Ablaufprotokollierung (2)

Im ersten Teil der Blogreihe zur Ablaufprotokollierung haben wir und mit der Trace-Klasse und der Debug-Klasse beschäftigt, um mit diesen unter Verwendung von Schaltern und Listenern Ablaufinformationen auf unterschiedliche Medien auszugeben. Zudem wurde erläutert, wie die Ausgabe über die app.config gesteuert und variiert werden kann. Im folgenden Beitrag widmen wir uns nun den Neuerungen des .NET Frameworks 2.0 und der darin eingeführten Klasse TraceSource.

Überblick

Mit der Einführung des .NET Frameworks 2.0 hat Microsoft dem Thema Ablaufverfolgung eine Überarbeitung spendiert. Unverändert bleibt das Prinzip, dass Meldungen mithilfe von Schaltern an Listener gesendet werden, die dann die Daten an das jeweilige Ausgabemedium weiterleiten. Neu ist die Klasse TraceSource, die die alten Trace und Debug Klassen ablösen soll. Trace und Debug stehen zwar immer noch zur Verfügung, Microsoft empfiehlt jedoch die TraceSource-Klasse für die Ablaufverfolgung zu verwenden.

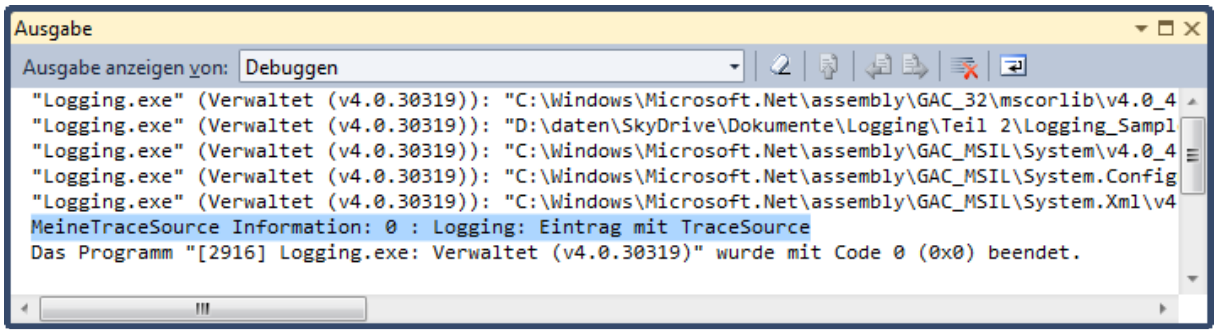
Wie auch die Trace-/Debug-Klasse, kann auch die TraceSource-Klasse sowohl mit einer Anwendungskonfigurationsdatei als auch programmgesteuert verwendet werden. In diesem Beitrag wird die Verwendung mit einer Konfigurationsdatei erläutert, da ich diese Vorgehensweise für besser erachte, da die Ablaufverfolgung beeinflusst werden kann, ohne das Programm anpassen zu müssen.

Die TraceSource Klasse

Im Gegensatz zu Trace und Debug ist TraceSource keine statische Klasse, d. h. in der Anwendung muss zunächst eine Instanz von TraceSource erzeugt werden, damit diese benutzt werden kann. Wie das geht, zeigt das folgende 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 8");
11
12              TraceSource loggingSource = new TraceSource("MeineTraceSource");
13              loggingSource.Switch.Level = SourceLevels.All;
14              loggingSource.TraceInformation("Logging: Eintrag mit TraceSource");
15              loggingSource.Close();
16
17              Console.ReadLine();
18          }
19      }
18  }
```

Wird das Programm gestartet erhalten wir folgende Ausgabe im Ausgabefenster



Wie gesagt, TraceSource ist keine statische Klasse, somit können natürlich auch mehrere Instanzen erzeugt werden. Doch was bringt das? Es ist z. B. möglich für unterschiedliche Anwendungsbereiche eine eigene TraceSource-Instanz zu verwenden, wenn man mag sogar bis auf Klassenebene hinab. Somit kann ganz gezielt der Trace-Level für diese Bereiche festgelegt werden, was die Fehlersuche merklich vereinfachen kann.

Zur Ausgabe der Meldungen stellt TraceSource die folgenden Methoden zur Verfügung:

- TraceEvent
- TraceData
- TraceInformation
- TraceTransfer

TraceInformation ist dabei nur ein vereinfachter Aufruf der Traceevent-Methode, somit entspricht der Aufruf

```
loggingSource.TraceInformation("Logging: Eintrag mit TraceSource");
```

dem Aufruf

```
loggingSource.TraceEvent(TraceEventType.Information, 0, "Logging: Eintrag mit TraceSou
rce");
```

In der TraceListener-Basisklasse sind auch TraceData und TraceTransfer spezialisierte Aufrufe von TraceEvent. Listener können diese Methoden überschreiben um spezielle Ausgaben zu ermöglichen. Im Folgenden werden wir stets TraceEvent verwenden.

Einstellungen in der Konfigurationsdatei

Das neue Konzept Instanzen von TraceSource zu verwenden bringt es mit sich, dass es auch in der Konfigurationsdatei zu einigen Änderungen kommt. Die nachfolgende Datei verwenden wir für alle weiteren Beispiele.

```
01 <?xml version="1.0" encoding="utf-8" ?>
02 <configuration>
03   <system.diagnostics>
04     <sources>
05       <source name="MeineTraceSource" switchName="MeinSchalter">
06         <listeners>
07           <add name="fileListener" />
08           <add name="consoleListener" />
09         </listeners>
10       </source>
11       <source name="MeineTraceSource2" switchValue="Warning">
12         <listeners>
13           <add name="fileListener" />
14         </listeners>
15       </source>
16       <source name="MeineTraceSource3" switchName="MeinSchalter">
```

```
17          <listeners>
18            <remove name="Default" />
19            <add name="lokalerDelimitedListener"
type="System.Diagnostics.DelimitedListTraceListener" delimiter=";"
initializeData="trace.csv" />
20            <add name="lokalerDelimitedListener"
type="System.Diagnostics.DelimitedListTraceListener" delimiter=";"
initializeData="enh_trace.csv"
21              traceOutputOptions="DateTime, ProcessId" />
22          </listeners>
23        </source>
24        <source name="MeineTraceSource4" switchName="MeinSchalter">
25          <listeners>
26            <remove name="Default" />
27            <add name="errorListener"
type="System.Diagnostics.TextWriterTraceListener" initializeData="error.log">
28              <filter type="System.Diagnostics.EventTypeFilter"
initializeData="Error" />
29            </add>
30            <add name="filterListener" />
31          </listeners>
32        </source>
33        <source name="MeineTraceSource5" switchName="MeinSchalter">
34          <listeners>
35            <add name="filterListener" />
36          </listeners>
37        </source>
38      </sources>
39      <sharedListeners>
40        <add name="consoleListener" type="System.Diagnostics.ConsoleTraceListener"
/>
41        <add name="fileListener" type="System.Diagnostics.TextWriterTraceListener"
initializeData="trace.log" />
42        <add name="filterListener"
type="System.Diagnostics.TextWriterTraceListener" initializeData="filter.log">
43          <filter type="System.Diagnostics.SourceFilter"
initializeData="MeineTraceSource5" />
44        </add>
45      </sharedListeners>
46      <switches>
47        <add name="MeinSchalter" value="Verbose" />
48      </switches>
49    </system.diagnostics>
50  </configuration>
```

Zunächst fällt der neue Abschnitt *<sources>* auf. In diesem werden alle verwendeten TraceSourcen definiert. Dabei müssen die dort angegebenen Namen genau mit denen übereinstimmen die beim Erstellen der TraceSource Instanzen in der Anwendung übergeben werden.

```
<source name="MeineTraceSource" switchName="MeinSchalter">
```

```
TraceSource loggingSource = new TraceSource("MeineTraceSource");
```

TraceListener

Je nach Verwendungszweck können TraceListener auf zwei verschiedene Arten festgelegt werden. Zum einen direkt innerhalb der Definition der Source, also genau wie bisher im *<trace>* Abschnitt.

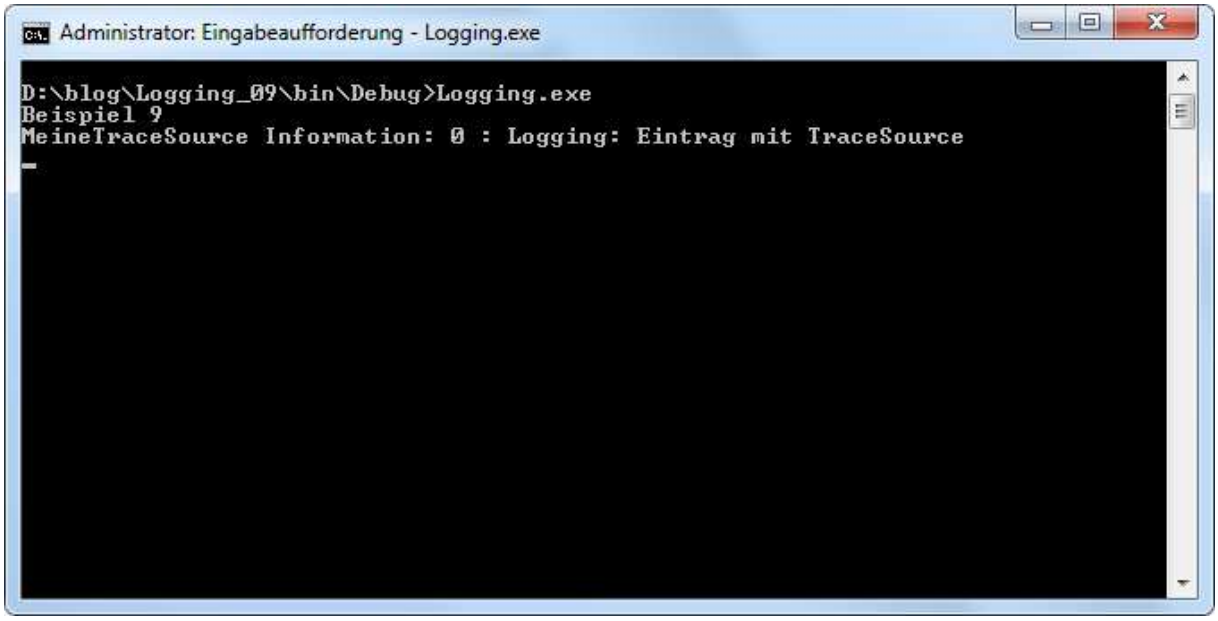
```
<listeners>
  <add initializeData="trace.log" type="System.Diagnostics.TextWriterTraceListener"
name="lokalerListener" />
</listeners>
```

Dieser Listener steht dann auch nur dieser TraceSource zur Verfügung. Soll ein Listener mehreren Sourcen zur Verfügung stehen, so muss bei der TraceSource nur der Name angeben werden. Die eigentliche Definition erfolgt im Abschnitt <*sharedListeners*>.

```
<listeners>
  <add name="consoleListener" />
</listeners>

<sharedListeners >
  <add type="System.Diagnostics.ConsoleTraceListener" name="consoleListener" />
</sharedListeners>
```

Lässt man nun das Programm nochmals laufen, erhält man nun folgende Ausgabe auf der Konsole



und zudem, je nach Build Konfiguration im Ordner bin/Debug oder bin/Release, die Datei *trace.log* mit folgendem Inhalt:

```
MeineTraceSource Information: 0 : Logging: Eintrag mit TraceSource
```

Schalter, Events und Filter

Wie die Listener können auch Schalter auf zwei Arten festgelegt werden. Zum einen kann ein switchValue angegeben werden. Dieser Schalter hat nur Auswirkungen auf die TraceSource bei der er angegeben ist - daher spricht man auch von einem *local switch*.

```
<source name="MeineTraceSource2" switchValue="Warning">
```

Die zweite Möglichkeit ist, einen switchName anzugeben, also den Namen eines Switches, der in der Konfigurationsdatei im Abschnitt <switches> definiert wird. Somit kann über einen Schalter die Ausgabe mehrerer TraceSourcen gesteuert werden; man spricht *von shared switches*.

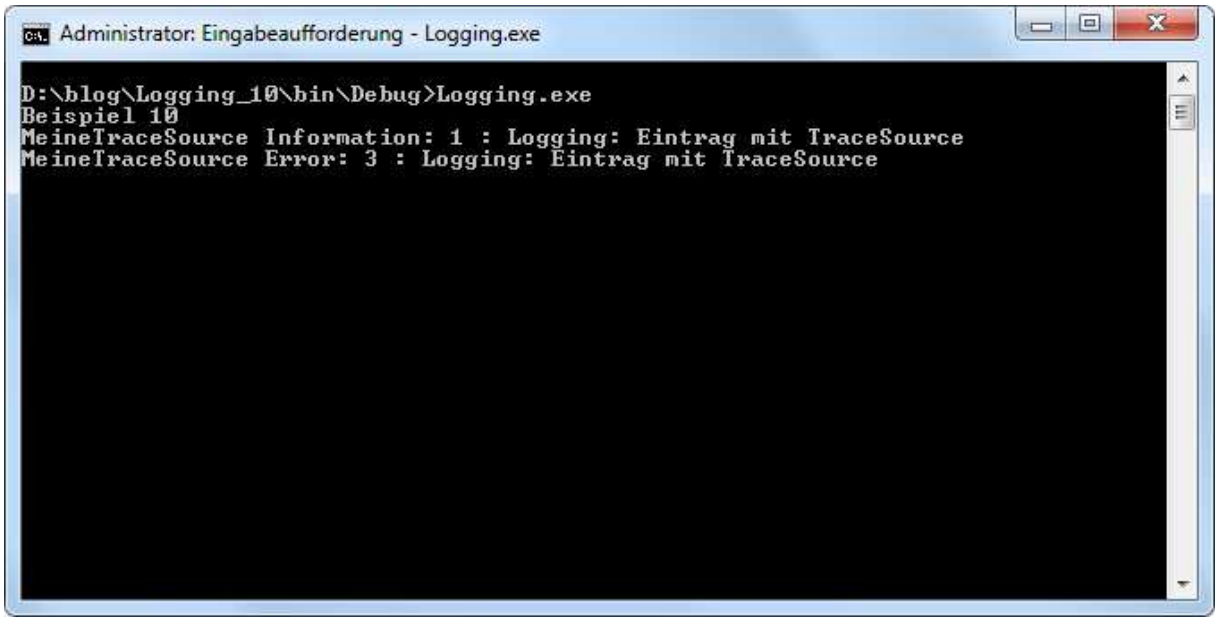
```
<switches>
  <add name="MeinSchalter" value="Verbose" />
</switches>
```

Das Beispielprogramm wird angepasst, um das Zusammenspiel zwischen TraceSourcen, Schaltern und Eventtypen zu verdeutlichen.

```
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 10");
11
12              TraceSource loggingSource = new TraceSource("MeineTraceSource");
13              TraceSource loggingSource2 = new TraceSource("MeineTraceSource2");
14              loggingSource.TraceEvent(TraceEventType.Information, 1, "Logging:
    Eintrag mit TraceSource");
15              loggingSource2.TraceEvent(TraceEventType.Information, 2, "Logging:
    Eintrag mit TraceSource");
16              loggingSource.TraceEvent(TraceEventType.Error, 3, "Logging: Eintrag
    mit TraceSource");
17              loggingSource2.TraceEvent(TraceEventType.Error, 4, "Logging: Eintrag
    mit TraceSource");
18              loggingSource.Close();
19              loggingSource2.Close();
20
21              Console.ReadLine();
22          }
23      }
24  }
```

Betrachten wir uns dazu nun die Ausgabe auf der Konsole und die Datei *trace.log*.



trace.log:

```
MeineTraceSource Information: 1 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 3 : Logging: Eintrag mit TraceSource
MeineTraceSource2 Error: 4 : Logging: Eintrag mit TraceSource
```

Wir wir aus Teil 1 von Trace bzw. Debug schon kennen, werden Meldungen, die über die TraceSource *MeineTraceSource* geschrieben werden, sowohl auf der Konsole als auch in der Datei ausgegeben, da dieser TraceSource zwei TraceListener zugeordnet sind. Neu ist, dass die Meldungen von *MeineTraceSource2* auch in der Datei *trace.log* landen – obwohl das aber auch irgendwie zu erwarten war, wird doch der gleiche TraceListener benutzt. Bei dem Beispiel sieht man auch, dass die Ausgabe aufgrund der Kombination von Schalter und verwendetem Eventtyp erfolgt. Die nachfolgende Tabelle

zeigt, bei welchen Kombinationen von SourceLevels und TraceEventType eine Ausgabe erfolgt oder eben nicht.

	Source Levels					
TraceEventType	*Critical*	*Error*	*Warning*	*Information*	*Verbose*	*ActivityTracing*
Critical	✔	✔	✔	✔	✔	
Error		✔	✔	✔	✔	
Warning			✔	✔	✔	
Information				✔	✔	
Verbose					✔	
Start						✔
Stop						✔
Suspend						✔
Resume						✔
Transfer						✔

Das folgende Programm macht diese Abhängigkeiten noch einmal deutlich.

```
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 11");
11
12              int id = 1;
13
14              foreach (SourceLevels sl in Enum.GetValues(typeof(SourceLevels)))
15              {
16                  Console.WriteLine("SourceLevel: " + sl.ToString());
17                  TraceSource loggingSource = new TraceSource("MeineTraceSource2");
18
19                  SourceSwitch switch1 = new SourceSwitch("switch1", sl.ToString())
    ;
20                  loggingSource.Switch = switch1;
21
22                  foreach (TraceEventType et in Enum.GetValues(typeof(TraceEventTyp
    e)))
23                  {
24                      loggingSource.TraceEvent(et, id, "Logging: Eintrag mit TraceS
    ource");
25                      id++;
26                  }
27                  loggingSource.Close();
28              }
29              Console.ReadLine();
30          }
31      }
32  }
```

Die erzeugte Protokolldatei *trace.log* sieht dann wie folgt aus.

```
MeineTraceSource Critical: 11 : Logging: Eintrag mit TraceSource
```

```
MeineTraceSource Critical: 21 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 22 : Logging: Eintrag mit TraceSource
MeineTraceSource Critical: 31 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 32 : Logging: Eintrag mit TraceSource
MeineTraceSource Warning: 33 : Logging: Eintrag mit TraceSource
MeineTraceSource Critical: 41 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 42 : Logging: Eintrag mit TraceSource
MeineTraceSource Warning: 43 : Logging: Eintrag mit TraceSource
MeineTraceSource Information: 44 : Logging: Eintrag mit TraceSource
MeineTraceSource Critical: 51 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 52 : Logging: Eintrag mit TraceSource
MeineTraceSource Warning: 53 : Logging: Eintrag mit TraceSource
MeineTraceSource Information: 54 : Logging: Eintrag mit TraceSource
MeineTraceSource Verbose: 55 : Logging: Eintrag mit TraceSource
MeineTraceSource Start: 66 : Logging: Eintrag mit TraceSource
MeineTraceSource Stop: 67 : Logging: Eintrag mit TraceSource
MeineTraceSource Suspend: 68 : Logging: Eintrag mit TraceSource
MeineTraceSource Resume: 69 : Logging: Eintrag mit TraceSource
MeineTraceSource Transfer: 70 : Logging: Eintrag mit TraceSource
MeineTraceSource Critical: 71 : Logging: Eintrag mit TraceSource
MeineTraceSource Error: 72 : Logging: Eintrag mit TraceSource
MeineTraceSource Warning: 73 : Logging: Eintrag mit TraceSource
MeineTraceSource Information: 74 : Logging: Eintrag mit TraceSource
MeineTraceSource Verbose: 75 : Logging: Eintrag mit TraceSource
MeineTraceSource Start: 76 : Logging: Eintrag mit TraceSource
MeineTraceSource Stop: 77 : Logging: Eintrag mit TraceSource
MeineTraceSource Suspend: 78 : Logging: Eintrag mit TraceSource
MeineTraceSource Resume: 79 : Logging: Eintrag mit TraceSource
MeineTraceSource Transfer: 80 : Logging: Eintrag mit TraceSource
```

Eine weitere Möglichkeit Ausgabe eines Listeners zu beeinflussen sind Filter. Dabei kann zum einen der SwitchLevel oder aber die TraceSource von der Ausgabe ausgeschlossen werden. Das Programm wird so angepasst, dass die TraceSourcen *MeineTraceSource4* und *MeineTraceSource5* verwendet werden.

```
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 12");
11
12              TraceSource loggingSource = new TraceSource("MeineTraceSource4");
13              TraceSource loggingSource2 = new TraceSource("MeineTraceSource5");
14              loggingSource.TraceEvent(TraceEventType.Information, 1, "Logging:
    Eintrag mit TraceSource");
15              loggingSource2.TraceEvent(TraceEventType.Information, 2, "Logging:
    Eintrag mit TraceSource");
16              loggingSource.TraceEvent(TraceEventType.Error, 3, "Logging: Eintrag
    mit TraceSource");
17              loggingSource2.TraceEvent(TraceEventType.Error, 4, "Logging: Eintrag
    mit TraceSource");
18              loggingSource.Close();
19              loggingSource2.Close();
20
21              Console.ReadLine();
22          }
23      }
24  }
```

Die Ausgabe in den beiden Dateien sieht dann wie folgt aus:

error.log

```
MeineTraceSource4 Error: 3 : Logging: Eintrag mit TraceSource
```

Wie man sieht wird in die Datei *error.log* nur der Eintrag mit dem Eventtyp *Error* geschrieben, nicht aber der Eintrag mit Eventtyp *Information*.

filter.log

```
MeineTraceSource5 Information: 2 : Logging: Eintrag mit TraceSource
MeineTraceSource5 Error: 4 : Logging: Eintrag mit TraceSource
```

Tatsächlich landet in der Datei *filter.log* nur die Ausgabe aus *MeineTraceSource5*, obwohl der Listener *filterListener* auch der TraceSource *MeineTraceSource4* zugeordnet ist.

Für alle, die an Sinn und Zweck von Filtern zweifeln (das kann man doch auch irgendwie mit Source, Switch und Listener bewerkstelligen) hier ein praktisches Anwendungsszenario:

Mehrere TraceSourcen (z. B. von jeder Assembly) schreiben Meldungen aller Eventtypen in eine Datei. Nun sollen aber alle kritischen Fehler auch in das Windows Event-Log geschrieben werden. Anstatt jetzt für jede Assembly eine weitere TraceSource zu definieren und alle Meldungen ein zweites Mal zu schreiben (was auch heißt das Programm anzupassen!) wird einfach zu jeder TraceSource noch ein weiterer Shared Listener hinzugefügt und dieser erhält einen EventTypeFilter für kritische Meldungen.

Zugegeben, für den SourceFilter ist mir auch kein wirklich sinniges Beispiel eingefallen, außer eben temporär die Ausgabe in einen Listener auf eine TraceSource zu begrenzen (und ja, das könnte man definitiv ebenso einfach durch einen zusätzlichen lokalen Listener bewerkstelligen).

Sonstiges

Zum Abschluss dieses Teils wollen wir noch einen kleinen Blick auf die TraceOptions-Enumeration werfen. Einem TraceListener können Optionen mitgegeben werden, so dass zusätzliche (System-) Informationen automatisch mit in das Ablaufprotokoll geschrieben werden. Folgende Optionen können angegeben werden:

- None
- DateTime
- Timestamp
- ProcessId
- ThreadId
- Callstack
- LogicalOperationStack

Das nachfolgende Programm schreibt über DelimitedListTraceListener zwei CSV-Dateien, einmal ohne sowie einmal mit erweiterten Informationen.

```
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 13");
11
12              TraceSource loggingSource = new TraceSource("MeineTraceSource3");
13              loggingSource.TraceInformation("Logging: Eintrag mit TraceSource");
14              loggingSource.Close();
15
16              Console.ReadLine();
17          }
18      }
19  }
```

Der Vollständigkeit halber hier auch noch die der Inhalt der Dateien nachdem das Programm gelaufen ist.

trace.csv - ohne Optionen

```
"MeineTraceSource3";Information;0;"Logging: Eintrag mit TraceSource";;;;;;;
```

enh_trace.csv - mit Optionen

```
"MeineTraceSource3";Information;0;"Logging: Eintrag mit
TraceSource";;2752;"";"8";"2012-07-26T18:24:52.8323267Z";2505725926857;"   bei
System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   bei System.Environment.get_StackTrace()
   bei System.Diagnostics.TraceEventCache.get_Callstack()
   bei System.Diagnostics.DelimitedListTraceListener.WriteFooter(TraceEventCache
eventCache)
   bei System.Diagnostics.DelimitedListTraceListener.TraceEvent(TraceEventCache
eventCache, String source, TraceEventType eventType, Int32 id, String format, Object[]
args)
   bei System.Diagnostics.TraceSource.TraceEvent(TraceEventType eventType, Int32 id,
String format, Object[] args)
   bei System.Diagnostics.TraceSource.TraceInformation(String message)
   bei Logging.Program.Main(String[] args) in D:\dev\web\code-bizarre\res\Logging\Teil
2\Logging_Samples_CS\Logging_13\Program.cs:Zeile 13.
   bei System.AppDomain._nExecuteAssembly(RuntimeAssembly assembly, String[] args)
   bei System.AppDomain.ExecuteAssembly(String assemblyFile, Evidence assemblySecurity,
String[] args)
   bei Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   bei System.Threading.ThreadHelper.ThreadStart_Context(Object state)
   bei System.Threading.ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state, Boolean ignoreSyncCtx)
   bei System.Threading.ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state)
   bei System.Threading.ThreadHelper.ThreadStart()"
```

Ausblick

Auch wenn es sicherlich noch vieles zum Thema Ablaufprotokollierung zu entdecken gibt, die Grundlagen dafür sollten nun bekannt sein. Die Bordmittel des.NET Frameworks sollten in jedem Fall ausreichen die Qualitätssicherung zu vereinfachen und somit die Qualität der Software zu verbessen. Doch was, wenn die Ausgabe der Listener einfach nicht den Anforderungen entspricht? Dann wird es Zeit einen eigenen Listener zu schreiben, und das ist das Thema des dritten Teils dieser Reihe.