kurz & bündig	
Inhalt	Qualitätsverbesserung durch die Simulation von Ausnahmesituationen
Zusammenfassung	Wenn Sie sicherstellen wollen, dass sich Ihre Applikation auch in unerwarteten Fehlersituationen stabil und zuverlässig verhält, bietet der Fault Simulator einen durchaus intelligenten Ansatz zur Simulation von Fehlersituationen.
Quellcode mit angeliefert	nein

Veröffentlchung: 12/2005.

Fehlersituationen simulieren

Wie reagiert Ihre Software auf Fehler? Stürzt die Applikation ab, wenn nicht ausreichend Hauptspeicher zur Verfügung steht? Was passiert, wenn die Netzverbindung abbricht oder Leseberechtigungen auf wichtige Dateien fehlen? Mit Hilfe des Fault Simulators können Sie Fehlersituationen simulieren, ohne die Stabilität der Applikationsumgebung zu gefährden.

Manu Carus

Der Test einer Software beschränkt sich meist auf die fachlich geforderte Funktionalität und vernachlässigt dabei die Verifizierung der Fehlerbehandlung in Ausnahmesituationen. Auf diese Weise gelangt ein erheblicher Anteil ungetesteter Code in die Produktionsumgebung. Es verwundert daher nicht, dass viele Software-Applikationen in unerwarteten Fehlersituationen unbeabsichtigt aussteigen. Die Folge sind Ausfallzeiten, Vertrauensverluste, Umsatzeinbußen und ein beschädigtes Image.

Ein vollständiger Funktionstest muss möglichst alle Pfade eines Programms berücksichtigen, insbesondere also die implementierten Fehlerbehandlungsroutinen. Wie aber testet man Fehlersituationen? Wie generiert man Fehler? Manuelle Lösungen ziehen erhebliche Konsequenzen nach sich: das Ziehen des Netzwerkkabels (Verbindungsabbruch) oder das Erzeugen eines hohen Netzwerktraffics (Timeouts) hat sehr unangenehme Seiteneffekte im Firmennetz, während massive Allokation des Hauptspeichers zwecks Simulation eines “out of memory” den Debugger und das Betriebssystem so sehr belasten kann, dass ein Test des hervorgerufenen Fehlers kaum mehr möglich ist.

In der Umgebung einer Applikation können unzählige Fehler auftreten, die nicht durch das falsche Verhalten des Benutzers verursacht werden. Die meist implementierten Fehlerbehandlungsroutinen protokollieren die Fehlermeldung in die Datenbank und geben dem Benutzer einen Warnhinweis aus. Was aber, wenn die Fehlerbehandlungsroutine selbst fehlerhaft ist? Wurden bei der Entwicklung überhaupt alle potentiellen Fehlerkandidaten berücksichtigt? Wann können im Code grundsätzlich Ausnahmen auftreten? Und an welchen Stellen?

Für die Durchführung solch umfassender Tests wird eine einfache, wiederholbare und sichere Methode benötigt, über die Fehlersituationen künstlich produziert werden können, ohne dass dabei die Applikationsumgebung selbst beeinflusst wird (Betriebssystem, Netzwerk, über Schnittstellen angebundene Systeme). Kann das Verhalten einer Applikation in einer instabilen Umgebung umfassend getestet werden, reduziert sich die Fehlerrate, und es ist möglich, kostspielige Problemfälle im Produktionsbetrieb der Software präventiv zu unterbinden.

Fault Simulator

An dieser Stelle kommt der Fault Simulator ins Spiel. Dieses Werkzeug simuliert Fehlersituationen (sog. “Faults”), indem die gewünschten Fehler zur Laufzeit in den Applikationscode injiziert werden. Auf diese Weise können Fehlersituationen auf kontrollierte Weise erzeugt werden und wirken sich ausschließlich auf die zu testende Applikation aus, ohne dabei die Betriebssystem-, Debugging- oder Netzwerk-Umgebung zu destabilisieren. Entwickler können ihren Code seiteneffektfrei auf die generierten Fehlersituationen testen und potentielle Anomalien analysieren.

Der Vorteil dieses Ansatzes der Code-Injektion ist, dass simulierte Fehler zu jeder Zeit während des Entwicklungs- und des Testzyklus erzeugt werden können. Mit Hilfe des mitgelieferten Kommandozeilentools lassen sich zudem automatisierte Regressionstests abbilden und in den Daily Build des Entwicklungsprozesses integrieren.

Grundsätzlich unterscheidet das Werkzeug zwischen Umgebungsfehlern und Exceptions, die durch Aufrufe der .NET Framework Class Library ausgelöst werden können.

Umgebungsfehler

Umgebungsfehler sind besonders schwierig zu simulieren. Oft handelt es sich um ein unerwartetes Ereignis in der Umgebung oder Hardware der Applikation. Manchmal ist es bei Auftritt solcher Fehler kaum möglich, die Anwendung “sachte” herunterzufahren (z.B. “low memory”), in den meisten Fällen jedoch kann das Problem entdeckt und entsprechend reagiert werden (bspw. bei gesperrten oder fehlenden Dateien).

Fault Simulator ist in der Lage, vielfältige Störungen in den wichtigsten Verarbeitungsbereichen einer Applikation zu simulieren:

- Netzwerkfehler (z.B. “connection failed”)
- Speicherfehler (z.B. “low memory”)
- Datenträgerfehler (z.B. “file access denied”)
- Registryfehler (z.B. “key not found”)
- COM Fehler (z.B. “DLL not found”)

Kasten 1 gibt eine detaillierte Übersicht über die simulierbaren Umgebungsfehler.

Exceptions

Exceptions können unterschiedlichste Ursachen haben. Unerwartete Rückgabewerte, falsche Übergabeparameter, Wertebereichsüberschreitungen, Pufferüberläufe, Null-Referenzen und Typverletzungen sind nur einige Beispiele, die harte Abbrüche verursachen können. Solche Fehlersituationen liegen in der Applikationslogik begründet als auch in den Aufrufen der verwendeten System- und Laufzeitbibliotheken selbst.

Zur Simulation von Exceptions verfolgt der Fault Simulator einen statisch-analysierenden Ansatz: Grundsätzlich ist in der .NET Framework Class Library dokumentiert, welche Methoden welche Exceptions auslösen können. Wird eine solche Methode von der zu testenden Applikation aufgerufen, kann das Werkzeug die vordefinierten Exceptions an entsprechender Stelle in den Source-Code injizieren. Programmierer können auf diese Weise ihre Exception-Handler austesten, ohne den Code der Applikation selbst an den Test anpassen zu müssen.

Kasten 2 führt (ohne Anspruch auf Vollständigkeit) einige Beispiele aus den Klassen der einzelnen Namespaces in der .NET-Framework auf, für die der Fault Simulator Exceptions simulieren kann.

Anwendungsbeispiel: Fehlersimulation für.NET-Komponenten

Im folgenden wird der Fault Simulator auf eine C# Komponente angewandt, die eine Bankleitzahl validiert und zu diesem Zweck die zugrunde liegende Bankleitzahlendatei lädt (Abbildung 1).

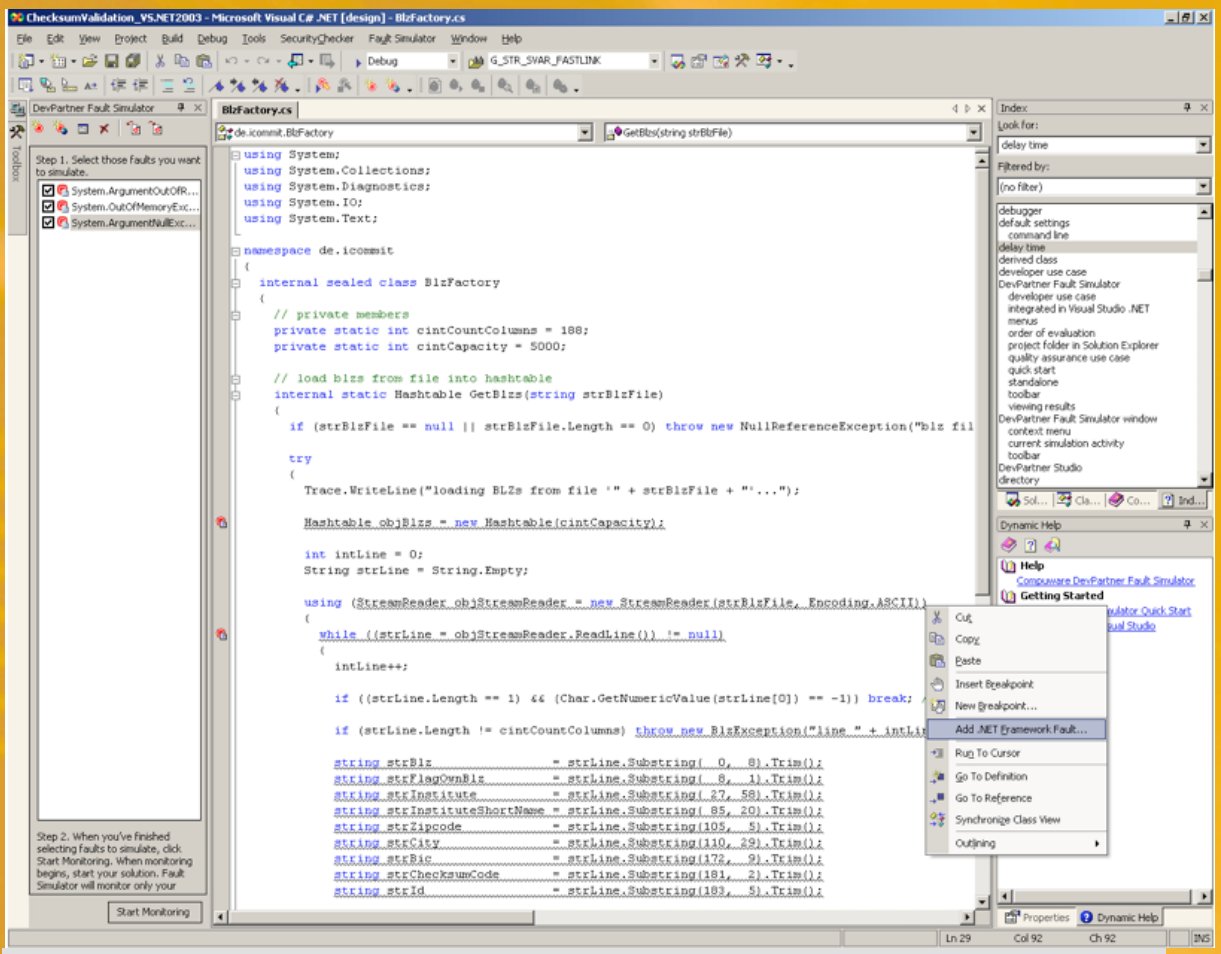

Abb. 1: Integration in Visual Studio.NET 2003

Im Code-Editor werden alle Zeilen unterstrichen dargestellt, für die der Fault Simulator eine Exception simulieren kann. Über das kontextsensitive Menu “Add .NET Framework Fault...” kann die gewünschte Exception spezifiziert werden (Abbildung 2). Im Beispiel soll eine “System.IO.FileNotFoundException” bei Aufruf des Konstruktors “StreamReader.StreamReader(String path, Encoding encoding)” erzeugt werden, falls der übergebene Pfad eine Datei namens “blz0506pc.txt” enthält.

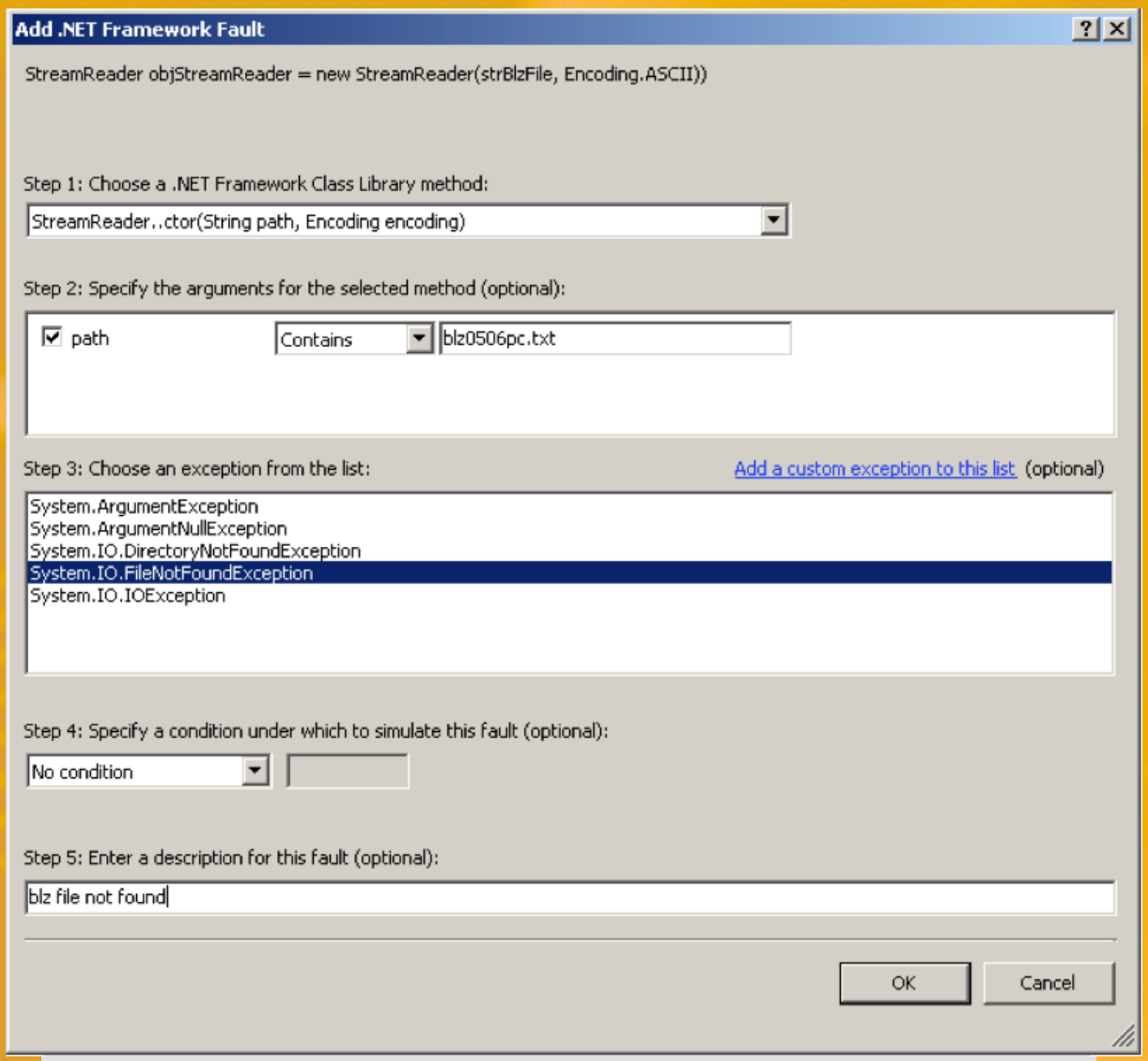

Abb. 2: .NET Framework Exception definieren

Jede so bearbeitete Code-Zeile wird in Visual Studio mit einem Icon ( ) links der Code-Zeile markiert. In dem “DevPartner Fault Simulator Pane” werden alle konfigurierten Fehlersimulation tabellarisch angezeigt und können dort vorübergehend jeweils deaktiviert und wieder eingeschaltet werden (siehe Abbildung 1).

Um mit der Simulation der gewünschten Fehler zu beginnen, muss vor dem Start der zu testenden Applikation das Monitoring des Fault Simulators aktiviert werden (“Start Monitoring”). Nun überwacht der Fault Simulator zur Laufzeit die ausgeführte Intermediate Language (IL) der Applikation und injiziert genau an den im Quelltext markierten Stellen die gewünschten Exceptions dynamisch in die ausgeführte IL hinein. Tritt eine Exception auf, erfolgt ein Sprung in den Exception-Handler der Applikation, ist dieser nicht vorhanden, terminiert das Programm. Nach Ablauf der Applikation präsentiert das Werkzeug eine tabellarische Übersicht der insgesamt aufgetretenen Fehler (Abbildung 3).

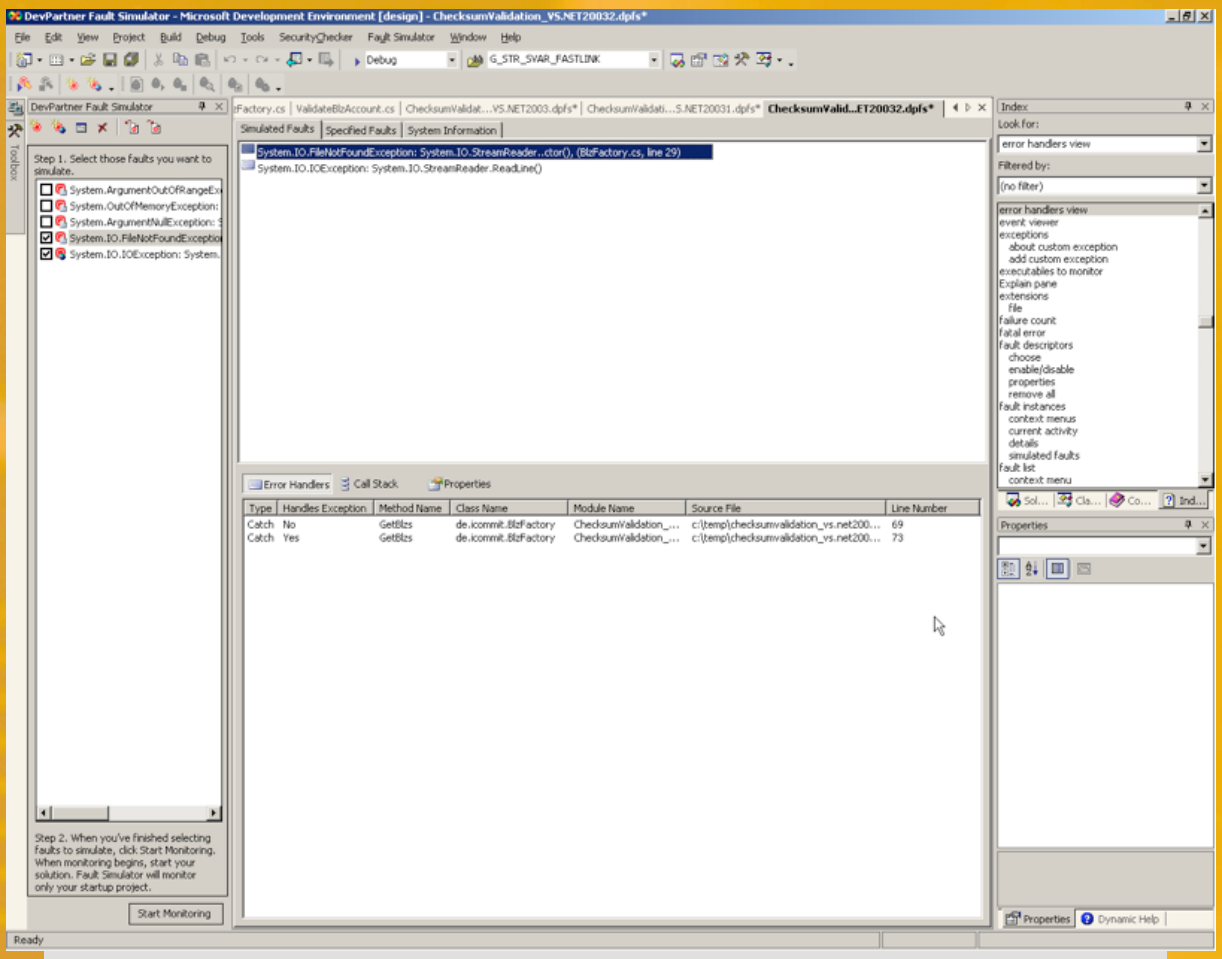

Abb. 3: Fehlersimulationsauswertung

In dem aufgeführten Beispiel wurden zwei simulierte Fehler ausgelöst:

- die gewünschte “FileNotFoundException” beim Auslesen der Bankleitzahlendaten
- eine “System.IO.IOException” bei Aufruf der Methode “StreamReader.ReadLine()”

Dem Fehler-Report kann entnommen werden, dass der Catch-Block in Zeile 69 der Komponente die aufgetretene Exception nicht abfängt, wohl aber der Catch-Block in Zeile 73.

Das Register “Call Stack” des Reports gibt zusätzlich Aufschluß über die einzelnen try/catch-Handler und den zur Laufzeit entstandenen Stack Trace (Abbildung 4).

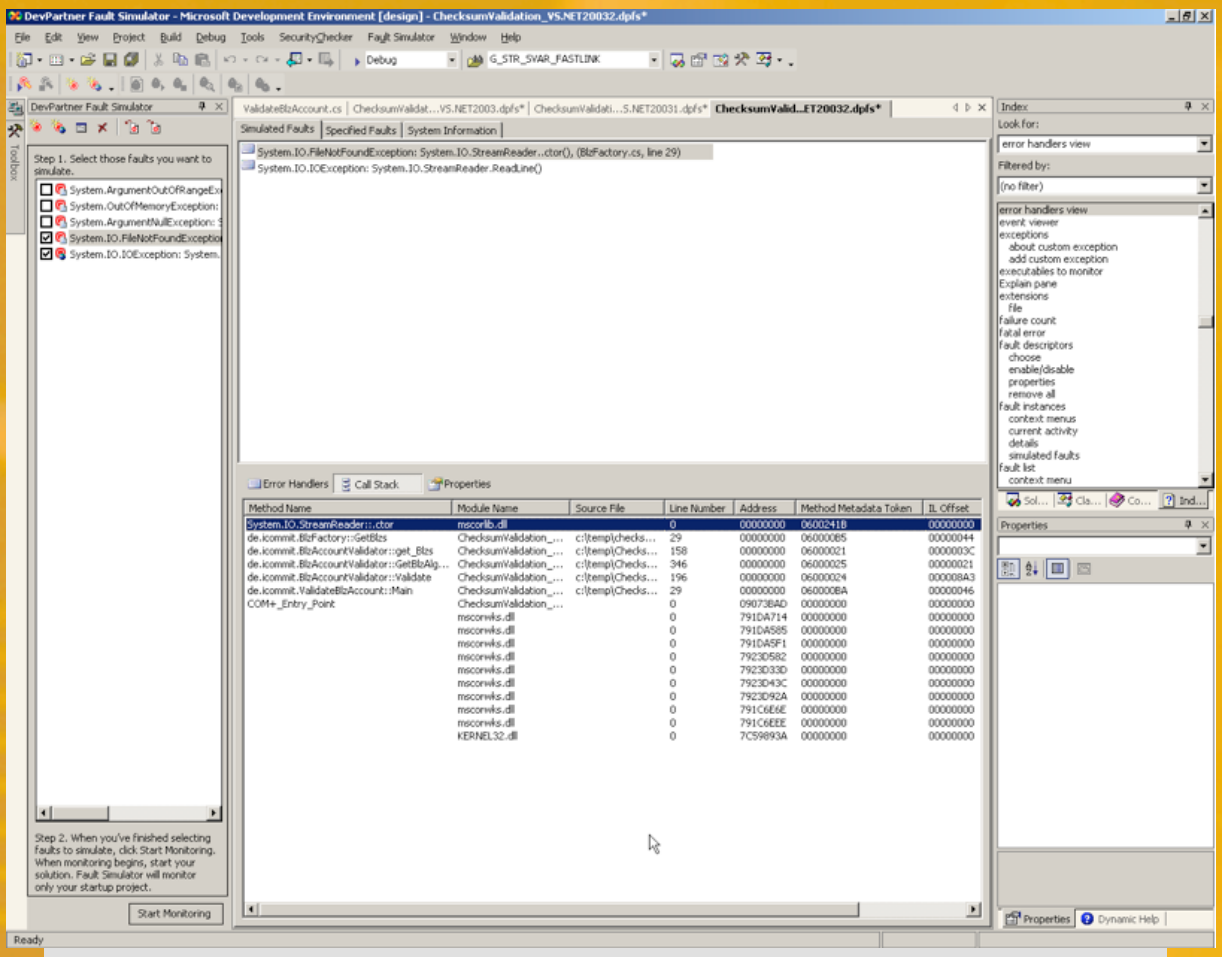

Abb. 4: Auswertung des Stack Trace

Durch Doppelklick auf eine der dargestellten Zeilen erfolgt der direkte Sprung in die gewünschte Stelle des Quelltexts, so dass der Fix dort vorgenommen werden kann oder Breakpoints zur weiteren Analyse der Fehlerbehandlung gesetzt werden können.

Anwendungsbeispiel: Umgebungsfehler für Black-Box-Tests

Mit Hilfe der Stand-Alone-Version des Werkzeugs lassen sich Umgebungsfehler für beliebige .EXE-Dateien, COM+ Applikationen und Web-Anwendungen simulieren. Eine Code-Injektion ist ohne Integration in Visual Studio allerdings nicht möglich, daher können ausschließlich allgemeine Umgebungsfehler simuliert werden (siehe Kasten 1).

Umgebungsfehler können nicht nur für .NET-Applikationen erzeugt werden, sondern auch für beliebige C-Applikationen und sonstige Executionables. Im Beispiel der Abbildung 5 werden verschiedene IO- und Netzwerk-Fehler beim Laden und Speichern einer Datei im Notepad simuliert.

Abb. 5: Stand-Alone-Version

Nach Beendigung des Notepads wird ein kurzer Report über die ausgelösten Fehler dargestellt (Abbildung 6). Im vorliegenden Fall hatte der Anwender keine Leseberechtigung für die Datei “D:\blz.txt”.

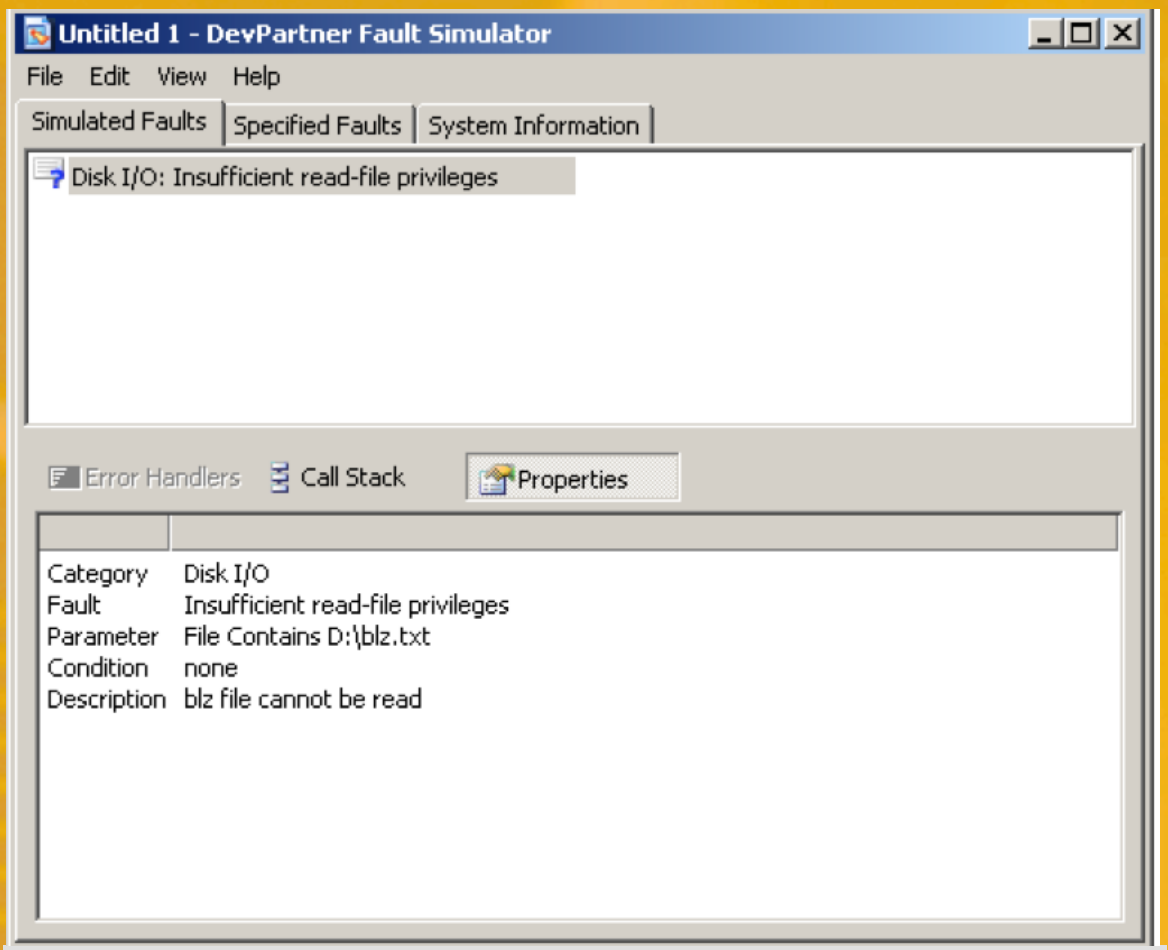

Abb. 6: Fehlersimulation im Notepad

Um das Fehlerverhalten von COM+ und Web-Applikationen zu analysieren, liest das Werkzeug den COM+ Katalog und die IIS-Metabase des lokalen Rechners aus (Abbildungen 7 und 8).

Abb. 7: Fehlersimulation für COM+ Applikationen

Bei der Störungssimulation von Web-Applikationen ist zu berücksichtigen, dass vor jedem Test der IIS neu gestartet werden sollte, da sich das Werkzeug nicht in laufende Prozesse einhaken kann, sondern eine frische “dllhost.exe” benötigt.

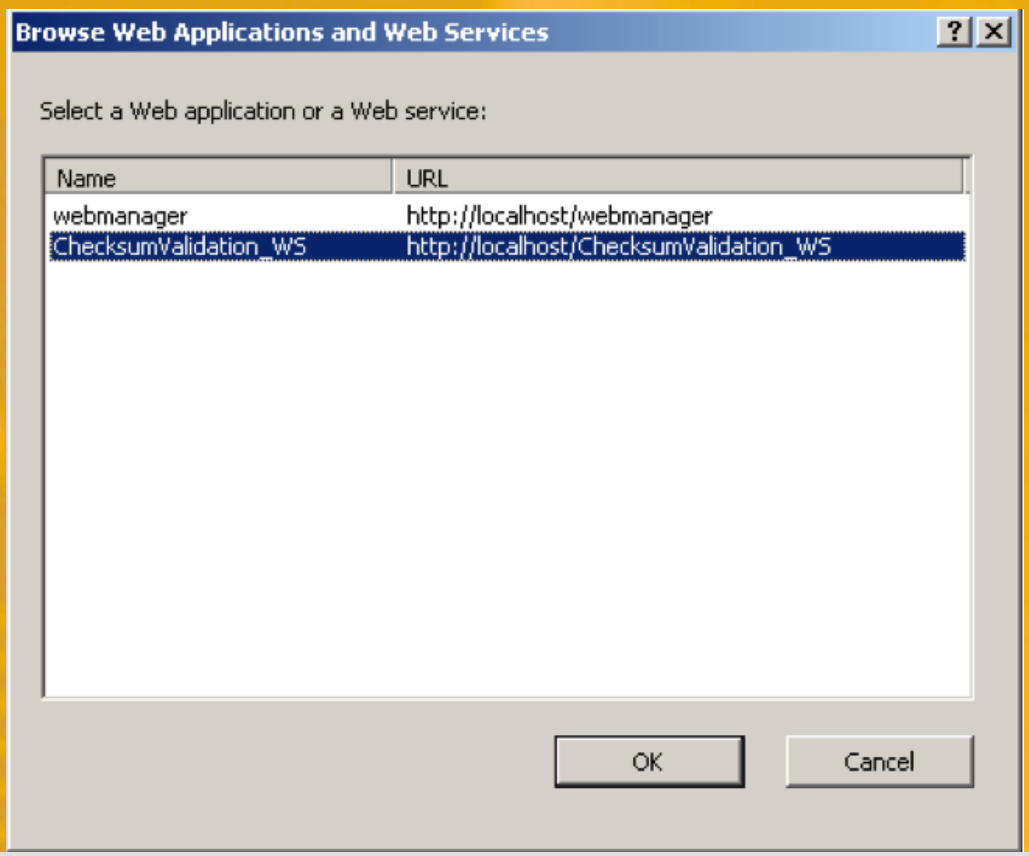

Abb. 8: Fehlersimulation für Web-Applikationen

Arbeitsweise des Fault Simulators

Wie können Fehlersituationen künstlich in einen bestehenden Quelltext eingefügt werden?

Eine mögliche Lösung wäre, die gewünschten Exceptions während des Kompiliervorgangs in die zu erstellende Assembly einzufügen. Letztere bestünde dann aus dem eigentlichen Applikationscode zuzüglich des Fehlersimulationscodes. Allerdings ergeben sich dabei ungewünschte Seiteneffekte:

1. Das Werkzeug müßte sicherstellen, dass die Applikation durch die transparente Erweiterung des IL-Codes determiniert bleibt, d.h. die Verarbeitung der Applikationslogik darf durch das künstliche Einfügen von Exceptions in die Intermediate Language in keiner Weise beeinflusst werden. Für Applikationen, die dynamische IL erst zur Laufzeit erzeugen oder "unmanaged code" verwenden, treten dabei hohe Unwägbarkeiten und Risiken auf.
2. Die erstellte Assembly enthält Code, der für die produktive Auslieferung nicht relevant ist. Schlimmer noch: bei Abschaltung der Fehlersimulation für die Erstellung einer Produktionsversion wird eine andere Assembly ausgeliefert als getestet wurde!

Beide Nachteile stellen für diesen Ansatz ein K.O.-Kriterium dar.

Der Fault Simulator macht sich einen intelligenteren Ansatz zu Nutze, der bei modernen Test-Tools bereits gang und gäbe ist: eine klare Trennung von Fehlerspezifikation und Applikationscode. Die zu erstellende Assembly bleibt unberührt. Stattdessen werden die vom Entwickler vorgegebenen Fehlersimulationen zunächst in einer XML-Konfigurationsdatei gespeichert. Ausgeführt wird bei Klick auf "Debug | Start" in Visual Studio nicht die eigentliche Applikation selbst, sondern ein sog. Jacket names "Compuware.DevPartner.FaultSimulator.Runtime.Jacket.dll". Diese Assembly lädt die zu testende Applikation

in den Hauptspeicher und führt die darin enthaltenen IL-Kommandos operationsweise aus. Trifft die Verarbeitung nun auf eine Stelle, die einer Source-Code-Zeile entspricht, für die der Entwickler eine Fehlersimulation spezifiziert hat (“IL Offset”), injiziert das Jacket dynamisch zur Laufzeit die gewünschte Exception in den Applikationscode und bringt diesen weiter zur Ausführung. Die .NET-Laufzeitumgebung behandelt die aufgetretene Exception dann wie gewohnt, indem in den entsprechenden Exception-Handler gesprungen wird (falls vorhanden) oder die Verarbeitung abgebrochen wird.

Für die Erzeugung der unterstützten Exceptions ist eine sog. Exception-Factory zuständig (“compuware.devpartner.faultsimulator.runtime.exceptionfactory.dll”).

Fehlerspezifikationen werden in einer XML-Konfigurationsdatei verwaltet (Abbildung 9). Dabei werden in einem sog. “FaultSet” alle gewünschten Fehlersimulationen zusammengefaßt. Jeder Fehler wird durch einen “FaultDescriptor” beschrieben, der u.a. die Klasse, die Methode sowie die Stelle beschreibt, an der die Exception zu injizieren ist. Beispiel für eine “System.OutOfMemoryException” bei Aufruf von “System.IO.Text.Reader.ReadLine()”:

```
<FaultDescriptor ...>
  <Condition ... />
  <ManagedLocationFault
     CallSiteAssemblyStrongName="ChecksumValidation_VS.NET2003,
                                 Version=2.0.0.0, Culture=,"
     CallSiteTypeName="de.icommit.BlzFactory"
     CallSiteMethodName="de.icommit.BlzFactory.GetBlzs(e)"
     CallSiteMethodSignature="System.String"
     CallSiteILRetOffset="456"
     StatementILOffsetInMethod="455"
     TargetTypeName="System.IO.TextReader"
     TargetMethodName="ReadLine"
     ExceptionTypeName="System.OutOfMemoryException"
     LineNumber="31"
     Column="11"
     SourceCode="while ((strLine = objStreamReader.ReadLine()) != null)"
     SourcePath="c:\temp\ChecksumValidation_VS.NET2003\BlzFactory.cs"
     UserDescription="reading blz file: out of memory"
     ...
  />
</FaultDescriptor>
```

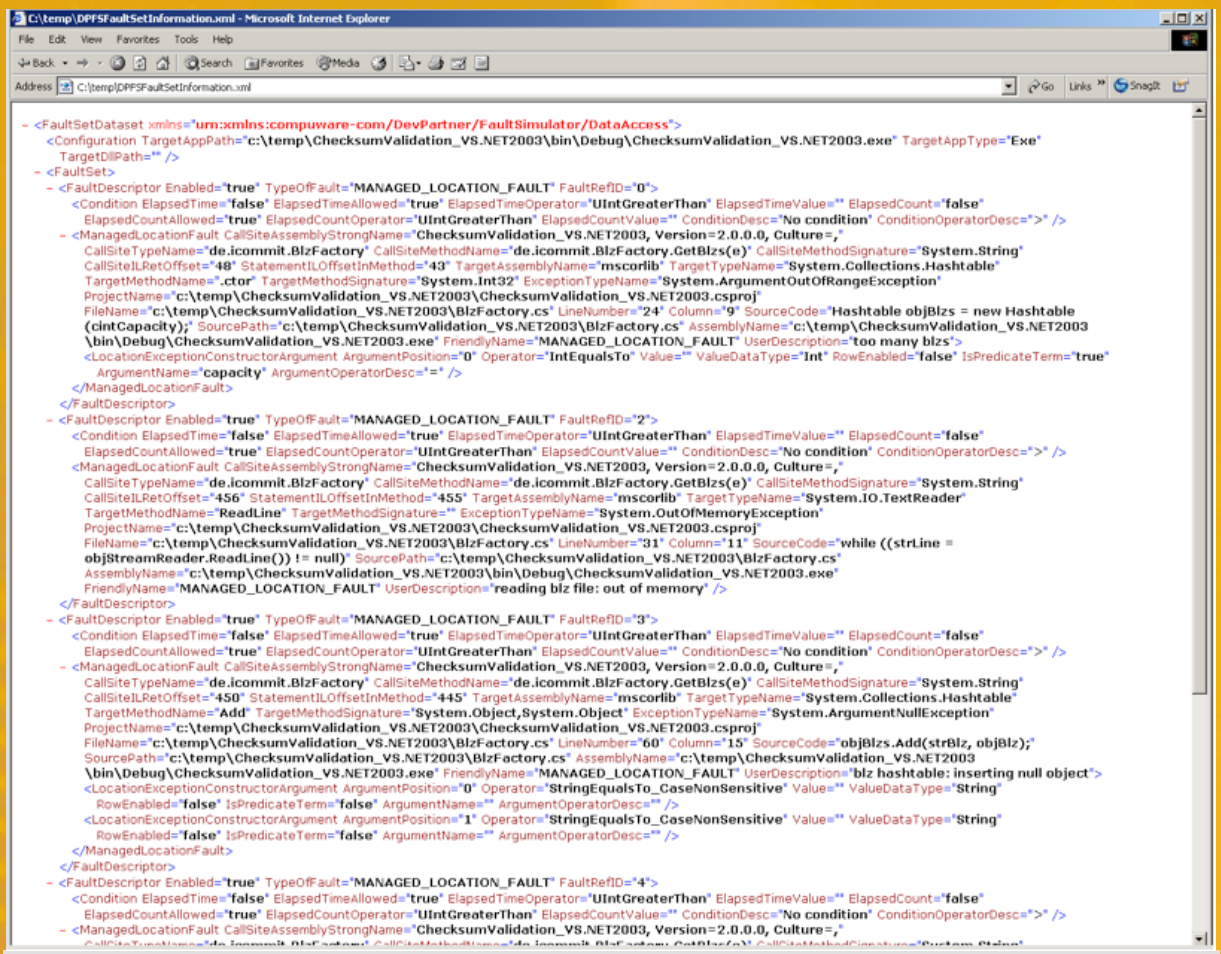

Abb. 9: XML-Beschreibung von Fehlersimulationen

Die XML-Simulationsbeschreibung enthält mehr Informationen als für die Injektion des Fehlercodes erforderlich ist, bspw. Angaben über den Source-Code. Letztere werden lediglich für das spätere Reporting benötigt.

Unterstützung des Entwicklungs- und Test-Zyklus

Durch die klare Trennung von Spezifikation und Implementation der Fehlersimulationen ist es möglich, einerseits den Entwicklungszyklus zu unterstützen und andererseits Testern und Qualitätsverantwortlichen geeignete Tools für den Testzyklus bereitzustellen. Zu diesem Zweck präsentiert sich der Fault Simulator in drei verschiedenen Versionen (Abbildung 10).

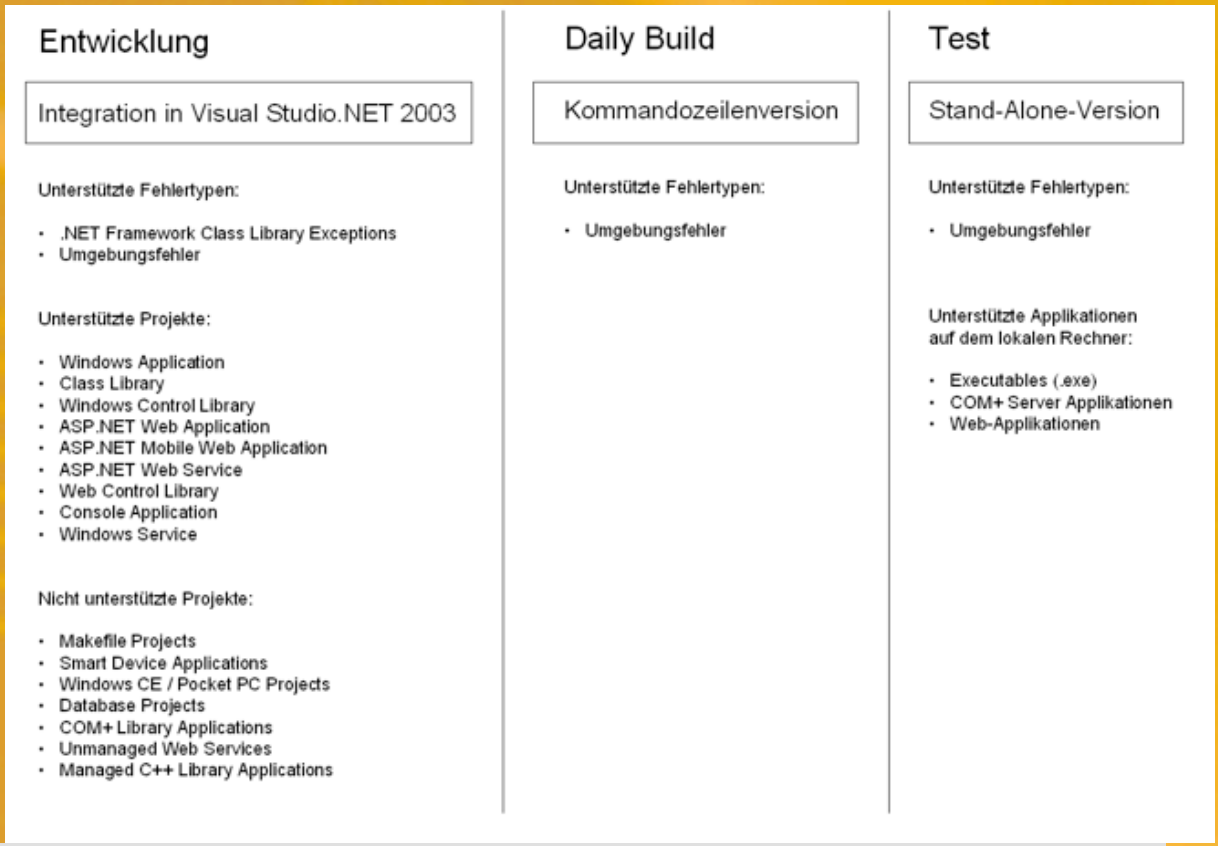

Abb. 10: Unterstützung des Entwicklungs- und Test-Zyklus

Integration in Visual Studio.NET

Zur Unterstützung des Entwicklungsprozesses läßt sich das Werkzeug nahtlos in Visual Studio.NET integrieren und ermöglicht dem Entwickler den Test und das Debuggen des Fehlerbehandlungscodes der Applikation. Statements, für die das Werkzeug Fehler simulieren kann, werden im Editor unterstrichen dargestellt, so dass allein die Markierung der relevanten Statements das Verständnis fördert, an welchen Stellen Fehler auftreten können und behandelt werden müssen. Der Fault Simulator erzeugt zur Laufzeit die gewünschten Fehler, ohne den Debugger, das Betriebssystem oder die .NET-Framework in ihrer Arbeitsweise zu stören. Die Reaktionen der Anwendung auf die Fehler werden detailliert aufgezeichnet und ausgewertet.

Stand-Alone-Version

Mit Hilfe der Stand-Alone-Applikation können Tester und Qualitätssicherungsverantwortliche einen Black-Box-Test der ausgelieferten Anwendung durchführen. In dieser Version ist allerdings keine Injektion von .NET-Exceptions möglich, es können lediglich die vordefinierten Umgebungsfehler ausgewählt werden (siehe Kasten 1). Allerdings ist es möglich, Vorbedingungen und Wiederholungsfaktoren anzugeben, so dass die gewünschten Umgebungsfehler nur in den bestimmten Fällen ausgelöst werden.

Kommandozeilenversion

Damit die Simulationstests nicht immer wieder erneut von Hand angestoßen werden müssen, läßt sich das Werkzeug in seiner Kommandozeilenversion nutzen und in automatisierte Testläufe oder Daily Builds integrieren.

Fazit

Im Durchschnitt besteht der Code einer Applikation zu ungefähr einem Drittel aus Fehlerbehandlungsroutinen. Nur ein geringer Teil dieses Codes wird vor dem Produktiv-Deployment überhaupt getestet. Und frei nach “Murphy's Law” geht grundsätzlich alles schief, was nicht getestet ist. Leider verursachen Fehler, die erst im Produktionsbetrieb einer Software entdeckt werden, in der Regel unverhältnismäßig hohe Kosten. Daher ist es für die Wirtschaftlichkeit einer Anwendung wichtig, möglichst viele potentielle Fehlersituationen bereits im Entwicklungs- und Testzyklus zu identifizieren. Auch der Entwickler gewinnt bei der Ausdehnung des Testaufwands: weniger Fehlermeldungen durch die Kunden, mehr Zeit für neue interessante Aufgaben und ein zufriedenes Gefühl, wenn man sich darauf verlassen kann, dass die Applikation auch in “schlechten Zeiten” zuverlässig läuft.

Mit der Tool-basierten Simulation von Fehlersituationen wird im Bereich der Software-Qualitätssicherung Neuland betreten. Dies macht sich bei der Evaluation des Fault Simulators bemerkbar. Während die Integration in Visual Studio eine ausgezeichnete Konfigurations- und Laufzeitumgebung bietet und darüber hinaus eine vorzügliche Simulationsarchitektur präsentiert, steckt die Unterstützung des Testbereichs noch in den Kinderschuhen. Die Stand-Alone-Version des Werkzeugs ist nicht in der Lage, .NET-Exceptions zu injizieren. Es können lediglich allgemeine Umgebungsfehler im Speicher-, Netzwerk-, I/O-, Registry- und COM-Bereich simuliert werden. Spezifische Code-Lokationen, an denen der Fehler ausgelöst werden soll, können nicht angegeben werden. An dieser Stelle sollte eine geeignete Tool-Umgebung den Tester dabei unterstützen, simulationsrelevante Stellen im IL-Code der Applikation zu identifizieren und an dedizierten IL-Offsets bestimmte .NET-Exceptions zu injizieren.

Zudem können nur Server-basierte COM+ Applikationen getestet werden, Library-Konfigurationen werden nicht unterstützt, ebenso wenig einfache COM Komponenten, die nicht im COM+ Katalog eingerichtet sind. Für COM+ Applikationen und für Web-Applikationen gilt gleichermaßen, dass die Tests lediglich auf dem lokalen Rechner durchgeführt werden können. Dies ist verständlich, da die künstliche Erzeugung von Umgebungsfehlern eine enge Integration mit dem Betriebssystem erfordert. Allerdings setzt die Installationsroutine des Fault Simulators voraus, dass Visual Studio.NET 2003 installiert ist. Will ein Tester die Applikation auf einer dedizierten Testumgebung untersuchen, macht dieses Szenario wenig Sinn: Testumgebungen sollen möglichst produktionsgleich aufgebaut sein, und auf solchen Umgebungen haben Entwicklungswerkzeuge herzlich wenig zu suchen.

Die Kommandozeilenversion unterstützt die Automatisierung der Testläufe. Leider muss der Entwickler zwecks Simulationsspezifikation die dem Testlauf zugrunde liegende XML-Konfigurationsdatei manuell editieren. Ein angepasster Editor könnte hier Fehlerquellen ausschließen, auch wäre das automatische Generieren von Skripten aus vordefinierten FaultSets wünschenswert. Bei Durchführung der Testläufe stört, dass die zu testende Applikation selbst vorher gestartet werden muss, dies erfolgt durch den Fault Simulator nicht automatisch.

Insgesamt unterstützt das Werkzeug die Simulation von Fehlern jedoch hervorragend, die Reports sind aussagekräftig und vereinfachen das Bugfixing. Es gibt grundsätzlich keinen Grund mehr, auf die tool-gestützte Durchführung von Fehlersimulationstests zu verzichten. Unter [1] können Sie eine 14tägige, kostenfreie Test-Lizenz erhalten (“Try It”).

Abschließend eine Bemerkung über den Anschaffungspreis des Werkzeugs. Software-Qualitätstools sind i.a. hochpreisig. Auch Microsoft stößt mit dem Team Foundation Server bei der Unterstützung von Entwicklern, Testern und Architekten in bisher ungewohnte Höhenordnungen. Wer in der Vergangenheit aber jemals die Mühen investiert hat, für ein Projekt die Infrastruktur der geforderten Qualitätssicherung zu schaffen, wird sich schnell nach Werkzeugen umschauen und den Anschaffungspreis mit dem eigenen Aufwand vergleichen. Preislich ist der Fault Simulator vergleichbar mit der neuen Microsoft Team Suite, die mit Visual Studio 2005 erscheinen wird. Enthalten ist eine einjährige Wartungspauschale für Patches und Updates. Da natürlich nur die .NET-Exceptions der aktuell verfügbaren .NET-Framework 1.1 unterstützt werden können, ist eine stetige Aktualisierung des Werkzeugs auf künftige .NET-Versionen erforderlich.

Manu Carus ist Software-Engineer der iCommit Integrationslösungen GmbH und stets auf der Suche nach Werkzeugen, die den Quellcode stabilisieren und die Fehlerrate reduzieren helfen. Sie erreichen ihn unter manu.carus@ethical-hacking.de.

Links & Literatur

[1] Bezugsquelle: http://www.compuware.com/products/devpartner/fault-simulator.htm

Sidebar: Voraussetzungen

Betriebssysteme

- W2K
- WinXP Pro
- Windows Server 2003

Entwicklungsumgebungen

- ausschließlich Visual Studio.NET 2003

.NET Framework

- ausschließlich .NET Framework 1.1 SP1