



Getting Started with DevPartner Fault Simulator™

Release 1.5



Technical support is available from our Technical Support Hotline or via our FrontLine Support Web site.

Technical Support Hotline:
1-800-538-7822

FrontLine Support Web Site:
<http://frontline.compuware.com>
(FrontLine Support login required)

This document and the product referenced in it are subject to the following legends:

Access is limited to authorized users. Use of this product is subject to the terms and conditions of the user's License Agreement with Compuware Corporation.

© 2005 Compuware Corporation. All rights reserved. Unpublished - rights reserved under the Copyright Laws of the United States.

U.S. GOVERNMENT RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in Compuware Corporation license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(OCT 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable. Compuware Corporation.

This product contains confidential information and trade secrets of Compuware Corporation. Use, disclosure, or reproduction is prohibited without the prior express written permission of Compuware Corporation.

TrackRecord and DevPartner Fault Simulator are trademarks or registered trademarks of Compuware Corporation.

Acrobat[®] Reader copyright © 1987-2005 Adobe Systems Incorporated. All rights reserved. Adobe, Acrobat, and Acrobat Reader are trademarks of Adobe Systems Incorporated.

All other company or product names are trademarks of their respective owners.

US Patent Nos.: 5,987,249 and 6,332,213

October 14, 2005



Table of Contents

Preface

Who Should Read This Manual	vii
Conventions Used In This Manual	vii
Accessibility	viii
For More Information	viii

Chapter 1

Installing DevPartner Fault Simulator

Minimum System Requirements	1
Coexistence with Other Compuware Products	2
Installing DevPartner Fault Simulator	3
Performing the Installation	3
New License Server Required for Concurrent Licenses	3
Installing DevPartner Fault Simulator SE	4
Performing the Installation	4
Upgrading to the Full Version	4

Chapter 2

Introducing DevPartner Fault Simulator

What Is Fault Simulation	5
Fault Simulation Enhances Software Testing	6
DevPartner Fault Simulator	7
Three Ways to Use Fault Simulator	7
Fault Descriptors	10
Fault Sets	11
Fault Simulation with Coverage Analysis	12
Using Fault Simulator in Visual Studio	14
Quick Start	14

A Fault Simulator Walk-Through	15
Using the Fault Simulator Standalone Application	19
Quick Start	19
A Fault Simulator Walk-Through	20
DevPartner Fault Simulator SE	22

Chapter 3

Improving Software Quality Through Fault Simulation

Software Development Objectives	25
Challenges to Software Quality	26
Product Instability	27
Explosion of New Technologies	27
Complexities of the Inner Workings of APIs and System Services	27
Software Vulnerabilities	28
Layers of Application Vulnerability	29
Alternatives to Achieve Software Quality	30
Conventional Testing	30
Non-Traditional Testing	30
Approaches for Testing Environmental Conditions	31
Three-Part Solution to Achieve Software Quality Using Fault Simulator	32
White Box Testing Using Fault Simulator in Visual Studio	33
Black Box Testing Using the Fault Simulator Standalone	38
Automated Testing Using Fault Simulator from the Command Line	43
Fault Simulator — Improving Software Quality	44

Chapter 4

Using Fault Simulator to Evaluate Error Handlers

Well-Constructed Error Handlers Promote Product Reliability	45
Error Handling for Function Calls	46
C++ Exception Handling	46
Structured Exception Handling	47
Configuring Fault Simulator to Capture Error Handler Data	49
Configuring a .NET Framework Fault in Managed Code	49
Configuring an Environmental Fault	51
Fault Simulation Results Views	52
Evaluating Error Handling Results	55
Determining the Path The Code Took to Unwind from an Exception	55
Identifying if Any Error Handlers Got Invoked	56
Viewing the Source Statement That Handled the Fault	57
Determining the Path The Code Took When a Function Failed	57
Assessing Whether the Intended Fault Was Handled	58

Confirming Where the Fault Was Handled	59
Fault Simulator Integral to Best Practices	60
Glossary	61
Index	63

Preface



- ◆ Who Should Read This Manual
- ◆ Conventions Used In This Manual
- ◆ Accessibility
- ◆ For More Information

Who Should Read This Manual

This manual assists software developers and quality assurance professionals who will install and use Compuware DevPartner Fault Simulator™. This manual presents the concepts and procedures fundamental to using DevPartner Fault Simulator.

This manual assumes your familiarity with the Microsoft Windows and Visual Studio technologies.

Conventions Used In This Manual

This manual uses the following conventions to present information.

- ◆ Screen commands and menu names appear in **bold typeface**. For example:
Choose **Add Environmental Fault** from the **Fault Simulator** menu.
- ◆ Computer commands and file names appear in `monospace typeface`. For example: `DPFS Getting Started.pdf`

- ◆ Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in *italic monospace type*. For example:
Enter `dpfs /help:option` at the command line prompt.

Accessibility

Prompted by federal legislation introduced in 1998 and Section 508 of the U.S. Rehabilitation Act enacted in 2001, Compuware launched an accessibility initiative to make its products accessible to all users, including people with disabilities. This initiative addresses the special needs of users with sight, hearing, cognitive, or mobility impairments.

Section 508 requires that all electronic and information technology developed, procured, maintained, or used by the U.S. Federal government be accessible to individuals with disabilities. To that end, the World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) has created a workable standard for online content.

Compuware supports this initiative by committing to make its applications and online help documentation comply with these standards. For more information, refer to:

- ◆ W3C Web Accessibility Initiative (WAI) at www.W3.org/WAI
- ◆ Section 508 Standards at www.section508.gov
- ◆ Microsoft Accessibility Technology for Everyone at www.microsoft.com/enable/

For More Information

You can access the following DevPartner Fault Simulator documentation for more assistance:

- ◆ **Fault Simulator in Visual Studio** — Online help integrates with Fault Simulator in Visual Studio.
- ◆ **Fault Simulator standalone application** — HTML-based online help accompanies Fault Simulator standalone.
- ◆ **Command line**
 - ◇ Console help available using Fault Simulator from the command line
 - ◇ HTML-based online help available from InfoCenter
From the **Start** menu, choose **All Programs > Compuware Dev-Partner Fault Simulator > InfoCenter**.

- ◆ **Getting Started with DevPartner Fault Simulator™** — Manual resides on your DevPartner Fault Simulator CD in Adobe Acrobat (.pdf) format.

Use these other resources for additional assistance:

- ◆ The HTML-based **Release Notes** provide late-breaking information and any known anomalies in this release.
- ◆ The manual, *Distributed License Management License Installation Guide* (**LicInst4.PDF**), provides specific details on licensing Compuware products.

We recommend the following books related to fault simulation:

- ◆ *Applied Microsoft .NET Framework Programming*, by Jeffrey Richter. Web link: <http://www.microsoft.com/mspress/books/5353.asp>
- ◆ *Software Fault Injection: Inoculating Programs Against Errors*, by Jeffrey M. Voas and Gary McGraw. Web link: <http://www.cigital.com/books/sfi>

Chapter 1

Installing DevPartner Fault Simulator



- ◆ Minimum System Requirements
- ◆ Installing DevPartner Fault Simulator
- ◆ Installing DevPartner Fault Simulator SE

This chapter describes the supported platforms, and operating development environments. It explains how to install the software.

Minimum System Requirements

The following table enumerates the system requirements and supported operating and development environments for Fault Simulator.

Table 1-1. Minimum System Requirements, Supported Operating Systems and Environments

Processor	Memory	Disk Space	Video	Other
Pentium III 733 MHz (1 GHz or faster recommended)	512 MB	500 MB	1024 X 768, 16-bit color	CD or DVD drive
Operating System	Service Pack	Edition	Browser	Internet Information Server (IIS)
Windows 2000	4	Professional, Server, Advanced Server	Internet Explorer 5.5	IIS 5.0
Windows XP (32 bit)	2	Professional ¹	Internet Explorer 5.5	IIS 5.1
Windows Server 2003 (32 bit)	1	Standard, Enterprise, Web	Internet Explorer 6.0	IIS 6.0

Table 1-1. Minimum System Requirements, Supported Operating Systems and Environments (Continued)

Integrated Development Environment (IDE) ²	Edition	.NET Framework Version	IDE Service Pack	Languages
Visual Studio .NET 2003	Architect Professional	.NET Framework 1.1 Service Pack 1 (Compact Framework not supported)	None	Visual Basic .NET, Visual C#, ASP.NET technologies, and native C++ (for environmental faults only)
Visual Studio 2005	Team Suite Professional Team Edition for Software Architects Team Edition for Software Developers Team Edition for Software Testers	.NET Framework 2.0	None	Visual Basic .NET, Visual C#, ASP.NET technologies, and native C++ (for environmental faults only)

¹Home and Media Center Editions are not supported.

²If no IDE is present, Fault Simulator will install the application without the IDE interface. If the .NET Framework is not installed, Fault Simulator will install .NET Framework 1.1 SP1.

Note: Fault Simulator does not support Windows 9.x, Windows ME or Windows NT operating systems, Windows 2000 Server Data Center, and Windows XP hardware-specific editions. In addition, Fault Simulator does not support Java, JavaScript or Visual J# in ASP.NET applications.

Coexistence with Other Compuware Products

Fault Simulator integrates with Compuware TrackRecord 6.2 and coexists with other Compuware DevPartner products, such as:

- ◆ DevPartner Studio 8.0 or later
- ◆ DevPartner SecurityChecker 1.0.1 or later
- ◆ Compuware TestPartner 5.3.0

Installing DevPartner Fault Simulator

DevPartner Fault Simulator detects your system configuration and installs the applicable software setup. If your system configuration does not meet the minimum system requirements, the pre-installer will alert you to the problem and exit from the installation.

Performing the Installation

To install DevPartner Fault Simulator:

- 1 Insert the product CD into your CD-ROM drive.
If you have autorun enabled, the setup runs automatically. If not, open the **Add or Remove Programs** control panel, click **Add New Programs**, and then click **CD or Floppy**.
- Note:** If you receive a message from the virus protection software installed on your system, disable that software until Fault Simulator completes installation.
- 2 Click **Install DevPartner Fault Simulator**.
- 3 Follow the on-screen instructions to perform the remainder of the installation.
- 4 If necessary, click **Configure a License** to enter your license file information.
See “[New License Server Required for Concurrent Licenses](#)” below for more information.
- Note:** You can skip this step and use the 14-day evaluation instead.
- 5 Click **Register Your Product** to complete the online product registration.

New License Server Required for Concurrent Licenses

DevPartner Fault Simulator 1.5 uses a new version of Compuware Distributed License Management (DLM 4.0). To obtain concurrent licensing, you must use the DLM 4.0 license server. For more information on license installation, refer to the *Distributed License Management License Installation Guide (LicInst4.PDF)* on the product CD.

Note: DevPartner Fault Simulator SE uses the same license as DevPartner Studio 8.0.

To proceed, follow the instructions provided in your product packaging. You can also submit your request for a license file by visiting:

www.compuware.com/license

If you do not have a license file at the time of installation, you can install DevPartner Fault Simulator as a 14-day evaluation.

Installing DevPartner Fault Simulator SE

DevPartner Fault Simulator SE contains a subset of DevPartner Fault Simulator features. See “[DevPartner Fault Simulator SE](#)” on page 22 for more information.

Performing the Installation

To install DevPartner Fault Simulator SE:

- 1 Insert the DevPartner Studio CD into your CD-ROM drive.
You will find the installation setup for DevPartner Fault Simulator SE on the DevPartner Studio Professional Edition 8.0 CD. If you do not have autorun enabled, use the **Add or Remove Programs** control panel to install.
- 2 Follow the on-screen instructions to perform the remainder of the installation.
- 3 Click **Register Your Product** to complete the online product registration.

Upgrading to the Full Version

At any time, you can upgrade to the full DevPartner Fault Simulator 1.5 product. However, in order to install DevPartner Fault Simulator, you must first completely uninstall DevPartner Fault Simulator SE. For details on upgrading, choose **All Programs > Compuware DevPartner Fault Simulator SE > How to Upgrade to Fault Simulator**. from the **Start** menu.

Chapter 2

Introducing DevPartner Fault Simulator



- ◆ What Is Fault Simulation
- ◆ DevPartner Fault Simulator
- ◆ Using Fault Simulator in Visual Studio
- ◆ Using the Fault Simulator Standalone Application
- ◆ DevPartner Fault Simulator SE

This chapter will discuss how fault simulation enhances software testing. It will introduce you to DevPartner Fault Simulator and its three interfaces. The chapter will also contrast the full version with DevPartner Fault Simulator SE, available in limited form in DevPartner Studio 8.0.

What Is Fault Simulation

Fault simulation provides a safe and reliable mechanism to simulate a wide range of failures in a running application (i.e., programming errors, network crashes, corrupt registry values, and improperly or unhandled exceptions). It helps developers evaluate potential anomalies. Fault simulation highlights possible weaknesses that can arise under diverse circumstances. It focuses on what happens in the application after a simulated fault occurs and how the application handles the fault condition.

Fault simulation serves as a safe, non-intrusive alternative to typical software testing:

- ◆ Helps to validate the robustness of a running application
- ◆ Tests the error handling capabilities in the code
- ◆ Assists quality assurance in regression testing to check the reliability of the running program in various operating environments

- ◆ Provides a better understanding of how software will behave under adverse circumstances, without jeopardizing the software under test
- ◆ Helps to anticipate future application anomalies missed during traditional testing cycles, that could surface after deployment

Consider this example. Say that you want to evaluate how your application withstands a network failure. You might physically disconnect a cable from the network or you might manually force excessive network traffic. These actions, while appropriate, could introduce other unintended and catastrophic problems.

However, using fault simulation, you can artificially cause these failures and then observe how your application handles the failed calls. This approach achieves the same objectives without disrupting the application under test or the operating environment.

Fault Simulation Enhances Software Testing

Undetected software bugs account for a substantial portion of software development costs. Traditional testing methods attempt to catch as many abnormalities in the code as possible. However, they inevitably overlook hidden bugs that could undermine application performance.

Tip: See “Conventional Testing” on page 30.

Traditional software testing can be inexact, unreliable, and costly to the entire development process, with little return on investment. Developers find it difficult to anticipate where bugs might occur, and when. They shy away from manually causing faults that disrupt the application and corrupt the debugging environment. Developers and testers alike are reluctant to manually trace bugs from their origin to their ultimate error handling.

Approximately one third of application code is dedicated to error handling. However, only a small portion of that code is actually tested prior to deployment. As a result, error handling code accounts for most of the undetected defects in application code.

Software development teams have dedicated quality assurance resources to testing and debugging code. Despite satisfactory success in catching real-world problems in the code, they have been less effective in testing error-handling code. Quality assurance technicians lack an easy, repeatable, and safe method to generate faults within the testing and debugging environment to uncover and fix defects.

Fault simulation provides an easy, repeatable, and safe alternative to artificially generating failures in the operating environment. Fault simulation helps you analyze, test, and debug error-handling code, by safely simulating faults into the code and checking for the desired result.

Using fault simulation, simulated faults help verify whether the program handles the error as expected.

DevPartner Fault Simulator

DevPartner Fault Simulator uses fault simulation to mimic real-world application errors. Fault Simulator helps developers and quality assurance professionals simulate faults in a running program. They can test an application's reaction to errors in a predictable and repeatable environment. Using actual simulation results, they can verify the application's ability to tolerate a variety of failure conditions prior to deployment, avoiding costly production errors afterwards.

Fault Simulator helps developers and quality assurance professionals by:

Tip: See “Fault Descriptors” on page 10 to learn about the two types of fault descriptors.

Tip: See “Configuring Fault Simulator to Capture Error Handler Data” on page 49 for examples of fault simulation results.

- ◆ Simulating faults:
 - ◇ You choose a fault descriptor on the **DevPartner Fault Simulator** window.
 - ◇ You can configure a .NET Framework fault, either on a source statement or independent of location in the solution.
 - ◇ You can configure an environmental fault to mimic real-world failures.
- ◆ Generating results:
 - ◇ Fault Simulator displays ongoing activity as the faults are simulated and handled.
 - ◇ Fault Simulator provides details results on simulated faults following the session.
- ◆ Providing troubleshooting tips:
 - ◇ Fault Simulator traces calls back to the point where a fault occurred and the error was handled.
 - ◇ Fault Simulator records program functions, stack tracing, and other fault details to aid troubleshooting.

Three Ways to Use Fault Simulator

You can test the error handling in your code in three interfaces.

Fault Simulator in Visual Studio

You can use Fault Simulator in Visual Studio to test and debug error handlers in managed code. Supporting hundreds of .NET Framework Class Library methods, Fault Simulator can simulate a wide range of

exceptions defined for those methods. Fault Simulator can also simulate environmental faults in running applications. Fault Simulator simulates faults without disrupting the debugger, operating system, or the IDE.

Fault Simulator Standalone Application

You can use Fault Simulator as a standalone by accessing it from the **Start** menu. With Fault Simulator, you can create and configure new environmental faults or modify existing .NET Framework faults (originally created in the IDE) to simulate error conditions in a running program.

Using the standalone application, quality assurance professionals can validate applications under development. Quality assurance can use fault sets configured in development for regression testing of the running program. They, in turn, can redirect simulation results back to development for error handling modifications.

Fault Simulator Command Line Interface

You can use Fault Simulator to automate fault simulations on projects that do not require user intervention. From the command line, you can execute scripts to automate regression testing. The command line supports any fault sets previously configured in Fault Simulator. Results generated from the command line are available for viewing in the Fault Simulator user interface.

From the command line, use the `dpfs.exe` command with one or more of the following command line options. Note that `/faultset` is required along with a choice of `/application`, `/COM+`, or `/url`.

```
>dpfs /faultset:<filespec>
    /application:<exepath> | /COM+:<component> | /url:<vroot>
    [/results:<filespec>
    [/launch | /launch:<exepath>] [/arguments:<"args args">]
    [/startin:<folderpath>] [/coverage]
    [/help | /help:<keyword> | /? | /?:<keyword>]
```

Note: See [Figure 3-14](#) on page 43 for guidance on automating Fault Simulator from the command line.

Refer to the DevPartner Fault Simulator Command Line online help for complete information about these command line options.

Supported Features in Each Interface

Features vary depending on how you use Fault Simulator. The following table summarizes these supported features:

Table 2-1. Supported Features in DevPartner Fault Simulator

Feature	In Visual Studio	From the Standalone	From the Command Line
Simulate faults in application code	Yes	Yes	Yes
Simulate faults in Visual Studio	Yes	No	No
Monitor the designated startup project	Yes	No	No
Add a .NET Framework fault to a source statement	Yes	No	No
Add a .NET Framework fault independent of location	Yes	No	No
Configure settings for a new .NET Framework fault	Yes	No	No
Modify an existing .NET Framework fault	Yes	Yes	No
Add an environmental fault	Yes	Yes	No
Modify an existing environmental fault	Yes	Yes	No
Review collected call stack and error handler simulation details	Yes	Yes	No
View source code associated with a fault instance configured at a specific source statement	Yes	No	No
Execute scripts to automate fault simulations from the command line	No	No	Yes
Use fault sets in command line batch processing	No	No	Yes

Fault Simulator simulates faults without disrupting the debugger, operating system, or the IDE. Fault Simulator records all program activities pertaining to how faults were handled in the source code. Fault Simulator displays the information as the simulation proceeds, with the final results available in a results file. You can view the results file immediately, or save to a user-defined file for later review. Fault Simulator

also lets you navigate to the original source statement, if available, helping you to troubleshoot error handling anomalies.

Fault Descriptors

Fault Simulator uses the concept of *fault descriptors* to define fault types it can simulate. Fault Simulator supports the following fault descriptors:

- ◆ NET Framework Faults
- ◆ Environmental Faults

Fault Simulator evaluates properties to determine whether to simulate a fault. A fault descriptor includes one or more properties.

Note: See [Table 4-1](#) on page 54 for more information on properties.

NET Framework Faults

.NET Framework fault represents a specific exception that a method call of a .NET Framework Class Library class can throw. You configure .NET Framework faults to test the exception handling in your application code.

Fault Simulator supports two types of .NET Framework faults. You can add a .NET Framework fault:

- ◆ To a source statement that contains a .NET Framework Class Library method call with an exception that can be simulated during a fault simulation.
- ◆ That is simulated independent of location.

Fault Simulator supports .NET Framework faults based on the following .NET Framework Class Library namespaces:

- ◆ System
- ◆ System.Collections
- ◆ System.Collections.Specialized
- ◆ System.Data
- ◆ System.Data.Common
- ◆ System.Data.SqlClient
- ◆ System.IO
- ◆ System.Net
- ◆ System.Net.Sockets
- ◆ System.Runtime.Remoting
- ◆ System.Runtime.Remoting.Channels
- ◆ System.Runtime.Remoting.Messaging
- ◆ System.Runtime.Remoting.Proxies
- ◆ System.Runtime.Remoting.Services
- ◆ System.Text

Note: Refer to the DevPartner Fault Simulator online help in Visual Studio for more information on .NET Framework faults.

Environmental Faults

Environmental faults let you validate the robustness of your monitored program by simulating external dependencies that your application requires. For example, you might configure a network offline fault to affect network-based method calls, such as connecting to a remote server, without affecting other applications running on the system.

The following table describes the environmental fault categories provided in Fault Simulator.

Table 2-2. Environmental Fault Categories

Fault Category	Fault Description
COM	Generated as a result of calls to the COM function call
Disk I/O	Related to file and directory access
Memory	Associated with memory management failures
Network	Associated with network failures
Registry	Related to any methods or functions that use Registry services

Note: Refer to the DevPartner Fault Simulator online help for more information on environmental faults.

Fault Sets

Fault Simulator lets you create and use *fault sets*, collections of saved fault descriptors used in fault simulation. Fault Simulator displays the current fault set contents in the **DevPartner Fault Simulator** window.

You can:

- ◆ Load a previously configured fault set file
- ◆ Save faults to a new file (.dpfsfault)
- ◆ Pass uniquely named fault set files back and forth between development and quality assurance groups

You can use a fault set, created in Visual Studio, in the Fault Simulator standalone application or vice versa. You can also execute a script and reference a fault set file from the command line.

Fault Simulation with Coverage Analysis

You can use Fault Simulator with DevPartner coverage analysis to collect coverage and fault simulation data during the same session.

Required: DevPartner Studio must reside on the same system as Fault Simulator to perform coverage analysis with fault simulation.

Tip: Refer to the Coverage Analysis online help in DevPartner Studio for more information.

DevPartner Studio provides coverage analysis to help developers and test engineers thoroughly test an application's code. DevPartner Studio can collect coverage data for managed code applications, including Web and ASP.NET applications. The coverage analysis feature gathers coverage data for applications, components, images, methods, functions, modules, and individual lines of code. The coverage session file uses the .dpcov file extension.

Coverage analysis is available in Fault Simulator if the following conditions are met:

- ◆ You have DevPartner Studio 8.0 or later installed on the same system.
- ◆ You have configured at least one fault descriptor.
- ◆ Another fault simulation is not already in progress.
- ◆ You have loaded a solution.¹
- ◆ At least one supported startup project meets the project requirements.¹

¹ These items apply to Fault Simulator in Visual Studio only. Consult the DevPartner Fault Simulator online help in Visual Studio for more information.

The following example shows how Fault Simulator combines a fault simulation session with coverage analysis.

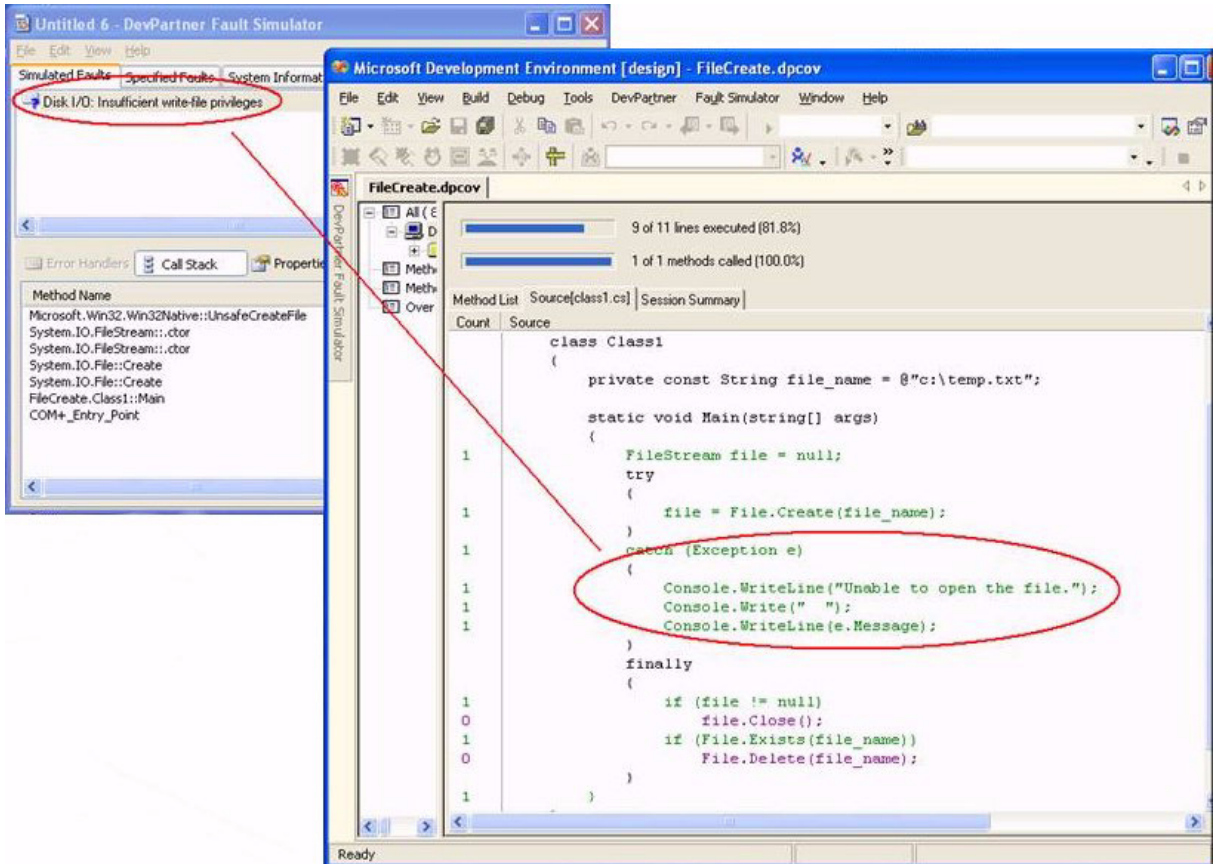


Figure 2-1. Example of Coverage Analysis with a Fault Simulation in Visual Studio

In this example, the developer used the *Disk I/O insufficient write-file privileges* environmental fault descriptor to cause the `File.Create()` API to fail.

Upon the completion of the fault simulation, Fault Simulator displays a results window (shown to the left in the previous illustration) with call stack and error handler data shown. Concurrently, the coverage results reveal whether applications and components have been thoroughly exercised under test conditions.

Tip: Consult *Understanding DevPartner* for more information on code coverage analysis.

Using Fault Simulator in Visual Studio

The first section gets you started using Fault Simulator in Visual Studio. The second section presents a user scenario.

Quick Start

Tip: See “Minimum System Requirements” on page 1 in Chapter 1, “Installing DevPartner Fault Simulator”.

This quick start guides you on using DevPartner Fault Simulator in Visual Studio. For more information, consult the DevPartner Fault Simulator online help in Visual Studio.

- 1 From Visual Studio, follow standard procedures to open a solution. If the **DevPartner Fault Simulator** window is not already open, click the **DevPartner Fault Simulator** tab (left margin) to display.



Figure 2-2. Example of **DevPartner Fault Simulator** Window in Visual Studio

- 2 Ensure that you have designated a valid startup project in your solution.
 - a Check that this project is a supported project type.
 - b Check that it meets the requirements for fault simulation.
 - c Check that you have properly configured the solution properties for monitoring.
- 3 Review the current fault list.

Tip: Consult the DevPartner Fault Simulator online help in Visual Studio for more information.

- 4 Select the check box of any fault descriptors you want activated during the next fault simulation.
 - 5 Optionally add and configure a new fault.
 - 6 Click the **Start with Fault Simulator** button.
This button is enabled as long as you have selected and properly configured at least one fault descriptor and also met the requirements listed in [step 2](#).
This action also launches the startup project targeted for fault simulation.
- Note:* If you designated more than one startup project in your solution, the **Choose Project to Simulate** dialog box will prompt you to pick a project, a requirement to start the fault simulation.
- 7 View current fault simulation details as they appear on the **DevPartner Fault Simulator** window.
 - 8 Click **End Simulation** to stop the current fault simulation.
Final results will immediately appear in a separate window.

Required: This action does not automatically stop the target program. You must terminate the process separately. The fault simulation will automatically stop if the targeted program terminates.

- 9 View the details in the results window.
You can view simulated faults, specified faults, or system information, as depicted in the next example. When you select a fault on the **Simulated Faults** pane, you can access its **Error Handlers**, **Call Stack**, or **Properties** views.

Tip: Refer to [Chapter 4](#), “Using Fault Simulator to Evaluate Error Handlers” for more details about results.

A Fault Simulator Walk-Through

As software development lead, you want to verify the exception handling for the .NET Framework method `DefineDynamicAssembly`. You decide to simulate the exception `System.ArgumentNullException` when a specific source statement executes and proceed with these steps:

From Visual Studio and with the desired solution open, you set properties for the startup project. You right-click on the source statement in the startup project that calls the `DefineDynamicAssembly` method.

To help you find the correct source line, Fault Simulator highlights any source statements with at least one supported method, such as in the following example.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WindowsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            System.Console.WriteLine();
            System.Console.WriteLine(styles());
            System.Console.WriteLine(m1());
        }
    }
}
```

Fault Simulator highlights source that contains a supported method.


Hover over the desired source line for the fault simulation tooltip.

Figure 2-3. Example Showing Source Highlighting

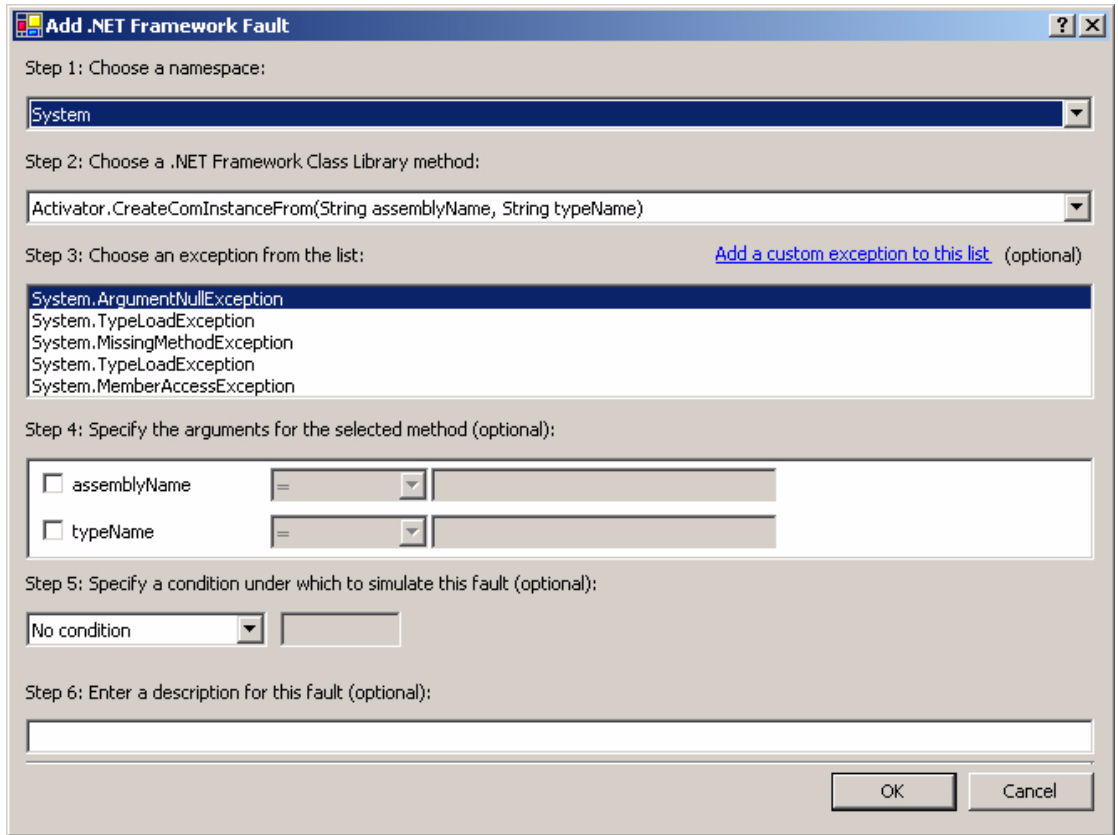
From the context menu on the source window, you select **Add .NET Framework Fault**. The **Add .NET Framework Fault** dialog box (Figure 2-4 on page 17) displays the specific information related to that fault descriptor.

You then select other configuration settings for the fault descriptor.

To track `DefineDynamicAssembly` elsewhere in the solution, you create a location-independent fault descriptor to be simulated any time your source code calls that method.

You click the  toolbar button on the **DevPartner Fault Simulator** window to display **Add .NET Framework Fault** dialog box.

Tip: Consult the DevPartner Fault Simulator online help in Visual Studio for details on configuring a new .NET Framework fault.



Configuration settings
(Argument, condition, and description are optional.)

Figure 2-4. Example of Add .NET Framework Fault Dialog Box

You proceed to configure the new fault descriptor. You decide not to specify an optional condition (delay time or skip count) or a description for this fault descriptor.

You verify the remaining settings on the **DevPartner Fault Simulator** window.



Figure 2-5. Example Shows List of Fault Descriptors

Once satisfied, you click the **Start with Fault Simulator** button. This action automatically launches the target program.

You view the ongoing fault simulation details appearing in the **DevPartner Fault Simulator** window. You observe that Fault Simulator simulates the exception at the specified source location.

Satisfied with the current outcome, you click **End simulation**. The Results window automatically displays a summary of the final simulation details.

You stop the target program. You use the information in the results window to troubleshoot problems with your error handling code.

Using the Fault Simulator Standalone Application

The first section provides steps gets you started using the Fault Simulator standalone application. The next section provides an example of a user scenario.

Quick Start

This section shows you how to start using the Fault Simulator standalone application.

- 1 Start Fault Simulator from the **Start** menu by choosing **Programs > Compuware DevPartner Fault Simulator > Fault Simulator**.

The **DevPartner Fault Simulator** window automatically appears.



Figure 2-6. DevPartner Fault Simulator Window in Fault Simulator Standalone

- 1 Select the simulation target by browsing for an executable, COM+ component, or Web application.
- 2 Determine the fault descriptors you want to use during the next simulation.

Note: You can create and configure a new environmental, but you can only modify an existing .NET Framework fault.

Tip: For more information on configuring the fault simulation and adding, modifying, enabling, or disabling faults, consult the DevPartner Fault Simulator online help.

- 3 When ready, click the **Start Simulation** button.
This button is disabled if you have not properly configured at least one fault descriptor.
- 4 Start the target program.
The **DevPartner Fault Simulator** window reminds you if you forget.
- 5 View current activity during monitoring.
The **DevPartner Fault Simulator** window displays current fault simulation details.
- 6 Click the **End Simulation** button.
Note: Fault Simulator automatically stops monitoring if the targeted program terminates.
- 7 View the details in the results window.
You can view simulated faults, specified faults, or system information, as shown next. When you select a fault on the **Simulated Faults** pane, you can access its Error Handlers, Call Stack, or Properties views.

A Fault Simulator Walk-Through

As quality assurance lead, you need to test the error handling of the application under development, and report deficiencies back to development. The software development team wants you to test possible registry anomalies in the executable.

You launch the Fault Simulator standalone application from the **Start** menu on the desktop.

From the **DevPartner Fault Simulator** window, you click **Add** to create a new environmental fault.

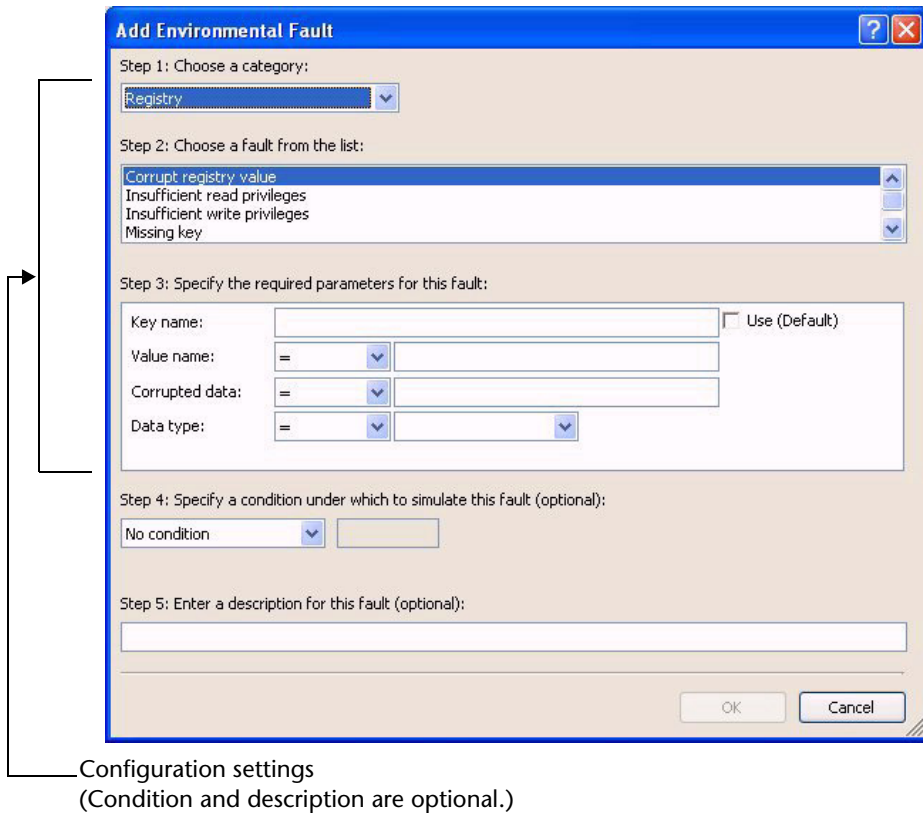


Figure 2-7. Example of Add Environmental Fault Dialog Box

Note: This dialog box is the same as the one that appears when using Fault Simulator in Visual Studio.

You proceed to configure the new fault descriptor. You decide not to specify an optional condition (delay time or skip count) or a description for this fault descriptor. Once satisfied, you click the **Start Simulation** button. The **DevPartner Fault Simulator** window begins the simulation and prompts you to start the target application.

You start the target executable. Fault Simulator simulates the registry fault occurrence in the **DevPartner Fault Simulator** window. You click the **End Simulation** button and also stop the executable. Fault Simulator automatically displays the result window.

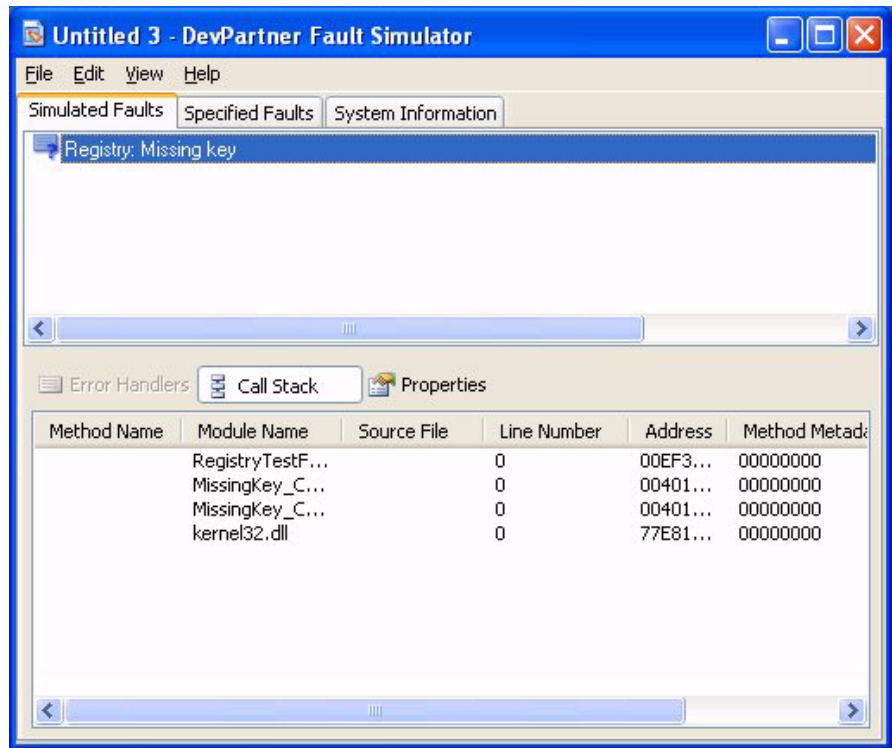


Figure 2-8. Example of Results Window Showing Call Stack Data Pertaining to Fault
 You view the details in the results window. Fault Simulator displays results on simulated faults, specified faults, and system information. In this example, Fault Simulator provides call stack and properties views for the registry fault in question.

DevPartner Fault Simulator SE

DevPartner Fault Simulator SE contains a subset of DevPartner Fault Simulator features. The following table differentiates functionality in the full version versus SE:

Table 2-3. Fault Simulator Full Versus SE

Functionality	Full Version	SE Version
Availability of Fault Simulator standalone application and command line interface	Yes	No
Ability to add or modify source-based .NET Framework faults	Yes	Yes

Table 2-3. Fault Simulator Full Versus SE (Continued)

Functionality	Full Version	SE Version
Ability to add or modify environmental faults	Yes	No
Saving of fault settings associated with .NET Framework or environmental faults created in a solution	Yes	No
Use of arguments, conditions, and/or parameters	Yes	No
Ability to save faults	Yes	No
Ability to load faults	Yes	No

Tip: See Table 2-1, *Supported Features in DevPartner Fault Simulator*, on page 9 for information about the full functionality.

DevPartner Fault Simulator SE must accompany DevPartner Studio 8.0 on the same system to perform coverage analysis with fault simulation. See “[Fault Simulation with Coverage Analysis](#)” on page 12 for more information.

Chapter 3

Improving Software Quality Through Fault Simulation



- ◆ Software Development Objectives
- ◆ Challenges to Software Quality
- ◆ Software Vulnerabilities
- ◆ Alternatives to Achieve Software Quality
- ◆ Three-Part Solution to Achieve Software Quality Using Fault Simulator
- ◆ Fault Simulator — Improving Software Quality

This chapter will contrast goals that software development teams share with the challenges they face. It will explore software vulnerabilities that affect application stability, along with alternatives that promote software quality. Finally, the chapter will present a three-part approach to improve software quality using the fault simulation capabilities in DevPartner Fault Simulator.

Software Development Objectives

Software development teams pursue similar objectives. For example, they strive to:

- ◆ Analyze and eliminate all software vulnerabilities
- ◆ Adapt to changing internal and external conditions
- ◆ Address interoperability, compatibility, and portability constraints
- ◆ Keep pace with changing technologies
- ◆ Identify and fix all possible software bugs prior to release to market

Today's applications should run reliably in different operating environments. However, in reality, conflicts could occur if applications are not tested thoroughly prior to deployment.

Internal and external factors help applications perform as expected. Some of these factors include:

- ◆ Data integrity
- ◆ Application integrity
- ◆ Data recovery

With data integrity, the application can gracefully handle unexpected incidences of invalid data. Data integrity protects against:

- ◆ Errors that result from bad data entered by a user
- ◆ Errors generated when bad data passes between computers or networks
- ◆ Errors caused by external forces, such as viruses or spyware
- ◆ Hardware failures, such as disk crashes

Application integrity means that an application will not stop running for unknown reasons, such as from changing conditions in the operating system. Application integrity ensures that the deployed application can successfully complete its tasks and return to its normal state. For example, if an application transaction only partially completes, this malfunction could place the application in a compromised state.

Data recovery incorporates mechanisms that salvage invalidated or lost data, such as those caused by disk crashes or virus attacks. Special utilities, either external or internally built into the application, can help restore the affected data.

Software developers can achieve these objectives using various testing methodologies, such as those highlighted in [“Conventional Testing”](#) on page 30. However, they can also address impediments to software quality directly using alternate approaches, such as those discussed in [“Non-Traditional Testing”](#) on page 30 and [“Approaches for Testing Environmental Conditions”](#) on page 31.

Challenges to Software Quality

Software development confronts many challenges to software quality, including:

- ◆ Product instability
- ◆ Explosion of New Technologies
- ◆ Complexities of the inner workings of APIs and system services

Product Instability

Product instability can create bottlenecks to future product development in several ways. For example, unstable products:

- ◆ Disrupt application performance, affecting customer perception
- ◆ Place additional, unintended burdens on the software development and quality assurance (QA) organizations
- ◆ Force software developers to fix unanticipated software bugs and quality assurance to retest them
- ◆ Prevent developers from researching and developing new technologies

Application defects become more costly to debug and repair after a product is released. Meticulous quality assurance testing conducted throughout the production phase, can help minimize the risk of out-of-bounds costs resulting from defects revealed after market release. The Gartner Group recently reported that “the average cost of unplanned downtime for a mission-critical application is \$100,000 per hour.”¹

¹Gartner Application Development Summit Presentation: “Software Quality in a Global Environment: Delivering Business Value.” by Theresa Lanowitz. September 2004.

Explosion of New Technologies

With the rapid advancement of the Web, new knowledge has proliferated at breakneck speed. These quantum leaps in technology demand never-ending mastery of new techniques. In addition, software development teams might not have ready access to the necessary subject matter consultants on the more subtle complexities of the applicable technology. Teams might also unwittingly underestimate the full extent of design requirements.

Complexities of the Inner Workings of APIs and System Services

Managing the complexities of the inner workings of APIs and system services poses another challenge for software development. For example, code calling into an API that checks for network connectivity might lack sufficient error handlers. If code is not built with proper error handling, an application might generate an unresponsive user interface or worse, an abrupt shutdown, placing that application and the operating environment at risk. A sufficient working knowledge about implementing effective error-handling code goes a long way to ensuring software reliability.

Software Vulnerabilities

Software problems can wreak havoc on today's applications and compromise stability. What developer does not dread the following symptoms?

- ◆ Blue screen crash
- ◆ System hang or freeze
- ◆ Lost data
- ◆ Failure of a critical process
- ◆ Network down

What software vulnerabilities might cause such symptoms to occur? These weaknesses stand out:

- ◆ Logic errors
Logic errors can generate invalid or unpredictable results, perhaps stemming from a misinterpretation of the intended workflow. While not fatal, logic errors can cause erratic behavior.
- ◆ Uninitialized data
Uninitialized data can cause intermittent problems in the application. If the program stores data in a location that was not properly initialized, the subsequent data can become corrupted. Such *dirty* data can lead to instability in the form of sporadically invalid results or erratic behavior.
- ◆ Invalid data
The underlying source code might not be able to process invalid external inputs (i.e., from a user or the operating environment). Failure to include provisions in the code to validate data can destabilize the program.

Layers of Application Vulnerability

The following illustration depicts three layers of application vulnerability:

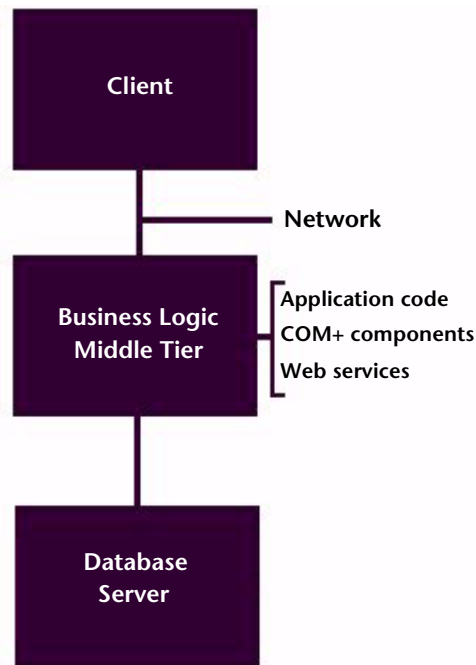


Figure 3-1. Layers of Application Vulnerability

Each layer (client, application logic, and database server) has the potential of impacting another. For example:

- ◆ Data passing from the client to the server might become corrupted, or the data might pass in a format or a size that the server cannot process properly.
- ◆ Client-side data might be susceptible to external security attack or unwanted manipulation.
- ◆ A network offline condition could prevent the client from integrating properly with the application logic.
- ◆ An application might fail to retrieve data from a missing directory on the database server.
- ◆ A missing or corrupted registry key might prevent the application from accessing a software program or the Windows operating environment.

Alternatives to Achieve Software Quality

Engineering teams (software developers and quality assurance alike) can work together to achieve software quality using various methodologies.

Conventional Testing

In conventional testing, engineering performs a variety of tests on predictable areas in the application to ensure expected outcomes. The following table summarizes common forms of testing:

Table 3-1. Common Forms of Testing

Type of Testing	Tester(s)	Description
Unit	Software developer	Isolates testing to specific code sections in the application
Integration	Software developer Quality assurance engineer	Verifies that different areas of the application work properly together
Functional	Quality assurance engineer	Executes certain user functions in the application and checks for the expected result
Regression	Quality assurance engineer	Runs tests to ensure that the latest fixes in the software safely correct the problem

Testing results help reviewers assess the risk and determine the bugs to fix, given that not all bugs can realistically be fixed. Engineering should consider fixing bugs that could negatively affect customer perception or hinder software reliability, even if rare. Engineering might elect to delay fixes for bugs that are bothersome, but not damaging to the application or customer perception.

Non-Traditional Testing

Non-traditional testing approaches include (but are not limited to):

- ◆ Stress testing
- ◆ Load testing

Stress Testing

Stress testing involves running an application and then monitoring program behavior under adverse, atypical conditions. To ensure that an application can reliably handle adverse scenarios, quality assurance should test the application under *adverse* conditions. Stress testing

executes the application under more demanding conditions, and in some cases under conditions that the application might never encounter. Stress testing evaluates how an application behaves as conditions become more acute. Stress testing tries to force the application to fail in order to observe how (or if) the application can recover.

Load Testing

Load testing assesses an application's tolerance to increased load (i.e., data input, transaction processing). Load testing analyzes the scalability and load balancing capabilities of an application. It attempts to cause failures that help an application's ability to perform reliably.

Contrasting Load and Stress Testing

Load testing and stress testing are not synonymous. Stress testing deliberately stresses an application, subjecting it to unreasonable conditions at extreme levels. Load testing measures software reliability by subjecting an application to a clearly defined, statistical load, such as to the maximum level that the application is specified to handle. Unlike stress testing, load testing does not push the application to its extreme levels.

Approaches for Testing Environmental Conditions

How do you safely force an API to fail in order to evaluate the application's response? Can you harmlessly alter the operating environment without destabilizing the application or the environment? Let's consider manual testing versus virtual testing, and see how they relate to robust error handling.

Manual Testing

Manual testing can be impractical and most likely harmful. For example, physically unplugging a network cable to test a network offline condition will disrupt the application and the operating environment. In this scenario, you will not be able to control the test variables. Also, if you wanted to repeat the testing, you probably will not be able to replicate the test exactly. On the other hand, if you left the API untested, you would have no way of knowing if it would fail in the released product.

Virtual Testing

With virtual testing, you subject the application to artificial errors without causing harm to the application or to the operating environment. Unlike manual testing, you do not *physically* change environmental parameters. However, you can virtually test how an application performs under various environmental conditions via fault

Tip: See "What Is Fault Simulation" on page 5.

simulation. With fault simulation, you run routines that artificially generate error conditions so that you can observe exactly how the application responds while it executes.

Tip: See “Well-Constructed Error Handlers Promote Product Reliability” on page 45.

Incorporation of Robust Error Handling

Software developers should incorporate robust error-handling code into the application code in order to ensure software quality. Error handlers allow the application to function responsibly by:

- ◆ Recovering gracefully from an unexpected condition
- ◆ Terminating without losing essential data
- ◆ Providing useful feedback to the user

Error handlers monitor and respond appropriately to external factors, such as interaction with system services, third-party applications, the operating system, and the installed environment. Proper error handling means managing adverse, unforeseen conditions gracefully without disrupting the application or the operating environment. Developers should build error handlers into the source code to ensure that the application can respond appropriately to a catastrophic event. Writing robust error handling code, followed by effective *testing* of the error handling, should become an early and integral part of the software development process, not an afterthought.

Three-Part Solution to Achieve Software Quality Using Fault Simulator

This section presents a three-part approach that illustrates how to improve software quality using the fault simulation capabilities with Fault Simulator. This approach incorporates white box, black box, and automated testing.

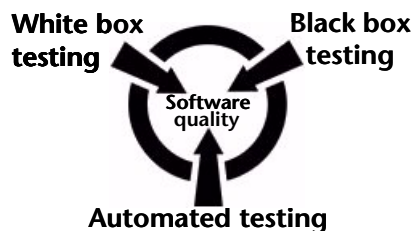


Figure 3-2. Three-Part Solution to Software Quality

White Box Testing

Software developers perform white box testing. In white box testing, the developer tests the source code throughout the development cycle.

White box testing focuses on the inner workings of the source code. It verifies program logic from the inside out. It checks how the application handles invalid or non-existent data and bad pointers. You can complement your white box testing using Fault Simulator in Visual Studio.

Note: See “White Box Testing Using Fault Simulator in Visual Studio” on page 33.

Black Box Testing

Quality assistance organizations perform black box testing. Black box testing seeks out bugs, such as user interface anomalies or problems with data handling (i.e., input or output, database problems). It also helps QA verify customer requirements outlined in the company’s engineering technical specification.

In black box testing, quality assurance engineers run tests that focus on specific environmental conditions to ensure application stability. Black box testing can be a prerequisite for final acceptance testing. You can complement your black box testing using the Fault Simulator standalone application.

Note: See “Black Box Testing Using the Fault Simulator Standalone” on page 38.

Automated Testing

In automated testing, QA engineers and software developers create scripts that test how the application handles environmental fault conditions. QA and development can run tests geared to address their unique concerns. You can perform automated testing using Fault Simulator from the command line.

Note: See “Automated Testing Using Fault Simulator from the Command Line” on page 43.

The sample code snippets appearing in the following sections, although simple, will demonstrate the concepts presented.

White Box Testing Using Fault Simulator in Visual Studio

Tip: See “Structured Exception Handling” on page 47.

As a developer, you have written an application that uses the .NET Framework. You have incorporated .NET Framework Structured Exception Handling methodology into your application code. You have also written additional error-handling code to catch and handle error conditions that might be generated within the application itself or the surrounding environment.

After you conduct unit testing on different areas in your code to get immediate feedback that your application code integrates properly, you perform fault simulation testing. The following procedure outlines the tasks you perform using Fault Simulator:

Tip: See “Using Fault Simulator in Visual Studio” on page 14 for a quick start.

- 1 You want to uncover any exception handling problems that might exist in your application code. From Visual Studio, you open the applicable solution.
- 2 In the source window, Fault Simulator identifies a try-catch code block that includes a supported .NET Framework Class Library method.

```
{
    try
    {
        Console.WriteLine("Your Method Code");
        FileStream fs = File.Open(@"C:\test.txt", FileMode.Open);
        Console.WriteLine("Attempted to open file");

        Class1 c = new Class1();
        c.Foo();
    }
    catch (ArgumentOutOfRangeException exc)
    {
    }
    catch (ArgumentException exc)
    {
    }
}

private void Foo()
{
    Console.WriteLine("Method Foo");
}
```

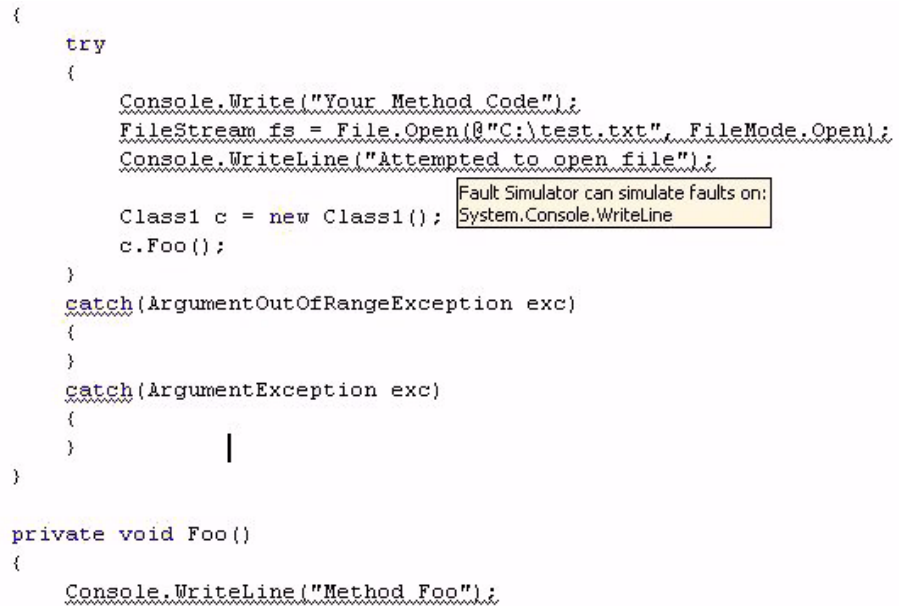


Figure 3-3. Highlighted Source Containing Supported Method with Tooltip

- 3 You add a new .NET Framework fault to test the exception handling at that location.

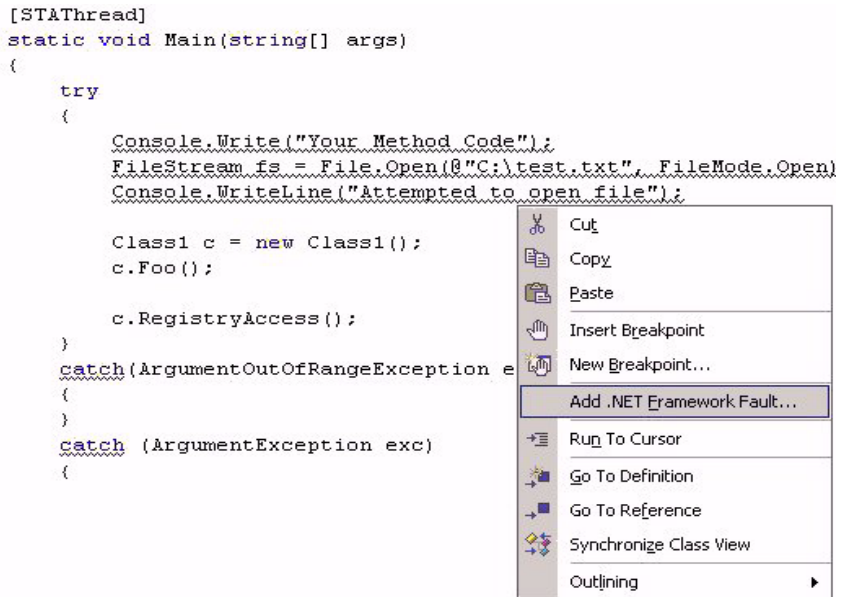


Figure 3-4. Context Menu to Add New .NET Framework Fault

- From the **Add .NET Framework Fault** dialog box, you choose to throw **System.IO.IOException** and then start the simulation.

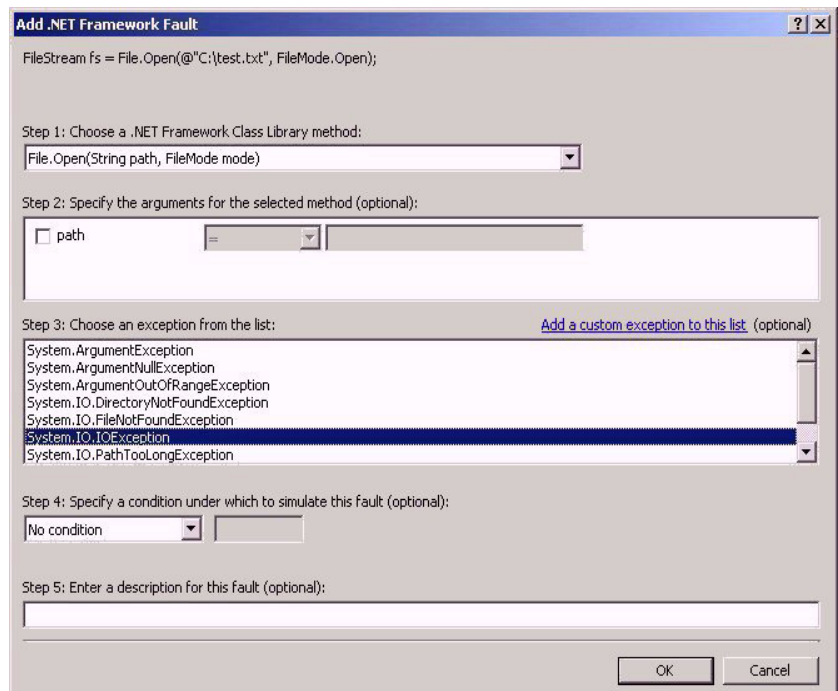


Figure 3-5. Selecting Exception for Supported .NET FCL Method

- 5 During the session, Fault Simulator throws the designated exception, automatically ending the session and stopping the executable.
- 6 You review the results. You see that two catch blocks in your code did not handle the exception.

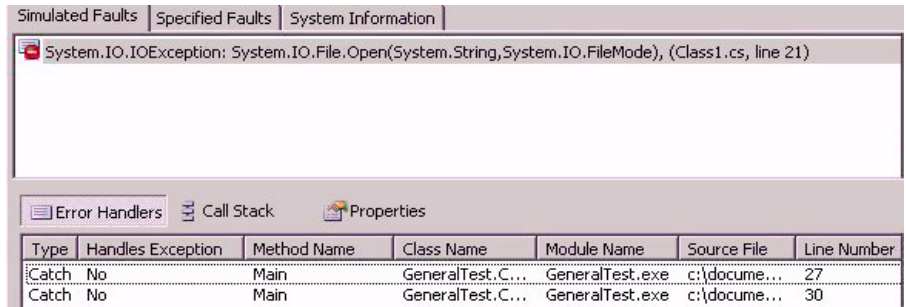


Figure 3-6. Error Handler View Shows Exception Was Not Properly Handled

- 7 You look at the **Call Stack** pane in the results window that shows the entire stack where the exception was or was not thrown. You trace up the method calls to troubleshoot the code.

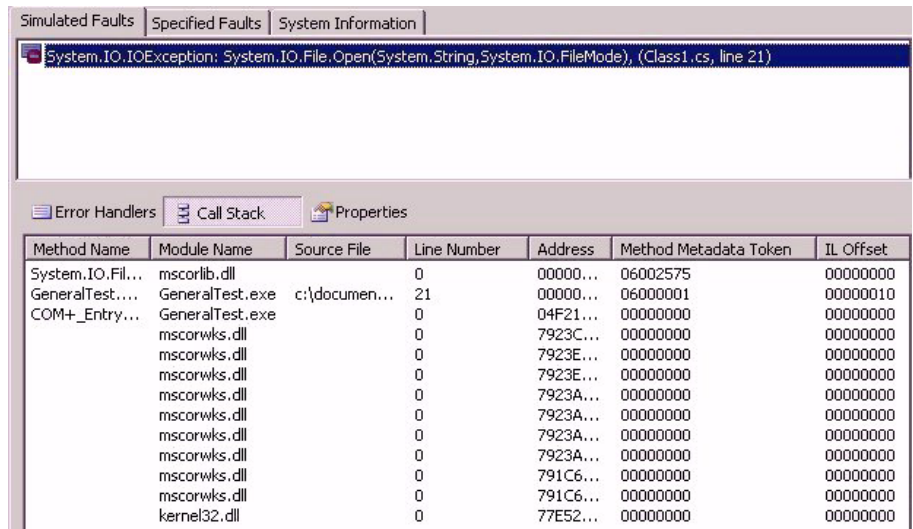


Figure 3-7. Call Stack View Lets You Trace Called Methods

- 8 You double-click on one of the unsuccessful catches in your code (shown in Figure 3-6 on page 36) to view the specific source. You conclude that you need to add a catch to handle the exception in that code block.

```

[STAThread]
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("Your Method Code");
        FileStream fs = File.Open(@"C:\test.txt", FileMode.Open);
        Console.WriteLine("Attempted to open file");

        Class1 c = new Class1();
        c.Foo();
    }
    catch(ArgumentOutOfRangeException exc)
    {
    }
    catch(ArgumentException exc)
    {
    }
    catch(IOException exc)
    {
    }
}

private void Foo()
{
    Console.WriteLine("Method Foo");
}

```

Figure 3-8. Changes to Code Deduced from Fault Simulation Results

- 9 You run another fault simulation with the same configuration. You want to verify that you corrected the problem revealed during the first session.
- 10 This time, following the session, you see that your code change worked. You noted the location in your source code where the exception has caught.

Shows location of handled exception

Type	Handles Exception	Method Name	Class Name	Module Name	Source File	Line Number
Catch	No	Main	GeneralTest.C...	GeneralTest.exe	c:\docume...	27
Catch	No	Main	GeneralTest.C...	GeneralTest.exe	c:\docume...	30
Catch	Yes	Main	GeneralTest.C...	GeneralTest.exe	c:\docume...	33

Figure 3-9. Fault Simulation Results Shows Location of Handled Exception

- 11 You save the original fault set and the subsequent, successful results to disk.

- 12 You conduct similar tests on other areas in your code where you suspect similar exception handling vulnerabilities. You use Fault Simulator to pinpoint exactly where in the code you have insufficient or non-existent error handling code.

Black Box Testing Using the Fault Simulator Standalone

As a QA engineer, you want to test that the developed application functions reliably under a variety of negative conditions. You know that *physically* causing environmental faults (such as unplugging a network connection or altering a registry key) could detrimentally disrupt the application under test. Moreover, you do not want such physically-imposed environmental fault conditions to adversely disturb other applications that might need to run on the same system.

Therefore, after you complete your standard QA test procedures, you test the application's tolerance to *simulated* fault conditions. The following procedure outlines the tasks you perform using the Fault Simulator standalone application:

Testing for Unintended Regression

- 1 From the Fault Simulator standalone application, you load the .NET-based fault set supplied by software development and start a fault simulation on the application.
- 2 Following the session, you look at the results and compare them to the results that the developer provided to you from his tests in Visual Studio. You confirm that regression has not occurred since the developer originally tested that aspect of the application code.

Tip: See "Using the Fault Simulator Standalone Application" on page 19 for a quick start.

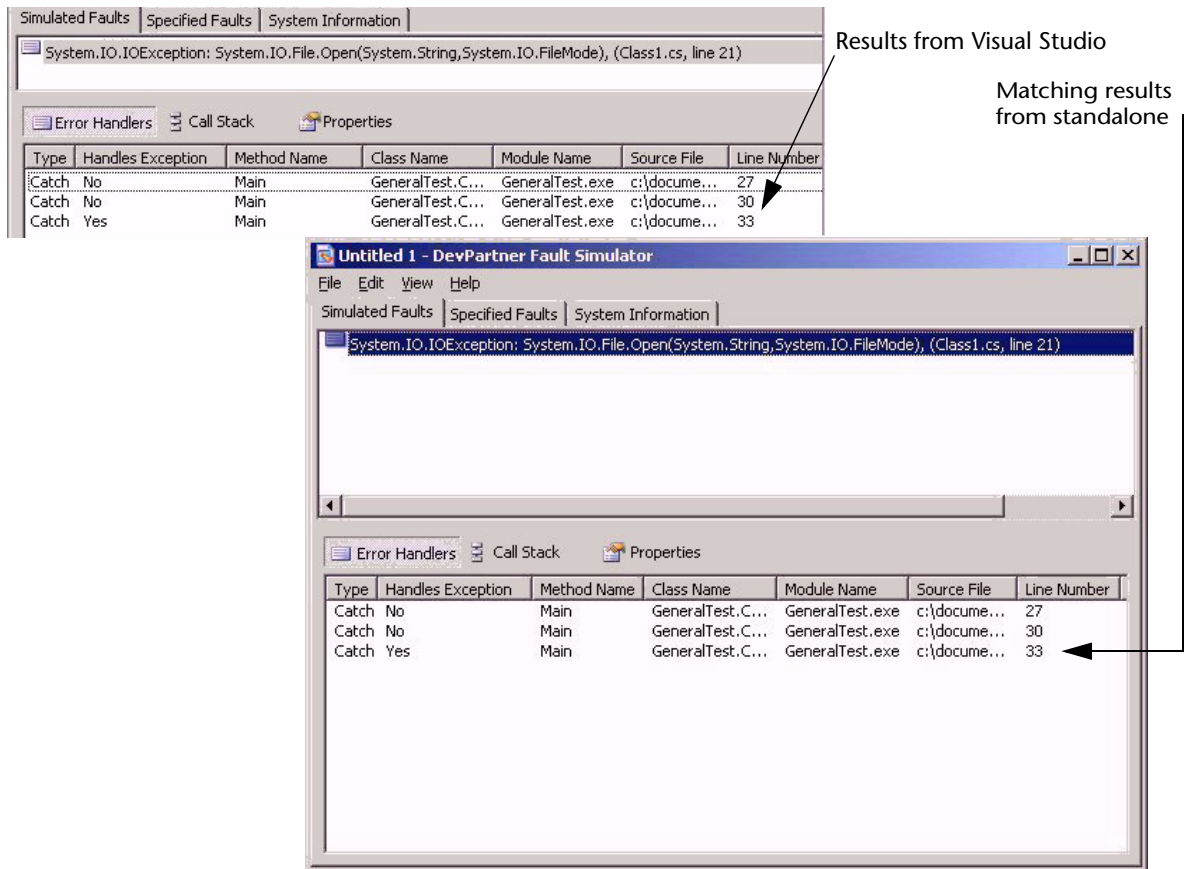


Figure 3-10. Results in Standalone Confirm No Regression Occurred

Testing Various Environmental Conditions

- 1 You test how the application handles a missing file condition. Because you hesitate to physically delete the target file, you create and configure a *Disk I/O* environmental fault descriptor, **Missing file**, and then run the simulation.

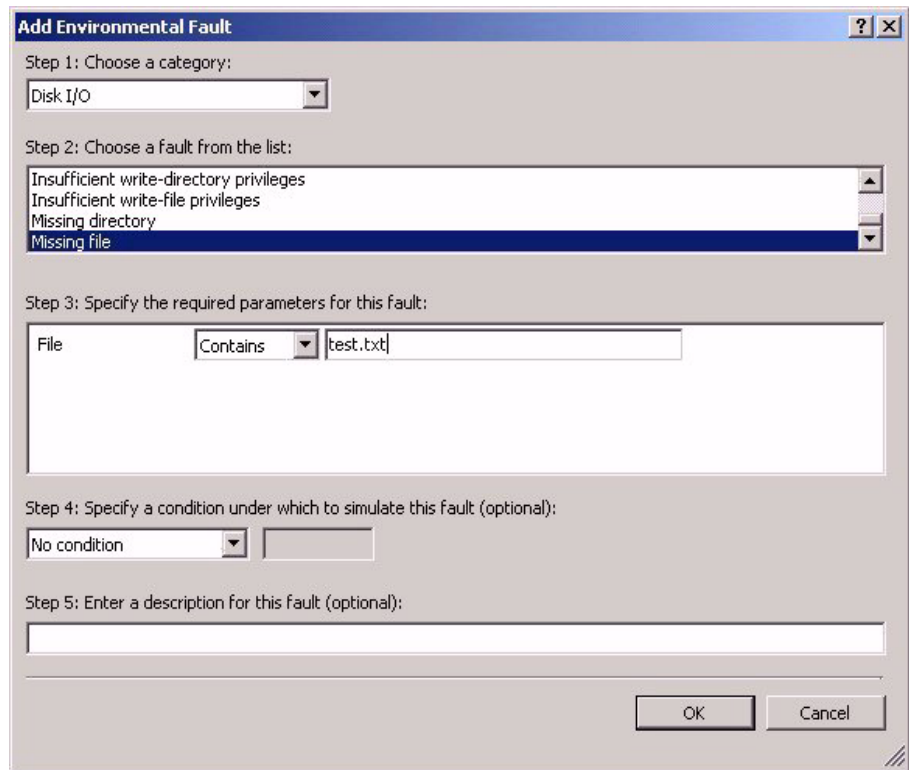


Figure 3-11. Configuring **Missing File** Environmental Fault

- 2 Following the conclusion of the session, you see that Fault Simulator simulated the designated fault condition. The **Properties** view confirms your original fault settings, while the **Call Stack** view gives method call stack details. You also conclude that the artificially missing file did not upset the application's normal operation.

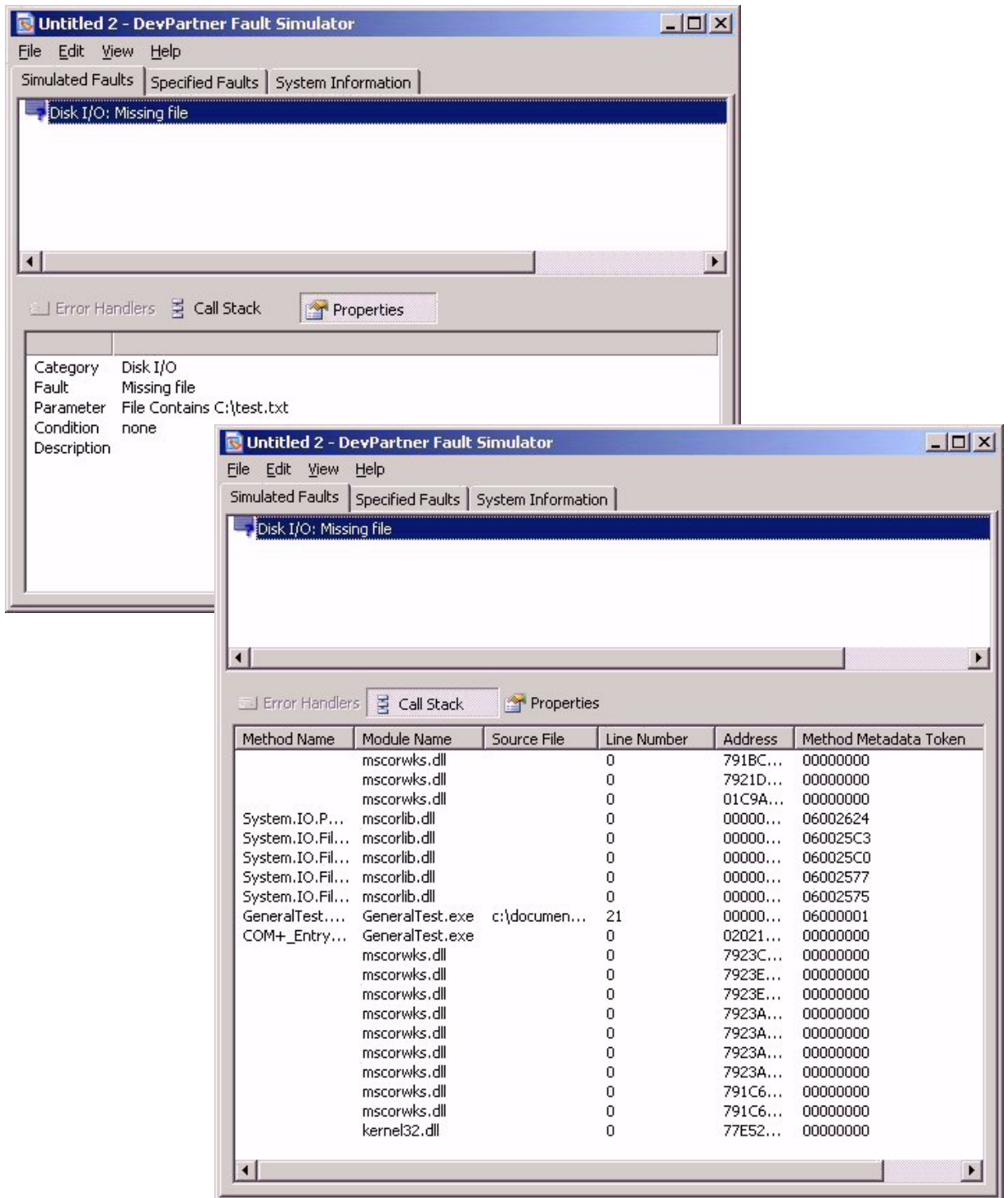


Figure 3-12. Missing File Properties and Call Stack Details

- 3 Next, you test how the application handles a missing registry value. Again, because you do not want to physically change settings in the Registry Editor, you create and configure a *Registry* environmental fault descriptor, **Missing value**. You run the fault simulation.
- 4 Following the session, you observe that the application did not crash or behave unfavorably when it encountered the missing registry value fault condition.
- 5 You look closely at the **Call Stack** view in the results window. Fault Simulator identifies where in the underlying code the application handles the registry condition.

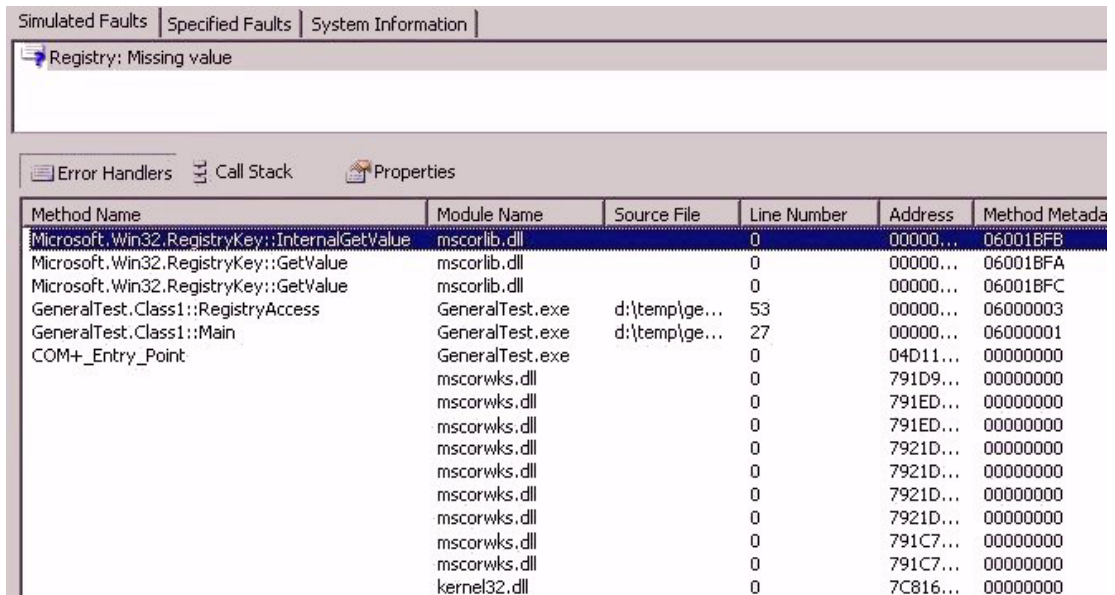


Figure 3-13. Example Showing Missing Registry Value Results

- 6 You save the fault set and results files and direct them back to the software developer so that he can run the test from the inside out to view the actual source referenced in those results.
- 7 You conduct additional tests on other environmental conditions that might also impact the application's performance, such as:
 - a You run a fault simulation on a **Network offline** fault condition to test how the application reacts without physically disconnecting the network cable.
 - b You run a fault simulation on a **Virtual memory allocation limit** fault condition to test if the application can sustain itself when it consumes the allocated virtual memory.

Automated Testing Using Fault Simulator from the Command Line

You plan to run automated testing on applications under development. To that end, you construct a script (using the following template as a model).

Note: Fault Simulator can execute fault simulations on Windows, console, and Web applications, as well as COM+ components. Fault Simulator also allows the testing of several instances of the same application, as long as each instance is unique.

```
<job id="DevPartnerFaultSimulatorAutomation">
<script language="JScript">
wsh = new ActiveXObject("WScript.Shell");
wsh.Run("Dpfs /f:<path>FaultSet.dpfsfault /a:<path>ConsoleApplication.exe
/r:<path>ResultFileName.dpfs
/l:<path>ConsoleApplication.exe", 1);
//After test is complete, exit target application so DPFS.exe can exit and generate results
WScript.Sleep("Appropriate Duration");
wsh.Run("Dpfs /f:<path>FaultSet.dpfsfault /a:<path>WindowsApplication.exe
/r:<path>Results.dpfs
/l:<path>WindowsApplication.exe", 1);
//After test is complete, exit target application so that DPFS.exe can exit and generate
results
WScript.Sleep("Appropriate Duration");
//Stop target COM+ server
wsh.Run("Dpfs /f:FaultSet.dpfsfault /com+:COM+Component /r:<path>Results.dpfs
/l:<path>COM+ClientApplication.exe", 1);
//Restart target COM+ server. After test is complete, stop target COM+ server so DPFS.exe can
exit and generate results
WScript.Sleep("Appropriate Duration");
wsh.Run("Dpfs /f:<path>FaultSet.dpfsfault /u:http://localhost/WebSite1/Default.aspx
/r:<path>Results.dpfs /l:http://localhost/WebSite1/Default.aspx", 1);
//After test is complete, run IISReset so DPFS.exe can exit and generate results
WScript.Quit()
</script>
</job>
```

Figure 3-14. Template to Automate Fault Simulator From Command Line

The next day, you launch the Fault Simulator standalone application and open results generated from the command line.

You repeat the nightly fault simulation tests to check for any regression with each round of changes from software development.

Fault Simulator — Improving Software Quality

Software developers strive to build and deploy applications that function reliably regardless of changing conditions. Software will never be perfect. However, software developers and quality assurance alike must make every effort to uncover and eliminate all possible software vulnerabilities.

Fault Simulator provides that solution. Throughout the development life cycle, Fault Simulator helps developers and quality assurance engineers ensure that critical applications can handle all sorts of adverse, atypical environmental conditions. Fault Simulator helps developers build applications that respond to unexpected internal or external failures without causing catastrophic outcomes. Where software vulnerabilities exist, Fault Simulator offers the tools to zero in on the problem areas and the means to correct them.

Chapter 4

Using Fault Simulator to Evaluate Error Handlers



- ◆ Well-Constructed Error Handlers Promote Product Reliability
- ◆ Configuring Fault Simulator to Capture Error Handler Data
- ◆ Evaluating Error Handling Results
- ◆ Fault Simulator Integral to Best Practices

This chapter will review different error handler approaches, including a detailed overview of Structured Exception Handling. The chapter will then show how Fault Simulator helps you evaluate the robustness of the error handlers in your code.

Well-Constructed Error Handlers Promote Product Reliability

Errors take time to detect and fix, but if left undetected, can disrupt the application and the operating environment. Errors fall into these broad categories:

- ◆ Logic — Problems with the program logic, causing unintended results
- ◆ Syntactic — Errors that the compiler uncovers in the code syntax
- ◆ Runtime — Bugs occurring in a running program

Well-constructed error handlers can deal with a multitude of error conditions, promoting a more reliable product. Error handlers are “sections of program code specifically provided to take remedial action in the event that other sections of code detect or cause error conditions in a running program.”

Error handlers can be grouped as follows:

- ◆ Error handling for function calls
 - ◇ Returns a success or failure status value
 - ◇ Checks for a successful status value
 - ◇ Makes provisions for a failed return

- ◆ C++ Exception Handling
 - ◇ Uses try-throw-catch statements
 - ◇ Throws C++ exceptions
 - ◇ Implemented by the Microsoft C++ compiler
- ◆ Structured Exception Handling
 - ◇ Uses try-catch-finally procedures
 - ◇ Throws structured exceptions

Error Handling for Function Calls

Historically, software development incorporated basic error handlers to manage Visual Studio 6 Win32 and COM APIs. These error handlers included error codes and generic `FALSE` returns from functions that encountered errors. For example, `GetLastError` might be used to troubleshoot the root cause. If the code was built to handle the error, the Win32 function still had to hunt for and handle any problems it encountered inline at every statement or method call that could fail.

C++ Exception Handling

Based on the ANSI C++ standard, the C++ exception handling model was the precursor to the Structured Exception Handling methodology. C++ exception handling applies the concept of throwing an exception and subsequently designating an exception handler to catch the thrown exception using `try`, `throw`, and `catch` statements.

What is an Exception

An exception is an unanticipated event occurring while a program is running that interrupts the normal operation of that program. When an error happens within the scope of a method, the affected method creates an exception object and then passes it to the runtime environment. The exception object consists of relevant data, such as the:

- ◆ Type of error
- ◆ Program state at the time the error occurred

After the method throws an exception, the runtime environment examines the call stack where the method resides to find a code block that can properly handle the exception. To qualify, the code block must be constructed to specifically handle the exception object. If the search is successful, the appropriate code block catches the exception. If the search is unsuccessful, the runtime environment terminates.

Structured Exception Handling

Following the release of the .NET Framework, Microsoft endorsed Structured Exception Handling as a necessary component of any well-written program. In this methodology, the exception handling code is organized into structured blocks using try-catch-finally procedures. Each procedure contains code that catches the types of exceptions that a procedure might generate. If a procedure cannot handle a particular exception, the procedure will pass the exception up the call stack to the calling method.

Best Practices for Structured Exception Handling

This section highlights key points of the Structured Exception Handling methodology, as advocated in the book *Applied Microsoft .NET Framework Programming*¹.

¹Richter, Jeffrey. *Applied Microsoft .NET Framework Programming*. Buffalo: Microsoft Press, 2002.

The techniques outlined in Richter's book represent best practices. Four fundamental guidelines for writing exception handlers emerge:

- ◆ Register for the `AppDomain.CurrentDomain.UnhandledException` and the `System.Windows.Forms.Application` type's static `ThreadException` events so that your managed `System.Windows.Forms` application will trap all unhandled exceptions.
- ◆ Whenever possible, avoid having your application catch `System.Exception`. However, if unavoidable, log information about the local variables that might assist in debugging and throw the exception to the calling method again.
- ◆ Only catch the exceptions where you can recover.
- ◆ Use at least one finally block to clean up resources.

Using Try-Catch-Finally Blocks

Try-catch-finally code construction forms the basis for the Structured Exception Handling best practices. You match a try block with one or more catch blocks, and at least one finally block, to handle the exception thrown by a method.

Try Block

The try block represents entry into the exception handling code. In the try block, you locate code to attempt a graceful recovery from an exception or else prepare for cleanup in the finally block. The try block

throws its own exception, that then causes each catch block to be evaluated. If the code in the try block does not throw an exception, the catch blocks are not evaluated, and the finally block executes.

Catch Block

The catch block contains code that handles a specific exception or group of exceptions. The more catch blocks you include in your exception handling code, the more precisely the code can respond when exceptions are thrown. The code only evaluates the catch blocks if the try block associated with them throws an exception.

Each catch block consists of the catch keyword, followed by:

- ◆ A parenthetical exception filter
- ◆ Code intended to handle or recover from the exception
- ◆ Code to throw the exception again

The exception filter lets you customize your error handling responses based on the specific exception thrown. For example, you might filter one catch block to handle:

```
System.ArgumentNullException
```

and the next to handle:

```
System.ArgumentException
```

Finally Block

The finally block completes cleanup operations in the exception handling code. The finally block will execute at the final stage of the try-catch-finally block regardless if the catch block actually handled the exception.

Try-Catch-Finally Execution

Executing a try-catch-finally mechanism follows this progression:

- ◆ The code evaluates the try block and throws an exception.
- ◆ The catch blocks are evaluated top to bottom.
- ◆ The finally block executes.
- ◆ The code executes if a catch block handled the exception.

Once all the code in the handling catch block executes, the code might take one of the following directions:

- ◆ Throw the same exception again to notify functions and methods higher up the call stack of the error condition
- ◆ Throw a different exception to provide more detail about the error condition to functions and methods higher up the call stack
- ◆ Stop throwing any more exceptions, letting the thread fall through the catch block once it has been handled

This option will not notify functions and methods higher up the call stack of an error condition.

You should integrate the try-catch-finally methodology at every stage of development to avoid unintended problems in the software product. You

should handle the error condition as close as possible to where the exception will be thrown. If an exception handler does not catch an error close to its source, the exception will fall out of the method. Should the calling function or method not have the proper exception handlers in place, the exception could repeatedly fall out unhandled until a generic exception handler catches the exception or the application terminates. This approach will degrade the application and lead to program instability.

Configuring Fault Simulator to Capture Error Handler Data

Fault Simulator helps you uncover weaknesses in your source code by letting you configure and execute an assortment of artificially generated error conditions. Fault Simulator directs you to specific problem areas in your code by simulating faults designed to test that code. Fault Simulator helps you identify the exceptions that your managed code should be catching. It also helps you prevent unhandled exceptions from falling through methods.

Configuring a .NET Framework Fault in Managed Code

You can configure settings on the **Add .NET Framework Fault** dialog box (shown next) to artificially throw an exception associated with a .NET Framework Class Library method in your managed code. Here, you can designate the exception to throw, plus other pertinent method signature properties.

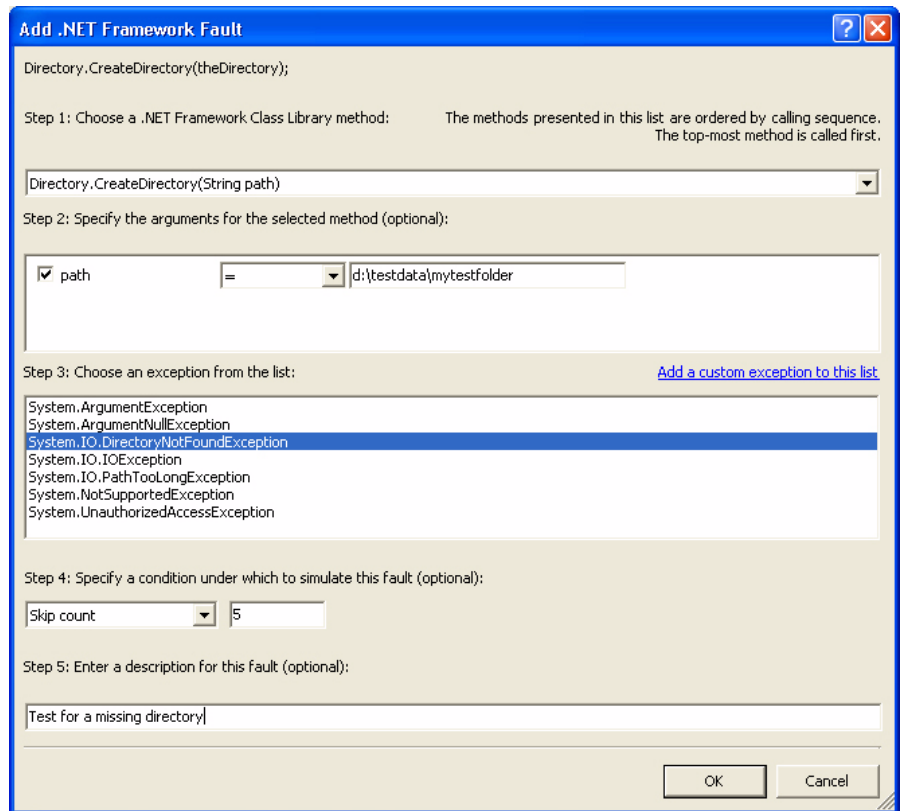


Figure 4-1. Add .NET Framework Fault Dialog Box

Note: See “A Fault Simulator Walk-Through” on page 15 for more information on using the **Add .NET Framework Fault** dialog box.

Fault Simulator also provides visible clues in the source view (shown next) where you can add and configure a .NET Framework fault for an upcoming fault simulation session.

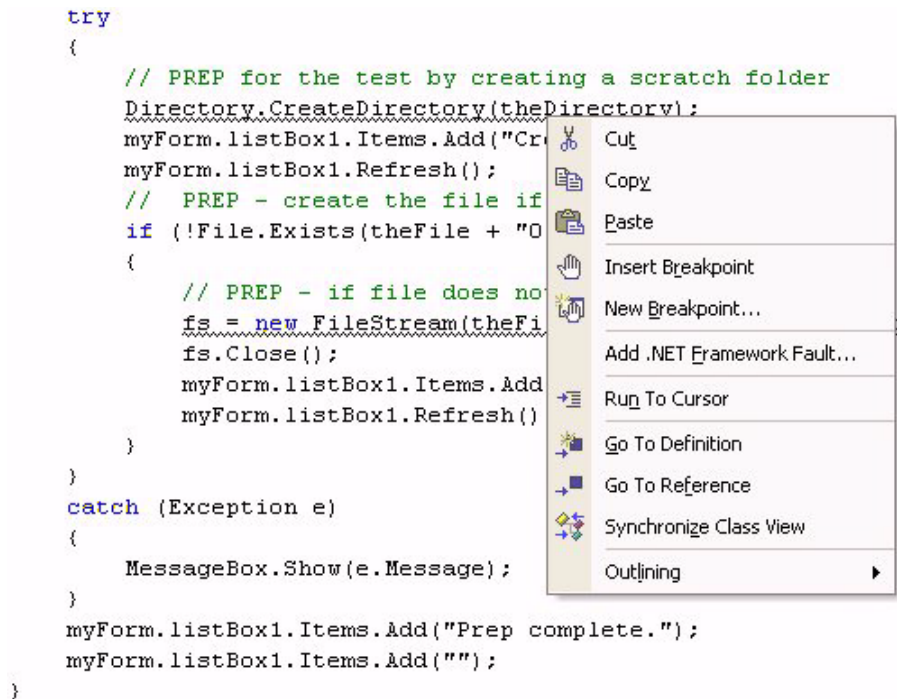


Figure 4-2. Source Location Where You Can Add Source-Based .NET Framework Fault

Configuring an Environmental Fault

You can configure settings on the **Add Environmental Fault** dialog box to simulate an error condition in the running program. Here, you can designate the type of environmental fault to simulate, plus other pertinent environmental properties.

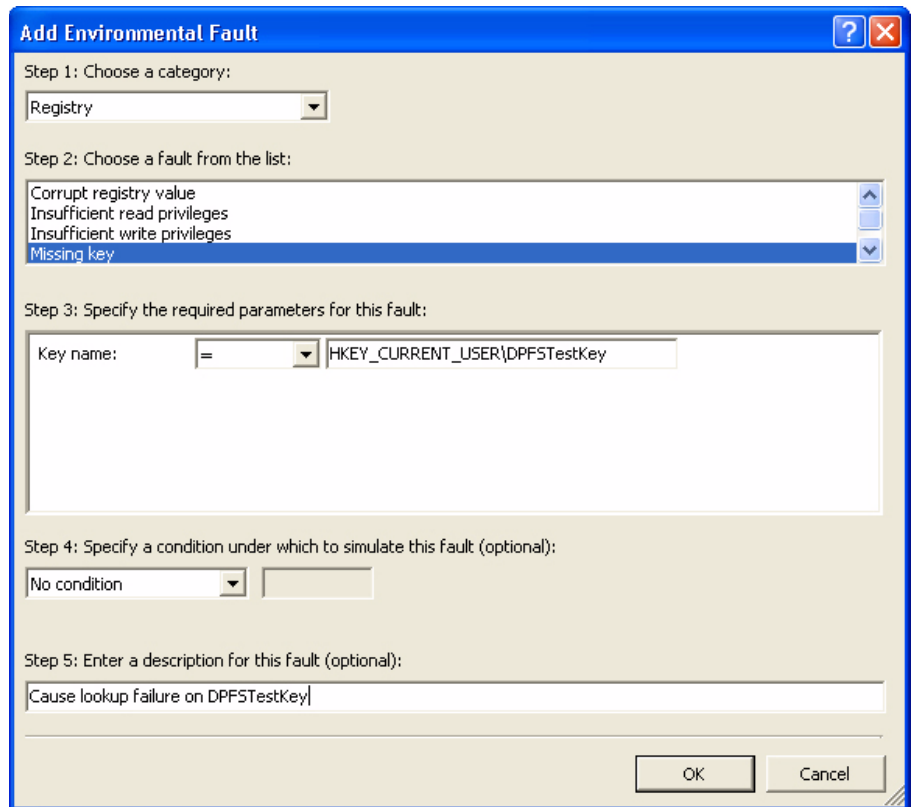


Figure 4-3. Add Environmental Fault Dialog Box

Note: See “A Fault Simulator Walk-Through” on page 20 for more information on using the **Add Environmental Fault** dialog box.

Fault Simulation Results Views

During an active fault simulation session, Fault Simulator displays current simulation activity. Following the completion of a fault simulation, Fault Simulator generates results summarizing the captured data.

Specified Faults Pane

The **Specified Faults** provides a summary of fault count information including:

- ◆ Count of every fault instance that occurred during the simulation
- ◆ Number of times the fault was evaluated (reflecting every attempt)

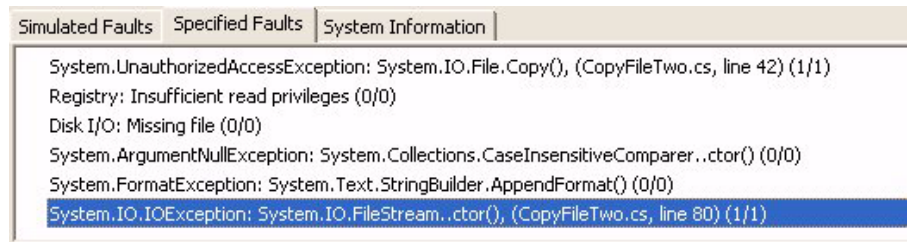


Figure 4-4. Specified Faults Pane

Fault Simulator shows additional details that you originally configured for that fault, such as:

- ◆ Argument(s) configured for .NET Framework faults
- ◆ Parameter(s) configured for environmental faults
- ◆ Category and fault name, for environmental faults only
- ◆ Optional conditions or description configured for each fault descriptor

Simulated Faults Pane

The **Simulated Faults** pane contains three views in the lower portion of the window:

- ◆ Error Handlers
- ◆ Call Stack
- ◆ Properties

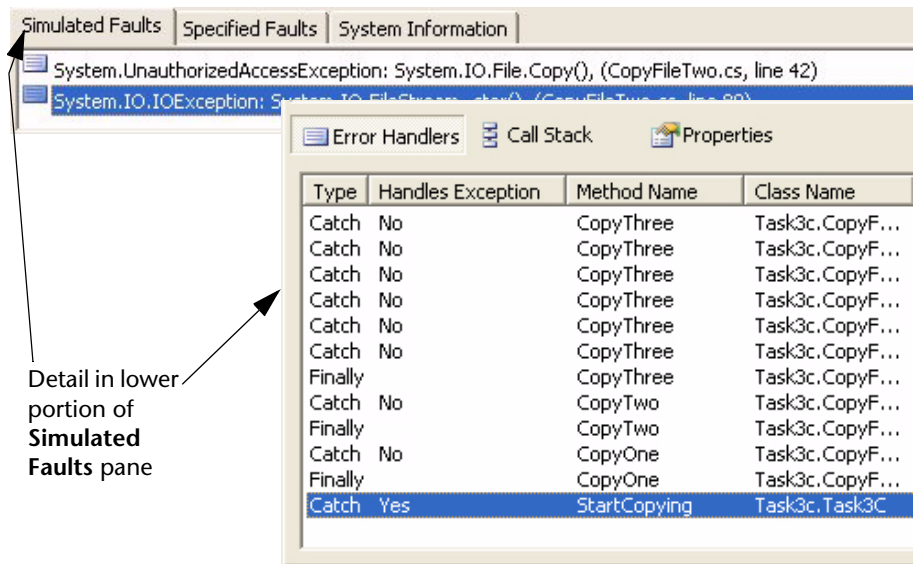


Figure 4-5. Simulated Faults Pane

Error Handlers View

Use the **Error Handlers** view to:

- ◆ Review evaluated and executed catch blocks
- ◆ Determine if the intended catch block, or a generic catch block, ultimately handled the fault
- ◆ Analyze the execution path that the error handling code took to see:
 - ◇ How your target application responded to the fault
 - ◇ How the fault was caught or fell through your catch blocks
 - ◇ How the error handling was resolved in the routine

Note: The **Error Handlers** view shows results on .NET Framework faults simulated in managed code only.

Call Stack View

Use the **Call Stack** view to:

- ◆ Analyze where in your code the exception was thrown or the function failed
- ◆ Trace the steps through your code that led to the exception being thrown or not

Properties View

Use the **Properties** view for a summary of the original fault configuration. The configuration varies depending on the type of fault displayed. The following table summarizes how the properties apply to the .NET Framework fault or the environmental fault:

Table 4-1. Fault Properties

Property	.NET Framework Fault	Environmental Fault
.NET Framework Class Library method	Yes	No
Argument	Yes	No
Parameter	No	Yes
Condition	Yes (optional)	Yes (optional)

Evaluating Error Handling Results

The following sections present examples that instruct you how to use fault simulation results to improve your code:

- ◆ [Determining the Path The Code Took to Unwind from an Exception](#)
- ◆ [Identifying if Any Error Handlers Got Invoked](#)
- ◆ [Viewing the Source Statement That Handled the Fault](#)
- ◆ [Determining the Path The Code Took When a Function Failed](#)
- ◆ [Assessing Whether the Intended Fault Was Handled](#)
- ◆ [Confirming Where the Fault Was Handled](#)

Determining the Path The Code Took to Unwind from an Exception

Fault Simulator details call stack information showing where each exception was thrown and where each fault occurred. The call stack also tells you:

- ◆ The steps through your code that led up to each exception being thrown
- ◆ Each fault causing a function to fail

For managed code, the **Error Handlers** view provides error handler information detailing how your code handled, or failed to handle, the designated .NET Framework fault.

You can follow the logic to see where the code properly handled the fault with a catch block, or where it fell through because of an incorrect or misplaced catch block. The next example shows evidence of an unhandled exception.

Type	Handles Exception	Method Name	Class Name	Module Name	Source File	Line Number
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	43
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	48
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	53
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	58
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	63
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	68
Finally		CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	75

Figure 4-6. Error Handlers View — Shows an Unhandled Exception

Identifying if Any Error Handlers Got Invoked

The **Error Handlers** view shows the catch and finally blocks that were evaluated and executed as each fault was simulated.

Type	Handles Exception	Method Name	Class Name	Module Name	Source File	Line Number
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	31
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	36
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	41
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	46
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	51
Catch	No	CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	56
Finally		CopyThree	Task3c.CopyF...	FileCopier.exe	c:\usability...	63
Catch	No	CopyTwo	Task3c.CopyF...	FileCopier.exe	c:\usability...	27
Finally		CopyTwo	Task3c.CopyF...	FileCopier.exe	c:\usability...	34
Catch	No	CopyOne	Task3c.CopyF...	FileCopier.exe	c:\usability...	27
Finally		CopyOne	Task3c.CopyF...	FileCopier.exe	c:\usability...	0
Catch	Yes	StartCopying	Task3c.Task3C	FileCopier.exe	c:\usability...	208

Figure 4-7. Error Handlers View

In the previous example, Fault Simulator reveals that your code did not handle the error as close as possible to where it was thrown, nor did it have the proper catch blocks in the correct places. It shows that six catch blocks were tried inside the lowest level method before the error handling fell out. Catch blocks were tried in three different methods before the error was handled in a fourth method's catch block as follows:

- ◆ Method `CopyThree` tried six catch blocks before executing the finally block and falling out of that method.
- ◆ Method `CopyTwo` tried one catch block, the finally block, and then fell out of the method.
- ◆ Method `CopyOne` also tried one catch block, executed the finally block, and then fell out into the main method `StartCopying`.
- ◆ `StartCopying` then handled the error with its generic catch block.

The next example shows how Fault Simulator depicts the catch blocks in your program that caught the exception in the **Call Stack** view.

Method Name	Module Name	Source File	Line Number
System.IO.Fil...	mscorlib.dll		0
Task3c.CopyF...	FileCopier.exe	C:\Usability\FileTester\FileCopier\CopyFileThree.cs	28
Task3c.CopyF...	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	25

Figure 4-8. Call Stack View — Location of Thrown Exception

Viewing the Source Statement That Handled the Fault

When available, Fault Simulator can take you back to the source location where the error handling was evaluated. Right-click on a line item in the lower portion of the **Error Handlers** view (Figure 4-6 on page 55) and choose **View Source** to go to the specific code, as depicted in the next example. From the original source location, you can troubleshoot whether the exception was properly handled, and if not, why not.

```

// create file names
try
{
    thisIndex = CF3.CopyThree(myForm, Index, fileIn);
}
catch (System.IO.FileNotFoundException e)
{
    MessageBox.Show(myForm, e.ToString());
    myForm.listBox1.Items.Add(e.ToString());
}

```

You can view source pertaining to an item listed in the **Error Handlers** view.

Figure 4-9. Viewing Applicable Source Location

Fault Simulator will inform you if it cannot find the specified code block due to a change in the original source.

Determining the Path The Code Took When a Function Failed

Fault Simulator can simulate environmental failures caused by one or more Win32 APIs. Unlike managed code, where best practices advocate the use of the Structured Exception Handling methodology, native code typically relies on error handlers that use return code values and inline error handling to detect errors.

Use the **Call Stack** view to trace the path your native code took when a function failed, as shown next.

Method Name	Module Name	Source File	Line Number	Address	Method Metadata Token	IL Offset
LookupRegValue	RegSample.exe	c:\projects\regsample\regsample.cpp	9	00411B6D	00000000	00000000
DisplayRegDefault	RegSample.exe	c:\projects\regsample\regsample.cpp	20	00411C59	00000000	00000000
_main	RegSample.exe	c:\projects\regsample\regsample.cpp	37	00411D46	00000000	00000000
_mainCRTStartup	RegSample.exe	f:\vs70builds\3077\vc\crt\src\src\crt0.c	259	0041213E	00000000	00000000
	kernel32.dll		0	77E814C4	00000000	00000000

Evidence of failed function

Figure 4-10. Call Stack View — Function Failure Location

The top-most entry in the **Call Stack** view will always show the location where Fault Simulator caused the function to fail.

Assessing Whether the Intended Fault Was Handled

You can compare the faults that you initially configured (on the **Specified Faults** pane) with the faults that were actually simulated (on the **Simulated Faults** pane).

Note: See “[Specified Faults Pane](#)” on page 52 and “[Simulated Faults Pane](#)” on page 53 for a description and example of each view.

Determine if you specified any faults that were never simulated and why those faults were not simulated. The following table provides some tips to consider:

Table 4-2. Comparing Specified Faults Versus Simulated Faults Results

Considerations	Possible Conclusions or Actions
Consider whether the code executed at the expected location where you expected it to occur.	If the code associated with the fault never executed, the fault would not occur.
Consider whether the managed code made a call to the relevant method.	<p>If you simulated a fault on a .NET Framework Class Library method, you could:</p> <ul style="list-style-type: none"> • Set a breakpoint on the targeted method call. • Run in the Visual Studio debug mode. • Check if the line of code executes. <p>If the statement was executed but Fault Simulator did not generate the fault, examine the arguments specified in the .NET Framework fault descriptor to see if those values evaluated true when the method was executed.</p>

Table 4-2. Comparing Specified Faults Versus Simulated Faults Results (Continued)

Considerations	Possible Conclusions or Actions
Consider whether the native code made a call to the relevant method.	If you simulated an environmental fault, ensure that the application exercised the code appropriately to cause the type of call associated with the fault. If Fault Simulator did not simulate the fault, check that the parameter(s) evaluated true when the call was executed.
Consider whether the code made a call to the correct overloaded method.	Verify that the signature (the number and type of arguments on the call) specified in the fault descriptor matches the actual method call in your code. If the signature does not, Fault Simulator would not simulate the fault.

For each simulated fault, examine the **Error Handlers** view (Figure 4-7 on page 56) and observe the catch and finally blocks that were evaluated and invoked. For example, you can:

- ◆ Check whether the catch blocks were properly specified to efficiently handle the fault.
- ◆ Check whether the fault fell through all the specific catch blocks but ended up being handled by a generic catch block within the routine where it was thrown.
- ◆ Check whether the fault fell through a series of routines before eventually being caught.
- ◆ Check whether the fault got handled outside the code (i.e., by catch blocks in some other external routine).

Confirming Where the Fault Was Handled

Did the catch blocks handle the intended exception? Or did it fall through routines, only to be caught by a generic catch elsewhere? If you find that the faults are being handled generically, you need to evaluate how you initially set up your catch blocks to handle exception.

The next example shows that an exception was not handled by the method where it occurred. Rather, it fell through all the catch blocks in the `CopyThree` method to be handled by a catch in the calling method, `CopyTwo`.

Type	Handles Exception	Method Name	Class Name	Module Name	Source File	Line Number
Catch	No	CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	31
Catch	No	CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	36
Catch	No	CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	41
Catch	No	CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	46
Catch	No	CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	56
Finally		CopyThree	Task3c.CopyFileThree	FileCopier.exe	c:\usability\filetester\filecopier\copyfilethree.cs	63
Catch	No	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	27
Catch	Yes	CopyTwo	Task3c.CopyFileTwo	FileCopier.exe	c:\usability\filetester\filecopier\copyfiletwo.cs	32

Indication that fault was handled

Figure 4-11. Fault Handled in Second Method

Looking at the results, you can assess whether the evidence showing that the exception was handled represents how you initially intended to handle the exception.

Fault Simulator Integral to Best Practices

This chapter showed you how you can verify the robustness of the error handling in your source code. Making Fault Simulator an integral part of product development ensures that your code will conform to best practices. Properly testing your code prior to deployment will help secure a more robust and stable product.

Glossary



.NET Framework fault

Represents a specific exception that a method call of a .NET Framework Class Library class can throw; can be configured at a specific source location or independent of location; see [“NET Framework Faults”](#) on page 10

Catch block

Contains code that handles a specific exception or group of exceptions; see [“Catch Block”](#) on page 48

Command line

Allows you to use scripts and batch files to automate the testing of applications with specific fault sets; requires the creation of a fault set before Fault Simulator is invoked on the command line; results are written to the results file for analysis; see [“Fault Simulator Command Line Interface”](#) on page 8

Environmental fault

Represents the emulation of failure conditions applied to several classes of methods that deal with runtime environments; not specific to source location for managed or native code; see [“Environmental Faults”](#) on page 11

Error handlers (Win32)

Log of unwind events or a log of return values applicable to the selected fault instance; see [“Error Handling for Function Calls”](#) on page 46

Exception

Unanticipated event that happens while a program is running that interrupts the normal operation of that program; see [“What is an Exception”](#) on page 46

Fault descriptor

Any combination of a specified fault, method, and/or properties (arguments, parameters, or conditions) used to trigger a fault instance during a fault simulation; see [“Fault Descriptors”](#) on page 10

Fault set

Collection of one or more fault descriptors; uniquely named file (stored as an XML document) that contains fault descriptors and configuration settings used in fault simulation; see [“Fault Sets”](#) on page 11

Fault simulation

Alternative to traditional testing methodology; validates the robustness of the software application code; functional basis for DevPartner Fault Simulator; see [“Fault Simulation Enhances Software Testing”](#) on page 6

Finally block

Completes cleanup operations in the exception handling code; see [“Finally Block”](#) on page 48

Load testing

Assesses an application’s tolerance to increased load; see [“Load Testing”](#) on page 31

Standalone

Separate application and accompanying user interface available in DevPartner Fault Simulator; helps quality assurance professionals validate applications under development; see [“Fault Simulator Standalone Application”](#) on page 8

Stress testing

Involves running an application and then monitoring program behavior under adverse, atypical conditions; see [“Stress Testing”](#) on page 30

Structured Exception Handling

Necessary component of any well-written program where developers organize the exception handling code in structured blocks using try-catch-finally procedures; see [“Structured Exception Handling”](#) on page 47

Try block

Represents entry into exception handling code; see [“Try Block”](#) on page 47

Index



Symbols

.NET Framework
 fault 8, 10
 namespace 10

A

accessibility [viii](#)
adding .NET Framework fault [16](#)
adding environmental fault [21](#)
analyzing error-handling code [6](#)
application integrity [26](#)
application logic [29](#)
application vulnerability layers [29](#)
automated testing [33](#)
automating fault simulations [8](#)

B

black box testing
 QA-centric [33](#)
 user scenario [38](#)
block
 catch [48](#), [54](#), [55](#), [56](#), [59](#)
 finally [56](#)
 try [47](#)

C

call stack data [22](#), [55](#)
call stack view [54](#), [56](#), [57](#), [58](#)
catch block [47](#), [48](#), [59](#)
client [29](#)
COM [11](#)
COM APIs [46](#)
command line [8](#)
costs from defects [27](#)

D

data integrity [26](#)
data recovery [26](#)
database server [29](#)
debugging error-handling code [6](#)
development environments [1](#)
DevPartner Fault Simulator
 command line [8](#)
 SE [22](#)
 standalone application [8](#), [19](#), [20](#)
 quick start [19](#)
 supported features [9](#)
 user interfaces [7](#)
 using in Visual Studio [15](#), [34](#)
 quick start [14](#)
 with coverage analysis [12](#)
DevPartner Fault Simulator window
 in Visual Studio [14](#)
 standalone application [19](#)
disk I/O [11](#)
 missing file [39](#), [42](#)

E

ending the simulation [18](#)
environmental faults [8](#), [11](#)
error handlers view [54](#), [56](#)
 catch block considerations [59](#)
 managed code [55](#)
error handling
 error handlers [45](#)
 incorporation into application code [32](#)
evaluation (14-day) [4](#)
exception [46](#)

F

failed calls [6](#)

- fault descriptors [10, 18](#)
- fault sets [11, 38](#)
- fault settings [16](#)
- fault simulation
 - alternatives [5](#)
 - end [18](#)
 - start [18, 21](#)
 - using DevPartner Fault Simulator [7](#)
 - with coverage analysis [12](#)
- fault types [10](#)
- finally block [47](#)
- functional testing [30](#)

H

- handling the correct faults [58](#)
- highlighted source [16, 50](#)

I

- installing Fault Simulator
 - full product [3](#)
 - license configuration [3](#)
 - SE [4](#)
- integration testing [30](#)
- interpreting results [52](#)
- invalid data [28](#)

L

- layers of application vulnerability [29](#)
- logic errors [28, 45](#)

M

- managed code
 - error handlers view [55](#)
 - source highlighting [51](#)
- manual testing [31](#)
- memory [11](#)

N

- namespaces [10, 47](#)
- native code
 - example [57](#)
- network [11](#)
 - excessive traffic [6](#)
 - failure [6](#)
- non-traditional testing [30](#)

O

- operating systems [1](#)

P

- properties view [54](#)

Q

- quick start
 - Fault Simulator in Visual Studio [14](#)
 - Fault Simulator standalone application [19](#)

R

- registry [11](#)
- regression testing [30](#)
- results file [9](#)
- results window [18, 22](#)
- runtime errors [45](#)

S

- SE (DevPartner Fault Simulator) [22](#)
- simulated faults pane [53](#)
- software development goals [25](#)
- source highlighting [16, 50](#)
- source statement [10, 15](#)
- standalone application [8](#)
- starting the fault simulation [18, 21](#)
- startup project [15](#)
- stress testing [30](#)
- structured exception handling [47](#)
- supported platforms [1](#)
- syntax errors [45](#)
- system requirements [1](#)

T

- testing
 - automated [33](#)
 - black box [33](#)
 - white box [32](#)
- testing error-handling code [6](#)
- traditional testing [6](#)
- try block [47](#)

U

- undetected software bugs [6, 45](#)
- uninitialized data [28](#)
- unit testing [30, 33](#)
- user scenario
 - standalone application [20, 38](#)
 - using Fault Simulator in Visual Studio [15, 33](#)

V

- virtual testing [31](#)

W

white box testing

- code scrutiny [33](#)

- developer-centric [32](#)

- user scenario [33](#)

- using Fault Simulator in Visual Studio [33](#)

Win32 APIs [46](#)