bereits andere Arbeiten vom aufrufenden Thread erledigt werden können.

Das kann beispielsweise ein Download in einem Browser sein. Der Anwender klickt auf eine Datei zum herunterladen, der Browser startet daraufhin

einen Downloadmanager, welcher die angegebene Datei aus dem Internet herunter lädt. Währenddessen kann der Benutzer den Browser ganz normal weiterverwenden. Ist der Download abgeschlossen, ruft der Downloadmanager den Browser zurück und informiert ihn über das Ergebnis (Abbildung 1).

- **Synchrone Aufrufe** sind immer dann von besonderem Interesse, wenn bis zum Eintreffen des Ergebnisses einer Routine nicht fortgefahren werden kann, statt einer fest eingebauten Routine jedoch eine beliebige benutzt werden können soll; Voraussetzung ist einzig die Äquivalenz der Schnittstelle. Beispielsweise kann ein Steuerelement (eine ListBox) eine Schnittstelle anbieten, so daß der Entwickler einer Software, der das Steuerelement benutzt, eigene Sortierfunktionen angeben kann (Abbildung 2).

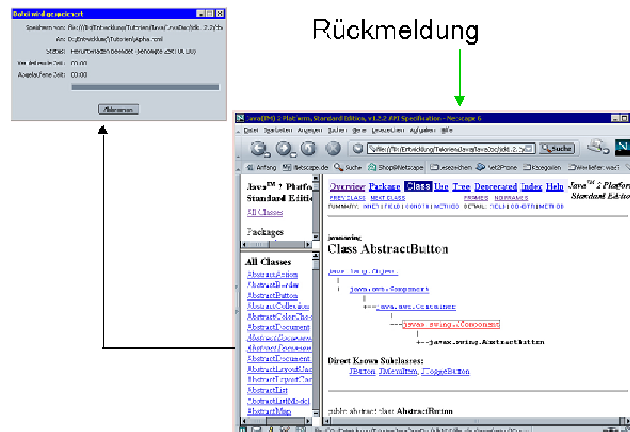


Abbildung 1: Asynchroner Aufruf

Mit Delegates lassen sich beide Variante realisieren. Wird im synchronen Fall ein Resultat erwartet, so wartet die aufrufende Methode auf das Ergebnis der Methode, die mit Hilfe des Delegates gestartet wurde. Im asynchronen Fall wird ein zweites Delegate übergeben, daß die Methode hält, die nach Beendigung der Arbeit aufgerufen werden soll, um das Ergebnis entgegen zu nehmen.

Ein weiterer Vorteil von Delegates ist, daß ein einzelnes Delegateobjekt mehrere Rückrufadressen halten kann. Dadurch kann man ein *Multicast* mit Hilfe des Delegateobjektes durchführen, also eine Reihe von Funktionsrückrufen an ein einzelnes Delegate binden - was ein großer Unterschied zur Benutzung von einfach mehreren Delegates wäre, die jeweils auf eine einzelne Methode verweisen [6]. Wird ein *Multicast*-Delegate verwendet, so kann man - wie bei einem einfachen

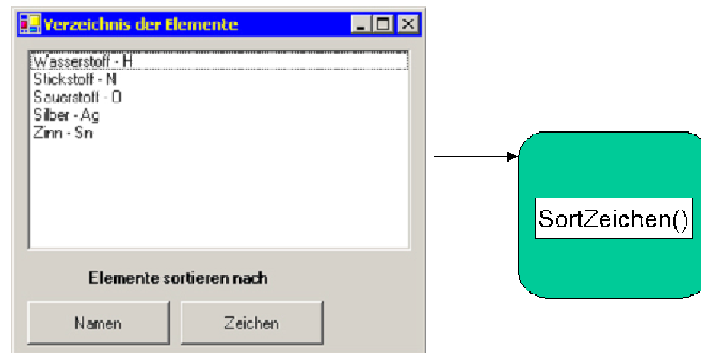


Abbildung 2: Synchroner Aufruf

Delegate - einen Wert von den Delegatefunktionen zurückgeben lassen, allerdings erhält man immer nur einen Rückgabewert, egal wie viele Methoden am *Multicast* beteiligt sind. Dies ist der Wert von der Funktion, die als letztes ausgelöst wurde. In allen Fällen (synchron oder asynchron, Einfach- oder Mehrfachaufruf), wird eine Delegateklasse von *System.MulticastDelegate* abgeleitet, bei der die Signatur eindeutig festgelegt wird. Dazu muß die Basisklasse nicht angegeben werden, dies geschieht automatisch durch das Schlüsselwort **delegate**:

```
delegate liste SortierDelegate (liste ListBoxElemente);
```

## 2.2 Arbeiten mit Delegates

Der Einsatz von Delegates in einem Programm erfolgt in drei Schritten.

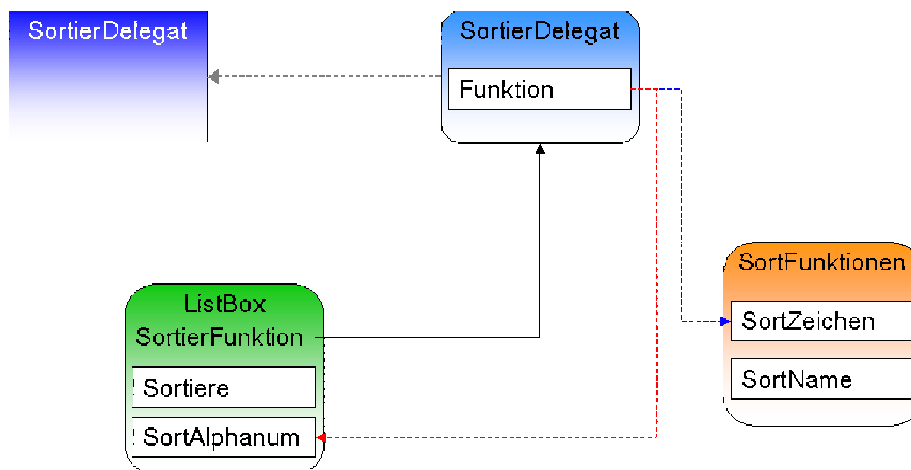
1. Deklaration einer abgeleiteten Delegateklasse
2. Instanziierung der Klasse als Delegateobjekt mit der Methode auf die verwiesen werden soll als Parameter
3. Aufruf des Delegateobjekts und dadurch indirekter Aufruf der in das Objekt verpackten Funktionen.

Bevor Delegates in ein Programm eingebunden werden, muß man sich Gedanken darüber machen, welche gemeinsame Schnittstelle all jene Methoden haben sollen, die von einem einzelnen Delegateobjekt übergeben werden können. Nur dann, wenn die Signatur einer Methode der Signatur des Delegates entspricht, kann das Delegateobjekt auf diese Methode verweisen; dazu gehört, daß sowohl Ereignistyp, wie auch Parametertypen identisch sind.

Der erste Schritt, das Ableiten einer Delegateklasse, erfolgt durch eine einzelne Kommandozeile. Das Schlüsselwort **delegate** erzeugt einen wohldefinierten Delegetypen, mit einer festgelegten Signatur. Dadurch wird eine Delegateklasse von der Basisklasse *System.MulticastDelegate* abgeleitet.

modifier **delegate** return-type *identifier* (formal-parameter-list);

Optional können als *Modifier* z.B. *public* für eine öffentliche Deklaration oder *private* für eine interne Deklaration benutzt werden. Der Ergebnistyp (*return-type*) ist entweder *void*, wenn es keinen Rückgabewert gibt, oder eine beliebige Klasse. Die Parameterliste ist optional, sie muß sich jedoch mit der Parameterliste der Methoden decken, die später in ein Delegateobjekt dieser Delegateklasse gepackt werden sollen.



**Abbildung 3:** Delegate als Funktionszeiger

Abbildung 3 zeigt ein Programmbeispiel, bei dem ein Delegate benutzt wird. Dabei handelt es sich um eine *ListBox*, deren *Sortierfunktion* frei gewählt werden kann. Die Variable *SortierFunktion* deren Typ die Delegateklasse *SortierDelegate* ist, instanziiert ein Delegateobjekt, das auf verschiedene Funktionen verweisen kann (z.B. *SortAlphanum* - Sortierung alphanumerisch, *SortZeichen* - Sortierung nach den Chemischen Zeichen der Periodentabelle, etc.). Wird die Funktion *Sortiere* der *ListBox* aufgerufen, so ruft sie ihrerseits die Funktion auf, welche im Delegateobjekt verpackt ist. Das kann die Funktion *SortAlphanum*, die in der *ListBox*-Klasse implementiert ist sein, oder eine Funktion aus einem anderen, unabhängigen entwickelten Programm, wie z.B. die Funktionen der Klasse *SortFunktionen*.

Die Ableitung einer Delegateklasse sähe für das Beispiel folgendermaßen aus:

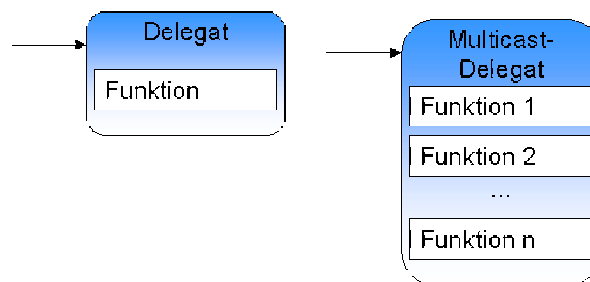
**delegate** liste *SortierDelegate* (liste *ListBoxElemente*);

Im zweiten Schritt können von der Delegateklasse nun Variablen angelegt und Objekte instanziiert werden. Bei der Instanziierung eines Delegateobjektes wird die Methode angegeben, auf die verzeigert werden soll; dabei handelt es sich um eine Methode eines bestimmten Objektes. Das Delegate enthält also auch die Referenz auf das Objekt, aus der die Methode stammt.

*DelegateKlasse* delegatevariable = **new** *DelegateKlasse*(methode);

*Multicast*-Delegates halten statt einer einzelnen Funktion eine Liste von Funktionen (siehe dazu Abbildung 4. werden auf die gleiche Weise erzeugt. Zusätzliche Methoden werden im Delegateobjekt verpackt, in dem man weitere Verweise „addiert“. Ebenso ist es möglich Verweise wieder zu entfernen.

```
multicastDelegate = delegate1 + delegate2;
multicastDelegate += delegate3;
multicastDelegate -= delegate2
```



**Abbildung 4:** einfaches Delegate und Multicast-Delegate

Die Methoden, die in einem Delegateobjekt verpackt sind, können durch den Aufruf des Delegateobjektes selbst (dritter Schritt), aufgerufen werden. Dabei müssen dem Delegate alle Parameter übergeben werden, die in der Signatur der Klasse festgelegt wurden.

delegatevariable(parameter-liste);

### 2.2.1 Beispiel: Einfache Delegates

Bereits in einem kleinen Programm kann man den Einsatz eines Delegates veranschaulichen (siehe Abbildung 3, der Programmcode wird in Abbildung 5 wiedergegeben). Dabei soll es sich um die Programmierschnittstelle einer *ListBox*

Steuerelement	<pre> <b>class</b> ListBox {     public <b>delegate</b> liste <i>SortierDelegat</i> (Liste ListBoxElemente);     public Liste Elemente;     ...     public Liste Sortiere(<i>SortierDelegat</i> SortierFunktion) {         Elemente = <i>SortierFunktion</i> (this.Elemente);     } } </pre>
Sortier-Methoden	<pre> <b>class</b> SortFunktionen {     public Liste SortZeichen(Liste ListBoxElemente) {         ...         return ListBoxElemente;     } } </pre>
Anwendung	<pre> ListBox listbox; public void SortiereListBox () {     listbox.<i>SortierDelegat</i> <i>SortierMethode</i> =         new ListBox.<i>SortierDelegat</i> (SortFunktionen.SortZeichen);     listbox.Sortiere(<i>SortierMethode</i>); } </pre>

**Abbildung 5:** Beispiel: Synchrones Delegate - Programmierschnittstelle

handeln, um neben den vorgegebenen Sortierverfahren, weitere eigene Algorithmen einbinden zu können ohne den Code des Steuerelements selbst zu verändern (siehe Abbildung 2). Das ist dann wichtig, wenn die ListBox universell einsetzbar sein soll und anderen Entwicklern zur Verfügung gestellt wird, welche die fertige Klasse in ihr Projekt einbinden können. Dazu wird eine Delegateklasse mit den Parametertypen und dem Ergebnistyp definiert. Im unseren Beispiel (Abbildung 5) wird dies durch die Deklaration

```
public delegate liste SortierDelegat (liste ListBoxElemente);
```

in der Klasse ListBox realisiert. Diese Delegateklasse stellt die Programmierschnittstelle zur Implementierung neuer Sortieralgorithmen dar. Als Parameter wird eine Liste angegeben, diese soll alle Elemente des Steuerelements, die ja sortiert werden sollen, enthalten. Als Ergebnis wird eine sortierte Liste zurückgegeben. Das Delegateobjekt selbst wird in der Anwendung, die das Steuerelement benutzt und die Sortierung anstoßen soll, angelegt. Als Verweis wird eine Sortiermethode einer dritten Klasse, die verschiedene Algorithmen mit der gleichen

Signatur enthält, angegeben. Diese Klasse kann unabhängig vom der ListBox entwickelt werden, einzige Voraussetzung für die Sortiermethoden ist, das deren Signatur mit der Delegateklassen-Signatur übereinstimmt. Beispielsweise könnte eine Methode *SortZeichen* implementiert werden, welche die Elemente der ListBox nach dem Chemischen Zeichen des Periodensystems sortiert.

```
ListBox.SortierDelegat SortierMethode = new ListBox.SortierDelegat (SortFunktionen.SortZeichen);
```

Sobald die Sortierung durchgeführt werden soll, wird die Methode *Sortiere* in der Klasse ListBox aufgerufen aus der Anwendung heraus aufgerufen. Die Methode akzeptiert ein Delegate und ruft es anschließend auf, wodurch die in das Delegateobjekt verpackte Methode gestartet wird.

```
ListBox.Sortiere(SortierMethode);
```

### 2.2.2 Beispiel: Multicast-Delegate

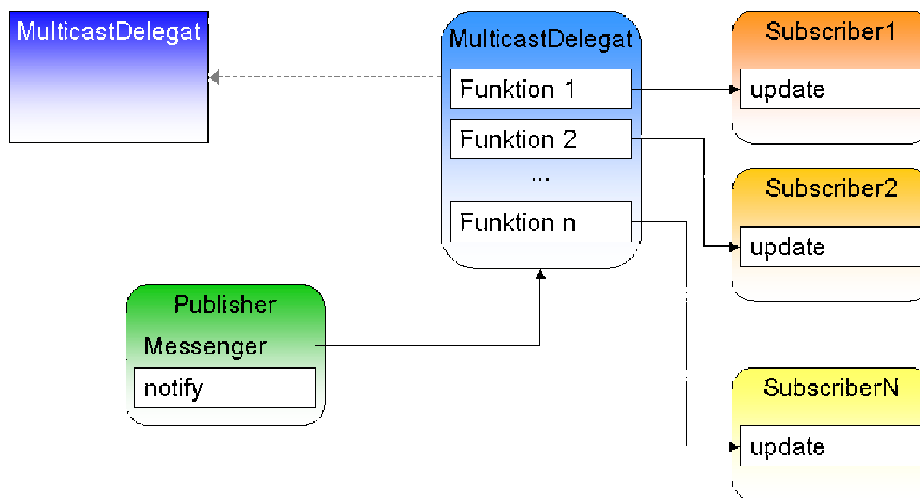


Abbildung 6: Ein Multicast-Delegate Beispiel

Möchte man statt einer einzelnen Funktion gleich mehrere aufrufen, so ist dies ebenfalls mit einem Delegate möglich, das statt der Referenz auf eine einzelne Methode, Referenzen auf verschiedene Methoden im gleichen oder in verschiedenen Objekten enthält. Diese sogenannten Multicast-Delegates enthalten eine Liste von Funktionen, die durch Aufruf des Delegates aufgerufen werden. Als Beispiel dient ein Publisher, der mehrere Beobachter (Subscriber) von einem Ereignis unterrichtet (siehe Abbildung 6).

Bei der Implementierung geht man zunächst wie beim einfachen Delegate vor und leitet eine Delegateklasse von *System.Multicast* ab.

```
public delegate void MulticastDelegate (string Message);
```

Bei der ersten Variante zur Erstellung eines *Multicast*-Delegates, wird für jeden Verweis eine Variable deklariert, bei der zweiten werden die Objekte direkt instanziiert zum *Multicast*-Delegate hinzugefügt.

Variante 1:

```
MulticastDelegate Messenger, Funktion1, Funktion2, ...;  
Funktion1 = new MulticastDelegate (Subscriber1);  
Funktion2 = new MulticastDelegate (Subscriber2);  
...  
MulticastDelegate = Funktion1 + Funktion2 + ...;
```

Variante 2:

```
MulticastDelegate Messenger;  
Messenger = new MulticastDelegate (Subscriber1);  
Messenger += new MulticastDelegate (Subscriber2);
```

*Multicast*-Delegates werden genauso aufgerufen wie ein einfaches Delegateobjekt, nur daß, statt einer einzelnen Methode, eine ganze Reihe von Methoden automatisch vom Delegateobjekt aufgerufen wird. Wurde bei der Ableitung der Delegateklasse ein anderer Ergebnistyp als void angegeben, so gibt der Aufruf des *Multicast*-Delegates das Ergebnis der Funktion zurück, deren Ausführung als letztes beendet wird. Alle anderen Ergebnisse werden ignoriert.

## 2.3 Delegates im Einsatz

Delegates können zu verschiedenen Zwecken eingesetzt werden, sowohl als asynchroner Rückruf, wie auch als Benutzererweiterung einer Komponente. Ein sinnvolles Einsatzgebiet für **synchrone Aufrufe** sind z.B. Steuerelemente, die in der Regel universell einsetzbar sein sollen. Sie werden meist so programmiert, daß sie von verschiedenen Programmen genutzt werden können. Entwickelt man z.B. eine *Listbox*<sup>3</sup>, könnte man zwei Sortiermethoden (z.B. alphabetisch und numerisch) implementieren, damit die Elemente geordnet ausgegeben werden können. Damit diese *Listbox* aber von allen Programmen genutzt werden kann, auch von jenen die Elemente anders sortiert haben möchten, kann man den Benutzern ermöglichen der *Listbox* eine Sortierfunktion zu übergeben. Die Sortierfunktion wird mit Hilfe eines Delegates verzeigert und hat eine fest definierte Signatur.

---

<sup>3</sup>Eine Textliste mit mehreren Einträgen

Anstelle einer numerischen Sortierung, kann ein Programmierer nun z.B. die Sortierung nach Wert (größte Zahlen vorn), nach Anzahl der Nachkommastellen oder Häufigkeit bestimmter Zeichen verwenden. Komponenten können so durch den Einsatz von Delegates ohne großen Aufwand flexibel gestaltet werden.

Auch für **asynchrone Aufrufe** lassen sich Delegates sinnvoll einsetzen. Spätestens seit der Einführung graphische Benutzeroberfläche kann ein Anwender viele Programme gleichzeitig ausführen. So kann man mit Hilfe eines Browsers Dateien aus dem Internet herunterladen. Während eine Datei heruntergeladen wird, kann man bereits andere Seiten betrachten. Ein Browser startet also ein internes oder externes Programm, daß den *Download* durchführt, während der Browser selbst ganz normal weiter genutzt werden kann. Ist das Herunterladen abgeschlossen informiert das Downloadprogramm den Browser über Erfolg oder Mißerfolg. Der Aufruf des Downloadprogrammes erfolgt also asynchron, denn der Browser wartet nicht auf die Rückmeldung (bzw. den Rückgabewert), der irgendwann später erfolgt.

### 2.3.1 Asynchrone Delegates

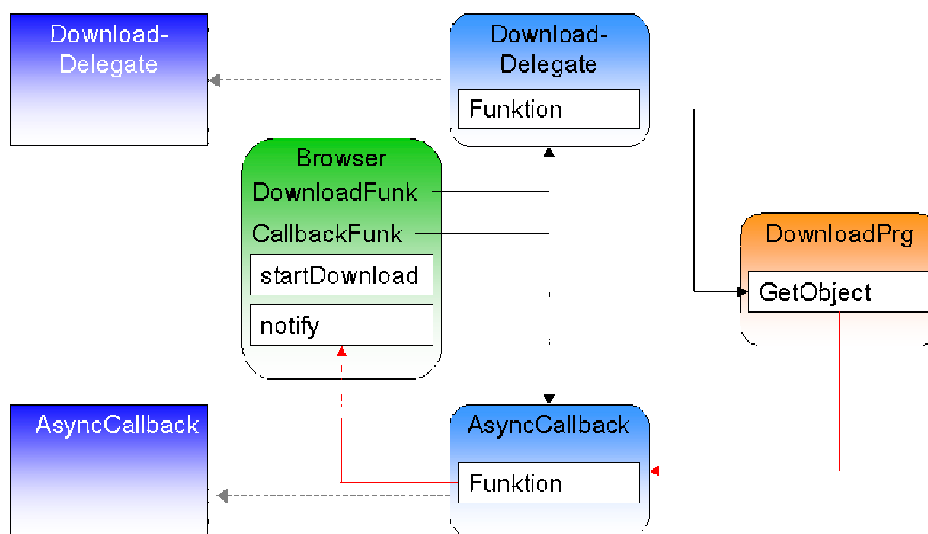


Abbildung 7: Ein asynchrones Delegate

Asynchrone Delegates bestehen eigentlich aus zwei Delegateobjekten, wobei das zweite an die Funktion übergeben wird, die das erste Delegateobjekt aufruft (siehe Abbildung 7). Die Deklaration der Delegateklasse und die Instanziierung des Delegatesobjekts sind zum synchronen Fall konform. Das Beispiel stellt einen

Browser da, der beim Herunterladen von Daten ein Downloadprogramm (Downloadmanager) asynchron startet.

```
public delegate int DownloadDelegate (string adresse);  
  
DownloadDelegate DownloadFunk = new DownloadDelegate (download.getObjekt);
```

Anders als bei synchronen Delegates muß nun noch ein zweites, asynchrones Delegateobjekt von der Klasse *AsyncCallback* instanziiert werden, dem man als Parameter die Rückruffunktion (in diesem Fall *notify*) übergibt. Diese Rückruffunktion wird dann aufgerufen, wenn die vom ersten Delegate referenzierte Methode beendet wird.

```
AsyncCallback CallbackFunk += new AsyncCallback(this.notify);
```

Soll der Downloadmanager gestartet werden, so muß man die von der Basisklasse *System.MulticastDelegate* implementierte Methode *BeginInvoke()* verwenden, um den asynchronen Rückruf durchzuführen. Zu den in der Delegateklasse definierten Parametern kommen zwei weitere; als erstes das asynchrone Rückrufdelegate, als zweites ein Statusobjekt<sup>4</sup>.

```
Object state = new Object();  
  
DownloadFunc.BeginInvoke(adresse, CallbackFunk, status);
```

In der Methode *notify*, die als einzigen Parameter eine Variable der Klasse *IAsyncResult* haben muß, muß der Rückruf abgefangen werden. Diese Klasse bietet Eigenschaften und Methoden, die einen darüber informieren, ob die Aufgerufene Funktion fehlerfrei durchgeführt wurde.

```
.   public void notify(IAsyncResult ergebnis) {  
.       int downloadStatus = DownloadFunk.EndInvoke(ergebnis);  
.   }
```

---

<sup>4</sup>Der Sinn des Statusobjektes wird weder in der Spezifikation, noch in der Onlinehilfe erläutert

## 3 Events

Seit der Entwicklung graphischer Benutzeroberflächen reicht die Interaktion nach dem E/A-Schema<sup>5</sup> nicht mehr aus. Moderne Oberflächen haben eine komplizierte und diffizile Interaktion mit dem Benutzer (und anderen Programmen), deren Reihenfolge nicht vom Programm vorhersehbar ist. Damit Oberflächen auf den Anwender reagieren können, benötigt man sogenannte *Ereignisse* oder *Events*, die immer dann ausgelöst werden, wenn ein Anwender agiert: das klicken auf einen Button, Texteingabe oder schließen eines Fensters. Aber auch Programme untereinander können mit Hilfe von Ereignissen kommunizieren und dadurch einer anderen Klasse oder Programm einen Statuswechsel melden. Wird zum Beispiel mit einem Internetbrowser eine Datei heruntergeladen, könnte das Downloadprogramm ein Ereignis *DownloadCompleted* besitzen.

Ereignisse werden von bestimmten Softwarekomponenten immer dann ausgelöst, wenn ein bestimmter Zustand eintritt. Wie ein anderes Programm, daß auf diese Ereignisse wartet, reagiert, ist dabei für die Anwendung, welche die Ereignisse auslöst, unwichtig. Der Knopf auf der Oberfläche meldet, daß er betätigt wurde, wie daraufhin (oder ob) das Programm an das die Meldung geht reagiert, ist für den Knopf nebensächlich.

### 3.1 Ereignisse in Java

Ereignisse in Java sind, im Gegensatz zu C# Objekte. Beim Java-Event-Modell<sup>6</sup> gibt es verschiedene Event-Klassen, die von verschiedenen Java-Klassen repräsentiert werden<sup>7</sup>. Die unterschiedlichen Events sind dabei von der Klasse *java.util.EventObject* abgeleitet. AWT Ereignisse sind wiederum Subklassen von *java.awt.AWTEvent*[8]. Das Quellobjekt eines Ereignisses kann mit Hilfe der Routine *getSource()* bestimmt werden. Das Java-Event-Modell basiert auf sogenannten *Event-Listnern*, Objekte, welche die Ereignisse abfangen sollen. Jede *Event-Quelle* pflegt eine Liste von Listnern, die über auftretende Ereignisse informiert werden sollen. Löst eine *Event-Quelle* ein Ereignis aus oder löst der Benutzer ein Ereignis in der *Event-Quelle* aus, so benachrichtigt diese alle entsprechenden Listener-Objekte, daß ein Ereignis eingetreten ist. Aus diesem Grund muß in ein Listener für einen Ereignistyp, das zugehörige *Interface* implementieren. Jedes Ereignis-Interface bietet die Methoden an, über die ein Listener-Objekt von der *Event-Quelle* informiert wird.

---

<sup>5</sup>Eingabe/Ausgabe-Schema: das Programm fragt - der Benutzer reagiert

<sup>6</sup>unter Java 1.1

<sup>7</sup>im Gegensatz zu Java 1.0, wo alle Ereignisse durch die Klasse *Event* repräsentiert werden

## 3.2 Events in C#

In C# wird die Ereignisbehandlung asynchron mit Hilfe von *Delegates* realisiert. Die ereignisauslösende Klasse leitet das Delegate von der Basisklasse ab, die empfangende Klasse, welche die ausgelösten Ereignisse abfangen möchte, instanziiert ein Objekt dieser abgeleiteten Delegateklasse. Diese Delegate nennen sich *Event-Handler*. Über sie werden die Ereignisse mit Hilfe des Schlüsselwortes **event** definiert.[6] *Event-Handler* geben in C# laut .NET Spezifikation *void* zurück und haben in der Regel zwei Parameter, laut ECMA sind aber auch Abweichungen von diesem Schema erlaubt[7]. Der erste Parameter ist die Quelle des Ereignisses, also das auslösende Objekt, der zweite Parameter ein von *EventArgs* abgeleitetes Objekt. Ereignisse gehören zur auslösenden Klasse (in der sie ja auch implementiert sind).

Die Deklaration des *Event-Handlers* erfolgt als Delegate.

```
delegate void EventHandler(object sender, System.EventArgs e);
```

Es wird ein *Interface* geschrieben, welches die Ereignisse mit Hilfe des *Event-Handlers* deklariert. Die Ereignisparameter müssen der Signatur des Delegates, also des *Event-Handler* entsprechen.

```
event EventHandlerName EreignisName (object sender, System.EventArgs e);
```

Außerdem werden im Interface weitere Methoden deklariert, die von der Klasse implementiert werden, welche das Interface einbinden und die Ereignisse auslösen soll.

In der Klasse, die das *Interface* einbindet, werden die Ereignis implementiert. Sobald diese Methode von der Klasse aufgerufen wird, wird das Ereignis ausgelöst. Die Implementation unterscheidet sich von der Deklaration, nur um die Möglichkeit einer zusätzlichen Angabe eines *Modifiers* (z.B. *public* oder *private*). Aus einer Methode der Klasse kann das Ereignis, wie jede andere Methode aufgerufen werden.

```
EreignisName(this, new System.EventArgs());
```

Eine anderes Programm (bzw. eine Klasse), welches die Klasse mit dem Ereignis einbindet, muß nun eine Abfangroutine implementieren, die durch das Ereignis aufgerufen werden soll, und das Ereignis auf diese Routine verzeigern. Die Abfangroutine muß die Signatur wie der *Event-Handler* haben.

```
void EreignisAbfangRoutine(object sender, System.EventArgs e) ...
```

Das Verweisen des Ereignisses auf die Abfangroutine erfolgt in zwei Schritten. Es wird ein Objekt vom *Event-Handler* abgeleitet und als jene Klasse, welche das

Interface implementiert und das Ereignis auslösen soll, instanziiert. Danach wird dem Ereignis der *Event-Handler* zugeordnet, als Parameter, muß die Abfangroutine übergeben werden.

```
EreignisInterface variablenName = new EreignisKlasse (<parameter>);  
variablenName.EreignisName += new Event-Handler(EreignisAbfangRoutine);
```

### 3.2.1 Ein einfaches Event-Beispiel

```
namespace EventTest1 {  
    public delegate void CounterEventHandler (object sender, System.EventArgs e);  
  
    interface CounterInterface {  
        event CounterEventHandler CounterFinished(object sender, System.EventArgs e);  
        void Add ();  
    }  
  
    class Counter : CounterInterface {  
        public event CounterEventHandler CounterFinished (object sender, System.EventArgs e);  
        ...  
        public void Start (int countUpTo) {  
            while (counterValue < countUpTo) {counterValue++;}  
            CounterFinished (this, new System.EventArgs());  
        }  
    }  
  
    class CounterTest {  
        static private void Finished (string msg) {  
            Console.WriteLine ("Ereignis Overflow ausgelöst: " + msg);  
        }  
  
        static void Main(string[] args) {  
            CounterInterface counter = new Counter();  
            counter.CounterFinished += new CounterEventHandler (Finished);  
            counter.Start(10000);  
        }  
    }  
}
```

Abbildung 8: Ein einfaches Event-Beispiel

Die Benutzung von Ereignissen wird an dieser Stelle anhand eines Counters erklärt, der von anderen Klassen eingebunden werden kann. Der Counter wird von einer anderen Klasse gestartet, zählt bis zu einem angegebenen Wert hoch und löst dann ein Ereignis aus (siehe Abbildung 8). Der *Event-Handler* wird von der Delegate-Basisklasse abgeleitet.

```
delegate void CounterEventHandler(object sender, EventArgs e);
```

Ein Interface deklariert die Ereignisse (Events) und Methoden, die vom Counter implementiert werden müssen. Die Ereignisse werden mit Hilfe des Schlüsselwortes **event** deklariert und als Typ, wird der *Event-Handler* angegeben.

```
event CounterEventHandler CounterFinished (object sender, EventArgs e);
```

Der Counter implementiert die Ereignisse und Methoden des Interfaces. Die Ereignisse werden durch den Aufruf aus einer der Methoden heraus ausgelöst.

```
CounterFinished(this, new System.EventArgs());
```

Die Klasse, die den Counter benutzt, instanziiert den Counter und weist dem Ereignis ein Delegate vom Typen des *Event-Handlers* zu, der auf die Methode verweist, die nach dem Auslösen der Ereignisses aufgerufen werden soll, sie muß die gleiche Signatur wie der *Event-Handler* und das Ereignis haben.

```
CounterInterface count = new Counter();
```

```
count.CounterFinished += new CounterEventHandler(Finished);
```

### 3.3 Zusammenfassung

Delegates sind Objekte, in denen ein oder mehrere Funktionen verpackt sind, die durch den Aufruf des Delegateobjektes synchron oder asynchron gestartet werden. Delegates eignen sich als synchrone Programmierschnittstelle, für den asynchronen Aufruf von Dienstleistern und zur asynchronen Behandlung von Ereignissen unter C#.

## Abbildungsverzeichnis

1	Asynchroner Aufruf . . . . .	5
2	Synchroner Aufruf . . . . .	6
3	Delegate als Funktionszeiger . . . . .	7
4	einfaches Delegate und Multicast-Delegate . . . . .	8
5	Beispiel: Synchrones Delegate - Programmierschnittstelle . . . . .	9
6	Ein <i>Multicast</i> -Delegate Beispiel . . . . .	10
7	Ein asynchrones Delegate . . . . .	12
8	Ein einfaches Event-Beispiel . . . . .	16

## Literatur

- [1] Andrew Troelsen  
*C# und die .NET-Plattform*  
Kapitel 1 Seite 30
- [2] Andrew Troelsen  
*C# und die .NET-Plattform*  
Die .NET Lösung Seite 32 ff
- [3] Andrew Troelsen  
*C# und die .NET-Plattform*  
Neuerungen durch C# Seite 34
- [4] Andrew Troelsen  
*C# und die .NET-Plattform*  
Kapitel 5 Seite 246-267
- [5] Tom Archer  
*Inside C#*  
Kapitel 14 - Delegates und Eventhandler Seite 253 - 267
- [6] J. Liberty  
*Programming C#*  
Chapter 12 - Delegates and Events Seite 263 - 292
- [7] ECMA  
*C# Language Specification*  
Abschnitt 8.7.5 Events
- [8] David Flanagan  
*Java in a Nutshell*  
Kapitel 7 - Events