Ablaufprotokollierung (1)

Bislang hatte ich dem Thema Ablaufprotokollierung eher wenig Aufmerksamkeit geschenkt. Meldungen per MessageBox, selbst gestrickte Protokolldateien – ja, natürlich. Aber das war es dann auch schon. Für meine Arbeit musste ich mich jetzt näher mit dem beschäftigen was das .NET Framework rund um dieses Thema so zu bieten hat. Und ich musste feststellen, dass ich mir wohl nie wieder Gedanken über selbst gestrickte Protokolldateien machen muss. So kam die Idee auf, das Thema in einer kleinen Blog-Reihe zu behandeln.

Überblick

Das Problem kennt jeder: die mit Mühen erstellte Anwendung (Klasse, Routine, …) macht irgendwie nicht (immer) das was sie sollte. Aber wo liegt der Fehler? Also Breakpoints setzen und dann Zeile für Zeile den Code debuggen, ab und an mal schauen, was so in den Variablen steht. Komisch, scheint doch alles richtig zu sein?! Also mal ein paar Nachrichtenfenster

```
MessageBox.Show("Hurz!");
```

an wichtigen Ablaufpunkten eingesetzt und das Ganze noch mal laufen lassen. Irgendwann ist dann der Fehler eingegrenzt, gefunden und behoben. Wieder laufen lassen; alles in Ordnung! Prima, dann noch die Nachrichtenfenster entfernen und fertig.

Mal abgesehen davon, dass diese Methode die Gefahr birgt, dass man beim Entfernen der kleinen hilfreichen Nachrichtenfenster mal eines vergisst, welches dann irgendwann den Anwender fröhlich begrüßt (und sicherlich ganz schnell nervt). Wie ermittelt man die Ursache, wenn Probleme erst beim Endanwender auftreten? Oder was tun, wenn die zu prüfende Datenmenge einfach viel zu komplex/umfangreich ist, als dass man sie einfach mal auf dem Bildschirm ausgeben könnte? Man wird wohl einfach aktuelle Werte oder sonstige Informationen in Textdateien schreiben (oder auf die Konsole, oder in das Windows Eventlog, oder, oder, oder) und somit den Ablauf des Programms protokollieren…

Wozu aber das Rad neu erfinden, wenn uns das .NET Framework genau dafür bereits vorgefertigte Lösungen in Form der Klassen **Trace** und **Debug** liefert?

Trace und Debug

Die beiden Klassen findet man im Namespace *System.Diagnostics*. Die Funktionalität der Klassen ist nahezu identisch, der wesentliche Unterschied ist, dass Debug nur dann Ausgaben erzeugt wenn die #DEBUG Konstante definiert ist und Trace nur dann wenn die #TRACE Konstante definiert ist. Im Standard sind für die Build Konfiguration Debug beide Konstanten definiert, für Release aber nur #TRACE. Sie können dies in den Projekteigenschaften auf dem Reiter *Erstellen* einstellen. Im Folgenden gehen wir von der Standardeinstellung aus.

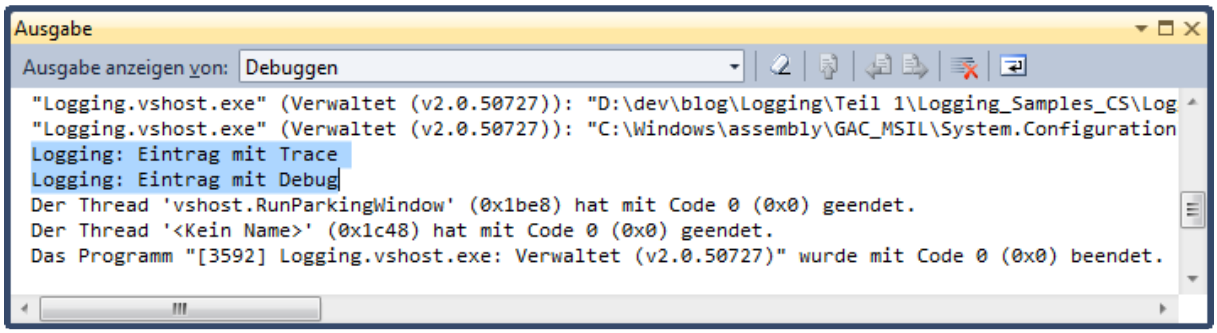
Ein einfaches Beispiel

Um einen ersten Eindruck zu bekommen, erstellen wir eine einfache Konsolenanwendung. Dabei sieht unsere *program.cs* wie folgt aus:

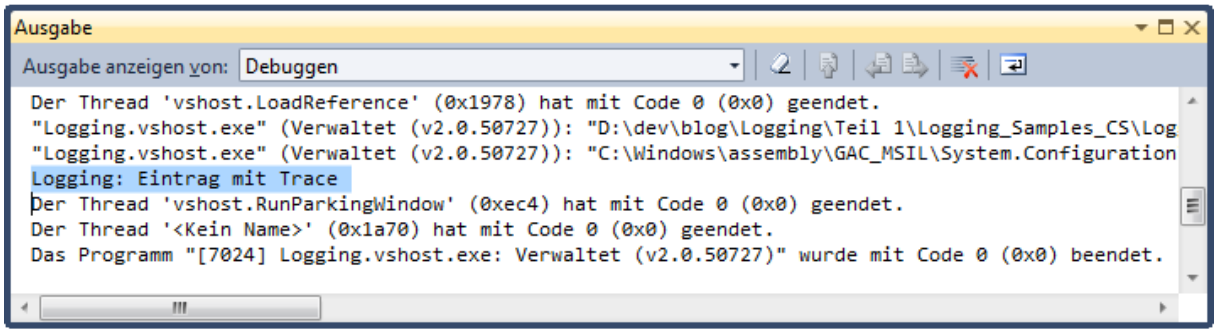
```
01 using System;
02 using System.Diagnostics;
03
04 namespace Logging
05 {
06     class Program
07     {
```

```
08          static void Main(string[] args)
09          {
10              Console.WriteLine("Beispiel 1");
11
12              Trace.WriteLine("Logging: Eintrag mit Trace");
13              Debug.WriteLine("Logging: Eintrag mit Debug");
14
15              Console.ReadLine();
16          }
17      }
18  }
```

Wenn wir die Anwendung nun in der Entwicklungsumgebung starten, erhalten wir je nach Build Konfiguration im Ausgabefenster die folgende Ausgabe



bzw.



Innerhalb der Entwicklungsumgebung können wir nun unsere Informationen verfolgen. Um außerhalb der Entwicklungsumgebung mit Trace und Debug zu arbeiten, müssen wir aber einen Mechanismus einrichten, der die gesendeten Meldungen an das gewünschte Ausgabemedium leitet.

Der Einfachheit halber werde ich in den nachfolgenden Beispielen nur noch die Ausgabe über Trace verwenden. Somit ist die Ausgabe unabhängig von der eingestellten Build Konfiguration.

TraceListener

Ablaufverfolgungsmeldungen werden von sogenannten TraceListenern empfangen. Dieser sogt dann dafür, dass die empfangenen Meldungen an ein entsprechendes Ausgabeziel weitergeleitet werden. Dies können Dateien, das Konsolenfenster oder auch Datenbanken sein. Wenn nötig formatiert der TraceListener auch die Ausgabe.

Obwohl nicht direkt ersichtlich benutzt auch unser erstes Beispiel einen TraceListener, nämlich einen vom Typ DefaultTraceListener. Dieser gibt die mit Write oder WriteLine gesendeten Meldungen über die Windows-API-Funktion OutputDebugString über den angeschlossenen Debugger aus - in unserem Fall das Ausgabefenster des Entwicklungsumgebung. Der DefaultTraceListener wird als einziger Listener automatisch in alle Listener-Ausflistungen eingetragen.

Wichtig: Debug und Trace nutzen eine gemeinsame Listeners-Auflistung, d. h. ein in eine

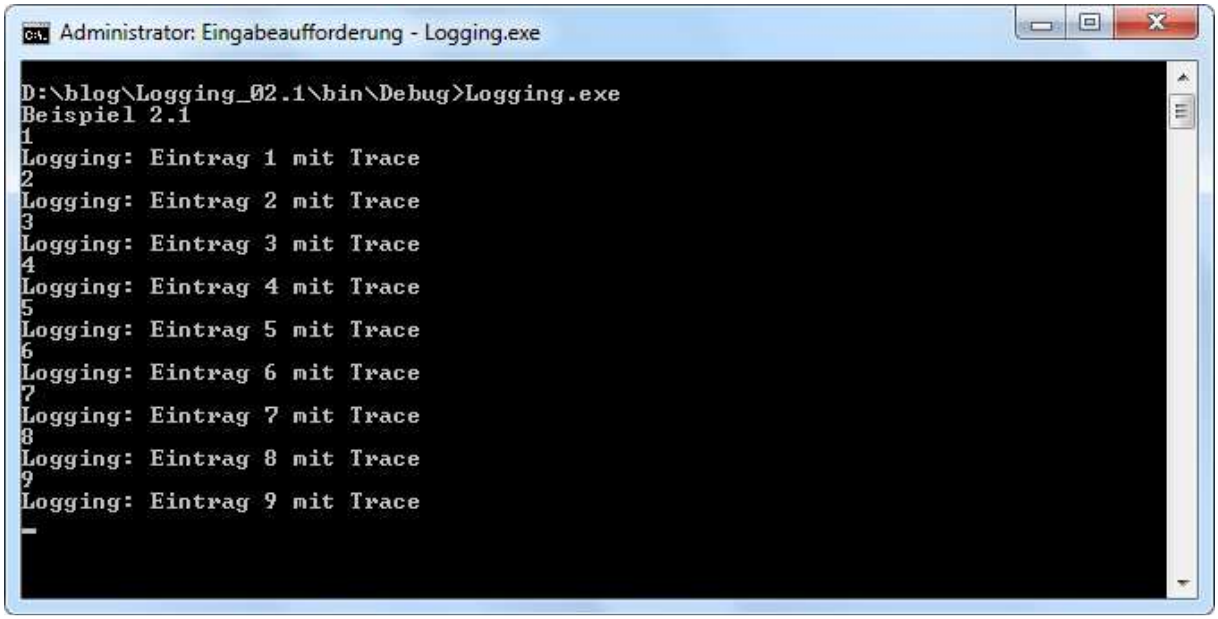
Debug.Listeners-Auflistung eingefügter Listener wird auch zur Trace.Listeners- Auflistung hinzugefügt und umgekehrt.

Ausgabe im Konsolenfenster

Bei unserer kleinen Konsolenanwendung bietet es sich an, die Trace/Debug-Meldungen auch auf der Konsole auszugeben. Dazu verwenden wir einen ConsoleTraceListener. Das Programm ändert sich wie folgt:

```
01  using System;
02  using System.Diagnostics;
03
04  namespace Logging
05  {
06      class Program
07      {
08          static void Main(string[] args)
09          {
10              ConsoleTraceListener listener = new ConsoleTraceListener();
11              Trace.Listeners.Add(listener);
12
13              Console.WriteLine("Beispiel 2.1");
14
15              for (int i = 1; i < 10; i++)
16              {
17                  Console.WriteLine(i);
18                  Trace.WriteLine(string.Format("Logging: Eintrag {0} mit Trace",
    i));
19              }
20              Console.ReadLine();
21          }
22      }
23  }
```

Wird das Programm gestartet, erhalten wir folgende Ausgabe:

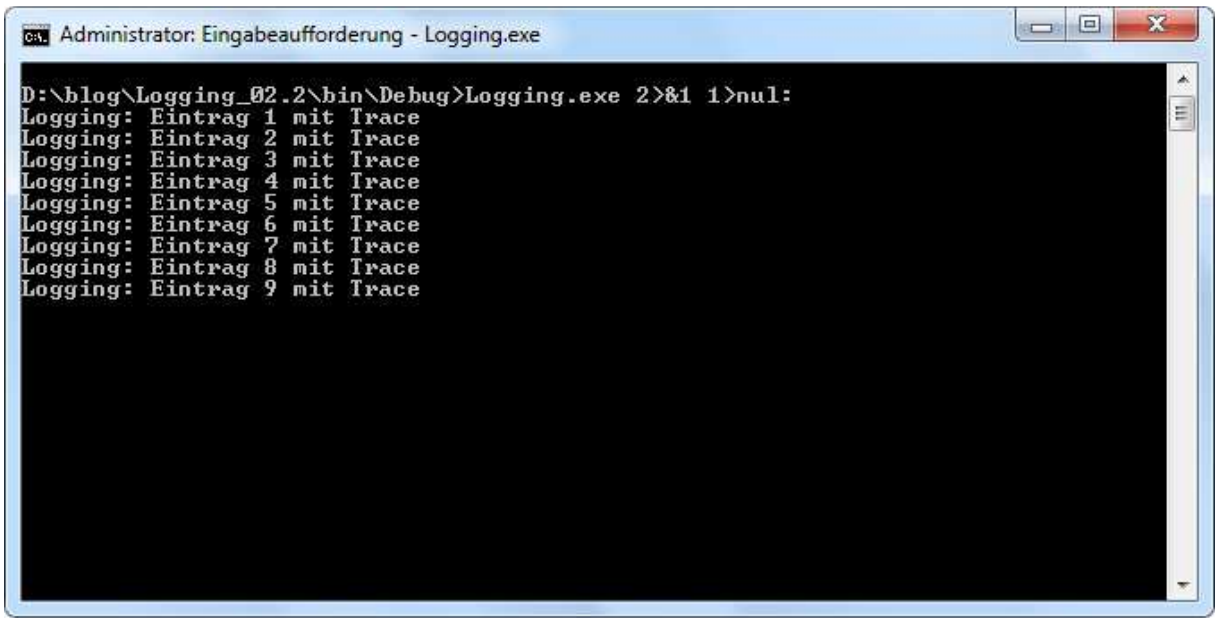


Zudem finden wir die Meldungen auch im Ausgabefenster.

Dadurch, dass Konsolenausgabe und Trace-Meldungen bunt gemischt auf der Konsole ausgegeben werden, ist diese Form nicht gerade übersichtlich. Zudem ist nicht zwingend zu erkennen, welche

Ausgaben denn gewünschte Programmausgaben und welche Trace- Meldungen sind. Daher kann der ConsoleTraceListener seine Ausgabe auch auf dem Fehlerkanal der Konsole ausgeben. Dazu muss nur beim Erstellen des Listeners der optionale Parameter useErrorStream angegeben und auf True gesetzt werden.

Jetzt kann die Trace-Ausgabe z. B. in eine Datei umgeleitet werden. Der nachfolgende Konsolenaufruf blendet z. B. die Standardausgabe aus und zeigt nur die Trace- Meldungen.



Direkte Ausgabe in eine Datei

Die Ausgabe auf die Konsole mag für Konsolenanwendungen ja noch sinnvoll sein, aber spätestens wenn das Programm nicht in einem Konsolenfenster läuft, müssen die Ausgaben irgendwie gesammelt werden. Die nächstliegende (und wahrscheinlich auch am weitesten verbreitete) Form ist sicherlich die Ausgabe in eine Datei.

Auch dafür gibt es einen vorgefertigten TraceListener, den TextWriterTraceListener.

```
01  using System;
02  using System.Diagnostics;
03
04  namespace Logging
05  {
06      class Program
07      {
08          static void Main(string[] args)
09          {
10              TextWriterTraceListener listener = new
    TextWriterTraceListener("trace.log");
11              Trace.Listeners.Add(listener);
12              Trace.Listeners.Remove("Default");
13
14              Console.WriteLine("Beispiel 3.1");
15
16              Trace.WriteLine("Beispiel 3.1 gestartet: " +
    DateTime.Now.ToString());
17              for (int i = 1; i < 10; i++)
18              {
19              Console.WriteLine(i);
20              Trace.WriteLine(string.Format("Logging: Eintrag {0} mit
    Trace", i));
```

```
21              }
22              Console.ReadLine();
23              Trace.Flush();
24          }
25      }
26  }
```

In diesem Beispiel werden die Meldungen nur in die Datei geschrieben und nicht wie bisher auch in das Ausgabefenster, da der Standard-Listener aus der Liste entfernt wurde.

Wichtig ist, dass nach der zuletzt abgesetzten Meldung ein Trace.Flush erfolgt, da sonst nicht alle Daten in die Datei geschrieben werden. Wenn Sie möchten, dass die Trace- Meldungen sofort in die Datei geschrieben werden und nicht nach jedem Trace-Aufruf ein Trace.Flush platzieren möchten (kann man ja mal vergessen), können Sie mit

```
Trace.AutoFlush = true;
```

auch dafür sorgen, dass nach jedem Trace Schreibzugriff automatisch ein Trace.Flush erfolgt.

Nach dem ersten Ausführen des Programms finden Sie je nach Build Konfiguration im Ordner *bin/Debug* oder *bin/Release* die Datei *trace.log*. Diese sieht wie folgt aus

```
Beispiel 3.1 gestartet: 22.05.2013 18:34:17
Logging: Eintrag 1 mit Trace
Logging: Eintrag 2 mit Trace
Logging: Eintrag 3 mit Trace
Logging: Eintrag 4 mit Trace
Logging: Eintrag 5 mit Trace
Logging: Eintrag 6 mit Trace
Logging: Eintrag 7 mit Trace
Logging: Eintrag 8 mit Trace
Logging: Eintrag 9 mit Trace
```

Führen Sie das Programm erneut aus, werden die Einträge am Ende der vorhandenen Datei angefügt.

```
Beispiel 3.1 gestartet: 22.05.2013 19:34:17
Logging: Eintrag 1 mit Trace
Logging: Eintrag 2 mit Trace
Logging: Eintrag 3 mit Trace
Logging: Eintrag 4 mit Trace
Logging: Eintrag 5 mit Trace
Logging: Eintrag 6 mit Trace
Logging: Eintrag 7 mit Trace
Logging: Eintrag 8 mit Trace
Logging: Eintrag 9 mit Trace
Beispiel 3.1 gestartet: 22.05.2013 19:37:39
Logging: Eintrag 1 mit Trace
Logging: Eintrag 2 mit Trace
Logging: Eintrag 3 mit Trace
Logging: Eintrag 4 mit Trace
Logging: Eintrag 5 mit Trace
Logging: Eintrag 6 mit Trace
Logging: Eintrag 7 mit Trace
Logging: Eintrag 8 mit Trace
Logging: Eintrag 9 mit Trace
```

Schalter für die Ablaufprotokollierung

Eine weitere Möglichkeit die Ablaufprotokollierung zu beeinflussen sind Schalter, die durch die beiden Schalterklassen BooleanSwitch und TraceSwitch zur Verfügung gestellt werden.

Die Schalterklasse BooleanSwitch kann dabei am ehesten mit einem An-/Ausschalter verglichen werden, die Ausgaben sind also entweder aktiviert oder deaktiviert.

```
01  using System;
02  using System.Diagnostics;
03
04  namespace Logging
05  {
06      class Program
07      {
08          static void Main(string[] args)
09          {
10              TextWriterTraceListener listener = new
    TextWriterTraceListener("trace.log");
11              Trace.Listeners.Add(listener);
12              Trace.Listeners.Remove("Default");
13              Trace.AutoFlush = true;
14
15              Console.WriteLine("Beispiel 4");
16
17              BooleanSwitch traceSwitch = new BooleanSwitch("MeinSchalter",
    "Ablaufprotokollierung mit Schalter");
18
19              traceSwitch.Enabled = true;
20
21              Trace.WriteLineIf(traceSwitch.Enabled, "Beispiel 4 gestartet: " +
    DateTime.Now.ToString());
22              for (int i = 1; i < 10; i++)
23              {
24                  Console.WriteLine(i);
25                  traceSwitch.Enabled = (i % 2 == 1);
26                  Trace.WriteLineIf(traceSwitch.Enabled, string.Format("Logging:
    Eintrag {0} mit Trace", i), "Kageorie");
27              }
28              Console.ReadLine();
29          }
30      }
31  }
```

Der Konstruktor der Schalterklasse erwartet lediglich eine Bezeichnung, sowie eine Beschreibung. Optional kann der Initialwert angegeben werden. Um die Ausgabe nun Abhängig vom Schalterwert zu beeinflussen bieten sich die Methoden Trace.WriteIf bzw. Trace.WriteIf an, aber natürlich kann auch ein Trace.WriteLine in einen if-Block verschachtelt werden.

Die Protokolldatei sieht dann wie folgt aus:

```
Beispiel 4 gestartet: 22.05.2013 19:47:30
Kageorie: Logging: Eintrag 1 mit Trace
Kageorie: Logging: Eintrag 3 mit Trace
Kageorie: Logging: Eintrag 5 mit Trace
Kageorie: Logging: Eintrag 7 mit Trace
Kageorie: Logging: Eintrag 9 mit Trace
```

Will man die Trace-Ausgabe hingegen nach Informationsebenen steuern, verwendet man ein TraceSwitch-Objekt. Die Eigenschaft Level, die vom Enumerationstyp TraceLevel ist, bestimmt dabei welche Informationen zur Ausgabe gelangen. Dabei gilt, dass eine
höhere Ebene immer aus die Informationen aller niedrigeren Ebenen anzeigt. Die nachfolgende Tabelle zeigt die möglichen Werte.

Konstante	Wert	Art der ausgegebenen Meldungen
Off	0	Keine
Error	1	Fehlermeldungen
Warning	2	Fehler- und Warnmeldungen
Info	3	Fehler-, Warn- und Informationsmeldungen
Verbose	4	Alle Meldungen

Der Konstruktor des TraceSwitch-Objekts ist identisch zum BooleanSwitch, auch hier werden nur Name, Beschreibung und optional der Startwert des TraceLevels benötigt.

Die TraceSwitch-Klasse besitzt mit den Methoden

- TraceError
- TraceWarning
- TraceInfo
- TraceVerbose

vier Eigenschaften die je nach Schalterstellung True oder False zurückgeben und an WriteIf bzw. WriteLineIf übergeben werden können. Auch hier wird berücksichtigt, dass höhere Trace-Level niedrigere Level enthalten. Hat z. B. Level den Wert TraceLevel.Info, so haben TraceError, TraceWarning und TraceInfo den Wert True und TraceVerbose den Wert False. Beim Wert TraceLevel.Off liefern alle vier Eigenschaften den Wert False.

Das nachfolgende Beispiel zeigt alle Ausgaben bis einschließlich Trace-Level Info.

```
01  using System;
02  using System.Diagnostics;
03
04  namespace Logging
05  {
06      class Program
07      {
08          static void Main(string[] args)
09          {
10              TextWriterTraceListener listener = new
    TextWriterTraceListener("trace.log");
11              Trace.Listeners.Add(listener);
12              Trace.Listeners.Remove("Default");
13              Trace.AutoFlush = true;
14
15              Console.WriteLine("Beispiel 5");
16
17              TraceSwitch traceSwitch = new TraceSwitch("MeinSchalter",
    "Ablaufprotokollierung mit Schalter");
18              traceSwitch.Level = TraceLevel.Info;
19
20              Trace.WriteLine("Beispiel 4 gestartet: " + DateTime.Now.ToString());
21
22              Trace.WriteLineIf(traceSwitch.TraceError, "Logging: Eintrag mit
    TraceLevel Error");
23              Trace.WriteLineIf(traceSwitch.TraceWarning, "Logging: Eintrag mit
    TraceLevel Warning");
24              Trace.WriteLineIf(traceSwitch.TraceInfo, "Logging: Eintrag mit
    TraceLevel Info");
25              Trace.WriteLineIf(traceSwitch.TraceVerbose, "Logging: Eintrag mit
    TraceLevel Verbose");
26
27              Console.ReadLine();
28          }
29      }
```

```
30  }
31
```

Hier die Protokolldatei:

```
Beispiel 5 gestartet: 22.05.2013 19:48:28
Logging: Eintrag mit TraceLevel Error
Logging: Eintrag mit TraceLevel Warning
Logging: Eintrag mit TraceLevel Info
```

Steuerung über Konfigurationsdateien

Wozu aber braucht man nun die Schalter, wenn die Ausgaben doch durch fest codierte Bedingungen gesteuert werden, was man dann ja auch mit selbst definierten Konstanten erledigen könnte? Bislang haben wir Schalter und Listener immer programmgesteuert erstellt und deren Werte gesetzt. Dies kann jedoch auch in der Anwendungskonfigurationsdatei geschehen.

Wir fügen unserem Programm eine Anwendungskonfigurationsdatei hinzu und diese sieht dann wie folgt aus:

```
01  <?xml version="1.0" encoding="utf-8" ?>
02  <configuration>
03      <system.diagnostics>
04          <switches>
05              <add name="MeinSchalter" value="INFO" />
06          </switches>
07
08          <trace autoflush="true">
09              <listeners>
10                  <add name="FileTraceListener"
11                      type="System.Diagnostics.TextWriterTraceListener"
12                      initializeData="trace.log" />
13                  <remove name="Default" />
14              </listeners>
15          </trace>
16      </system.diagnostics>
17  </configuration>
```

Da wir ja nun die Einstellungen über unsere Konfigurationsdatei holen, ändert sich auch unser Programm

```
01  using System;
02  using System.Diagnostics;
03
04  namespace Logging
05  {
06      class Program
07      {
08          static void Main(string[] args)
09          {
10              Console.WriteLine("Beispiel 6");
11
12              TraceSwitch traceSwitch = new TraceSwitch("MeinSchalter",
    "Ablaufprotokollierung mit Schalter");
13
14              Trace.WriteLine("Beispiel 4 gestartet: " + DateTime.Now.ToString());
15
16              Trace.WriteLineIf(traceSwitch.TraceError, "Logging: Eintrag mit
    TraceLevel Error");
17              Trace.WriteLineIf(traceSwitch.TraceWarning, "Logging: Eintrag mit
    TraceLevel Warning");
18              Trace.WriteLineIf(traceSwitch.TraceInfo, "Logging: Eintrag mit
```

```
TraceLevel Info");
            Trace.WriteLineIf(traceSwitch.TraceVerbose, "Logging: Eintrag mit
TraceLevel Verbose");

            Console.ReadLine();
        }
    }
}
```

Die Ausgabe in der Protokolldatei ist identisch zur vorhergehenden Version ohne Konfigurationsdatei.

```
Beispiel 6 gestartet: 22.05.2013 19:52:19
Logging: Eintrag mit TraceLevel Error
Logging: Eintrag mit TraceLevel Warning
Logging: Eintrag mit TraceLevel Info
```

Ändern wir nun z. B. den Wert des Trace-Levels in “ERROR“ und lassen das Programm erneut laufen, sieht die Protokolldatei wie folgt aus:

```
Beispiel 6 gestartet: 22.05.2013 19:52:19
Logging: Eintrag mit TraceLevel Error
Logging: Eintrag mit TraceLevel Warning
Logging: Eintrag mit TraceLevel Info
Beispiel 6 gestartet: 22.05.2013 19:53:12
Logging: Eintrag mit TraceLevel Error
```

Welchen Vorteil bietet aber die Vorgehensweise, Einstellungen über die Konfigurationsdatei vorzunehmen? Ganz einfach, man kann die Trace-Ausgabe verändern ohne das Programm verändern zu müssen!

Beispiel: In der Einrichtungsphase werden alle Nachrichten angezeigt. Während der Eingewöhnungsphase der Endanwender werden alle Nachrichten bis zum Level Info angezeigt, um z. B. Bedienungsfehler und Verhaltensmuster zu protokollieren Im normalen Betrieb protokolliert man dann z. B. nur noch Fehler an. Wenn es dann zu Problemen kommt, kann der Kunde einfach die Protokolldateien übermitteln, anhand derer dann ein eventueller Programmfehler entdeckt werden kann.

Weitere Möglichkeiten

Mit dem bisher Gezeigten sind die Möglichkeiten der Ablaufprotokollierung natürlich noch nicht ausgeschöpft.

Um die Lesbarkeit der Trace-Ausgabe zu erhöhen kann diese eingerückt werden. Wie einfach das geht zeigt das folgende Beispiel:

```
01 using System;
02 using System.Diagnostics;
03
04 namespace Logging
05 {
06     class Program
07     {
08         static void Main(string[] args)
09         {
10             Console.WriteLine("Beispiel 7");
11
```

```
12              Trace.WriteLine("Beispiel 7 gestartet: " + DateTime.Now.ToString());
13              Trace.IndentLevel = 0;
14              Trace.IndentSize = 4;
15
16              traceIndentTest(0);
17
18              Trace.WriteLine("Beispiel 7 beendet.");
19
20              Console.ReadLine();
21          }
22
23          private static void traceIndentTest(int level)
24          {
25              if (level < 10)
26              {
27
28
27                  Trace.WriteLine("Trace mit IndentLevel: " +
    Trace.IndentLevel.ToString());
28              Trace.Unindent();
29                  traceIndentTest(level + 1);
30              }
31              Trace.Unindent();
34          }
33      }
34  }
```

Die Ausgabe in der Log-Datei sieht dann so aus:

```
Beispiel 7 gestartet: 22.05.2013 19:54:05
Trace mit IndentLevel: 0
    Trace mit IndentLevel: 1
        Trace mit IndentLevel: 2
            Trace mit IndentLevel: 3
                Trace mit IndentLevel: 4
                    Trace mit IndentLevel: 5
                        Trace mit IndentLevel: 6
                            Trace mit IndentLevel: 7
                                Trace mit IndentLevel: 8
                                    Trace mit IndentLevel: 9
Beispiel 7 beendet.
```

Zudem gibt es auch noch weitere TraceListener, von denen ich auf drei noch ganz kurz eingehen möchte.

EventLogTraceListener
Mit diesem Listener können die Trace-Meldungen an die Windows Ereignisanzeige gesendet werden. Dabei sollte man aber mit den Meldungen etwas sparsam umgehen, damit das Ereignisprotokoll nicht zu schnell vollläuft (z. B. Warnungen, Fehlermeldungen, und Start- und Stopp-Meldungen). Besonders geeignet ist diese Variante z. B. für Windows- oder Web-Dienste.

XmlWriterTraceListener
Ein weiterer Dateibasierter TraceListener, der die Ausgabe in einer XML-Struktur einträgt.

DelimitedListTraceListener
Dieser TraceListener würde mit dem .NET Framework 2.0 eingeführt und schreibt seine Informationen - wie der Name schon vermuten lässt - mit Trennzeichen (Delimiter) in eine Datei. Damit eignet sich dieser Listener sehr gut dafür CSV-Dateien zur weiteren Verarbeitung/Auswertung zu erzeugen. Auch wenn mit den Methoden Trace.TraceInformation, Trace.TraceWarning und Trace.TraceError und den Eigenschaften Trace.TraceOutputOptions und Trace.Delimiter eien Ausgabe über diesen TraceListener möglich ist, ist dieser doch eher für die Trace-Ausgabe mit der Klasse TraceSource ausgelegt. Auf diese Klasse und auch den DelimitedListTraceListener wird im nächsten Teil dieser Blog-Serie näher eingegangen.

Ausblick

Dies war der erste Teil der Artikelreihe über Ablaufprotokollierung. Im zweiten Teil dieser Reihe werden wir uns der Klasse TraceSource widmen, die mit dem .Net Framework 2.0 eingeführt wurde und sich flexibler als die Trace-Klasse erweist.