

Kochen mit Patrick

Nach dem großen Erfolg des ersten Teils folgen weitere Tipps rund ums Debugging. Diesmal geht es um Variablen-Fenster, Object-IDs und die verzwickte Fehlersuche in Multithread-Anwendungen. Zum krönenden Abschluss gibt's dann Pangasiusfilet aus dem Backofen mit Rosmarinkartoffeln.



dnpCode
A0904Kochen

Patrick A. Lorenz ist Geschäftsführer der PGK GmbH, einem auf .NET spezialisierten Technologiedienstleister. Daneben ist er als Autor tätig. Sein neuestes Buch „ASP.NET 3.5 mit AJAX“ beschreibt die Neuerungen in .NET 3.5 für Webentwickler. In seiner Freizeit ist Patrick Hobbykoch. Sie erreichen ihn unter www.pgk.de oder lorenz@pgk.de.

Das Watch-Fenster

Variablen lassen sich zum Watch-Fenster während des Debuggens durch Rechtsklick und Auswahl von *Add Watch* hinzufügen. Das klappt gleichermaßen im Quelltext wie aus den Locals- und Autos-Fenstern heraus. Steht die Variable im Ausführungskontext zur Verfügung, ist sie schwarz, ansonsten ausgegraut. Hat sich der Wert geändert, wird dieser rot markiert. Sofern änderbar, können Sie den Wert direkt überschreiben.

Auch manuell können Sie eine Variable hinzufügen, indem Sie deren Namen in die letzte, leere Zeile der *Watch*-Tabelle eingeben. Weniger bekannt ist, dass Sie auch Methoden und Ausdrücke überwachen können. Dazu tragen Sie den gewünschten Ausdruck ein, zum Beispiel folgende Werte:

```
DateTime.Now
DateTime.Now.GetType()
this.DoSomething()
DateTime.Now.Minute * 60
```

Beachten Sie, dass der Debugger tatsächlich die entsprechende Eigenschaft oder Methode ausführt. Die aufgerufenen Ausdrücke sollten daher lediglich Leseoperationen durchführen. Etwaige Breakpoints werden dabei ignoriert.

Während der Fehlersuche wird das Watch-Fenster nach Ausführung jeder einzelnen Zeile automatisch aktualisiert. Gerade am ersten Beispiel lässt sich dies gut erkennen. Dies gilt allerdings nur für Variablen und Eigenschaften, Ausdrücke müssen manuell aktualisiert werden. Hierzu erscheint ein kleines Refresh-Icon hinter dem letzten Wert. Sie können in diesem Fall die Aktualisierung über das Icon oder die Leertaste forcieren.

Eigentlich waren für diesen Monat weitere Tipps zu ASP.NET AJAX angekündigt. Aufgrund der positiven Rückmeldungen zum Februar-Thema Debugging [1] möchte ich mich in diesem Artikel aber noch einmal der Fehlerfindung widmen. AJAX wird jedoch nachgeholt, voraussichtlich im nächsten Heft.

Die Variablen-Fenster

Visual Studio bietet vier Fenstertypen zum Überwachen von Variablen. Auf die Variablen kann zugegriffen, ihr Wert angezeigt und verändert werden, zudem lassen sich Objektstrukturen aufgliedern.

Das Fenster Locals listet Variable der aktuellen Methode auf. Über *this* steht eine Referenz der ausführenden Klasse bereit. Das Fenster Autos passt sich während der Ausführung jeder Codezeile an und listet

die dort und in der vorangegangenen Zeile verwendeten Variablen.

Vier gleichwertige Watch-Fenster erlauben die gezielte Überwachung von Variablen. Die Konfiguration der Fenster bleibt im Projekt gespeichert, steht also auch beim nächsten Debugging wieder zur Verfügung.

Das Immediate-Fenster dient der expliziten Auswertung von Variablen beziehungsweise Ausführung von Ausdrücken.

Name	Value	Type
DateTime.Now	{15.02.2009 11:21:26}	System.D
DateTime.Now.Minute,h	0x00000015	int
typeof(DateTime),raw	{Name = "DateTime" FullName = "System.DateTime"}	System.T
DateTime.Now.GetType()	This expression causes side effects and will not be evaluated	System.T
DateTime.Now.Minute * 60	1260	int
p	{Debugging2.Person}	Debuggin

[Abb. 1] Mit dem Watch-Fenster lassen sich auch Ausdrücke überwachen.

Listing 1

Identifizieren eines Objekts per Object-ID.

```
static void Main(string[] args) {
    var persons = new List<Person>();
    persons.Add(new Person { FirstName = "Max" });
    persons.Add(new Person { FirstName = "Moritz" });
    // Zweites Element wird markiert als 1#

    persons.ForEach(p => {
        var processor = new PersonProcessor(p);
        processor.Process();
    });

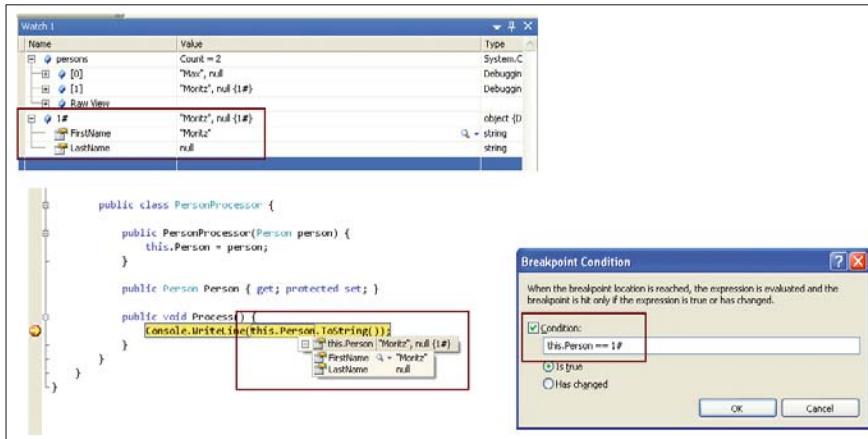
    Console.Read();
}

public class PersonProcessor {

    public PersonProcessor(Person person) {
        this.Person = person;
    }

    public Person Person { get; protected set; }

    public void Process() {
        // Breakpoint nur, wenn Person = 1#
        Console.WriteLine(this.Person.ToString());
    }
}
```



[Abb. 2] Object-IDs identifizieren Objekte eindeutig.

Je nach Anforderung können Sie per Kontextmenü zwischen dezimalen und hexadezimalen Werten umschalten. Das ist auch gezielt pro Variable möglich, indem Sie dieser eine Formatierungsanweisung mitgeben. *h* gibt den Wert hexadezimal aus (*DateTime.Now.Minute, h*), *d* zeigt den Dezimalwert (*DateTime.Now.Minute, d*).

Object-IDs nutzen

Anders als in nicht verwaltetem Code kann ein Objekt in verwaltetem Code nicht eindeutig über seine Adresse identifiziert werden. Hier hilft das wenig bekannte und selten verwendete Konzept der Debugger-Object-IDs. Diese erlauben die eindeutige Identifizierung eines Objekts während seiner gesamten Lebenszeit und auch über den aktuellen Ausführungskontext hinweg.

Um ein Objekt mit einer ID zu versehen, wählen Sie während des Debuggens in einem der Variablenfenster oder im Quelltext aus dem Kontextmenü den Befehl *Make Object*. Hinter dem Wert der Variable wird nun eine fortlaufende Zahl in der Form *{n#}* aufgeführt. Die ID wird automatisch überall angezeigt, wo das Objekt (nicht die Variable) evaluiert wird.

Sie können das Objekt nun während des Debuggens über den Platzhalternamen nutzen und etwa als *1#* zum Watch-Fenster hinzufügen. Auf diese Weise können Sie das Objekt unabhängig vom Ausführungskontext und einer darin vorhandenen Referenz beobachten. Das ist äußerst nützlich! Object-IDs lassen sich als Bedingungen in Breakpoints verwenden. Die Ausführung wird dann nur bei der Verarbeitung des markierten Objekts unterbrochen. Gerade bei umfangreichen Datenmengen oder Rekursionen ist dies ausgesprochen hilfreich.

```
this = 1#
```

Tatsächlich können Object-IDs überall dort verwendet werden, wo der Debugger Ausdrücke evaluieren kann. Beachten Sie dabei, dass Object-IDs nicht persistiert werden, sondern bei Bedarf neu anzulegen sind. Als Bedingung oder Filter für Breakpoints verwendete Object-IDs können bei einer weiteren Ausführung nicht evaluiert werden und führen daher zu einem entsprechenden Fehlerhinweis der IDE.

Abbildung 2 und Listing 1 zeigen Object-IDs im Einsatz. Zwei Instanzen einer Klasse *Person* werden einer generischen Liste hinzugefügt. Das zweite Element wird mit einer ID versehen. In einer Schleife werden die Elemente durchlaufen und einer neuen Instanz der Klasse *PersonProcessor* übergeben. Auf deren Methode *Process* wurde ein Breakpoint gesetzt, der jedoch nur im Falle des markierten zweiten Elements zur Unterbrechung der Ausführung führt.

Breakpoints gezielt setzen

Zum Thema Breakpoints und Tracepoints habe ich in [1] schon einige Tipps gegeben. Oft werden Breakpoints bereits vor dem Debugging per Klick in den linken Fensterand oder mittels [F9] angelegt. Beide Varianten scheinen äquivalent, sind es jedoch nicht. Beim Klick auf den Fensterand wird der Breakpoint an den Zeilenbeginn gesetzt, bei [F9] wird er abhängig von der aktuellen Cursor-Position an den Start des aktiven Kontexts gesetzt. Dies lässt sich etwa bei einer einzeiligen Bedingung nachvollziehen (der Zeilenumbruch ist hier nur durch den Druck bedingt):

```
if(this.Person.FirstName.EndsWith("x"))
    Console.WriteLine(this.Person.ToString());
```

Befindet sich der Cursor im Ausführungsteil nach der Bedingung, wird der Breakpoint mit [F9] nur für diesen Teil der

Zeile erstellt. Dies lässt sich gezielt durch Selektion eines Zeilenbereichs vor dem Drücken der Funktionstaste verfeinern.

Darüber hinaus gibt es weitere Möglichkeiten, einen Breakpoint gezielt zu setzen, beispielsweise aus dem Call-Stack-Fenster heraus. Markieren Sie eine Ausführungszeile und betätigen Sie die [F9]-Taste. Oder Sie öffnen das Breakpoint-Fenster mittels [Strg]+[D], [N] und geben den Namen einer Methode ein.

Der Dialog unterstützt Sie mittels Intellisense und erlaubt die Eingabe eines Methodennamens ohne Angabe des Klassennamens. Ist der Name nicht eindeutig, zeigt Visual Studio einen Auswahldialog (Abbildung 3), über den sich auch mehrere Breakpoints auf einmal anlegen lassen.

Run to Cursor versus Set Next Statement

Wie im ersten Teil beschrieben, können Sie die aktuelle Ausführungsposition innerhalb des aktuellen Kontexts per Drag-and-drop des gelben Pfeils im Seitenrand verändern. Dies ist nützlich, um sonst nicht aufgerufene Bereiche explizit zu durchlaufen, eine Codeausführung zu wiederholen oder zu überge-

TIPPS & TRICKS



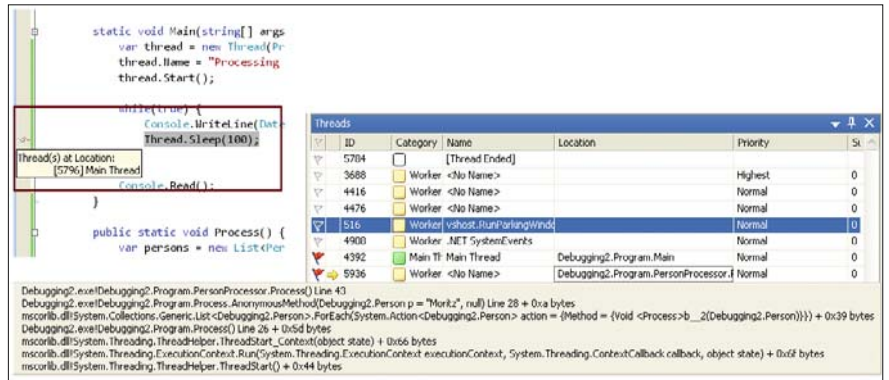
[Abb. 3] Anlegen eines Breakpoints über einen Methodennamen.

hen. Im Kontextmenü steht die Funktion als *Set Next Statement* zur Verfügung.

Einen anderen Ansatz verfolgt die ebenfalls im Kontextmenü angebotene Funktion *Run to Cursor*. Diese legt quasi einen temporären Breakpoint an der gewünschten Stelle an und setzt die Codeausführung bis zum ersten Erreichen der markierten Position fort – es sei denn, Breakpoints sind „im Weg“. Die Funktionalität lässt sich übrigens auch im Kontextmenü der einzelnen Frames im Call-Stack-Fenster nutzen.

Multithreading debuggen

Das Debuggen von Multithreading-Anwendungen ist nicht unbedingt einfach, klappt mit Visual Studio aber mittlerweile



[Abb. 4] Debugging mehrerer Threads.

relativ gut. Ausgangspunkt ist das Threads-Fenster beziehungsweise die Debug-Location-Symbolleiste, die bei Bedarf angezeigt werden kann.

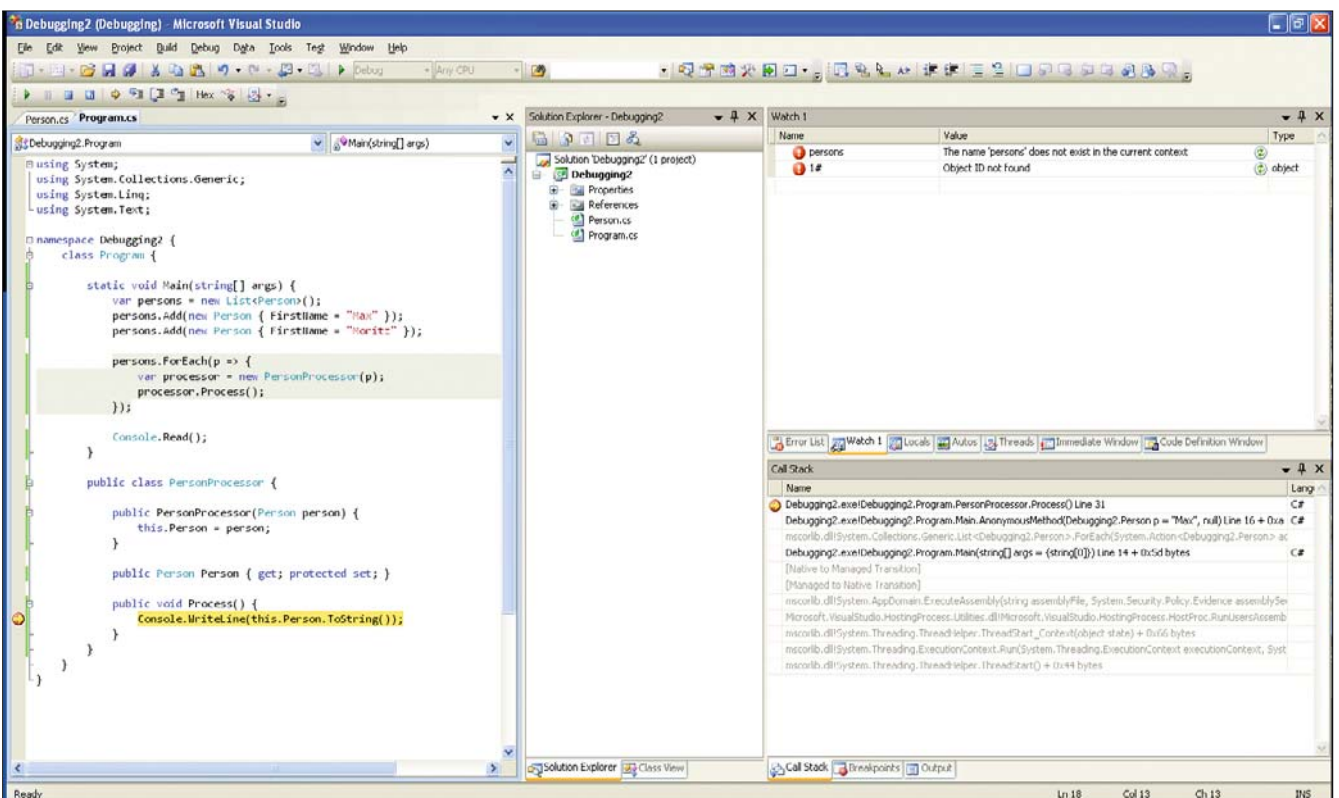
Das Threads-Fenster (Abbildung 4) listet alle Threads. Dabei lässt sich auch die Art des Threads und dessen aktuelle Position erkennen. Per Tooltip kann ein Stack Trace eingblendet und mittels Doppelklick in den jeweiligen Thread gesprungen werden.

Zur besseren Identifizierung von Threads bietet Visual Studio zwei Möglichkeiten: Einerseits können Sie einzelne Threads markieren (rote Flagge, rechte Spalte). Andererseits können Sie einen freien Namen vergeben. Dies ist zur Laufzeit über das Fenster möglich, sollte jedoch generell be-

reits zur Entwicklungszeit stattfinden, beispielsweise so:

```
var thread = new
Thread(Program.Process);
thread.Name = "Processing Thread";
thread.Start();
```

Wird die Ausführung einer Applikation angehalten, werden automatisch sämtliche Threads unterbrochen. Wird die Ausführung fortgesetzt, gilt dies ebenso für alle Threads. Bei parallelen Aufrufen kann das Debuggen dadurch arg erschwert werden. Hilfreich ist es, Threads gezielt für kurze Zeit zu unterbrechen. Klicken Sie dazu mit der rechten Maustaste auf den gewünschten Eintrag in der Liste und wählen



[Abb. 5] Mögliche Fensteraufteilung für optimales Debugging.

Sie *Freeze*. Mittels *Thaw* können Sie die Ausführung später fortsetzen.

Interessant ist auch die Kennzeichnung der aktuellen Ausführungsposition innerhalb von Quelltextdateien. Stellen Sie zunächst sicher, dass die Option *Show Threads in Source* in der Debug-Symboleiste aktiviert ist. Unterbrechen Sie nun die Ausführung einer Multithread-Anwendung und werfen Sie einen Blick auf den linken Fensterrand des Quelltexteditors. Durch zwei geschweifte Linien werden hier die Ausführungspositionen der anderen Threads angezeigt. Um welche Threads es sich dabei jeweils handelt, verrät ein Tooltip.

.NET-Quelltexte herunterladen

In [1] habe ich beschrieben, wie Sie Visual Studio zum Debuggen der Quelltexte von .NET konfigurieren können. Leider lädt die IDE jeweils nur die benötigte Datei von den Microsoft-Servern herunter. Das kostet einerseits jedes Mal etwas Zeit und verhindert andererseits, dass Sie ein wenig in den Quelltexten der FCL stöbern.

Mit dem .NET Mass Downloader [2] können Sie alle gewünschten Quelltexte und

Debug-Symbole auf einmal laden. Anschließend lassen sich die Quelltextdateien direkt öffnen, zudem können Sie die Quellen auch offline debuggen.

Die optimale Fensteraufteilung

Ich wurde nach der optimalen Fensteraufteilung für die Fehlersuche gefragt. Die Frage lässt sich nur sehr subjektiv beantworten, und die Antwort hängt vor allem vom verfügbaren Platz auf dem Monitor ab. Ich habe an der Docking-Station für die Entwicklung zwei Monitore angeschlossen, einen mit 27 Zoll für die IDE und einen mit 19 Zoll für die Applikation. Als wirklich hilfreich hat sich das Breitbildformat des 27-Zoll-Monitors erwiesen.

Abbildung 5 zeigt meine aktuelle Entwicklungsumgebung. Diese ist zunächst zweigeteilt: Quelltext links und rechts die Debugging-Tools. Der rechte Bereich ist wiederum vertikal und horizontal gegliedert und enthält pro Position jeweils eine Gruppe von Tool-Fenstern:

- Links befindet sich die physikalische Navigation im Quelltext in Form von Solution Explorer und Class View.

- Rechts oben liegen die Zustandsfenster Watch, Autos, Locals und Threads.
- Rechts unten sind Stack Trace und Breakpoints zu finden. Das Pending-Changes-Fenster zeigt geänderte Dateien, die oftmals Ausgangspunkt für das Debugging sind. Außerdem liegt hier das Output-Fenster, in dem sich Trace-Ausgaben prüfen lassen.

Vor der Arbeit mit dem zusätzlichen Breitbildmonitor hatte ich zwei normale 19-Zoll-Monitore im Einsatz und auf dem einen den Quelltext, auf dem anderen die Fenster positioniert. Das klappte auch sehr gut.

Wussten Sie übrigens, dass Visual Studio mehrere Fenstereinstellungen parallel speichert? So können Sie für Entwicklungs- und Laufzeit unterschiedlich optimierte Fenster anzeigen und positionieren. **[bl]**

[1] Kochen mit Patrick, dotnetpro 02/2009, Seite 116 ff.

[2] .NET Mass Downloader, [dnplink SL0904Kochstudio1](#)

[3] Greenpeace Fischratgeber, [dnplink SL0904Kochstudio2](#)

Nieder mit dem Kabeljau!

Betrachtet man Fischtheke und Eisfach im Supermarkt, könnte man meinen, dem Kabeljau soll endgültig der Garaus gemacht werden. Dort findet man jede Menge Kabeljau und Dorsch, der aber mittlerweile nahezu überfischt ist. Eine günstige Alternative, die man guten Gewissens essen kann, ist der Pangasius aus Vietnam, den auch Greenpeace in seinem sehr guten Fischratgeber [3] empfiehlt.



Ofenfisch

Fisch ist ja nicht jedermanns Sache, sowohl auf dem Teller als auch in der Küche. Dieses Gericht ist wirklich einfach zuzubereiten und gelingt selbst Fischanfängern wie mir.

Würzen Sie 500 bis 750 Gramm Fischfilet von beiden Seiten mit Salz und Pfeffer und legen Sie sie nebeneinander in eine leicht eingefettete Ofenform.

Schneiden Sie eine Zwiebel, zwei bis drei Zehen Knoblauch und eine getrocknete Chili ganz fein. Verfahren Sie ebenso mit je einem halben Bund Basilikum und Petersilie sowie einigen Zweigen frischem Rosmarin. Mischen Sie die Zutaten und geben Sie sie über den Fisch. Nun 400 Gramm Tomaten waschen, in feine Streifen schneiden und ebenfalls über den Fisch geben. Zu guter Letzt geben Sie noch einen Schuss Olivenöl darüber. Der Fisch kommt anschließend für 15 bis 20 Minuten in den auf 180 Grad vorgeheizten Backofen.

Zu dem Gericht passen leckere Rosmarinkartoffeln. Die werden gekocht, geviertelt und zusammen mit Rosmarin in der Pfanne golden gebraten. Guten Appetit!