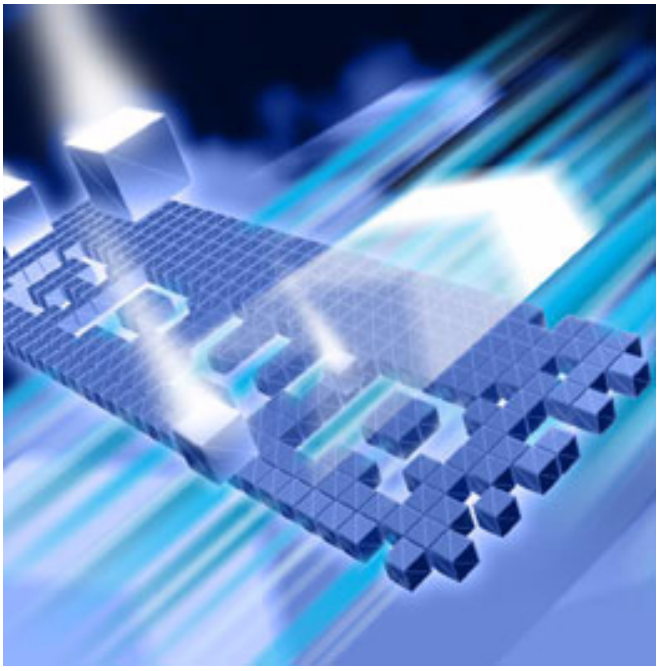


Understanding DevPartner Fault Simulator®

Release 2.0

COMPUWARE®

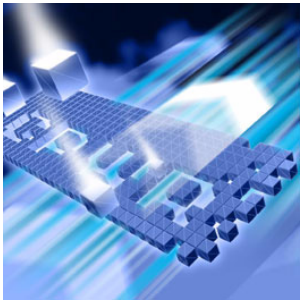
Technical support is available from our Technical Support Hotline or via our FrontLine Support Web site.

Technical Support Hotline:
1-800-538-7822

FrontLine Support Web Site:
http://frontline.compuware.com

August 25, 2006

Table of Contents



Chapter 3

Understanding Fault Simulator Fundamentals

Chapter 4

Performing Quality Assurance Tasks

Chapter 5

Enhancing Quality Assurance Testing

Chapter 6

Setting Up Exception Handler Tests in Visual Studio

Chapter 7

Evaluating Error Handlers

Chapter 8

Improving Software Quality

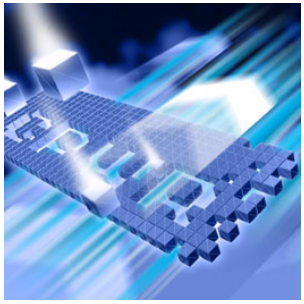
Appendix A

Troubleshooting

Appendix B

Command Line Quick Reference

Preface



The Preface introduces you to the *Understanding DevPartner Fault Simulator* manual and outlines what is contained in this book. The Preface summarizes the many new features in this release.

Who Should Read This Manual

This manual assists software developers and quality assurance engineers 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 familiarity with Microsoft Windows and Visual Studio.

What This Manual Covers

This manual contains the following chapters and appendixes:

Chapter 1, "Installing DevPartner Fault Simulator" outlines system requirements, supported environments, product dependencies. It also provides installation instructions for each of the product editions of DevPartner Fault Simulator.

Chapter 2, "Introducing DevPartner Fault Simulator" acquaints you with DevPartner Fault Simulator, and describes the supported functionality. It also highlights the new features in this release, including an introduction to the QA Edition.

Chapter 3, "Understanding Fault Simulator Fundamentals" explains common concepts, terminology, and functionality in Fault Simulator.

Chapter 4, "Performing Quality Assurance Tasks" summarizes the features geared to quality assurance engineers and also explains how to perform fundamental tasks in Fault Simulator.

Chapter 5, "Enhancing Quality Assurance Testing" considers various testing methodologies used by quality assurance engineers and explores the challenges of effective testing. It then presents a user scenario that shows how Fault Simulator enhances quality assurance testing objectives using fault simulation.

Chapter 6, "Setting Up Exception Handler Tests in Visual Studio" acquaints you with Fault Simulator in Visual Studio. This chapter also introduces you to new functionality that helps improve your exception handling code.

Chapter 7, "Evaluating Error Handlers" reviews different exception handling approaches, with an emphasis on structured exception handling. The chapter then helps you understand fault simulation results so that you can evaluate the robustness of the error handlers in your code.

Chapter 8, "Improving Software Quality" helps software developers enhance software quality by providing advice to improve your exception handling code.

Appendix A, "Troubleshooting" provides assistance to resolve issues you might encounter using Fault Simulator either in Visual Studio or the standalone application, along with other general issues.

Appendix B, "Command Line Quick Reference" introduces the Fault Simulator command line interface. It also includes a quick reference of the Fault Simulator commands, as well as the return codes that the command line interface might generate.

What's New in This Release

DevPartner Fault Simulator 2.0 introduces several new features geared to assist software developers and quality assurance engineers. These features vary depending on where they are available in the product.

- Fault Simulator in Visual Studio and in the Standalone
  - Improved Display and Order of Fault Information

    Fault Simulator organizes fault information in predefined groups and sorting order. You can optionally customize how faults are organized based on criteria such as fault descriptor name, fault type, checked status, etc., to help you better understand the contents being displayed. See "Why Might I Reorganize the Display of Fault Information?" on page 21.
  - Integration into Visual Studio Team System

    If you are already using Visual Studio Team System, you can submit a work item from Fault Simulator. Submitted data reflects error handler and call stack results collected from a fault simulation session. See "Visual Studio 2005 Team Foundation Server Integration Requirements" on page 4. See also "How Do I Submit Defects Generated from Fault Simulator?" on page 24.
  - Ability to Suspend an Active Simulation

    You can suspend an active fault simulation session. Suspending a session after the simulation has started disables all fault descriptors set to activate in the target application. You can resume the session when appropriate to your testing requirements. See "Why Would I Suspend a Fault Simulation Session?" on page 20.
- Fault Simulator in Visual Studio only
  - Advice to Improve Your Exception Handling Code

    Fault Simulator analyzes your managed code and identifies areas where you can improve your exception handling code. Fault Simulator then suggests how you can correct code constructs within the current method scope, such as inserting try/catch blocks and/or adding XML appropriate XML `<exception>` tags where needed. See "Walk Through Focusing on Exception Handlers in Your Code" on page 63.
  - Enhanced .NET Support

    You can simulate .NET faults associated with the .NET Framework 1.1 and 2.0 using Fault Simulator in Visual Studio. You can also simulate .NET faults on methods in third-party software and e-

commerce managed applications, as well as methods created in user-written managed code. Refer to the DevPartner Fault Simulator online help for more information on .NET support.

The standalone application also supports e-commerce applications, as well as limited .NET support. Refer to the online help that accompanies the standalone application for more information.

- Fault Simulator Standalone Application only
  - Automatic Creation of a Batch Script

    You can use the Fault Simulator standalone application to generate a batch file that represents your fault preferences. This feature takes the guesswork out of building a working batch script. See "Automatically Generating a Batch Script" on page 41.
  - Automatic Generation of Environmental Faults

    The Fault Simulator standalone application can watch your target application for program activities and resources used by your application as you use it normally. From that data, Fault Simulator can generate a collection of environmental faults that you can use in a subsequent fault simulation. See "Automatically Generating Environmental Faults" on page 29.
- Fault Simulator Command Line Interface

  Fault Simulator has added new commands to facilitate multiple executions of the same fault set. See Appendix B, "Command Line Quick Reference" for an overview.

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: **`file.txt`**
- 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 /?:`***`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:

- The **Fault Simulator in Visual Studio** online help is integrated with Fault Simulator in Visual Studio.
- The **Fault Simulator standalone application** online help accompanies the Fault Simulator standalone application.
- The **Command line** help is available in the following formats:
  - As console help from the command line
  - As online help from InfoCenter
- The ***Understanding DevPartner Fault Simulator*** manual resides on your DevPartner Fault Simulator CD in Adobe Acrobat (.pdf) format.

Use these other resources for additional assistance:

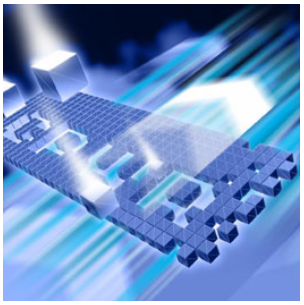
- The **Release Notes** provide current information. It also links directly to a late-breaking known issues file on the Web.
- The ***Distributed License Management License Installation Guide*** manual provides specific details on licensing Compuware products.

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



- System Requirements
- Supported Environments and Product Dependencies
- Licensing
- DevPartner Fault Simulator Installation
- Accessing DevPartner Fault Simulator
- Coexistence with Other Compuware Products

This chapter outlines the system requirements, supported operating systems, and development environments, along with installation instructions.

System Requirements

Table 1-1 lists the minimum hardware requirements.

Table 1-1. Minimum Hardware Requirements

Item	Specification
Processor	Pentium III, 733 MHz
Memory	128 MB above the memory requirements for the system and Visual Studio
Disk space	600 MB
Video	1024x768, 16-bit color
Other	CD drive

Table 1-2 lists the supported operating systems.

Table 1-2. Operating Systems

Operating System	Editions	Internet Explorer Browser and Internet Information Server
Windows Vista (Beta 2) (32-bit only)	Business Enterprise Ultimate	IE 7.0 IIS 7.0
Windows XP (SP2) (32 bit only)	Professional[1] Tablet PC	IE 5.5 IIS 5.1
Windows Server 2003 (SP1) (32 bit only)	Standard Enterprise Web	IE 6.0 IIS 6.0

Supported Environments and Product Dependencies

Table 1-3 lists details about the Visual Studio integration.

Table 1-3. Visual Studio Integration

Development Environments[1]	Editions	.NET Framework	Languages
Visual Studio 2005	Professional Edition Team Edition for Software Architects Team Edition for Software Developers Team Edition for Software Testers Team Suite (64-bit development not supported)	2.0	Visual C# Visual Basic .NET ASP.NET technologies[1] Unmanaged languages[2] (primarily C++)
Visual Studio .NET 2003	Developer Professional Enterprise Architect	1.1[3]	Visual C# Visual Basic .NET ASP.NET technologies Unmanaged languages (primarily C++)

[1]Fault Simulator does not support Java, JavaScript, or Visual J# in ASP.NET applications.
[2]Fault Simulator supports unmanaged projects for environmental faults only.
[3]Compact Framework is not supported.

Table 1-4 identifies other products that integrate into Fault Simulator.

Table 1-4. Other Product Dependencies

Product	Version	Details
Compuware TrackRecord	6.2	Required for defect submission only
Visual Studio Team Foundation Server	8.0.50727.147	Required for Work Item submission only Team Explorer must be installed. The *Work Item* template must include *Bug* type. See "Visual Studio 2005 Team Foundation Server Integration Requirements" on page 4

Licensing

Complete information on installing and managing licenses can be found in *Distributed License Management License Installation Guide* on the product CD. For additional information, visit the Compuware Web site at `http://frontline.compuware.com/sw/license_default.asp` or call Worldwide License Management at 1-800-538-7822.

Note: DevPartner Fault Simulator SE uses the DevPartner Studio license. See "Installing DevPartner Fault Simulator SE" on page 5 for more information.

DevPartner Fault Simulator Installation

DevPartner Fault Simulator detects your system configuration and will install the applicable software setup. If your system configuration does not meet the minimum system requirements, the software installation will not proceed. Prior to the installation, review the following installation considerations:

Previous Versions of DevPartner Fault Simulator

If a previous version of DevPartner Fault Simulator resides on your system, you must uninstall that version (via **Add or Remove Programs**) and then restart your machine.

Visual Studio 2005 Team Foundation Server Integration Requirements

To submit a work item into Visual Studio Team System, ensure that:

- The Team Explorer client is installed.
- You are connected to a Team Foundation Server.

  DevPartner Fault Simulator supports the version of Team System installed at your site, and that a compatible version of Team Explorer is installed on your client system.
- Your Visual Studio Team System project supports the **Work Item** of the Type **Bug**.

  Visual Studio 2005 Team System includes two default project templates, Microsoft Solutions Framework (MSF) for Agile Software Development and MSF for CMMI Process Improvement. Both templates support **Bug** as a **Work Item** type and are compatible with Fault Simulator.
- If the default **Work Item Type** definition in the project schema has changed or a custom project without this **Work Item Type** exists, you will not be able to submit a work item to Visual Studio Team System.
- A project must be selected.

Consult the Microsoft Visual Studio 2005 Team System documentation for more information.

Note: Fault Simulator does not support Visual Studio Team System in Visual Studio .NET 2003.

Installing DevPartner Fault Simulator

To install DevPartner Fault Simulator:

1. Insert the product CD into your CD-ROM drive.

   If you have autorun enabled, the setup proceeds automatically. If not, open **Add or Remove Programs**, click **Add New Programs**, and then click **CD or Floppy**.
2. To install DevPartner Fault Simulator, click **Install DevPartner Fault Simulator**.

   This action will install:

   - Fault Simulator in Visual Studio
   - Fault Simulator standalone application
   - Command line interface
3. Follow the on-screen instructions to complete the installation.

Installing the DevPartner Fault Simulator QA Edition

To install the QA Edition:

1. Insert the product CD into your CD-ROM drive.

   If you have autorun enabled, the setup proceeds automatically. If not, open **Add or Remove Programs**, click **Add New Programs**, and then click **CD or Floppy**.

2. To install the DevPartner Fault Simulator QA Edition, click **Install DevPartner Fault Simulator QA Edition**.

   This action will install:

   - Fault Simulator standalone application
   - Command line interface

3. Follow the on-screen instructions to complete the installation.

Note: You cannot use the QA Edition in Visual Studio.

Installing DevPartner Fault Simulator SE

To install DevPartner Fault Simulator SE:

1. Insert the DevPartner Studio Professional Edition CD into your CD-ROM drive.

   If you have autorun enabled, the setup proceeds automatically. If not, open **Add or Remove Programs**, click **Add New Programs**, and then click **CD or Floppy**.

2. Follow the on-screen instructions to complete the installation.

Upgrading from Fault Simulator SE to the DevPartner Fault Simulator

At any time, you can upgrade to the full product. To do so, first uninstall DevPartner Fault Simulator SE. For more information, choose **Programs > Compuware DevPartner Fault Simulator SE > How to Upgrade to Fault Simulator** from the **Start** menu.

Note: See "DevPartner Fault Simulator SE" on page 12 for more information.

Accessing DevPartner Fault Simulator

Following a successful installation, you can access DevPartner Fault Simulator as indicated in Table 1-5.

Table 1-5. Accessing DevPartner Fault Simulator

Edition	How to Access
DevPartner Fault Simulator as a standalone application	**Programs** > **Compuware DevPartner Fault Simulator** > **Fault Simulator**
DevPartner Fault Simulator in Visual Studio	Open Visual Studio
DevPartner Fault Simulator QA Edition as a standalone application	**Programs** > **Compuware DevPartner Fault Simulator QA Edition** > **Fault Simulator**
DevPartner Fault Simulator command line interface	From the command line
DevPartner Fault Simulator SE in Visual Studio	Open Visual Studio (Requires DevPartner Studio)

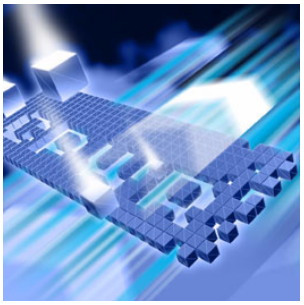
Coexistence with Other Compuware Products

Fault Simulator integrates with Compuware TrackRecord 6.2 .1 and 6.2.2 and coexists with other Compuware DevPartner products, such as:

- DevPartner Studio 8.1 or later
- DevPartner SecurityChecker 1.0.1 or later
- Compuware TestPartner 5.4 or later

Chapter 2

Introducing DevPartner Fault Simulator



This chapter introduces you to DevPartner Fault Simulator. It highlights how Fault Simulator uses fault simulation to assist software developers and quality assurance engineers. This chapter differentiates the supported functionality depending on where functionality is available.

Introducing DevPartner Fault Simulator

Application code is often written to address error handling, but little if any is properly scrutinized or tested prior to deployment. DevPartner Fault Simulator provides a workable solution to this long-standing problem.

DevPartner Fault Simulator is a software development and quality assurance tool that uses fault simulation to mimic real-world application failures. Fault Simulator helps developers and quality assurance engineers simulate faults in a running program. Software developers can debug the exception handling code, without risking the application under test or disrupting the operating or debugging environment. Quality assurance engineers can test an application's reaction to errors in a predictable and repeatable environment. Using actual simulation results, they both can verify the application's ability to tolerate a variety of failure conditions prior to deployment, avoiding costly production errors afterward.

Fault Simulator helps ensure that your application is thoroughly tested.

- You can simulate real-world fault conditions without affecting the operating environment or the application being tested:
  - You can configure an environmental fault to test how your application will respond to unexpected environmental failures.
  - You can configure a .NET fault to test the exception handling code directly in your source.
- Fault Simulator generates results on these simulated faults. Results reveal the success or failure of error handlers in your code, helping you troubleshoot problems in your application code.
  - Fault Simulator shows evaluated and executed catch blocks, letting you analyze the execution path that the exception handling code took.
  - Fault Simulator traces the steps through your code that led to the exception being thrown or not.

DevPartner Fault Simulator Supported Functionality

You can use Fault Simulator in Visual Studio, as a standalone application (outside Visual Studio), and from the command line.

Table 2-1 summarizes where Fault Simulator functionality is supported.

Table 2-1. Supported Functionality in DevPartner Fault Simulator

Feature	Available in Visual Studio	Available in the Standalone	Available from the Command Line	Where to Learn More
Simulate faults in application code outside Visual Studio	Yes	Yes	Yes	"Available as a Standalone Application" on page 10
Simulate faults in Visual Studio	Yes	No	No	"Integrated in Visual Studio" on page 10
Add a .NET fault	Yes	No	No	"Configuring a .NET Fault in Managed Code" on page 78
Add an environmental fault	Yes	Yes	No	"Configuring an Environmental Fault" on page 80

Table 2-1. Supported Functionality in DevPartner Fault Simulator (Continued)

Feature	Available in Visual Studio	Available in the Standalone	Available from the Command Line	Where to Learn More
Modify an existing fault (.NET or environmental)	Yes	Yes	No	"How Do I Use Fault Descriptors to Simulate Fault Conditions?" on page 14
Incorporate code coverage in a fault simulation session	Yes	Yes	Yes	"Can I Collect Coverage Analysis During a Fault Simulation?" on page 23
Reorganize the display of fault information	Yes	Yes	No	"Why Might I Reorganize the Display of Fault Information?" on page 21
Get advice on improving your exception handling code	Yes	No	No	"Walk Through Focusing on Exception Handlers in Your Code" on page 63
Suspend a fault simulation session in progress	Yes	Yes	No	"Why Would I Suspend a Fault Simulation Session?" on page 20
Review collected call stack and error handler simulation details	Yes	Yes	No	"Evaluating Error Handler Results" on page 84
Integrate fault simulation session results into Visual Studio Team System or Compuware TrackRecord	Yes	Yes	No	"How Do I Submit Defects Generated from Fault Simulator?" on page 24
View source code associated with a fault instance	Yes	No	No	"Viewing the Source Statement That Handled the Fault" on page 87
Execute scripts to automate fault simulations from the command line	No	No	Yes	"Fault Simulator Commands" on page 118
Have Fault Simulator watch your target application and generate environmental faults	No	Yes	No	"Automatically Generating Environmental Faults" on page 29
Have Fault Simulator generate a script that you can run from the command line	No	Yes	No	"Automatically Generating a Batch Script" on page 41

Available as a Standalone Application

Using the Fault Simulator standalone application helps quality assurance validate the stability of applications under development *outside* Visual Studio. The standalone application complements your ongoing functional and regression testing with functionality geared to supplement your quality assurance objectives. See Table 4-1, Supported Functionality in the Standalone Application, on page 28 for a list of features.

You can safely mimic real-world environmental failure conditions and see how the application might respond. You can create and configure new environmental faults (see "Using Environmental Faults to Simulate Application Failures" on page 14) to validate applications under development.

You can share your testing results with development, along with the original fault set that you used to generate these results, to help software development conduct code modifications more quickly and efficiently. You can also reuse fault sets configured in development for subsequent regression testing.

Note: The standalone application is available in DevPartner Fault Simulator, as well as in the QA Edition. See "Accessing DevPartner Fault Simulator" on page 6.

Integrated in Visual Studio

Using Fault Simulator in Visual Studio helps you track and troubleshoot the exception handlers in your managed source code. Fault Simulator simulates faults without disrupting the operating or debugging environments.

You can create .NET faults to test exception handling either at a specific source location or independent of location. This means that you can artificially throw exceptions in your source and see how the executable reacts to the simulated exception. See "Using .NET Faults to Simulate Thrown Managed Exceptions" on page 19.

You can also configure environmental faults that simulate failure conditions, such as registry, COM, network, disk I/O, or memory anomalies.

You can reuse the fault set files you create for each fault simulation test, and also share these files with quality assurance for subsequent test verification.

Available from the Command Line

Fault Simulator extends its fault simulation capabilities to the command line. From the command line, you can automate fault simulations on projects that do not require user intervention. You can use this functionality to enhance the unit testing, functional testing, and regression testing you perform on applications under development. For example, from the command line, you can run scripts repeatedly to augment regression testing. The command line uses fault sets previously configured in Fault Simulator. Results generated from the command line are available for viewing in the Fault Simulator user interface.

Note: See Appendix B, "Command Line Quick Reference" for an overview of the command line interface.

DevPartner Fault Simulator QA Edition

DevPartner Fault Simulator 2.0 has introduced the DevPartner Fault Simulator QA Edition, a new product edition in this release. The focus of the QA Edition is to enhance your quality assurance testing. The QA Edition helps you uncover environmental vulnerabilities in your target application. Used outside Visual Studio, the QA Edition includes the following components:

- Standalone application (see "Available as a Standalone Application" on page 10)
- Command line interface (see "Introducing the Command Line Interface" on page 117)

The DevPartner Fault Simulator QA Edition lets you:

- Use Fault Simulator as a standalone application outside Visual Studio
- Add, modify, and simulate environmental faults in application code
- Modify existing .NET faults
- Reorganize the display of fault information
- Collect coverage information during a fault simulation session
- Submit a work item to Visual Studio Team System
- Submit a defect to Compuware TrackRecord
- Suspend a fault simulation session in progress
- Review fault simulation details
- Have Fault Simulator watch your target application and generate environmental faults
- Have Fault Simulator generate a working batch script and then run it from the command line

DevPartner Fault Simulator SE

Tip: See "DevPartner Fault Simulator Supported Functionality" on page 8 for more information.

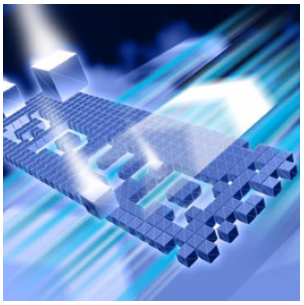
DevPartner Fault Simulator SE contains a limited feature set. It gives you an opportunity to sample some of the features in the full product. DevPartner Fault Simulator SE lets you:

- Add or modify a source-based .NET faults (in the .NET Framework 1.1 and 2.0)
- Reorganize the display of fault information
- Collect coverage information during a fault simulation session
- Submit a work item to Visual Studio Team System
- Submit a defect to Compuware TrackRecord

DevPartner Fault Simulator SE is available in Visual Studio when DevPartner Studio is also installed. DevPartner Fault Simulator and DevPartner Fault Simulator SE cannot be installed on the same machine. To upgrade to the full product, see "Installing DevPartner Fault Simulator SE" on page 5.

Chapter 3

Understanding Fault Simulator Fundamentals



- How Does Fault Simulation Help Ensure Application Stability?
- How Do I Use Fault Descriptors to Simulate Fault Conditions?
- Why Would I Suspend a Fault Simulation Session?
- Why Might I Reorganize the Display of Fault Information?
- Why Would I Have Fault Simulator Create Environmental Faults for Me?
- How Do I Edit Environmental Faults Created for Me?
- Can I Reuse Fault Sets?
- Can I Collect Coverage Analysis During a Fault Simulation?
- How Do I Submit Defects Generated from Fault Simulator?

This chapter explains concepts and terminology used in Fault Simulator.

How Does Fault Simulation Help Ensure Application Stability?

To consider the significance of fault simulation, it helps to first look at fault and simulation individually.

A fault is an abnormal, possibly unstable condition or defect that could lead to a failure. In technology, a fault refers to an event that occurs during the execution of an application that could cause an unexpected behavior. The behavior could be manageable such as a temporarily missing menu option or a brief *Please Wait* message, or more extreme, such as data corruption, a frozen user interface, or a sudden crash.

A simulation imitates a real-world event. In technology, a simulation demonstrates the possible effects of certain conditions in order to gain

insight into the test item's functionality, stability, and subsequent reaction to the original event.

Software developers build applications to perform a specific function and handle subsequent failure conditions. Quality assurance engineers conduct testing to verify that the application can withstand environmental abnormalities. However, software developers and quality assurance engineers might not know for certain whether the software will work satisfactorily once it is deployed.

Fault Simulator provides a workable solution, using the concept of fault simulation to test for failures in an application. Fault Simulator can simulate a variety of fault conditions in a running program. Fault simulation testing can artificially uncover whether an application can appropriately recover from a failure. It can cause a line of code to artificially fail in order to test the program's ability to recover from whatever adverse conditions might ensue. Fault Simulator simulates faults without changing the operating environment or the debugging session.

How Do I Use Fault Descriptors to Simulate Fault Conditions?

Fault Simulator uses the concept of fault descriptors to designate the ability to define fault preferences that you want Fault Simulator to use when simulating a particular fault condition. You can set criteria, such as specifying the managed method for a .NET fault that you want artificially called, the exception you want thrown against that method, or the argument or parameter that you want to trigger an instance of that particular fault.

Fault Simulator supports two kinds of fault descriptors:

- Environmental faults
- .NET faults

Using Environmental Faults to Simulate Application Failures

Unintended failure conditions, such as a corrupt file, a full disk, an unreadable registry value, or an unavailable server, can sabotage an application's performance. Fault Simulator lets you create and configure environmental faults to uncover these scenarios.

An environmental fault is an error condition that results from the environment where the target program is executing. You set an environmental fault to validate the robustness of a program under test by simulating faults that couple with external dependencies that the

program requires. For example, you might configure a network-related fault to affect network-based method calls, such as remote server crash or connection timeout, without affecting the operating environment of the application being tested or other applications running on the system. Fault Simulator supports five categories of environmental faults, as described next:

- Disk I/O
- Network
- COM
- Registry
- Memory

Disk I/O Environmental Faults

Failures related to data loss, disk I/O operations, or read-write privileges can have a significant impact on an application's functionality. The possible downtime resulting from a file-related failure condition (such as a corrupted or missing file, disk full, or missing directory), can have a detrimental effect, especially if a failure incapacitates the application.

An application relies heavily on file-related tasks that either the user performs or the application performs internally. While commonplace, an inability to perform file-related tasks could detrimentally affect the application's performance. The user interface might become unresponsive, data might get lost, or worse, the application could crash.

Disk I/O faults simulate common, real-world problems related to disk I/O operations and privileges. Disk I/O faults affect file and directory access that prevent the application from executing, but do not physically alter existing files or directories.

Table 3-1 summarizes the disk I/O faults.

Table 3-1. Disk I/O Faults

Disk I/O Fault	Simulates
Corrupt file	An inability to access a file whose contents became corrupted
Disk full	An inability to allocate more disk space on a specified drive
DLL or assembly not found	An inability to load a pre-existing module into memory
File locked	A failure condition where another application locks the designated file, preventing the target application from accessing it

Table 3-1. Disk I/O Faults (Continued)

Disk I/O Fault	Simulates
Insufficient read-directory privileges	An inability to gain read access to a pre-existing directory
Insufficient read-file privileges	An inability to gain read access to a pre-existing file
Insufficient write-directory privileges	An inability to gain write access to a pre-existing directory
Insufficient write-file privileges	An inability to gain write access to a pre-existing file
Missing directory	An inability to locate a pre-existing directory
Missing file	An inability to locate a pre-existing file

Network Environmental Faults

Loss of network connectivity is a common but serious problem in a network-aware environment. Network-related vulnerabilities result from problems with network access or resources. However, to test these failure conditions by physically altering the environment (such as, removing a network interface card, unplugging a network cable, or disconnecting a network server) could put the application, and any other applications or machines that also depend on that network, at greater risk.

Table 3-2 summarizes the network faults.

Table 3-2. Network Faults

Network Faults	Simulates
Connection timed out	A disruption to network operation and performance
Network offline	A variety of network failures
Remote server crashed	A failed connection between the monitored program and the associated network
Server not available	A failure to connect to the network

COM Environmental Faults

The Microsoft Component Object Model (COM) technology has revolutionized how software components link with external applications and interact with Windows services. However, COM might introduce another set of unanticipated environmental issues. Applications can succumb to COM-related anomalies arising from installing or uninstalling software, the presence of third-party components that share a COM object, or changes to existing COM components.

Table 3-3 summarizes the COM faults.

Table 3-3. COM Faults

COM Faults	Simulates
CLSID not found	A failed COM function call attempt to query the registry for the CLSID
DLL not found	A failure with class instantiation that prevents a DLL from loading
Interface not registered	A failure where the entry for a pre-existing interface is missing
ProgID not found	An inability for COM to query a registry key value

Registry Environmental Faults

The Registry controls many executable and internal structures for an application and the Windows environment. An application and the underlying operating environment rely on configuration settings in the Registry. Registry data that an application accesses could inadvertently get altered during the course of normal program activity. Registry failures could occur when the application is started, when internal drivers are loaded, or when a user logs into the system that accesses the application. An altered or corrupted Registry setting could impact the application's ability to function. However, testing for Registry-related environmental failures by manually changing Registry settings could place the application at greater risk.

Table 3-4 summarizes the registry faults.

Table 3-4. Registry Faults

Registry Faults	Simulates
Corrupt registry value	The pre-existence of a corrupt value or a legitimate value with the wrong data type
Insufficient read privileges	The monitored program's failure to gain read access to the pre-existing registry key
Insufficient write privileges	The monitored program's failure to gain write access to the pre-existing registry key
Missing key	An inability to open a pre-existing registry key
Missing value	An inability to access a pre-existing registry key value or one of the key's subkeys

Memory Environmental Faults

Memory is not limitless. However, applications are often written as if memory will never run out. Given that memory issues have the capacity to incapacitate an application, can you risk releasing the application without sufficient and confident testing? How will your application respond to memory failures that arise from:

- Targeting memory-management facilities in Windows XP or later
- Limited available virtual memory that the program can allocate
- Preventing the program from allocating (or re-allocating) memory from its default heap regardless of the actual memory previously allocated

Table 3-5 summarizes the memory faults.

Table 3-5. Memory Faults

Memory Faults	Simulates
Heap allocation limits	A condition that prevents the monitored program from allocating (or re-allocating) memory from its default heap regardless of the actual memory previously allocated
Low memory notification	A low-memory condition that targets memory-management facilities in Windows XP or later
Virtual memory allocation limit	A limitation of available virtual memory that the monitored program can allocate

Using .NET Faults to Simulate Thrown Managed Exceptions

Software developers attempt to incorporate sufficient exception handling into the applications they build. However, testing the validity of the exception handling code can be difficult. Fault Simulator facilitates the testing of exception handling via .NET faults.

A .NET fault is a managed exception that is artificially thrown after a supported method has been called, either at a source statement or independent of a source location. You can configure a .NET fault to target a specific method call in your managed code. For example, you might set a .NET fault on `String.Format()` to simulate `ArgumentNullException` even if none of the runtime parameters would normally cause that exception.

Fault Simulator supports two types of .NET faults:

- .NET fault that you can add to a source statement that includes a supported method (see "Adding a .NET Fault to a Source Location" on page 68 for more information)
- .NET fault that can be simulated independent of location in the source code

What Does a Fault Instance Represent?

A fault instance represents a fault descriptor that was simulated during a fault simulation session.

- For a .NET fault, a fault instance represents a thrown managed exception that occurs during program execution.
- For an environmental fault, a fault instance represents the simulation of an environmental failure in the target application.

You can review results of fault instances that Fault Simulator simulated to evaluate how the application handled the failure condition during execution. See Chapter 7, "Evaluating Error Handlers" for examples of various results.

Why Would I Suspend a Fault Simulation Session?

Suspending a session gives you the flexibility to have greater control over the fault simulation. There are several reasons why this feature can enhance your testing efforts. You can suspend fault simulation activity to perform other tasks in the target application while the session is suspended. You can suspend a session in order to focus your testing on specific areas of the application during program execution. You can suspend the start of a simulation if you want to test newly added code but you want to bypass existing code that you have already tested. You can also suspend data collection while the Fault Simulator standalone application watches the target application and identifies potential environmental failures that you can incorporate into your testing (see "Automatically Generating Environmental Faults" on page 29).

Another benefit is when simulating the *Heap allocation limits* memory environmental fault. You can use the suspend simulation feature to bypass program initialization errors and prevent unrelated faults from firing prematurely. Pausing a fault simulation gives the target application an opportunity to execute to a point where you would actually want to start simulating faults (such as, giving it time to load system DLLs first). Once the target application has reached an acceptable point of execution, you can resume the fault simulation activity.

Pausing a session does not terminate the session. However, all active fault descriptors are temporarily halted. While in this suspended state, no fault instances will occur even if all other properties (arguments, parameters, and/or conditions) of the fault descriptor have been met. Resuming the session allows the simulation to proceed normally. Suspending or

resuming does not affect fault descriptors that were in a disabled state when the session started.

When you start a fault simulation session, the simulation automatically starts in a non-suspended state. You can, however, suspend simulation before you start the session, in effect starting a simulation in a suspended state. You can suspend or resume a simulation up until the time that the simulation stops (either when you explicitly stop the simulation or the monitored program exits).

If you suspend a session, Fault Simulator will maintain that suspended state while Fault Simulator remains open. In the case of Visual Studio, Fault Simulator also maintains that state until you close the current solution.

Note: Suspending a fault simulation has no effect on the collection of coverage information. See "Can I Collect Coverage Analysis During a Fault Simulation?" on page 23.

Why Might I Reorganize the Display of Fault Information?

Fault Simulator organizes fault information in predefined groups and sorting order. You can optionally customize how faults are organized based on different fault criteria by clicking **Arranged by** (when available) for a list of sorting options (such as, fault category, fault descriptor name, targeted method for .NET faults, and enabled/disabled state).

You might sort fault descriptors as you set up a fault simulation for several reasons. You can rearrange the fault list to create a more logical order to help you prioritize which faults you want to include in your testing. You can reorganize the list of environmental faults (such as by category) that the Fault Simulator standalone application created for you to determine which faults you want to use in your next fault simulation (see "Automatically Generating Environmental Faults" on page 29).

You can also sort the data following a fault simulation session to simplify analysis of the fault simulation results. For example, you might arrange fault instances on the **Specified Faults** pane based on the number of times each fault instance successfully fired, the number of times each one was attempted, or by fault descriptor type, fault descriptor name, or checked status.

Why Would I Have Fault Simulator Create Environmental Faults for Me?

The capability of Fault Simulator to simulate environmental failures in real-world situations is invaluable, but it can be difficult for you to create and configure just the right environmental fault descriptors to fortify your testing matrix.

The Fault Simulator standalone application can watch your target application during program execution and record program activities and resource strings it encounters. From that data, Fault Simulator will generate environmental faults you can use in a subsequent fault simulation. See "Automatically Generating Environmental Faults" on page 29 for more information on using this feature.

How Do I Edit Environmental Faults Created for Me?

If you prefer to create new fault descriptors or modify existing fault properties yourself, rather than simply using those that the Fault Simulator standalone application created for you, you can easily transfer generated faults to a fault editor window. Choosing **Switch to Fault Editor** from the **Edit** menu, you can make the desired changes in the **DevPartner Fault Simulator** window.

In addition, choosing this menu selection enables you to have Fault Simulator automatically generate a batch script file, based on your preferences, for subsequent automated testing. See "Automatically Generating a Batch Script" on page 41.

Can I Reuse Fault Sets?

You can save and reuse fault sets. Fault sets include fault data that you configured for a fault simulation session. Fault sets can be used interchangeably between Visual Studio and the standalone application.

You can also execute a script and reference a fault set file from the command line. See Appendix B, "Command Line Quick Reference" for an overview of the command line interface. Refer to the DevPartner Fault Simulator Command Line online help from the InfoCenter for detailed usage information.

Can I Collect Coverage Analysis During a Fault Simulation?

You can use Fault Simulator with DevPartner coverage analysis to collect coverage and fault simulation data during the same session. This feature is available in Visual Studio and in the standalone application when DevPartner Studio is also installed.

Tip: Refer to the DevPartner coverage analysis online help for more conceptual information about the DevPartner Studio coverage analysis feature.

DevPartner Studio provides coverage analysis to help developers and quality assurance 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.

Figure 3-1 illustrates how Fault Simulator combines a fault simulation session with coverage analysis.

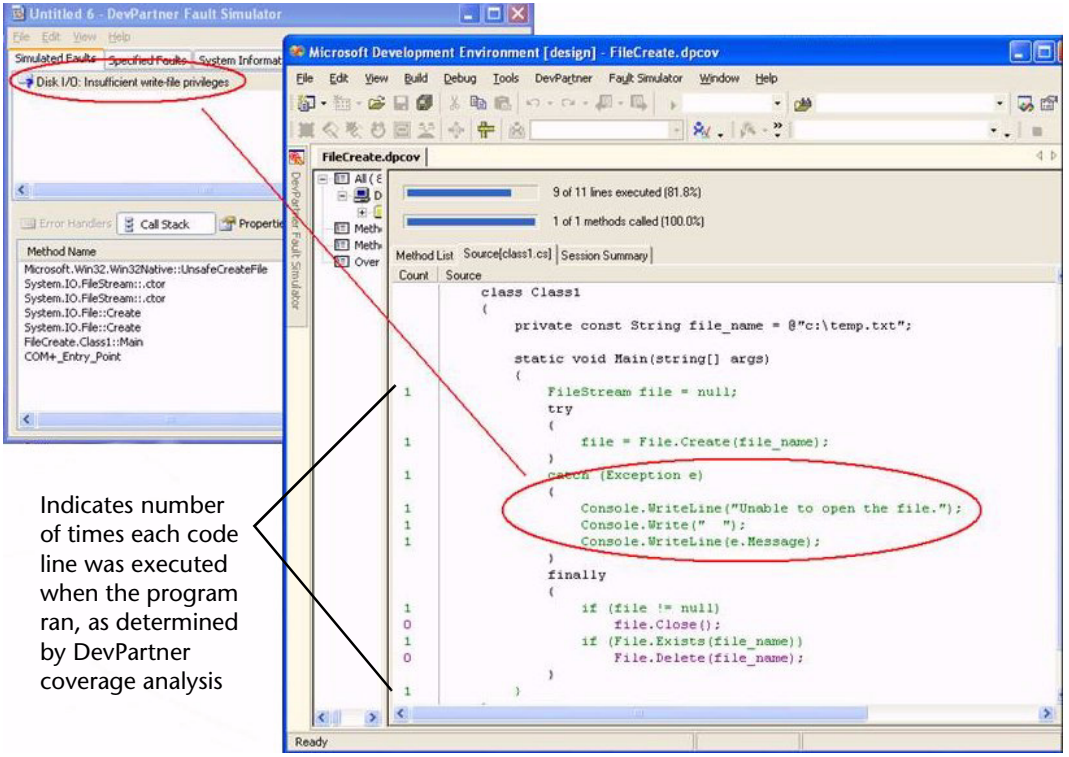


Figure 3-1. Example of Coverage Analysis with a Fault Simulation in Visual Studio

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. Following the simulation, Fault Simulator displays results (shown to the left in Figure 3-1) with call stack and error handler data.

Concurrently, the coverage results reveal whether applications and components have been thoroughly exercised under test conditions. Notice the numbers in the left margin. They indicate the number of times each code line was executed while the program ran. Consult the *Understanding DevPartner* manual that accompanies DevPartner Studio for comprehensive information on code coverage analysis.

Combining Coverage Analysis from the Command Line

You can also combine coverage analysis with a fault simulation from the command line. See Appendix B, "Command Line Quick Reference" for an overview of the command line interface using the `/v` switch. Refer to "Fault Simulator Commands" on page 118 for a quick overview of this and other commands. Refer to the DevPartner Fault Simulator Command Line online help from the InfoCenter for more information.

How Do I Submit Defects Generated from Fault Simulator?

You can submit defects in two ways using Visual Studio Team System or Compuware TrackRecord.

Submitting a Work Item to Visual Studio Team System

Fault Simulator integrates into Visual Studio Team System, the Microsoft software development version control, defect tracking, and process management software. In order to submit a **Work Item** of the type **Bug** from DevPartner Fault Simulator:

- The Team Explorer client must be installed
- An active connection to the Team Foundation Server must exist
- A project must be selected

Note: See "Visual Studio 2005 Team Foundation Server Integration Requirements" on page 4 for more details.

Fault Simulator automatically adds error handler and call stack results collected during a fault simulation.

Consult the Microsoft Visual Studio 2005 Team System documentation for more information.

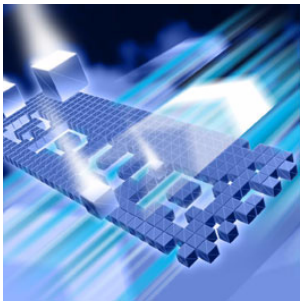
Note: Fault Simulator does not support Visual Studio Team System in Visual Studio .NET 2003.

Submitting a Defect to Compuware TrackRecord

You can submit a defect from Fault Simulator, as long as Compuware TrackRecord, a software defect management tool from Compuware Corporation, is on the same system as Fault Simulator. Fault Simulator automatically adds error handler and call stack results collected during a fault simulation. Refer to TrackRecord on the Compuware Web site for more information about this product. Consult the TrackRecord online and hard-copy documentation for comprehensive information.

Chapter 4

Performing Quality Assurance Tasks



This chapter summarizes the features geared to quality assurance engineers and also explains how to perform fundamental tasks in Fault Simulator.

DevPartner Fault Simulator Supports Quality Assurance

Used outside Visual Studio, the DevPartner Fault Simulator standalone application helps quality assurance engineers improve application quality by ensuring the robustness of applications prior to deployment. Fault Simulator can safely mimic various environmental failure conditions to show how the application responds. Fault Simulator complements functional and regression testing, helping you uncover areas in the application code where the error handling is either flawed or non-existent. With Fault Simulator, you can test more of your application code, ensuring higher-quality applications, lowering project risk, and maximizing productivity.

Functionality that Supports Quality Assurance

Table 4-1 lists the supported functionality in the standalone application that is geared to support quality assurance testing efforts.

Table 4-1. Supported Functionality in the Standalone Application

Functionality	Where to Learn More About this Feature
Use of the Fault Simulator application (outside Visual Studio)	"Available as a Standalone Application" on page 10
Creation and configuration of environmental faults	"Configuring an Environmental Fault" on page 80
Modification of existing .NET faults, (originally created using Fault Simulator in Visual Studio)	"Using .NET Faults to Simulate Thrown Managed Exceptions" on page 19 See Note.
Saving, loading, and reuse of fault sets in subsequent fault simulations	"Can I Reuse Fault Sets?" on page 22
Incorporation of code coverage in a fault simulation session	"Can I Collect Coverage Analysis During a Fault Simulation?" on page 23
Automatic generation of environmental faults derived from observation of program activities	"Automatically Generating Environmental Faults" on page 29
Suspension of an ongoing fault simulation session	"Why Would I Suspend a Fault Simulation Session?" on page 20
Execution of fault simulation scripts from the command line	"Fault Simulator Commands" on page 118
Automatic creation of a batch script that you can subsequently use from the command line	"Automatically Generating a Batch Script" on page 41
Submission of a work item to Team System	"Submitting a Work Item to Visual Studio Team System" on page 24
Submission of a defect to Compuware TrackRecord	"Submitting a Defect to Compuware TrackRecord" on page 25

Note: The standalone application provides limited .NET support. You can modify a .NET fault that was originally created using the Fault Simulator in Visual Studio. However, you cannot create a .NET fault in the standalone application. Refer to the DevPartner Fault Simulator online help for more information.

Automatically Generating Environmental Faults

Fault Simulator can mimic environmental failures to show how an application might react in a real-world situation. While this data is invaluable, it can be difficult to identify all possible environmental failure conditions that you should include in your application's testing matrix. Moreover, you might not be able to identify the precise resources to target, such as the Program ID associated with the invalid COM object, the corrupt registry string, or the missing file name. As an alternative, you can let Fault Simulator handle this task for you. You simply use the application normally as Fault Simulator does the rest.

Watching Your Target Application for Potential Environmental Weaknesses

The Fault Simulator standalone application watches your target application during program execution and records program activities and resource strings it encounters. Program activities might include those you initiate, such as opening a file or accessing a directory, as well as those executed by the program itself, such as the program accessing the Registry or creating a COM object. Resource strings identify the objects that are affected, such as file name, registry value, COM object, etc.

Fault Simulator can learn and report where your application might exhibit potential environmental weaknesses. After the application exits or if you end the observation phase, Fault Simulator will list activities related to registry, Disk I/O, COM, and network access that it watched. Each activity is associated with an environmental fault. For example, opening a file could fail as:

- Insufficient read-file privileges
- Insufficient read-directory privileges
- Missing file
- Missing directory
- File locked
- Corrupt file

Even though Fault Simulator will assign one environmental fault to each activity, additional environmental faults might also apply. For example, Fault Simulator will assign the *Insufficient write-directory privileges* fault to the *opening a file* activity. You can optionally choose a different environmental fault from the list of environmental faults for each activity. You can include or exclude environmental faults to be applied to the simulation and then start a fault simulation session. The next section walks you through the steps to use this feature.

Walk Through to Generate Environmental Faults

Read on to learn how to have Fault Simulator watch your target and generate environmental faults, and then perform the fault simulation session using the recorded data.

Watching Your Target

1 From the **DevPartner Fault Simulator** window in the standalone application, click **Browse** and select the target application to watch.

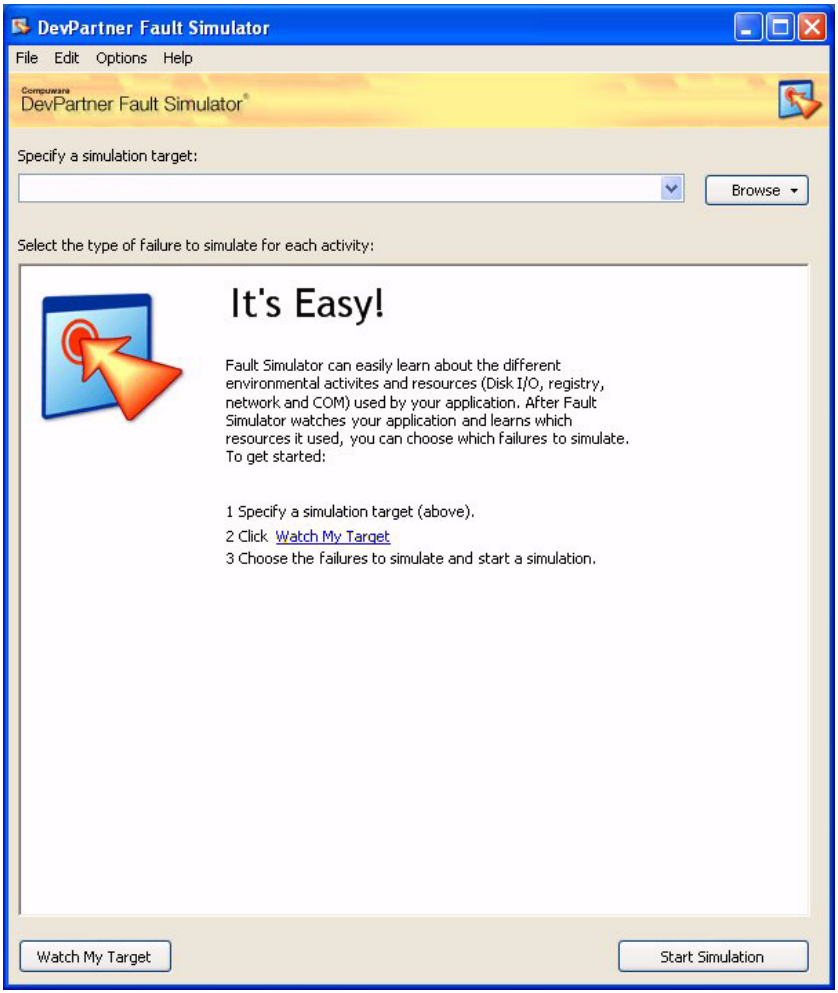


Figure 4-1. Watch Your Target

2 Click **Watch My Target** to begin the session.

The window displays the current status of the session and prompts you to launch the program.

3 Launch your target application and use it normally.

You can click **Suspend** to pause the recording of program activity. For example, you might suspend some portion of the session if you wanted Fault Simulator to observe a specific execution point in the application but bypass other parts of the application.

4 Stop the session either by exiting the application or clicking **End Observation**.

Note: If Fault Simulator is watching a Web application, closing the Web browser alone does not stop the observation. You must end the session in order to completely stop the process.

5 Click a category name (such as *Disk I/O*), to expand its list of activities, and then review how each activity relates to registry, Disk I/O, COM, and network fault categories.

Each activity will be associated with an environmental fault, and will be grouped under a category name.

Tip: Notice that activities with multiple faults will be hyperlinked.

6 Click on an item (to the right) to choose the fault to simulate.

There could be one or more faults in the list, as shown in Figure 4-2 on page 32. For example, the activity, `Deleting file3.txt`, can be simulated to fail as follows:

- Insufficient write-directory privileges
- Missing file
- Missing directory
- File locked

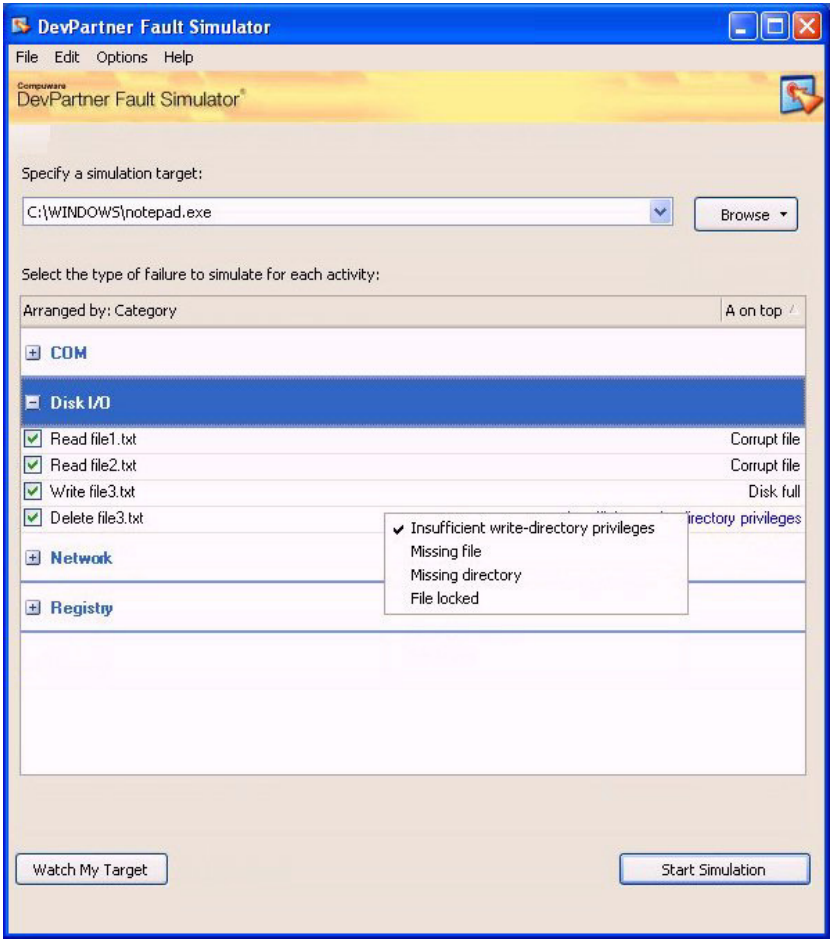


Figure 4-2. Choosing a Fault to Simulate for the Activity — **`Deleting file3.txt`**

7 Repeat step 5 through step 6 as often as necessary.

8 Optionally save the faults that Fault Simulator generated for you in a uniquely named fault set file (**`.dpfsfault`**).

This action allows you to load these faults and reuse and/or reconfigure them in a subsequent fault simulation. See "Can I Reuse Fault Sets?" on page 22.

Performing a Fault Simulation with the Collected Data

1. Click **Start Simulation** to start a fault simulation on the subset of environmental faults that you selected.
2. Launch the target application.

   The fault simulation proceeds until you either stop the session or exit the monitored program. For Web applications, you must explicitly stop the simulation, rather than just close the browser.
3. When appropriate, stop the session, either by exiting the application or stopping the simulation.
4. Review the results of the fault simulation.

   In the example below, the **Specified Faults** pane shows that various file-related fault conditions were simulated during monitoring.

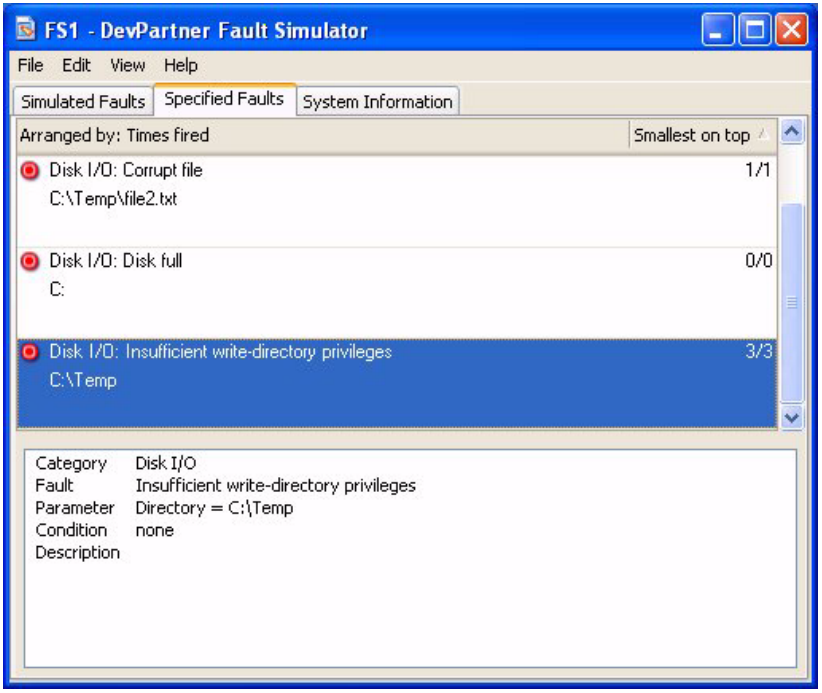


Figure 4-3. Results on the **Specified Faults** Pane

In Figure 4-4 on page 34, the **Simulated Faults** pane traced the path your unmanaged code took when a function failed. The top-most entry in the **Call Stack** view will always show the location where Fault Simulator caused the function to fail.

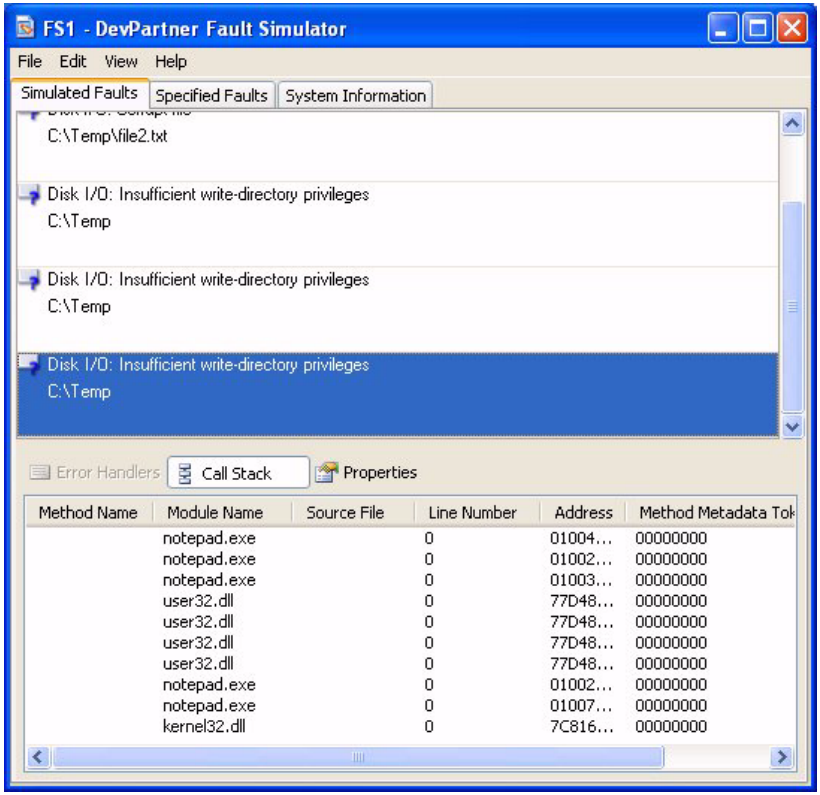


Figure 4-4. Call Stack Results on the **Simulated Faults** Pane

Optional Actions

1. Save the results to a uniquely named results file (`.dpfs`).

   This action lets you review the results at a later time or share the results with others.

2. Choose **Switch to Fault Editor** from the **Edit** menu (optional).

   This option allows you to make changes to some or all of the environmental faults collected in the previous fault set as well as to add new fault descriptors. It also allows you to use the current fault set in a batch script that Fault Simulator can automatically generate for subsequent automated testing.

Notice that Fault Simulator automatically selects an environmental fault for each activity (some in an unselected state) which is then preserved in the current fault set. However, you can enable other faults in this fault set to another fault simulation. Read on to learn how to configure fault settings yourself that you can use in a subsequent fault simulation.

Manually Configuring a Fault Simulation

Previously, you saw how the Fault Simulator standalone application watched your target application as you used it normally and generated environmental faults that you could use in a subsequent fault simulation. You can also configure a fault simulation yourself, either from scratch or by loading and modifying a previously saved fault set.

Configuring Fault Settings to Manage Your Environmental Testing

Using Fault Simulator, you can create and configure environmental faults yourself that focus on particular failure conditions in a running application, without changing the operating environment or disrupting the application. Fault Simulator guides you as you create fault descriptors to test specific areas in the program execution. See "How Do I Use Fault Descriptors to Simulate Fault Conditions?" on page 14.

The following section walks you through the steps to configure fault settings yourself.

Walk Through to Set Up a Fault Simulation

The first section gets you started to create new fault descriptors and/or to modify existing fault properties. The following section guides you to perform the fault simulation session using the fault descriptors you either created or modified.

Note: See "How Do I Use Fault Descriptors to Simulate Fault Conditions?" on page 14 for more information.

Creating and Modifying Fault Descriptors

1 From the Fault Simulator standalone application, choose **Switch to Fault Editor** from the **Edit** menu.

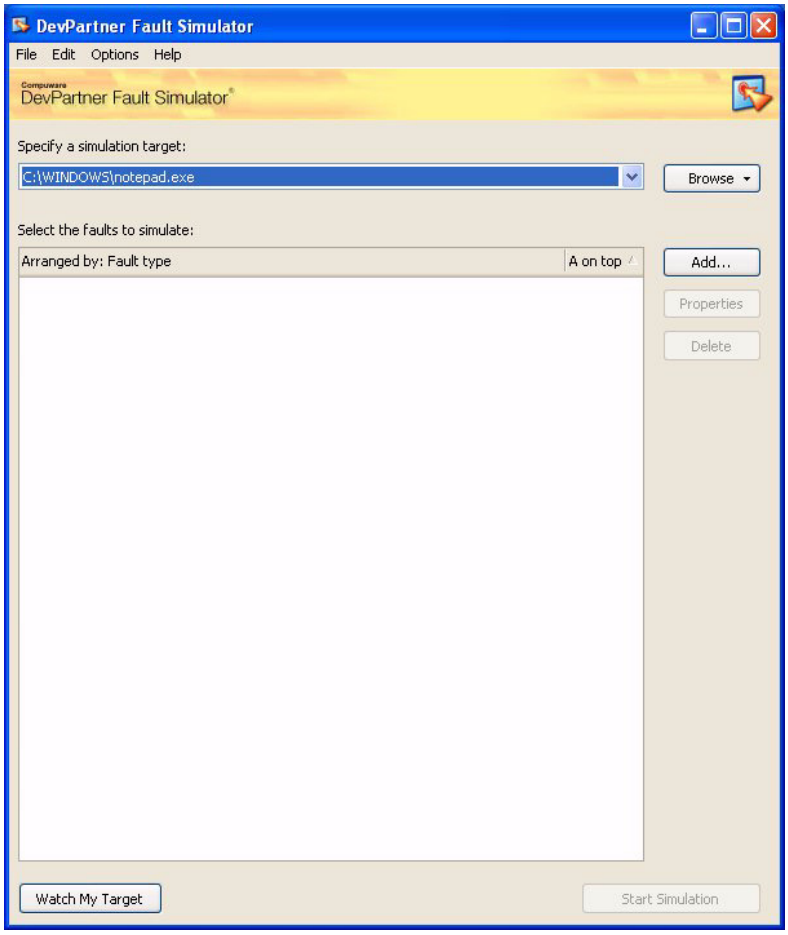


Figure 4-5. **DevPartner Fault Simulator** Window in the Fault Editor View

2 Choose **Load faults** from the **File** menu to load the contents of a previously saved fault set.

This step populates the **DevPartner Fault Simulator** window with a pre-existing collection of fault descriptors that you can use, as is, or modify.

3 Choose the next action:

- **a** To use the faults in the previous step, as is, skip to step 1 on page 39.
- **b** To make changes or additions to the currently loaded fault set, proceed to the next step.

4 Browse for the target application (executable, COM+ component, or Web application) where you want to conduct the simulation.

5 Check any faults in the expanded list that you want included in the fault simulation.

To rearrange the checked items to the top of the list, select **Arranged by** along the column header and then choose **Checked** from the list.

6 Double-click a fault descriptor in the list to modify its fault properties. In this example:

a We changed the designated file parameter to: `C:\Temp\file1.txt`.

b We added an optional fault description: **Testing temp file 1**.

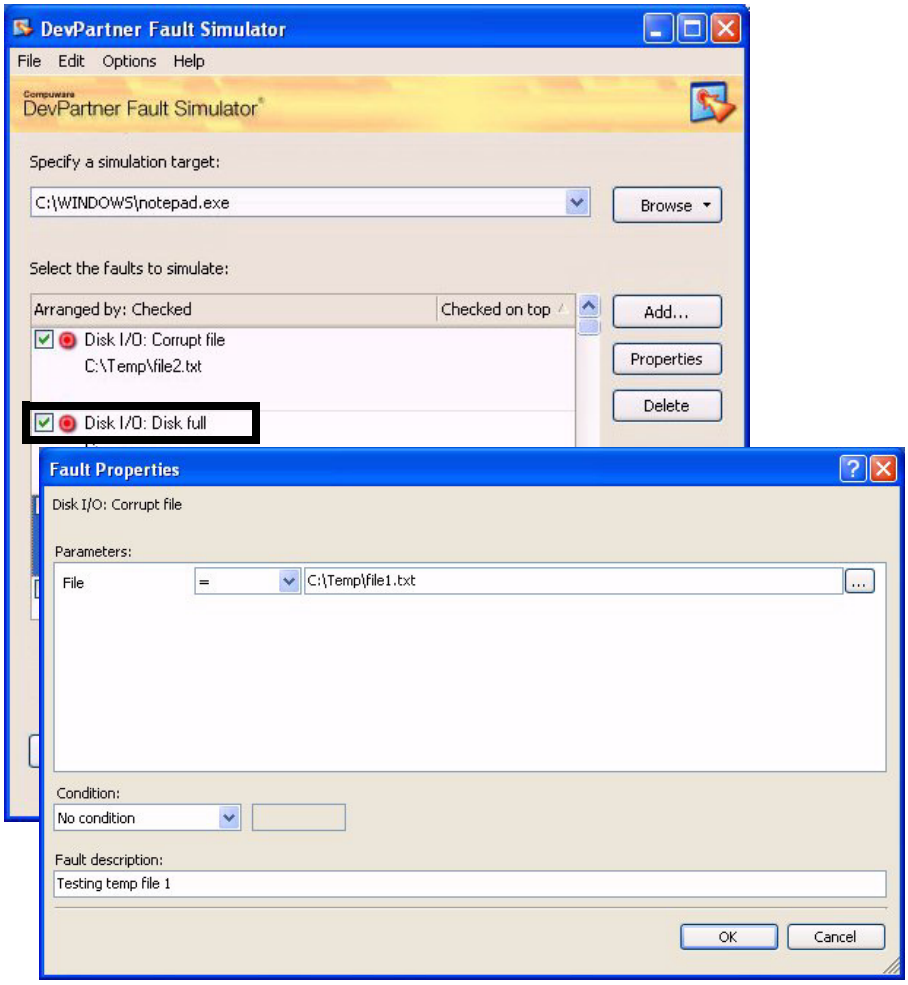


Figure 4-6. Modifying Fault Properties

7 Create a new environmental fault — such as, a memory-related fault, shown in Figure 4-7.

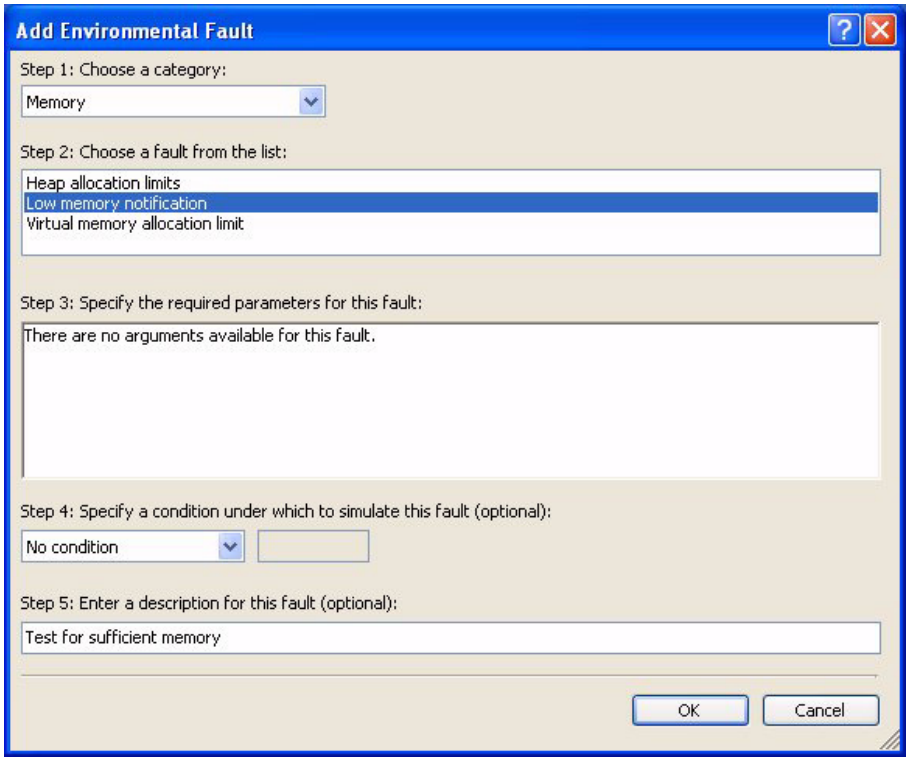


Figure 4-7. Adding and Configuring a Memory Environmental Fault

8 Review the remainder of the fault descriptors and decide if you want to include any of them in the next fault simulation.

9 Optionally click **Arranged by** to customize the fault display.

For example, as shown in Figure 4-8 on page 39, you might choose **Checked** and **Checked on top** to only see selected faults.

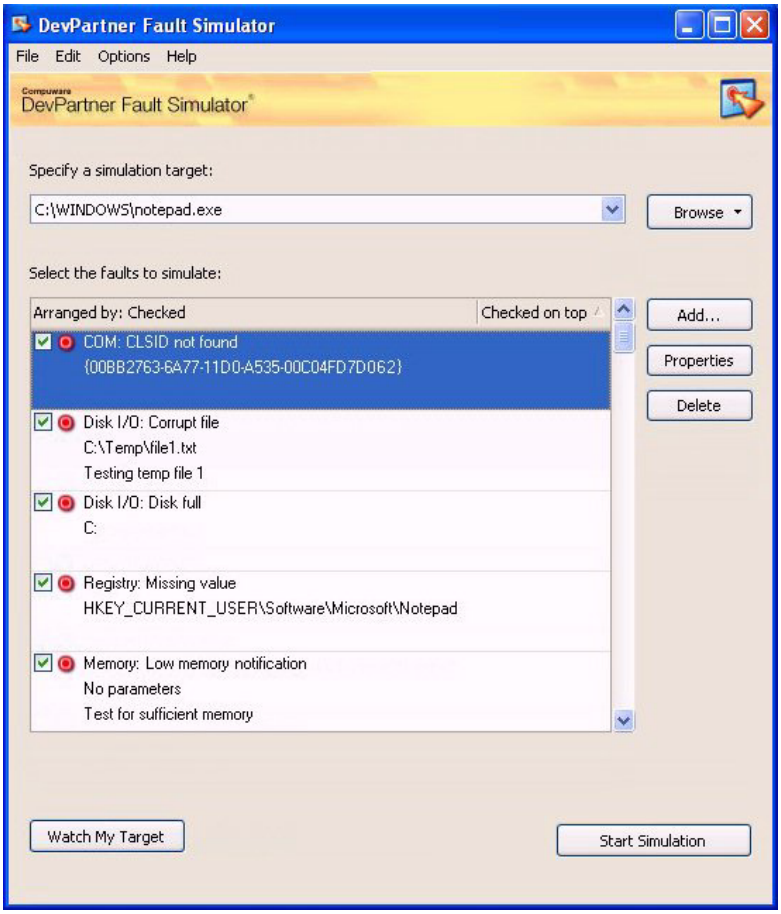


Figure 4-8. Selecting Fault Descriptors for a Fault Simulation

10 Optionally save the fault set to a uniquely named fault set file to retain this configuration (similar to step 8 on page 32).

Performing a Fault Simulation

Tip: You must have configured at least one fault descriptor to proceed.

1 Click **Start Simulation**.

2 Launch the target application.

3 Stop the session, when desired, either by exiting the application or stopping the simulation.

4 Review the simulation results.

In Figure 4-9, notice that Fault Simulator simulated the following fault conditions:

- The designated COM object could not be found.
- `C:\Temp\file1.txt` was corrupted.
- The `C:` drive was full.
- A registry key value was missing.

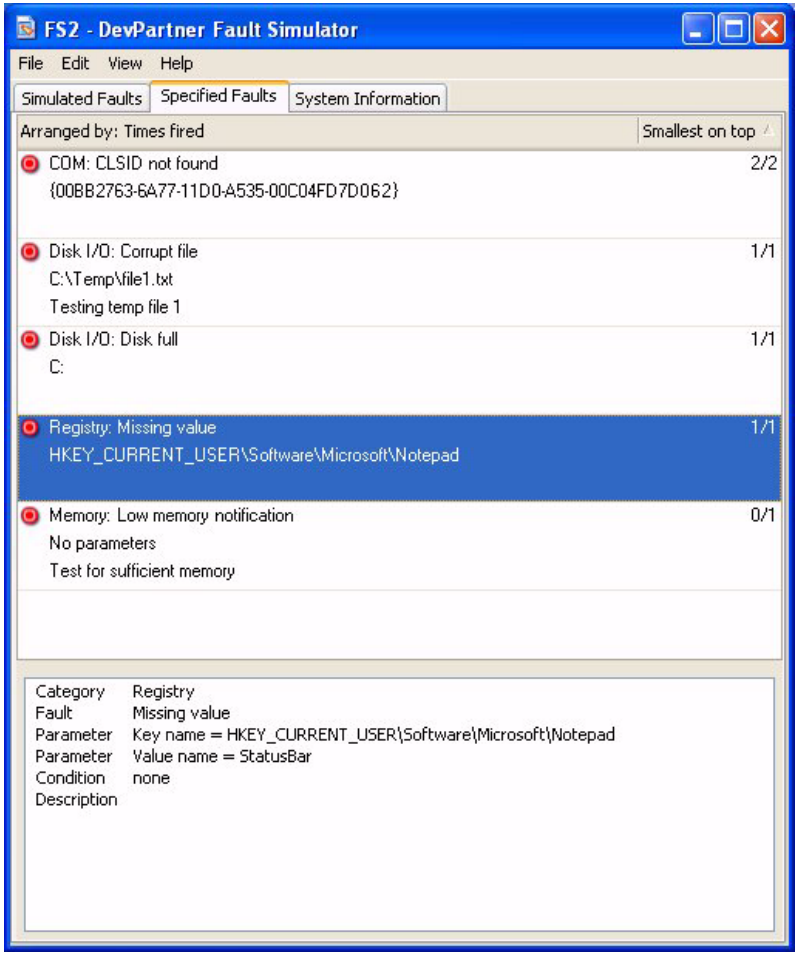


Figure 4-9. Results in the **Simulated Faults** Pane

You can review the results generated during this session and also save the results to a uniquely named results file (`.dpfs`) for future reference. See "Evaluating Error Handler Results" on page 84 for an explanation of fault simulation results.

Automatically Generating a Batch Script

DevPartner Fault Simulator extends its fault simulation capabilities to the command line, allowing you to automate fault simulations.

Tip: See Appendix B, "Command Line Quick Reference" on page 117 for an overview of the Fault Simulator commands.

You might find it difficult to build a batch script that incorporates the right command line switches in Fault Simulator. The Fault Simulator standalone application eliminates the guesswork and the potential mistakes by generating the script for you. Mirroring your preferences, it will create a script file that you can use as part of your automated tests. The following section walks you through the steps to generate a batch script.

Walk Through to Create a Batch Script

This section guides you to make a few required choices, and the following section explains optional settings you can also make.

Configuring Simulation Target Information

Tip: In order to use this feature, the standalone application must be in the Fault Editor view and a fault set must be loaded.

1. Choose **Switch to Fault Editor** from the **Edit** menu.

   Ensure that you also have loaded a fault set. See "Can I Reuse Fault Sets?" on page 22.

2. Choose **Generate Batch Script** from the **File** menu in the standalone application.

   The **Simulation Target Information** pane includes settings in the current fault set and your target application. If you want to make changes to the default entries, follow the remaining steps. If not, skip to step 4 on page 43.

3. Specify the application, component, or Web application or service to be monitored (if different from the default preference).

4. Specify the location of the fault set file (`.dpfsfault`).

5. Choose whether to save the session results (`.dpfs`), and if so, specify where to save it.

6. Specify where to save the generated batch script file (`.bat`).

7. Review the settings you have made so far in this window.

8. If finished, skip to step 4 on page 43.

   If not, click **Additional Options**.

Setting Additional Options

1 Choose whether to launch a separate executable.

If so, you can specify these optional selections:

a Browse for the executable file (`/l`).

b Specify a valid working directory for the process pertaining to the previous entry (`/s`).

c Identify any command line arguments for the launched process (`/g`).

2 Choose whether to include coverage analysis in the automation (`/v`).

__Tip:__ See "Fault Simulator Commands" on page 118 for an overview of command line options.

3 Decide how to simulate faults. Choose one of the following options:

- ◇ Simulate all the faults in the designated fault set file (default)
- ◇ Perform an individual simulation of each fault, in the order in which they are positioned in the file (`/e`)
- ◇ Simulate one specific fault (`/n:x`)

  You will be prompted to choose a fault from the list.

Notice in Figure 4-10 on page 43 that the script designated the switch,
`/n:23`, to correspond with your selection of the 23rd fault, *Network: Connection timed out*.

Note: Fault Simulator previews the batch script for you as it is generated. Although you cannot alter it, you can copy its contents to the clipboard.

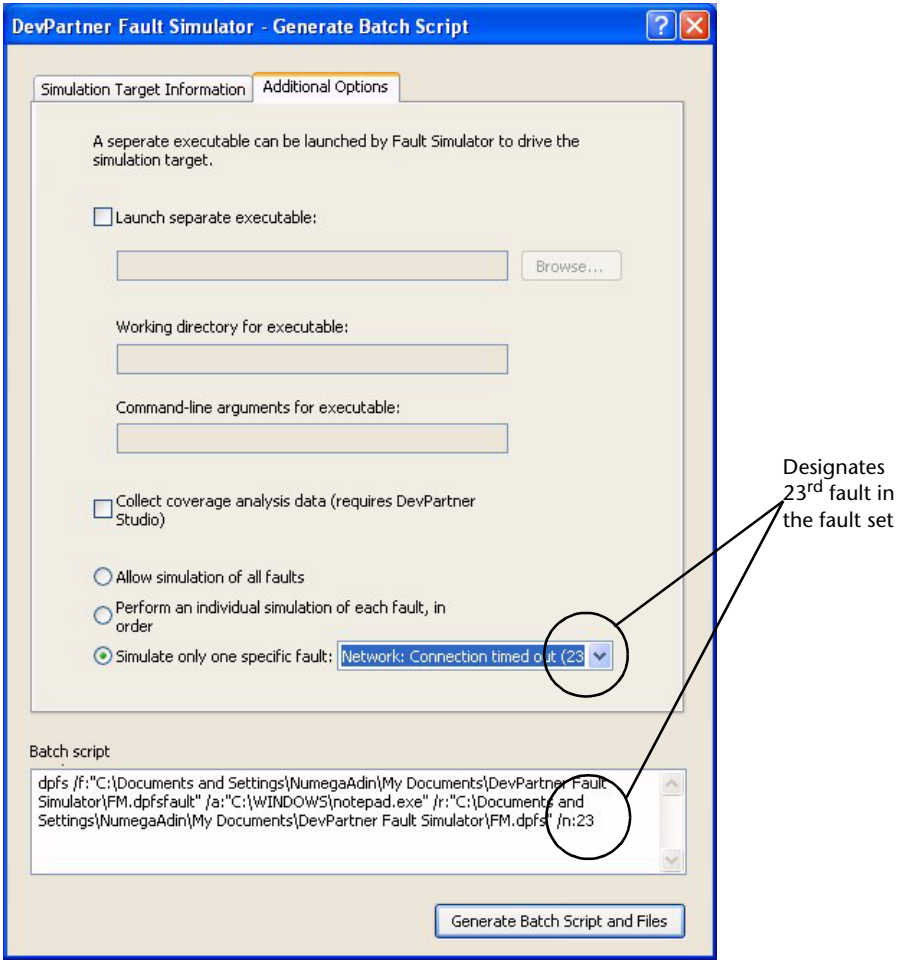


Figure 4-10. **Simulation Target Information** Pane

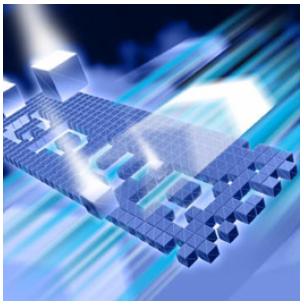
4 When finished, click **Generate Batch Script and Files** to instruct Fault Simulator to build the script.

You can run the batch script from the command line.

5 To view results generated from this script file, launch the standalone application and open the results file you designated in step 5 on page 41.

Chapter 5

Enhancing Quality Assurance Testing



- Traditional Software Testing Methodologies
- When Traditional Software Testing Is Not Enough
- Fault Simulator Enhances Software Quality Through Fault Simulation
- User Scenario — Testing Software Quality with Fault Simulator

This chapter considers various testing methodologies used by quality assurance engineers and explores the challenges of effective testing. It then presents a user scenario that shows how Fault Simulator enhances quality assurance testing objectives using fault simulation.

Traditional Software Testing Methodologies

As a quality assurance engineer, you perform various types of software testing, as summarized in Table 5-1.

Table 5-1. Common Software Testing

Testing	Purpose
Integration	Verifies that different areas of the application work properly together
Functional	Executes certain user functions in the application and checks for the expected result
Regression	Selectively retests areas in the software to ensure that fixes made to the application code have not negatively altered previously working functionality

You also perform stress testing and load testing. Stress testing involves running an application and then monitoring program behavior under

adverse, atypical 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 assesses an application's tolerance to increased load (such as, data input and 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.

Load testing and stress testing are not synonymous. Stress testing deliberately subjects the application 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.

When Traditional Software Testing Is Not Enough

Despite painstaking efforts to validate software, the product can still be deployed with untested code. Several permutations of operating systems, service packs, system configurations, network layouts, user privilege configurations, and third-party components can wreak havoc on an application's stability. While a user might overlook the occasional disabled menu option or incorrectly displayed dialog box, the Web-based transaction that hangs or the application that crashes could undermine the user's overall impression of the software.

Your quality assurance team might be overextended in its attempts to test for every conceivable failure. It could be impractical to assume that you can test an application's reaction to all system or environmental failure conditions using traditional testing methodologies. Moreover, you could be flooded with a stream of bug reports about software instability or poor performance from the field. Consequently, software developers might resist being pulled away from their development activities to switch back to resolving defects after deployment.

Fault Simulator Enhances Software Quality Through Fault Simulation

Enhancing quality assurance testing with Fault Simulator provides a benchmark for ensuring that your application can reliably handle unanticipated environmental anomalies. With Fault Simulator, you can better assess whether your application is truly ready for deployment.

DevPartner Fault Simulator offers a meaningful solution to age-old testing challenges. Fault Simulator can simulate environmental failures in Windows-based applications, without actually interfering with physical settings or altering file structures or source code. Fault Simulator safely allows the application to demonstrate its response to real-world, and possibly catastrophic environmental conditions. You can either set up the environmental fault criteria yourself, or let the Fault Simulator standalone application watch the program execution, collect data as it is running, and then generate possible environmental failure scenarios that you can subsequently test.

Fault Simulator supports your quality assurance objectives with virtually little or no learning curve. All that is required is that you understand and can use the application you are testing. The following section presents a user scenario showing how to test software with Fault Simulator.

User Scenario — Testing Software Quality with Fault Simulator

You have been assigned to test your company's Window-based task tracking application[1], currently under development but close to deployment. The application tracks hours spent working on different projects. It allows you to clock in and out during the workday, and optionally add comments to work time. The application has a reporting capability that totals on a weekly or monthly basis, or by task. You can also view, print, save the reporting details.

You have conducted various standard tests on the application, as summarized in Table 5-1 on page 45. You must test your application's ability to withstand unanticipated environmental failures, but you are unsure how to proceed. Consequently, you use the Fault Simulator standalone application to generate a collection of environmental fault conditions that you can use in a fault simulation to test for application weaknesses prior to product release.

1. Source: *Chris Oldwood's Home Page*; URL: http://www.cix.co.uk/~gort/default.htm

Scoping Out Areas to Test

You scope out areas where you want to focus your testing. You consider four key environmental categories: Disk I/O, registry, COM, and network.

Disk I/O

The application depends on file-related tasks that either your end user will perform or that the application performs internally. An inability to perform even simple file-related tasks could frustrate the user experience at minimum or cripple the application at worst. You want to test your application's ability to withstand these kinds of file-related problems:

- How will the application react to a file that becomes corrupt?
- Will the application assume that the end user has write privileges to a file?
- How will the application react if the local disk is full?
- Will the application display any kind of message to the user if it has any of these issues with a file?

Note: See "Disk I/O Environmental Faults" on page 15 for more information.

Registry

The application uses settings in the Windows registry. These registry settings could get altered during normal operation, such as at program launch, when internal drivers are loaded, or system login. Problems with registry settings could result in application crashes or other environmental failure conditions. You want to be able to test for registry issues without physically changing registry settings, so as not to destabilize the application.

Note: See "Registry Environmental Faults" on page 17 for more information.

COM

This Windows-based application uses the Microsoft COM technology. You admit that your application could succumb to COM-related problems, resulting from the absence of third-party components or changes to existing COM components. You want to test COM issues, but you need assistance from Fault Simulator to pinpoint the actual COM objects to target.

Note: See "COM Environmental Faults" on page 17 for more information.

Network

Network-related vulnerabilities, resulting from problems with network access or resources, pose a serious problem you want to understand better before releasing the product. However, to test for network failures, in the past you would have removed a network interface card, unplugged a network cable, or disconnected a network server, putting the application, as well as any other applications or machines that use the same network, at greater risk. You hope to safely test for these scenarios:

- Can your application access the network?
- Does your application communicate anything to the end user if it cannot access the remote server?

Note: See "Network Environmental Faults" on page 16 for more information.

Having Fault Simulator Watch Your Target and Record Program Activities

You run your application, letting Fault Simulator watch the target and create environmental faults based on its analysis of normal program activities.

Note: Learn more about this functionality at "Automatically Generating Environmental Faults" on page 29.

You launch the standalone application and browse to the location of the target application. You then instruct Fault Simulator to watch your target application. After you launch the target application, you perform these normal program activities:

- Adding new tasks to the current session
- Editing or deleting existing task items
- Importing or exporting data files (.csv)
- Performing reporting functions, including viewing and saving to disk
- Accessing a URL from the program

Evaluating the Collection of Environmental Faults

After you exit from the program, Fault Simulator generates a set of environmental faults. You save the complete fault set to a uniquely named file (`TT0.dpfsfault`) that you can use later. You decide to first focus on disk I/O-related fault conditions. Therefore, you uncheck all other environmental faults in the COM, registry, and network categories, and only select the disk I/O faults, as shown in Figure 5-1 on page 50.

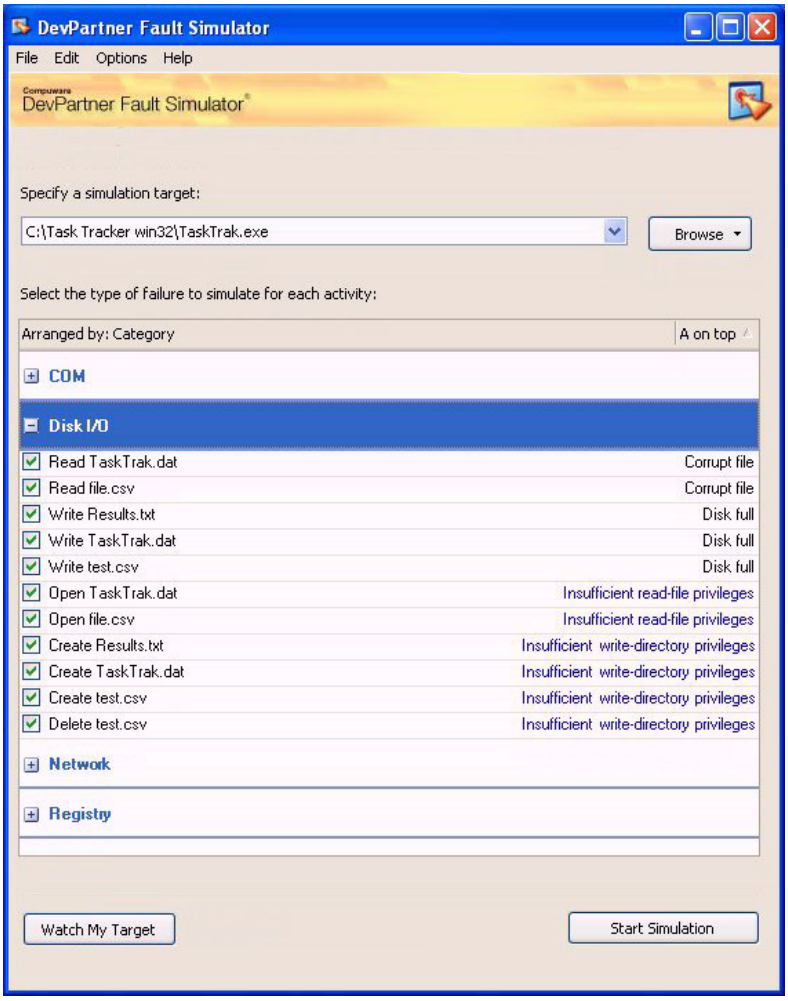


Figure 5-1. Enabling Disk I/O Faults

You review the list that Fault Simulator generated. To the left, you see program activities that either you performed or that the application performed internally. To the right, you see how Fault Simulator converts the program activity to an environmental fault. For example, attempting to write to the results file, **`Results.txt`**, could cause a *Disk full* condition.

You notice that some program activities include a hyperlinked environmental fault. In these instances, Fault Simulator has determined that more than one environmental fault can occur as a result of a particular activity. You can click on the hyperlink to either keep the selection that Fault Simulator made (checked) or to choose another fault from the list. You can only assign one fault per program activity.

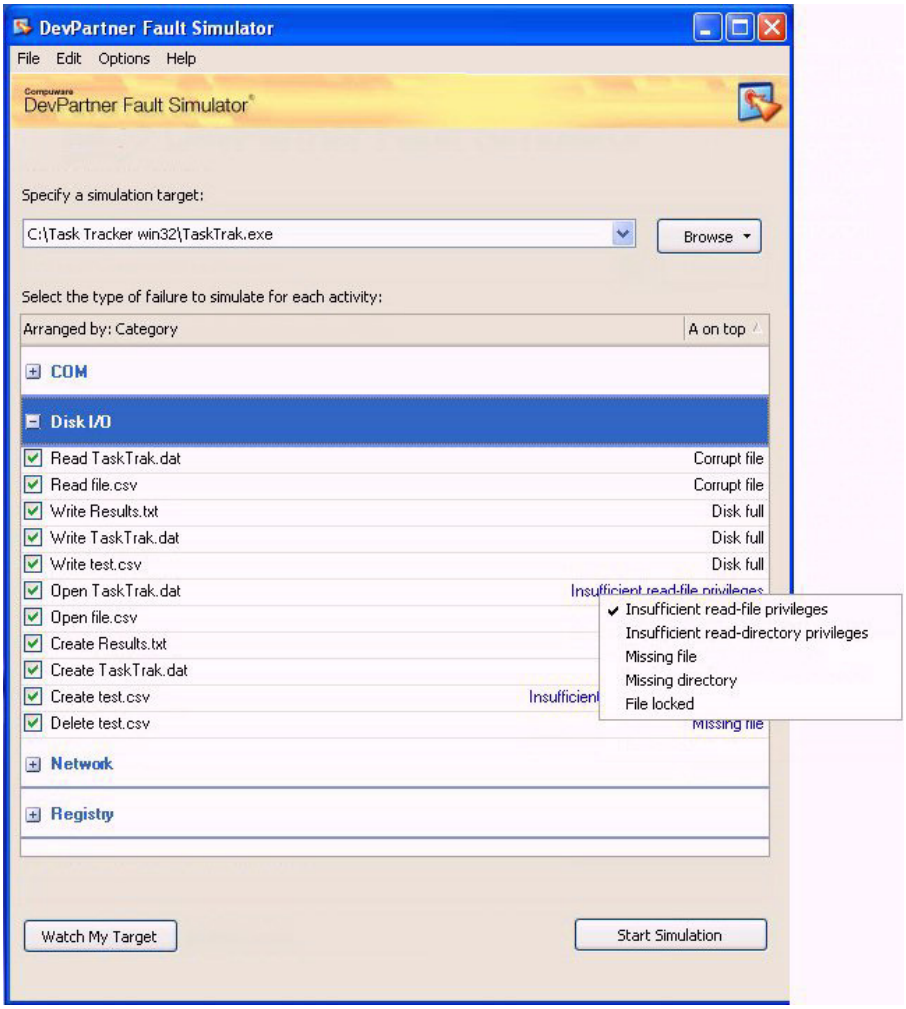


Figure 5-2. Evaluating the List of Environmental Faults

You notice also that for each activity, Fault Simulator includes a resource string (if applicable), such as a file name associated with the activity (such as, **`test.csv`**, or **`TaskTrak.dat`**, shown in Figure 5-2).

After you make other selections on this window, you save this fault set configuration to another uniquely named file, **`TT1.dpfsfault`**. You are now ready to start a fault simulation, in order to see how the application reacts to simulated fault conditions.

You start the fault simulation and again launch your application, when prompted to do so. You repeat the same tasks you performed earlier (in "Having Fault Simulator Watch Your Target and Record Program Activities" on page 49). You infer that Fault Simulator is preventing you from completing these tasks. You also notice that as you do try to proceed, Fault Simulator records the faults that it is simulating in parallel with your actions. You observe that the application subsequently terminates and displays a generic Windows message. At this point, Fault Simulator displays the results it collected thus far. You first look at the **Specified Faults** pane.

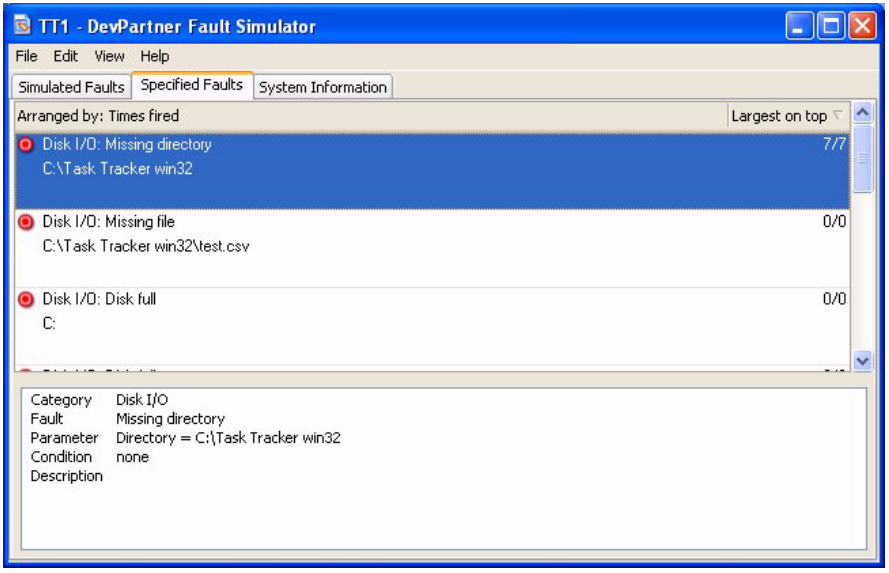


Figure 5-3. Reviewing How Many Times a Fault Was Simulated on **Specified Faults**

Fault Simulator confirms that it attempted and successfully simulated the *Missing directory* fault seven times. However, you also notice that Fault Simulator did not simulate the other faults in the fault set before the application terminated.

Next, you look at the results on the **Simulated Faults** pane, as shown in Figure 5-4 on page 53.

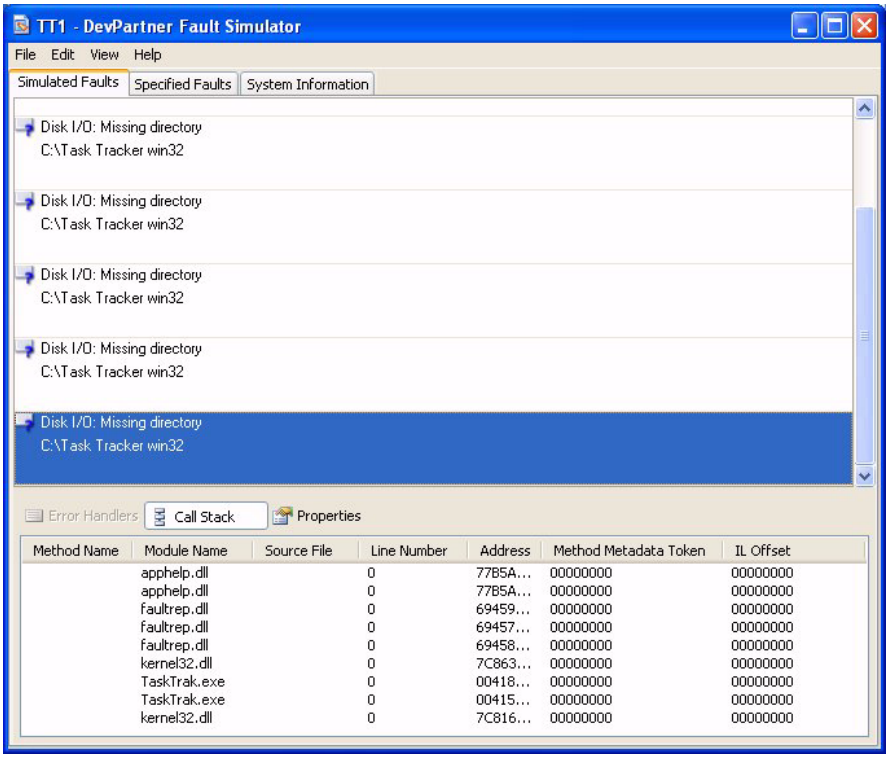


Figure 5-4. Call Stack Results on the **Simulated Faults** Pane

Fault Simulator presents information related to faults it simulated during program execution. For example, in the previous example, Fault Simulator traced the method and DLL invocations in the call stack pertaining to a specific fault, in this case the *Missing directory* fault condition. You save the results (`TT1.results`) to allow software development to also review the results and understand exactly what might have caused the application to terminate abruptly (for example, if it was caused by the operating environment and not by the application itself).

You configure another variation of the disk I/O-related fault set. This time, you uncheck the *Missing directory* fault so that you can simulate other faults in the collection (such as, Corrupt file, Disk full, Insufficient read-file privileges, Insufficient write-directory privileges, and Missing file). You save that fault configuration to another fault set file named `TT1a.dpfsfault` (see the configuration in Figure 5-5 on page 54).

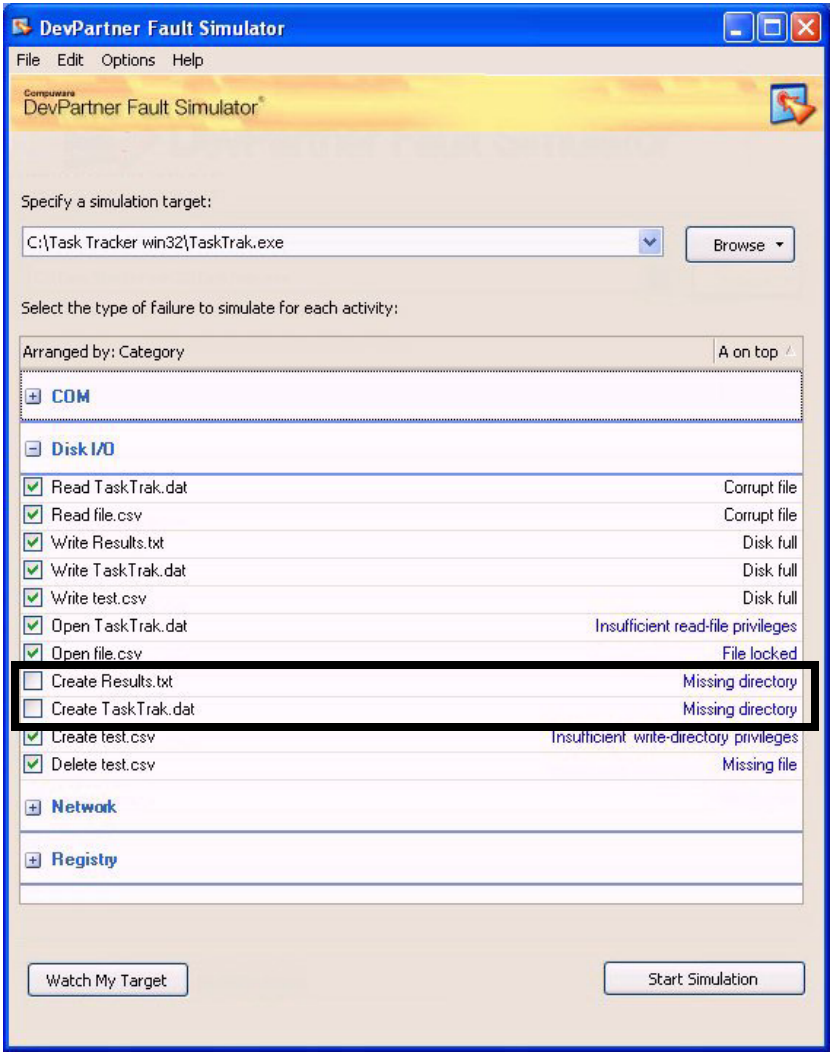


Figure 5-5. Excluding the Missing Directory Faults from the Next Fault Simulation

You run another simulation and perform similar user tasks in the program once again. You quickly discover that you are prevented from completing each task. You also notice that the application conveys immediate feedback upon each attempt. When done, you end the simulation and review the results.

The **Specified Faults** pane confirms successful attempts at simulating several disk I/O faults that parallels the messages you encountered when you interactively used the program. Recall that as you tried to perform file-related tasks, the application dutifully displayed clear messages to inform you that you could not complete each task. Fault Simulator confirms these attempts in its results.

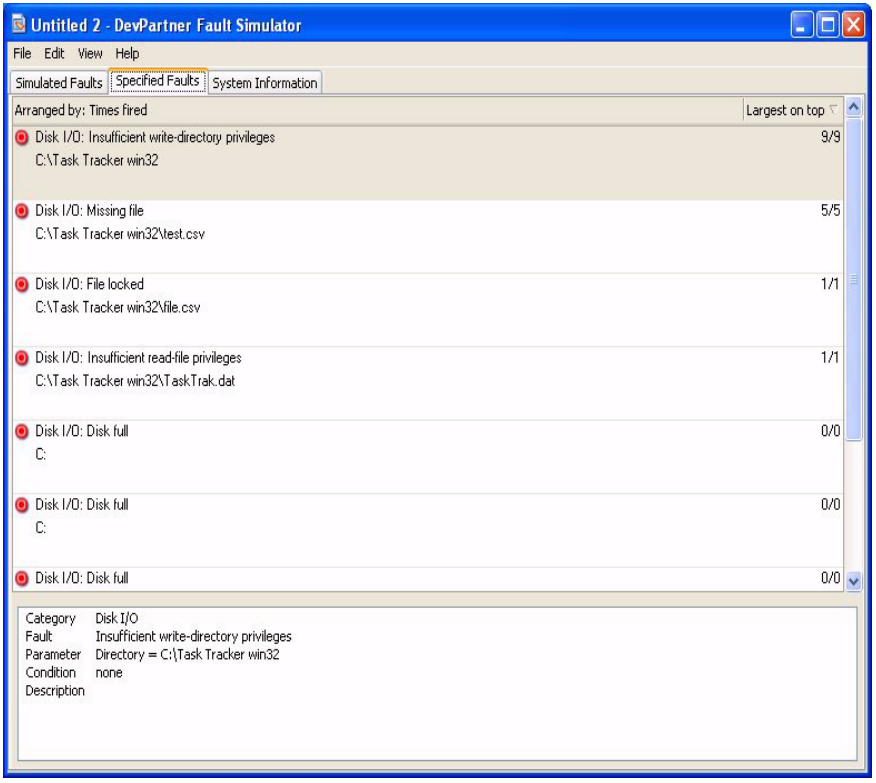


Figure 5-6. **Specified Faults** Pane Showing a Successful Simulation of Disk I/O Faults

You might question why Fault Simulator simulated a fault multiple times. This might have occurred because the API was called several times which then triggered that fault each time it was encountered. In addition, Fault Simulator simulated the fault each time it encountered a particular parameter, such as a specified file name, that you originally configured. As a quality assurance engineer, you wonder why the application has responded in this way, as evidenced by the results Fault Simulator captured. You save the results (`TT1a.dpfs`) and direct them along with the original fault set file to the responsible software developer to verify the original intent of the application code.

Simulating COM, Registry, and Network-Related Environmental Faults

Next, you enable environmental faults under the other categories of COM, Registry, and Network. You concede that these types of environmental fault conditions are more difficult and far riskier to set up manually because doing so requires that you must physically alter critical settings in order to generate the fault. Thus, you clearly see the benefit of simulating these faults, as shown in Figure 5-7.

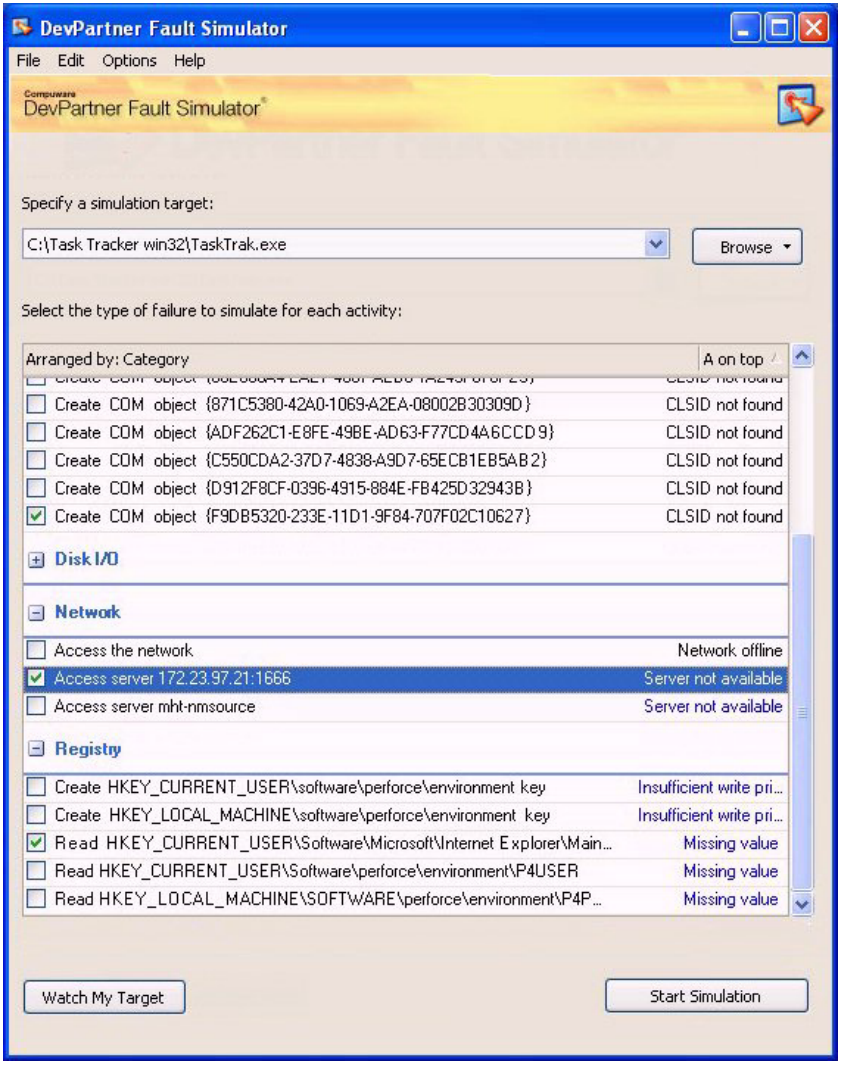


Figure 5-7. Setting Up a Fault Simulation with COM, Network and Registry Faults

You run a fault simulation and use the application normally once again. Following this session, you look at the results, starting with the **Specified Faults** pane, as shown in Figure 5-8 on page 57.

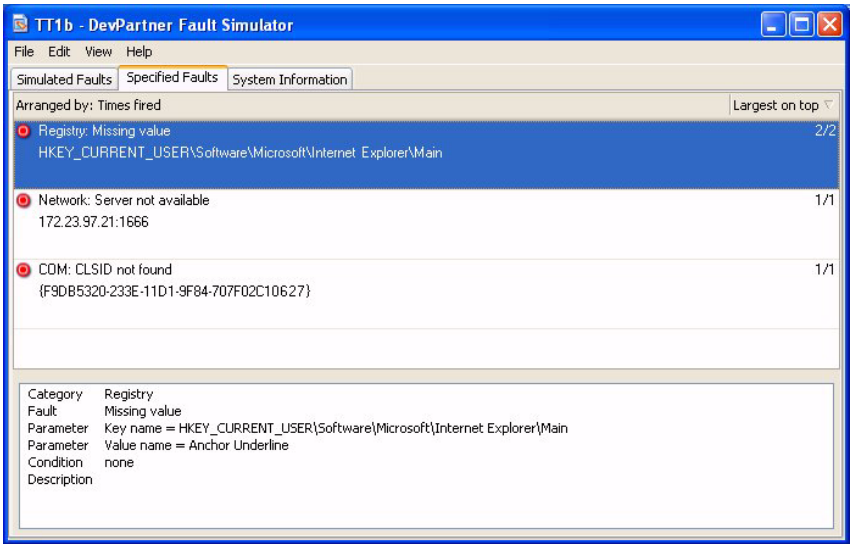


Figure 5-8. Simulated Registry, Network, and COM Environment Faults

The results show that twice Fault Simulator could successfully simulate a missing registry key value that the application uses. It also simulated a condition where a specific COM object could not be found, and that a specific server IP address was unavailable. You think back during the actual session when you were using the application, and you do not recall getting any alerts related to those environmental events. While you are not certain that this is problematic, you save these results as well to a file named **`TT1b.dpfs`**, so that you can forward it, along with the original fault set file, to the software development team for further scrutiny.

Performing Repeatable Testing

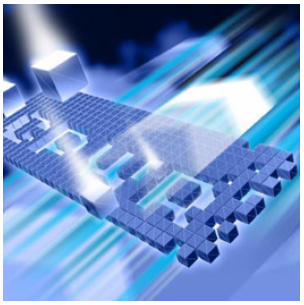
You can reuse each fault set file you created in the standalone application. You can also share these fault sets and the corresponding results files with software development to give them direct feedback on improving their underlying code. Similarly you can execute fault simulations from the command line.

Note: See Appendix B, "Command Line Quick Reference" for an overview of the command line interface. Refer to the DevPartner Fault Simulator Command Line online help from the InfoCenter for detailed usage information.

Moreover, if you find it difficult to construct a valid script from scratch, you can instead use the standalone application to build the script file for you. Learn more about this capability in "Automatically Generating a Batch Script" on page 41.

Chapter 6

Setting Up Exception Handler Tests in Visual Studio



- Testing Exception Handlers in Fault Simulator
- Using Fault Simulator in Visual Studio
- Walk Through Focusing on Exception Handlers in Your Code

This chapter acquaints you with Fault Simulator in Visual Studio. This chapter also introduces you to new functionality that helps you to improve your exception handling code.

Testing Exception Handlers in Fault Simulator

You can use Fault Simulator in Visual Studio to test and debug exception handlers in managed code. Fault Simulator simulates faults without disrupting the debugger, operating system, or Visual Studio. Fault Simulator analyzes 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 exception handling anomalies.

Fault Simulator can also simulate environmental faults in unmanaged applications. Fault Simulator helps you to test how the application reacts to a variety of environmental failure conditions.

Using Fault Simulator in Visual Studio

This quick start shows you how to use Fault Simulator 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.

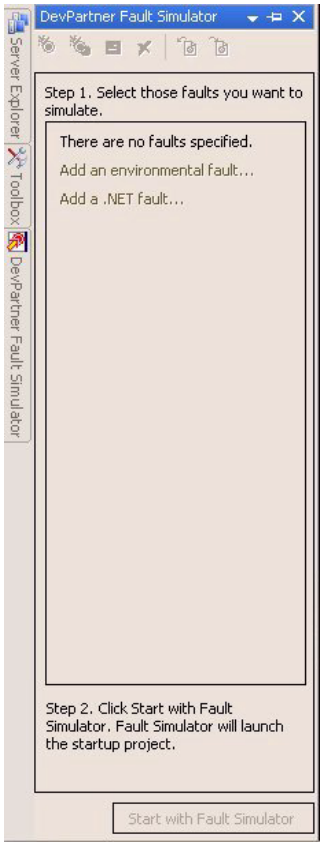


Figure 6-1. **DevPartner Fault Simulator** Window in Visual Studio

2 Ensure that you have designated a valid startup project in your solution.

Check that this project is a supported project type and that it meets the requirements for fault simulation. Refer to the DevPartner Fault Simulator online help in Visual Studio for more information on project requirements and supported project types.

3 Review the current fault descriptor list, and select the check box of any fault descriptors you want activated during the next fault simulation.

Note: If this is your first time configuring a fault simulation, the window will be empty (as pictured in Figure 6-1 on page 60) and you will need to add and configure at least one fault descriptor.

4 Add and configure a .NET fault on the **Add .NET Fault** dialog box, as shown in Figure 6-2.

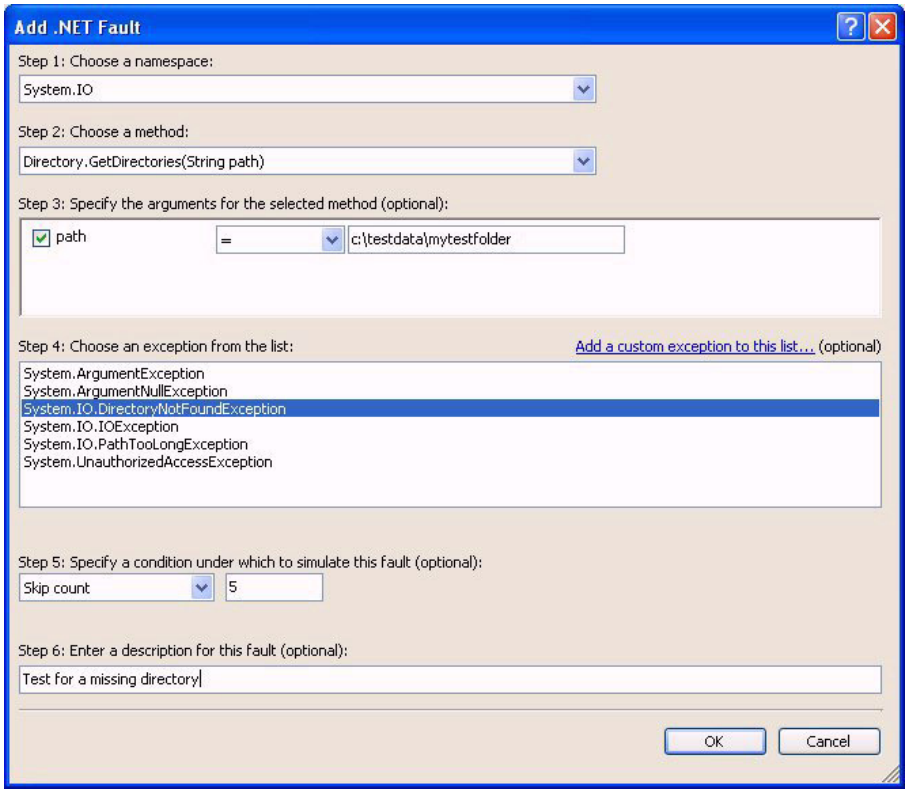


Figure 6-2. **Add .NET Fault** Dialog Box

5 Configure as follows:

- **a** Choose the namespace and method from each list provided.
- **b** If prompted, specify argument(s) for those selections.
- **c** Choose the exception to be artificially thrown.
- **d** Optionally designate skip count or delay time.
- **e** Optionally enter a fault description.

6 Add and configure an environmental fault on the **Add Environmental Fault** dialog box.

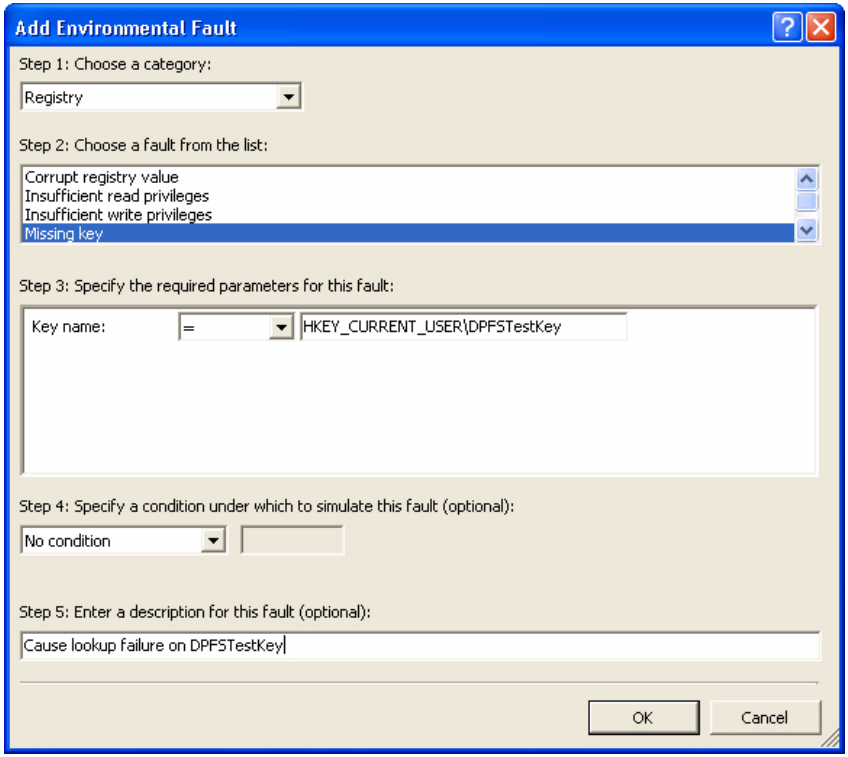


Figure 6-3. **Add Environmental Fault** Dialog Box

7 Configure as follows:

- **a** Choose the environmental fault category.

  Choices include: COM, Disk I/O, Network, Registry, and Memory.
- **b** Choose an environmental fault to simulate.
- **c** If prompted, specify parameter(s) for the previous selection.
- **d** Optionally designate skip count or delay time.
- **e** Optionally enter a fault description.

8 Repeat step 3 on page 60 through step 7 as needed.

9 Click **Start with Fault Simulator** on the **DevPartner Fault Simulator** window.

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 on page 60.

This action also launches the startup project targeted for fault simulation.

10 View current fault simulation details as they appear on the **DevPartner Fault Simulator** window.

11 Click **End Simulation** to stop the current fault simulation.

Final results will immediately appear in a separate window.

12 View the details in the results window.

Refer to Chapter 7, "Evaluating Error Handlers" for various examples of fault simulation results.

Walk Through Focusing on Exception Handlers in Your Code

Fault Simulator can analyze your managed code and identify areas where you can improve your exception handling code. It then suggests actions you can take to make improvements within the current method scope — such as where you can add missing try/catch blocks or add XML `<exception>` tags to your managed code. The following sections explain this functionality.

Showing Fault and Exception Handler Indicators in the Source Window

Following a clean build (and with **Show Fault and Exception Handler Indicators** selected), Fault Simulator can analyze your managed code and identify areas where you can improve your exception handling code. It then suggests actions you can take to make improvements within the current method scope.

Fault Simulator enhances Visual Studio analysis capabilities by scanning your code constructs for exception code or XML `<exception>` tags. To use this functionality, save any changes to the source code, and, if necessary, rebuild the solution before proceeding.

Note: The target executable must have a clean build for source lines to be properly identified. Icons will only appear in build configurations that generate debug information.

Fault Simulator identifies eligible statements. An icon () will appear in the source window where actions can be performed. See Figure 6-4 on page 64.

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

        Class1 c = new Class1();
        c.Foo();

        //c.RegistryAccess();
    }
    catch(ArgumentOutOfRangeException exc)
    {
    }
    catch(IOException exc)
    {
    }
}

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

Figure 6-4. Shows Fault and Exception Handler Indicators in the Source Window

Click on one of the icons for a list of further actions you can take at that location, including:

- Inserting appropriate exception code
- Adding XML `<exception>` tags
- Adding a .NET fault to test the exception handling at that location

Inserting Appropriate Exception Code

Fault Simulator analyzes your managed code in Visual Studio and identifies where you can improve the exception handling code within the current method scope. Fault Simulator enhances Visual Studio analysis capabilities by focusing on your exception handling code constructs and spotting unhandled exceptions within the current method scope. Fault Simulator identifies exceptions that could propagate from a try block, from a catch block that results from the exception handling code, as well as from a finally block. Similarly, Fault Simulator examines code statements containing supported methods and will identify exceptions that could propagate from that method.

When Fault Simulator detects exceptions that can propagate from a code statement and the statement is contained within a try block that has one or more associated catch blocks, Fault Simulator will insert a stub catch block to that try block. If the statement is nested within multiple try blocks, Fault Simulator will insert the stub catch block to the innermost try block that contains one or more catch blocks. Otherwise, Fault Simulator will insert a try block around the entire method source body with the stub catch blocks.

To understand the scope of advice that Fault Simulator will provide, consider the difference between using a try/finally or a try/catch. A try/finally construct is used to ensure that one or more statements are executed after the code in the try block is executed (usually to dispose of objects). Conversely, a try/catch construct is used to catch exceptions that might escape from the try block. Fault Simulator might advise you to add a new try/catch around a try/finally to preserve the original intent of the code.

Figure 6-5 shows your selection to add a surrounding try/catch block to a specific source location.

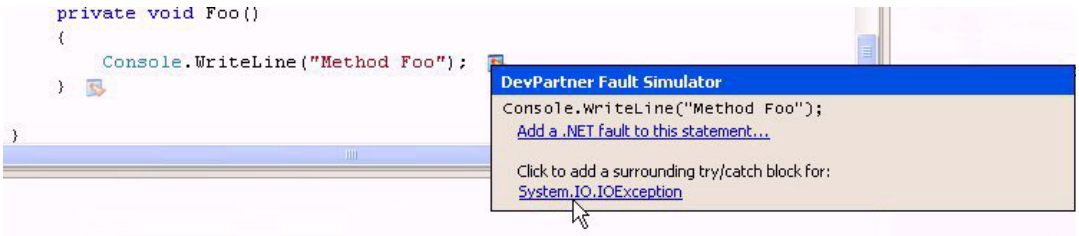


Figure 6-5. Advice to Add a Try/Catch Block

Fault Simulator inserts a try block and stub catch blocks into the source code, as shown in Figure 6-6.

```
private void Foo()
{
    try
    {
        Console.WriteLine("Method Foo");
    }
    catch (System.IO.IOException)
    {
        throw; // TODO: Add error handling here
    }
}
```

Figure 6-6. Shows Where to Insert the Appropriate Try/Catch Code

Note: By default, the stub catch block will throw the exception that is caught.

You might replace that code block as shown in Figure 6-7:

```
private void Foo()
{
    try
    {
        Console.WriteLine("Method Foo");
    }
    catch (System.IO.IOException) EX)
    {
        Console.WriteLine(EX.TOString());
    }
}
```

Figure 6-7. Revised Code Block

Note: If you have one or more assemblies that define the exception type being caught by the new catch block(s) that are not referenced in the current project, Fault Simulator will add a code comment within the catch block. The comment would read:

```
TODO: Add a reference to the <assembly name> assembly to
the current project.
```

If Fault Simulator cannot identify the assembly that defines the exception type, the comment would read:

```
TODO: Add a reference to an assembly that defines the
<exception type> type.
```

Refer to the DevPartner Fault Simulator online help in Visual Studio for additional information about this feature.

Adding XML Documentation <Exception> Tags to a Source Statement

Fault Simulator analyzes your source code for missing XML `<exception>` tags in Visual Studio projects where XML documentation is supported.

Note: Ensure that the current project, containing the method where you want to simulate a .NET fault, has been enabled to support XML documentation.

Fault Simulator uses XML documentation to determine the exception types that can be simulated. This is especially useful in third-party and user-written managed code.

If Fault Simulator detects escaping exceptions that are not caught within the current method, Fault Simulator will advise you to add `<exception>`

markup in a stubbed-out form to the method declaration. Fault Simulator will detect exceptions that can escape from a method. When exceptions are detected, you can add XML documentation comments to the applicable method.

Notice in Figure 6-8 that Fault Simulator lists every source line in the code block where it determines that XML could be added.

Figure 6-8. Shows Where to Add an XML Documentation Comment

When you click on a hyperlink, Fault Simulator adds `<exception>` markup in a stubbed-out form to the method declaration, as shown in Figure 6-9, prompting you to insert the appropriate comments.

```
/// <exception cref="ArgumentNullException"></exception>
/// <exception cref="FormatException"></exception>
/// <exception cref="IOException"></exception>
void Foo()
{
[snip]
}
```

Figure 6-9. Insertion of <exception> Tags

Adding a .NET Fault to a Source Location

Fault Simulator will also tell you where to add a .NET fault at a source statement to test the exception handling at that location. Click to add a .NET fault to that source location, and then click on the **Add a .NET fault to this statement** hyperlink, as shown in Figure 6-10.

Figure 6-10. Shows Where to Add a .NET Fault to a Source Statement

This action opens the **Add .NET Fault** dialog box, which reflects the method/exception criteria at that source location.

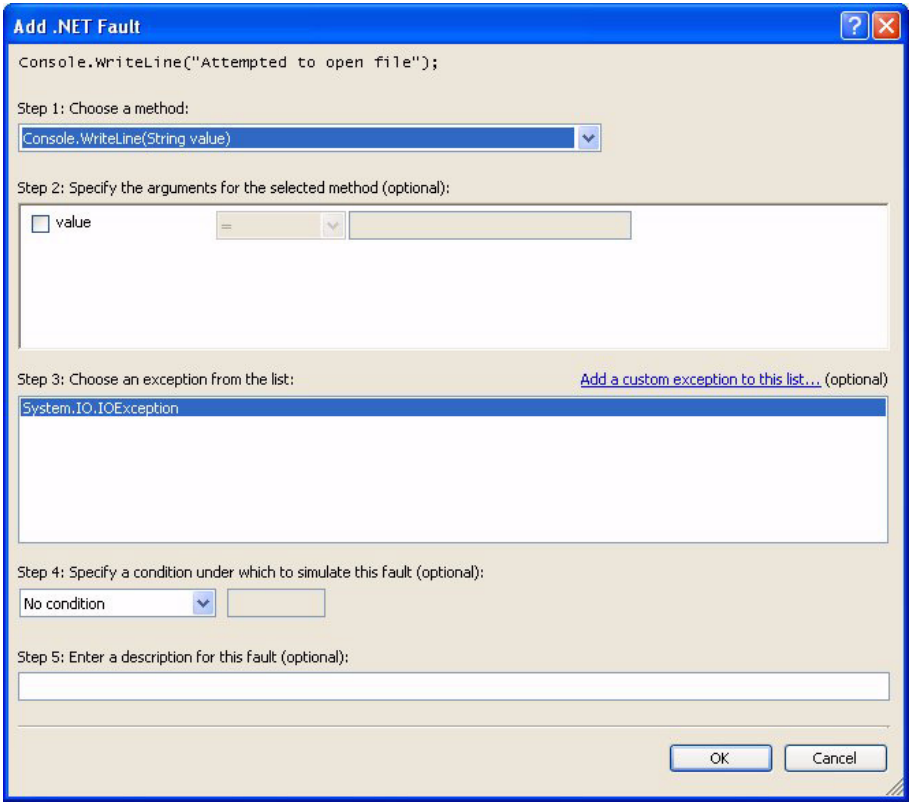


Figure 6-11. Settings Reflect Criteria at a Source Location

After you click **OK**, the new .NET fault automatically will appear in the **DevPartner Fault Simulator** window, as shown in Figure 6-12.

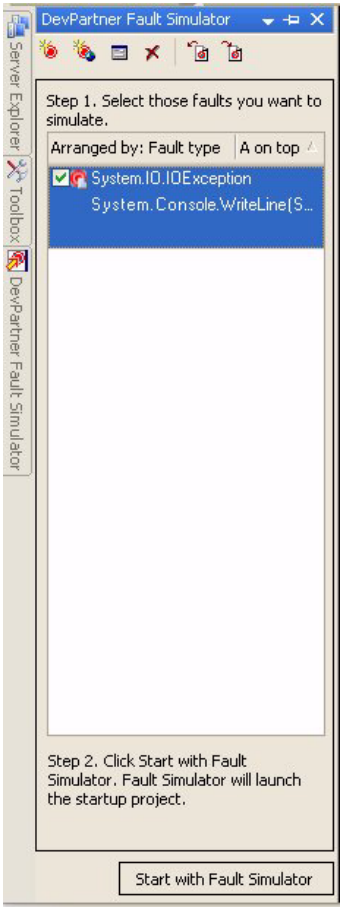


Figure 6-12. DevPartner Fault Simulator Window Shows a New .NET Fault

Click other indicators in the source window and repeat the previous steps if you want to add and configure more .NET faults for a subsequent fault simulation.

Performing a Fault Simulation to Test Exception Handlers

Start a simulation on the .NET fault(s) you previously added in "Adding a .NET Fault to a Source Location" on page 68, and then review the results.

Note: During a fault simulation, Fault Simulator will simulate a thrown exception as *soon* as the associated method is called. This means that no code from the target method will be executed. Therefore, if you add another .NET fault and then you run the program, the first exception that is encountered will be thrown immediately following the method call. However, any activity after the first method call will not occur.

Figure 6-13 shows that one of the four .NET faults was actually attempted and successfully simulated on the **Specified Faults** pane.

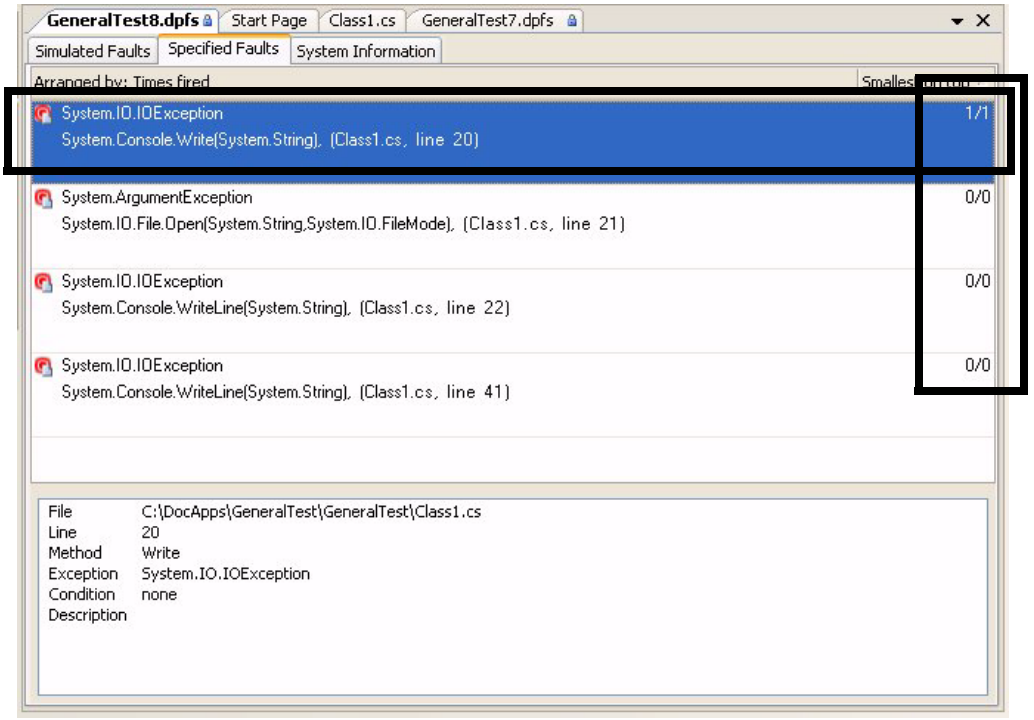
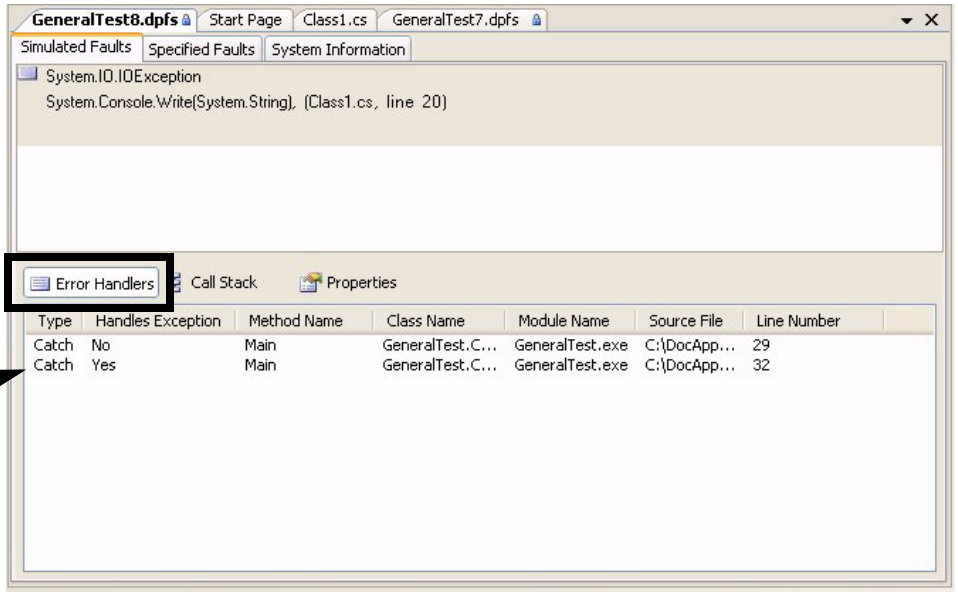


Figure 6-13. Shows that the `SystemIO.IOException` Was Successfully Simulated

Go to the **Simulated Faults** pane, and click on **Error Handlers**.



Error Handlers shows exact location where the exception was handled.

Figure 6-14. Error Handlers View Shows a Handled Exceptions

This view records the exact source location where the exception was handled along with other pertinent information. You can double-click on a line item to view the source associated with that occurrence.

Next, view the **Call Stack** view, as shown in Figure 6-15.

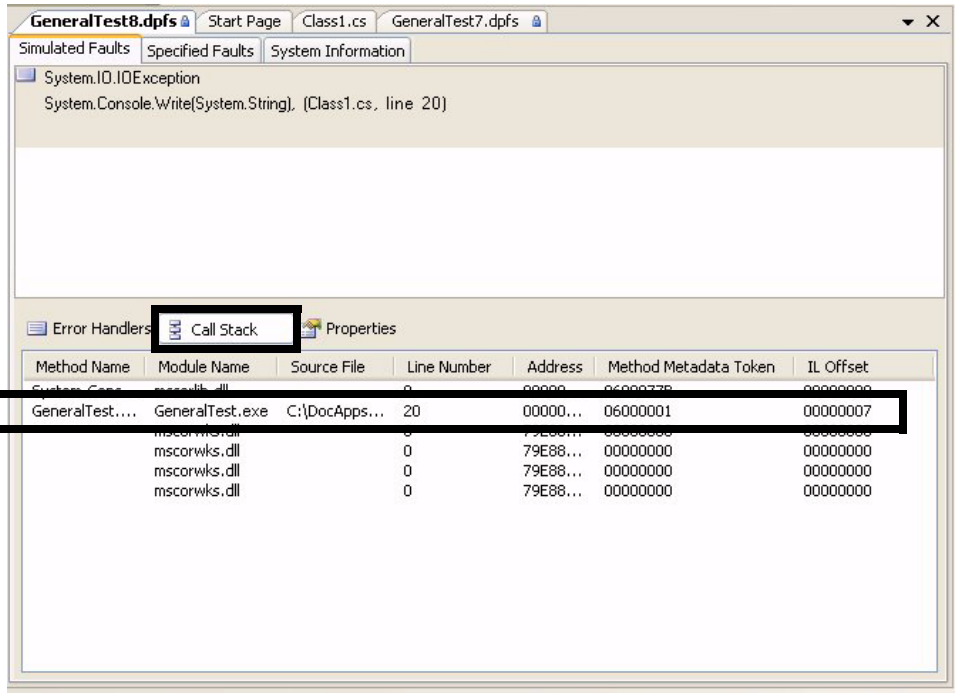
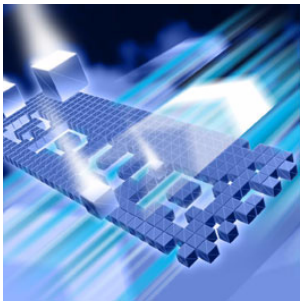


Figure 6-15. Call Stack View Traces Method Calls For a Thrown Exception

The **Call Stack** view analyzes where the exception was thrown in your code, and traces the steps through your code that led to the thrown exception. If the DLL or method is within the scope of your code, Fault Simulator will provide applicable source information, such as line number, source file, and called method.

Chapter 7
Evaluating Error Handlers



- Well-Constructed Error Handlers Promote Product Reliability
- Using Fault Simulator to Achieve Best Practices
- Evaluating Error Handler Results

This chapter reviews different approaches to evaluate error handlers, including an overview of structured exception handling. The chapter then shows 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. As defined in *Toward a Framework for Testing Error Handlers*[1], 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."

1.Source: Eliza LeCours and Robert Meagher, *Toward a Framework for Testing Error Handlers*, URL: <http://www.compuware.com/products/devpartner/1426_ENG_HTML.htm#NET>, Compuware Corporation, 2005

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

Incorporation of Robust Error Handling

Software developers should incorporate robust error handling code into the application code 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 effective exception 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.

Error Handling for Function Calls

Historically, software developers 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].

1.Source: Richter, 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 contains your executing code and 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 your exception handling code and only gets executed when code in your try block throws an exception. 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 exception 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 code block regardless whether the catch block actually handled the exception.

Try/Catch/Finally Execution

The try/catch/finally code follows this progression when an exception is successfully thrown:

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

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.

Using Fault Simulator to Achieve Best Practices

You can use Fault Simulator in Visual Studio as an integral part of product development to ensure that your code conforms to best practices. With Fault Simulator, you can verify the robustness and accuracy of the exception handling code. Fault Simulator helps you uncover weaknesses in your source code by letting you configure and execute 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 Fault in Managed Code

You can configure settings on the **Add .NET Fault** dialog box (example shown in Figure 7-1) to artificially throw an exception associated with a supported method in your managed code. Here, you can designate the exception to throw, plus other pertinent method signature properties.

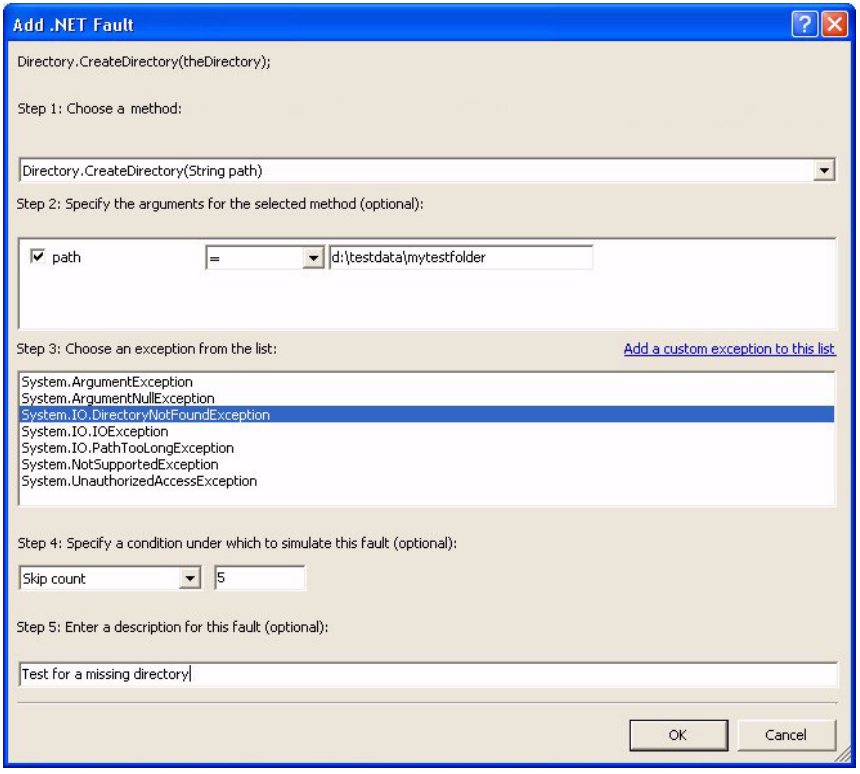


Figure 7-1. Example Adding a Source-Based .NET Fault

Note: See step 5 on page 61 to configure a .NET fault in Visual Studio.

Fault Simulator provides hints in the source window where you can add a .NET Fault directly to a source location. You can:

- Right-click on a source window and choosing **Add .NET fault** from the context menu (see Figure 7-2).
- Use visual indicators via the **Show Fault and Exception Handler Indicators** option on the **Fault Simulator** menu in Visual Studio (see Figure 7-3).

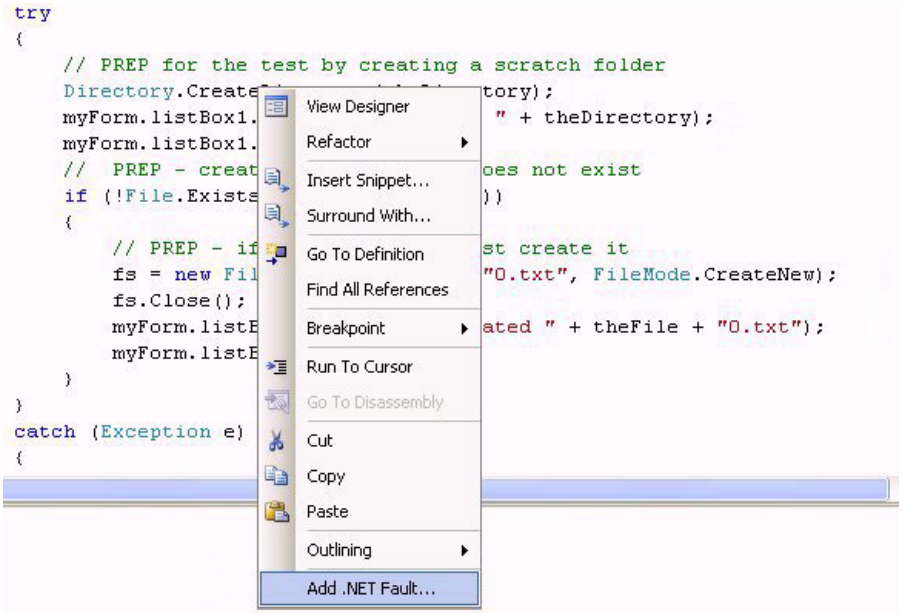


Figure 7-2. Shows the Context Menu to Add a .NET Fault to a Source Statement

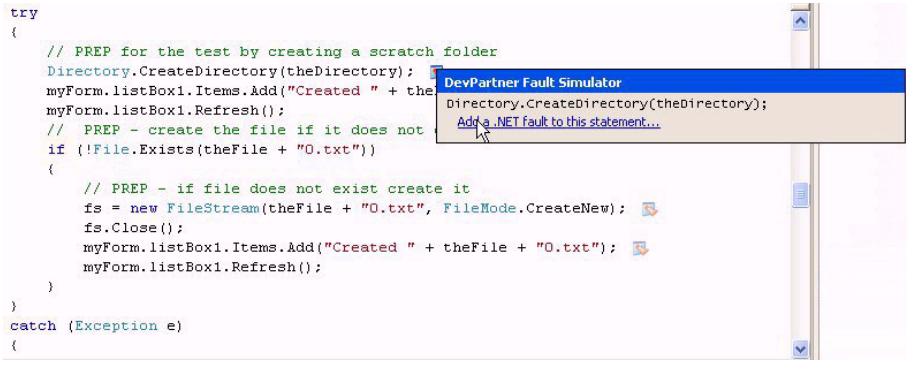


Figure 7-3. Shows the Show Fault and Exception Handler Indicators Option

Note: Learn more about using this feature in the IDE in "Walk Through Focusing on Exception Handlers in Your Code" on page 63.

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 (as shown in Figure 7-4).

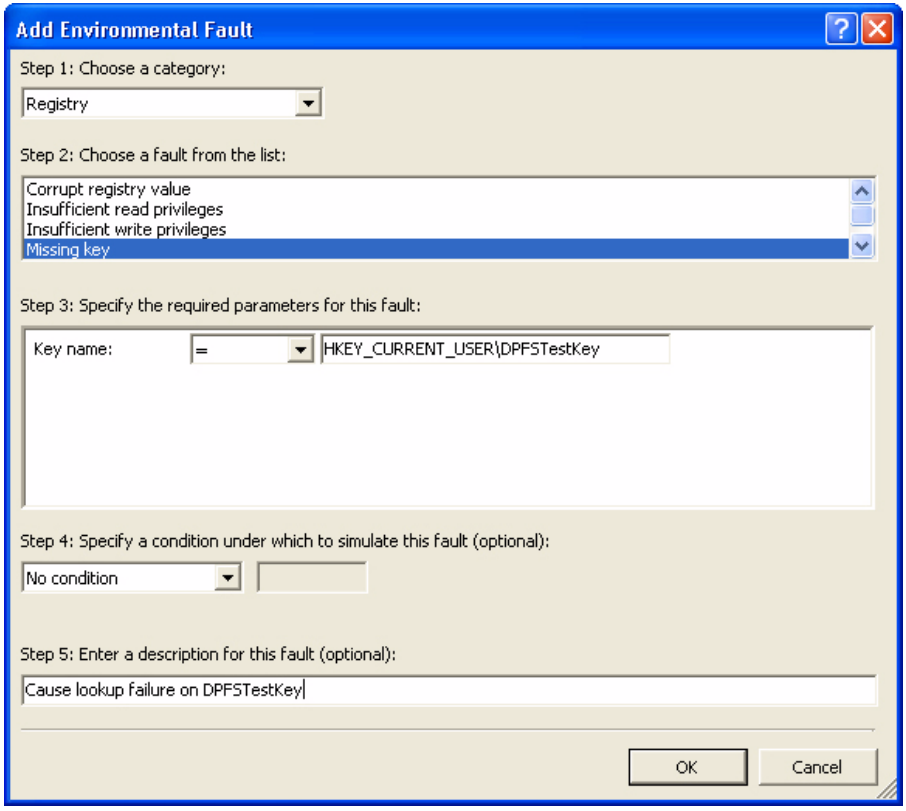


Figure 7-4. Example Adding an Environmental Fault

Note: See step 7 on page 38 to add and configure an environmental fault.

Reviewing Fault Simulation Results Views

Fault Simulator displays current simulation activity during a fault simulation session. Following its completion, Fault Simulator generates results summarizing the captured data.

Specified Faults Pane

The **Specified Faults** pane 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)

Table 7-1 summarizes how *Disk I/O: Disk full 1/1* is represented.

Table 7-1. How Faults Are Defined on Specified Faults Pane

Example of Text	Represents
Disk I/O	Environmental fault category
Disk full	Environmental fault name
1/1	Count of fired fault instances
1/**1**	Count of all attempts

Figure 7-5 on page 82 shows an example of the **Specified Faults** pane:

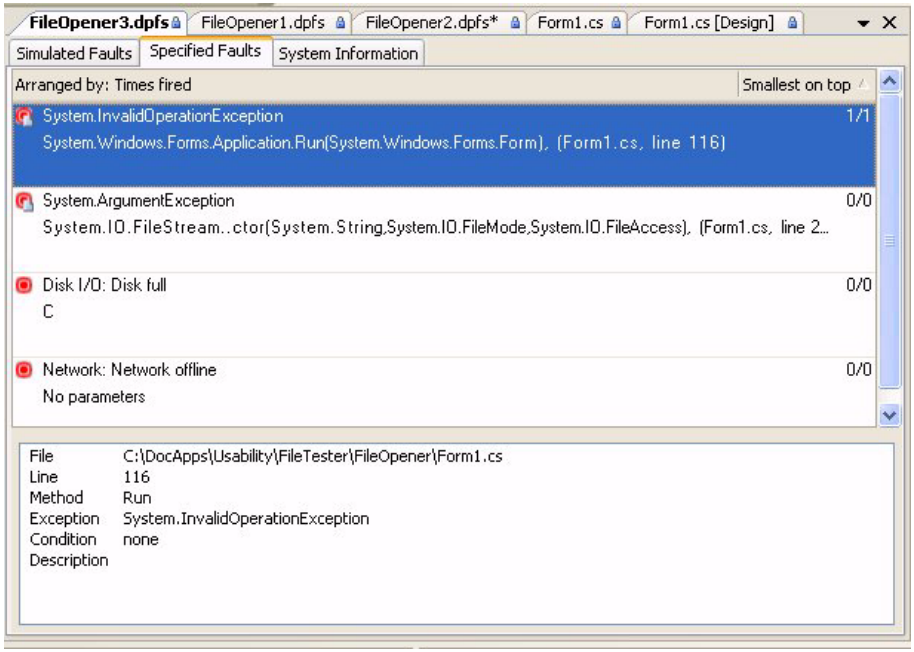


Figure 7-5. **Specified Faults** Pane

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

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

Simulated Faults Pane

The **Simulated Faults** pane, as shown in Figure 7-6 on page 83, lets you view three result tabs in the lower panel:

- Error Handlers
- Call Stack
- Properties

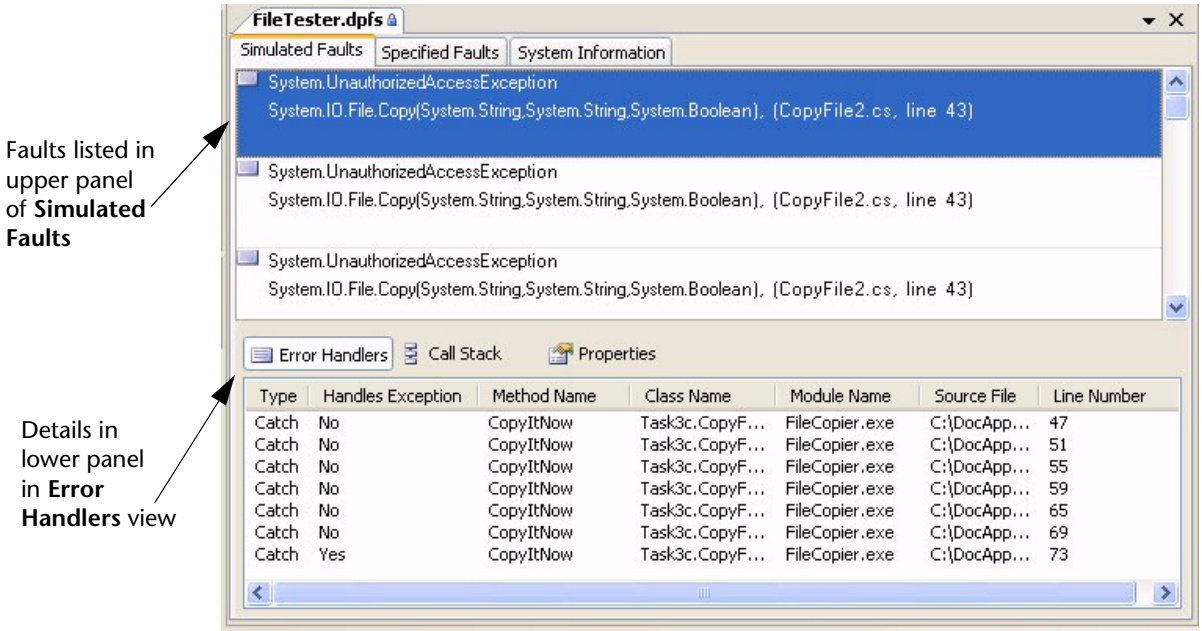


Figure 7-6. **Simulated Faults** Pane

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 managed 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 exception handling was resolved in the routine

Note: The **Error Handlers** view shows results on .NET 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. Table 7-2 summarizes how the properties apply to the .NET fault or the environmental fault.

Table 7-2. Fault Properties

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

Evaluating Error Handler Results

The following sections explain different examples of fault simulation results:

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 identifies:

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

For managed code, the **Error Handlers** view details how your try/catch code handled, or failed to handle, the designated .NET fault.

You can follow the logic to see where the code handled the fault with a catch block, or where it fell through because of an incorrect or misplaced catch block. Figure 7-7 on page 85 shows that the designated .NET fault was not handled at the intended locations.

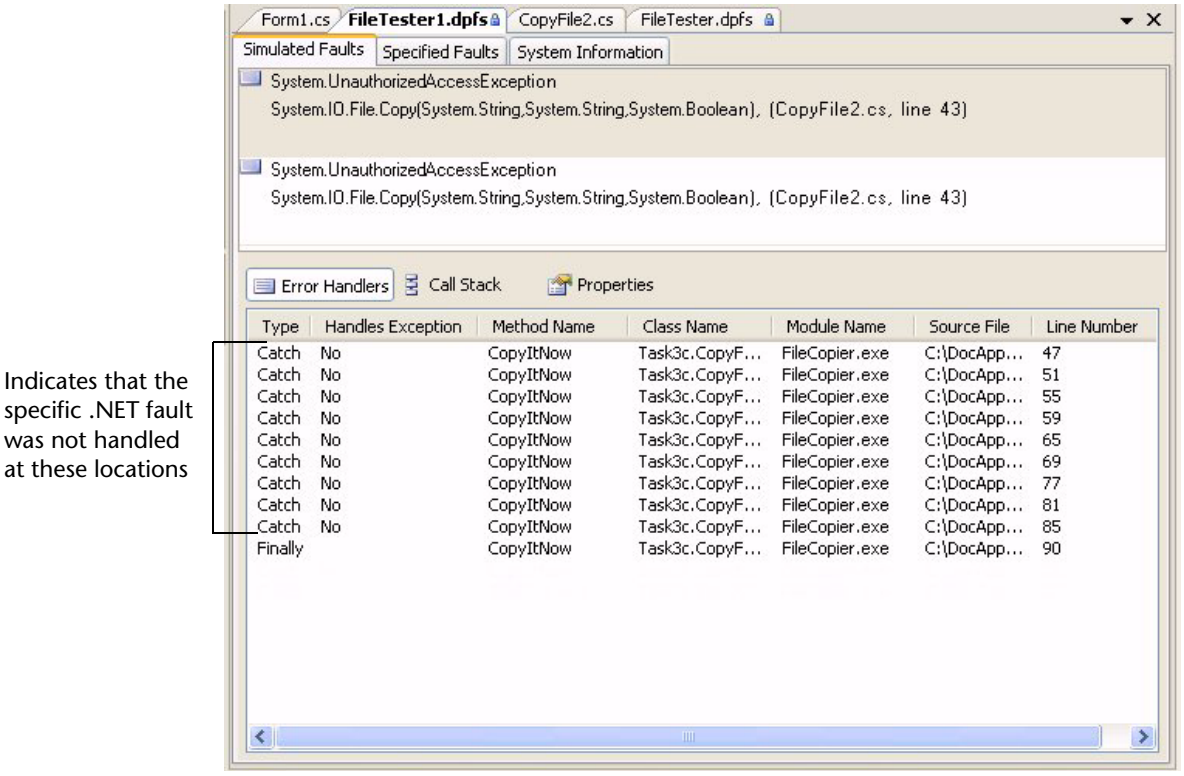


Figure 7-7. **Error Handlers** View — Shows Exception Handling Analysis

Identifying if Any Error Handlers Got Invoked

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

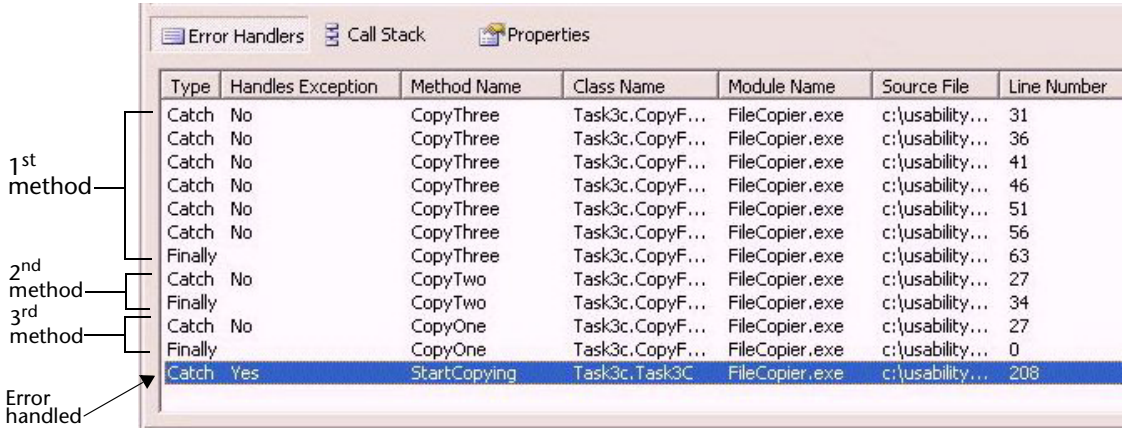


Figure 7-8. Error Handlers View Showing Catch and Finally Blocks Evaluated

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.

Figure 7-9 shows the catch blocks in your program that caught the exception in the **Call Stack** view.

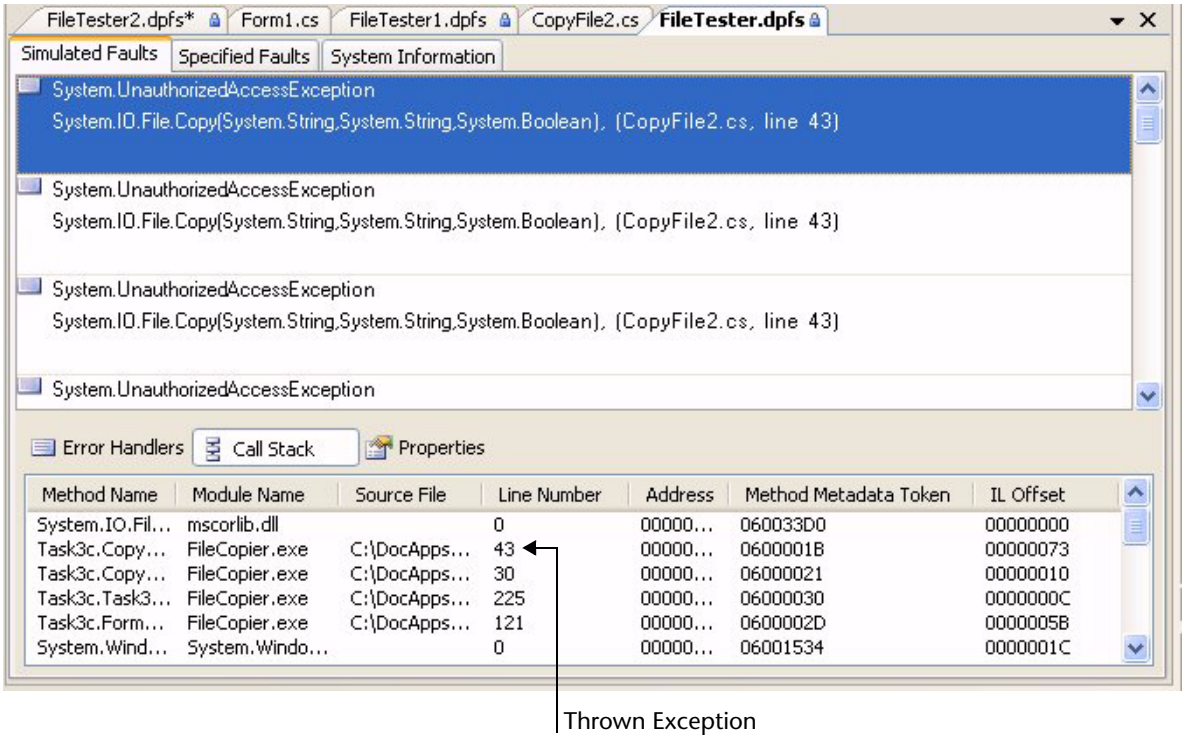


Figure 7-9. Call Stack View — Location of a Thrown Exception

Viewing the Source Statement That Handled the Fault

When available, Fault Simulator can take you back to the source where the exception handling was evaluated. Right-click on a line item in the lower portion of the **Error Handlers** view and choose **View Source** to go to the original code, as shown in Figure 7-10. From the original source location, you can troubleshoot whether the exception was properly handled, and if not, why not.

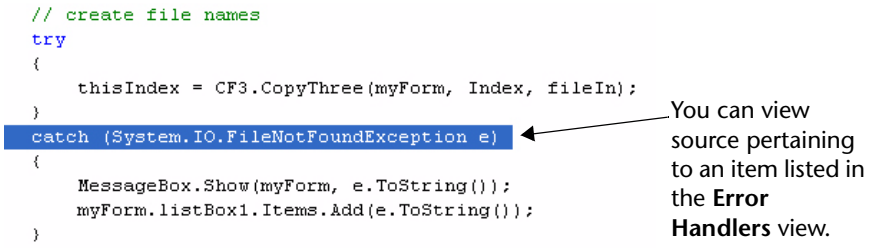


Figure 7-10. Viewing a Source Location Associated with a Fault Instance

Fault Simulator will also 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, unmanaged code typically relies on error handlers that use return code values and inline error handling to detect errors. Refer to "Error Handling for Function Calls" on page 74.

Use the **Call Stack** view to trace the path your unmanaged code took when a function failed, as shown in Figure 7-11 on page 88.

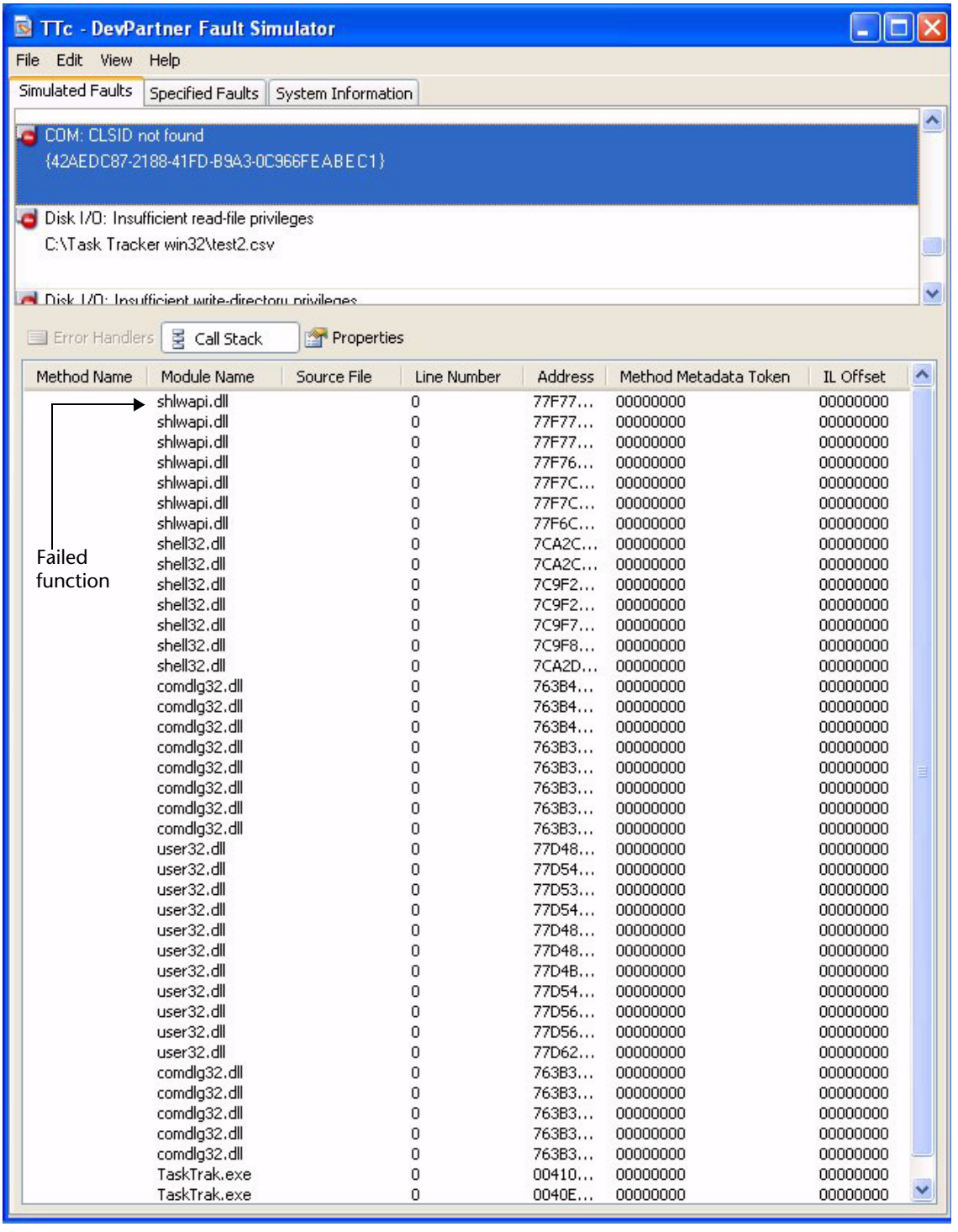


Figure 7-11. Call Stack View — Top-Most Item Designates Where a Function Failed

In the previous example, the **Call Stack** view shows the progression of API and DLL invocations that Fault Simulator monitored. Notice that 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 81 and "Simulated Faults Pane" on page 82 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. Table 7-3 provides some tips to consider.

Table 7-3. Comparing Specified Faults Versus Simulated Faults Results

Considerations	Possible Conclusions or Actions
Did the code execute at the location where you expected it to occur?	If the code associated with the fault never executed, the fault would not occur.
Did the managed code make a call to the relevant method?	If you simulated a fault on a supported method, you could: • Set a breakpoint on the targeted method call • Run in the Visual Studio debug mode • Check if the line of code executes If the statement was executed but Fault Simulator did not generate the fault, examine the arguments specified in the .NET fault descriptor to see if those values evaluated true when the method was executed.
Did the unmanaged code make 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.

Table 7-3. Comparing Specified Faults Versus Simulated Faults Results (Continued)

Considerations	Possible Conclusions or Actions
Did the code make 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 7-8 on page 85) 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 (such as, 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.

Figure 7-12 on page 91 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`.

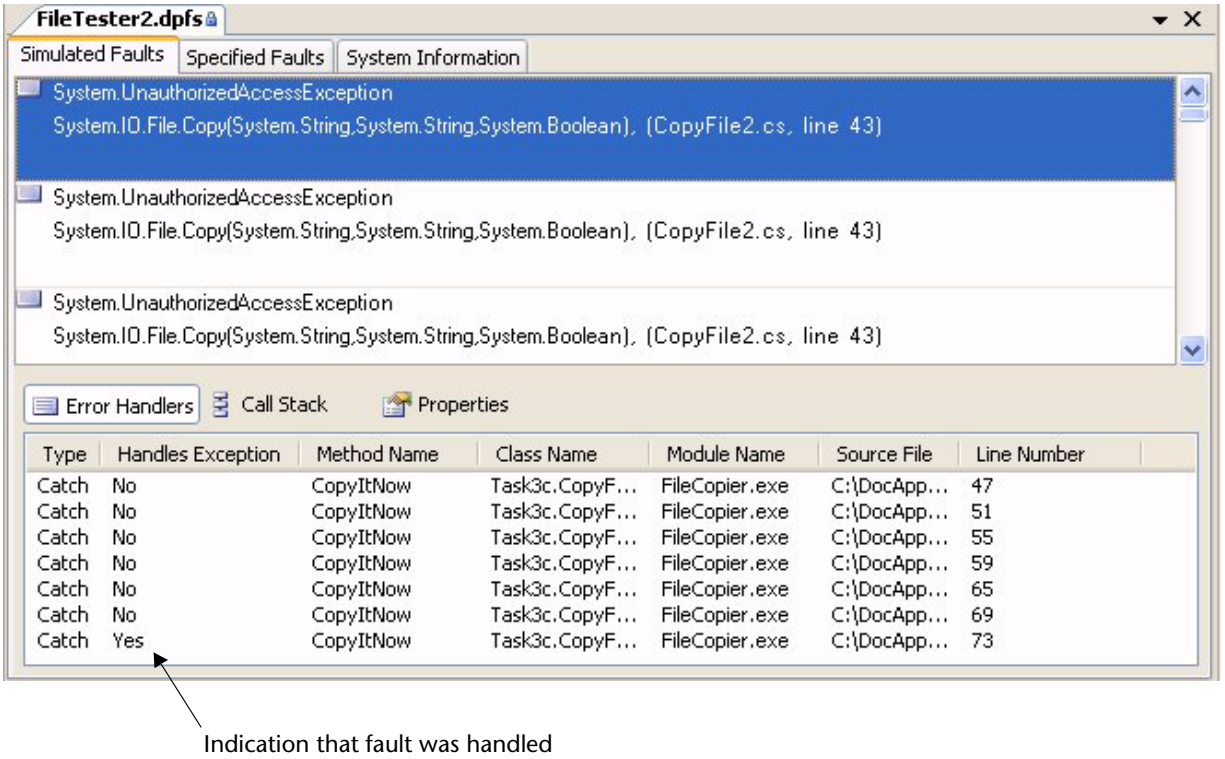
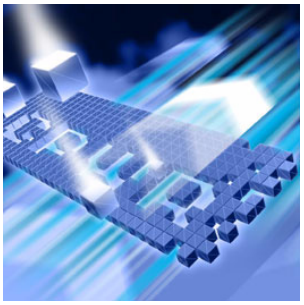


Figure 7-12. Fault Handled in the Second Method on the Error Handlers View

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.

Chapter 8

Improving Software Quality



- Objectives Software Developers Share
- Obstacles to Software Quality
- Software Vulnerabilities
- Testing for Predictable Outcomes
- Using Fault Simulator to Ensure Software Quality
- Three-Part Solution Using Fault Simulator

This chapter demonstrates how Fault Simulator optimizes your software development goals. It contrasts these goals with the challenges you also face. It then explores software vulnerabilities that affect application stability, along with alternatives that promote software quality. Finally, this chapter presents a three-part approach to improving software quality using Fault Simulator.

Objectives Software Developers Share

You share common objectives with other software developers, such as:

- Analyzing and eliminating all software vulnerabilities
- Adapting to changing internal and external conditions
- Addressing interoperability, compatibility, and portability constraints
- Keeping pace with changing technologies
- Identifying and fixing 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

Data Integrity

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

With application integrity, an application will continue to run, regardless of unknown or 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

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.

Obstacles to Software Quality

Several obstacles can jeopardize software quality, including:

- Product instability
- Explosion of new technologies
- Complexities of the inner workings of APIs and system services

Product Instability

Product instability creates bottlenecks to future product development in several ways. Unstable products:

- Disrupt application performance, affecting customer perception
- Place additional, unintended burdens on software development
- Force you to backtrack and fix unanticipated software bugs from the field
- Prevent you from performing your forward-thinking research of new technologies

Application defects become significantly more costly to debug and correct *after* a product is released. The Gartner Group recently reported[1] that "the average cost of unplanned downtime for a mission-critical application is $100,000 per hour."

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. You might not have ready access to the necessary subject matter consultants on the more subtle complexities of the applicable technology. You might also unwittingly underestimate the full extent of design requirements.

Complexities of the Inner Workings of APIs and System Services

You face another challenge, that of managing the complex inner workings of APIs and system services. For example, you might not have included sufficient error handlers inside a code block that calls into an API that checks for network connectivity. If that code is in fact insufficient, an unresponsive user interface might result, or worse — an abrupt shutdown that places the application and the operating environment at risk. Possessing the knowledge and the means to implement and test effective error-handling code goes a long way to ensuring software reliability.

---

1. Source: Theresa Lanowitz, *Gartner Application Development Summit Presentation: Software Quality in a Global Environment: Delivering Business Value*, URL: https://www.gartner.com/2_events/conferences/ad6_agenda.jsp?day=2, Gartner, Inc., September 2004.

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

These symptoms might result from the following software vulnerabilities:

- Logic errors

  Logic errors generate invalid or unpredictable results, perhaps stemming from a misinterpretation of the intended workflow. While not fatal, logic errors can lead to 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 (such as, 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

Figure 8-1 depicts three layers of application vulnerability:

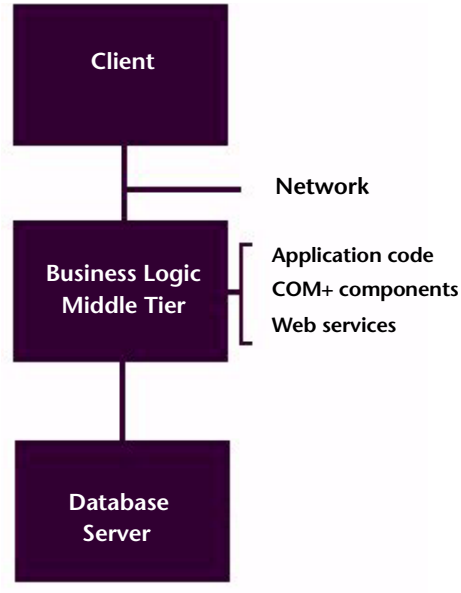


Figure 8-1. Layers of Application Vulnerability

Each layer has the potential of impacting the 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 an external security attack or an unwanted manipulation.
- A network offline condition could prevent the client from integrating properly with the business 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.

Testing for Predictable Outcomes

You perform tests on predictable areas in the application to ensure expected outcomes, including:

- Unit testing

  Isolates testing to specific code sections in the application
- Integration testing

  Verifies that different areas of the application work properly together

Testing results help you assess the risk and determine the bugs to fix, given that in reality not all bugs can realistically be fixed. You attempt to fix bugs that could negatively affect customer perception or hinder software reliability, even if rare. On the other hand, you might elect to delay fixes for bugs that are bothersome, but not damaging to the application or customer perception.

Using Fault Simulator to Ensure Software Quality

Building and deploying reliable applications but not sufficiently testing for all potential weaknesses in the product can challenge software development and quality assurance. Using Fault Simulator throughout the entire development life cycle ensures that critical applications can handle a multitude of adverse, atypical conditions. Fault Simulator lets you artificially expose application components to real-world failure conditions without causing actual harm to the application or the operating environment. You do not *physically* change environmental parameters. Rather, you let Fault Simulator mimic conditions, allowing you to examine how the application responds to unanticipated failures.

The next section illustrates how you can use Fault Simulator to ensure software quality using a three-part solution.

Three-Part Solution Using Fault Simulator

This section shows how you can use the fault simulation capabilities in Fault Simulator to improve software quality. This three-part approach incorporates white box, black box, and automated testing.

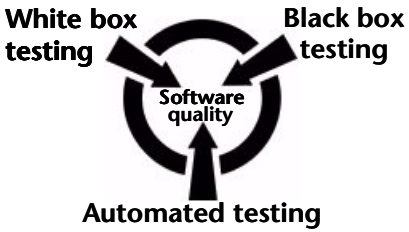


Figure 8-2. Three-Part Solution to Software Quality

Note: The code snippets in the following sections are intentionally simple but serve to demonstrate fault simulation concepts and capabilities.

White Box Testing Using Fault Simulator in Visual Studio

In white box testing, you test the source code during the development cycle, verifying program logic from the inside out. You focus on the inner workings of the source code.

Consider the scenario where you are building an application in Visual Studio. You hope to include sufficient exception handling. You need guidance on where you should insert try/catch blocks and add missing XML `<exception>` tags. Visual Studio provides a broad list of exceptions, but not necessarily choices that are context-specific.

You use Fault Simulator to indicate areas in your managed code where you can make direct improvements. After a clean build and with the **Show Fault and Exception Handler Indicators** feature selected (enabled by default). Fault Simulator displays icons ( ) wherever it determines one or more actions can be taken.

Note: Learn more about this functionality at "Walk Through Focusing on Exception Handlers in Your Code" on page 63.

Fault Simulator advises you of several areas in your managed code where you can pursue further actions. For example, you click on one of the icons to see where Fault Simulator detects that XML `<exception>` tags can be added, as shown in Figure 8-3 on page 100.

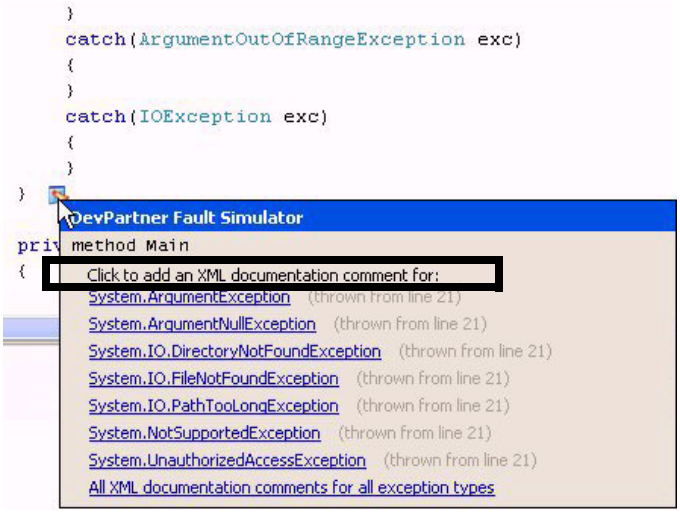


Figure 8-3. Detection of Missing XML Documentation

You then click on a hyperlink for one of the exception types to navigate to that source line. Fault Simulator inserts XML markup in a stubbed-out form. You insert the appropriate comments. You can repeat this step for the rest of the items on the list.

You click on another icon in the source window. Fault Simulator indicates where you can insert a catch block.

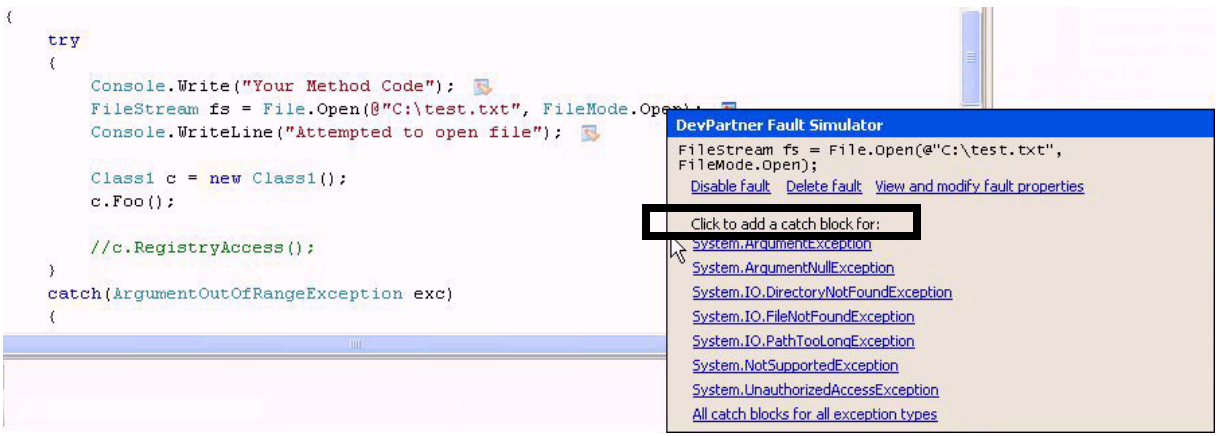


Figure 8-4. Detection of Catch Blocks That Should Be Inserted In the Source Code

You click on a hyperlinked item in the list. Fault Simulator takes you to the source line and inserts a stub catch block, as shown in Figure 8-5 on page 101.

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

    Class1 c = new Class1();
    c.Foo();

    //c.RegistryAccess();
}
catch (ArgumentOutOfRangeException exc)
{
}
catch (System.ArgumentException)
{
    throw; // TODO: Add error handling here
}
catch(IOException exc)
{
}
```

Figure 8-5. Insertion of a Catch Block in Stubbed-Out Form

You insert the appropriate exception handling code, and then repeat this step for the rest of the items in the list.

You click on another icon in the source window where you can add a .NET fault to test the exception handling at a source line that includes the method `File.Open` (see "Adding a .NET Fault to a Source Location" on page 68).

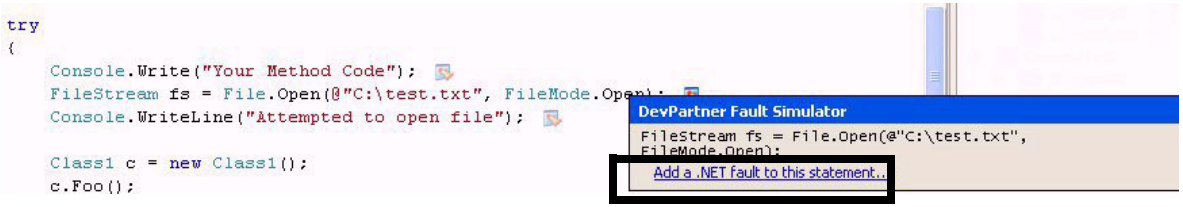


Figure 8-6. Indication to Test an Error Handler at a Specific Source Location

After you make that selection, the **Add .NET Fault** dialog box (see example at Figure 6-11 on page 68) confirms the string and other method/exception details. Fault Simulator lists the exceptions that can be simulated at that location.

After you ensure that the target executable is current with the source (and if necessary — rebuild), you start a fault simulation on the executable.

During the session, Fault Simulator throws that exception and terminates the executable, thus ending the session.

You review the results to determine the execution path. Fault Simulator captures and displays the call stack data as it unwound during the session, along with other error handler details. The results confirm whether the call successfully threw the exception. If so, it shows what catch block handled it. You trace up the method calls. You also double-click on an unsuccessful catch in the results view to go to the original source to troubleshoot the code. You conclude that you need to add a catch to handle the exception in that code block. You save the fault configuration data to a fault set file (`.dpfsfault`). You also save the results (`.dpfs`).

Note: Refer to Chapter 7, "Evaluating Error Handlers" for an explanation of error handler and call stack results.

You run another fault simulation with the same configuration. You want to verify that you corrected the problem revealed during the first session. You also save the subsequent fault set and results to disk.

You conduct similar tests on other areas in your code where you can improve your exception handling.

Black Box Testing Using the Fault Simulator Standalone

In black box testing, you test your software for bugs, such as environmental failures resulting from data handling, network access, or registry access. You run tests to ensure application stability and as a precursor for functional, regression, and final acceptance testing.

You intend to test that your application functions reliably under a variety of negative conditions. You know that *physically* causing environmental faults (such as reaching to the back of your box to unplug the network connection) could cause unintended consequences to the application or other applications also using the same network.

You decide to test the application's tolerance to simulated fault conditions. You start with disk I/O faults and choose to test a missing file condition using the Fault Simulator standalone application (see "Available from the Command Line" on page 11).

Because you hesitate to physically delete the target file, you create and configure a *Disk I/O* environmental fault descriptor, **Missing file**.

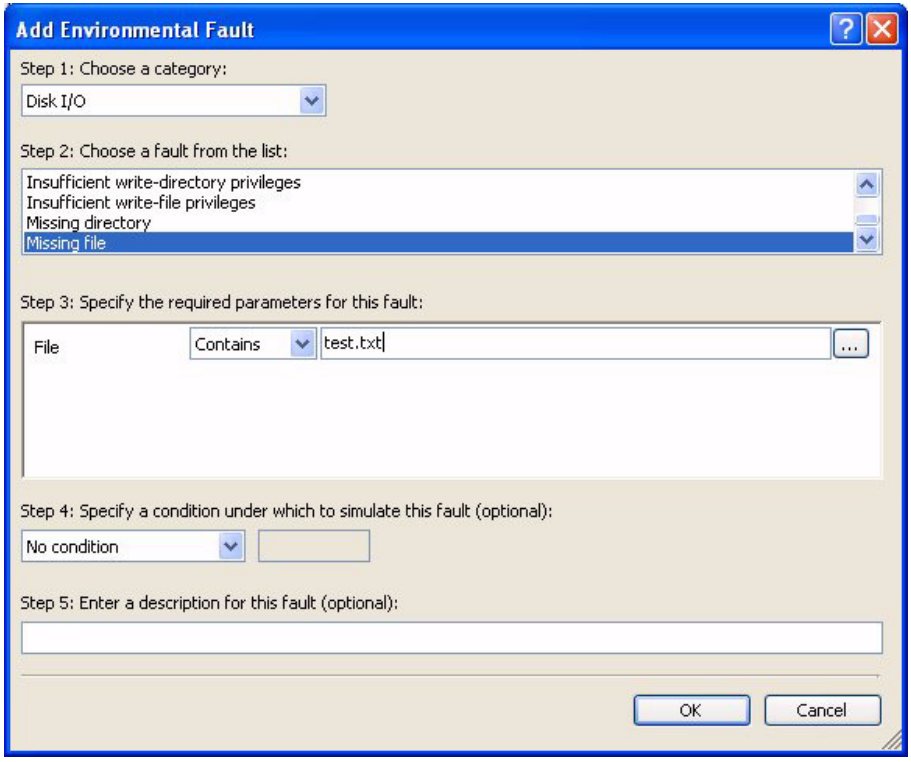


Figure 8-7. Configuring a **Missing File** Environmental Fault

You start the fault simulation and then launch the executable. Following the session, you see that Fault Simulator simulated the designated fault. 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 disrupt the application's normal operation.

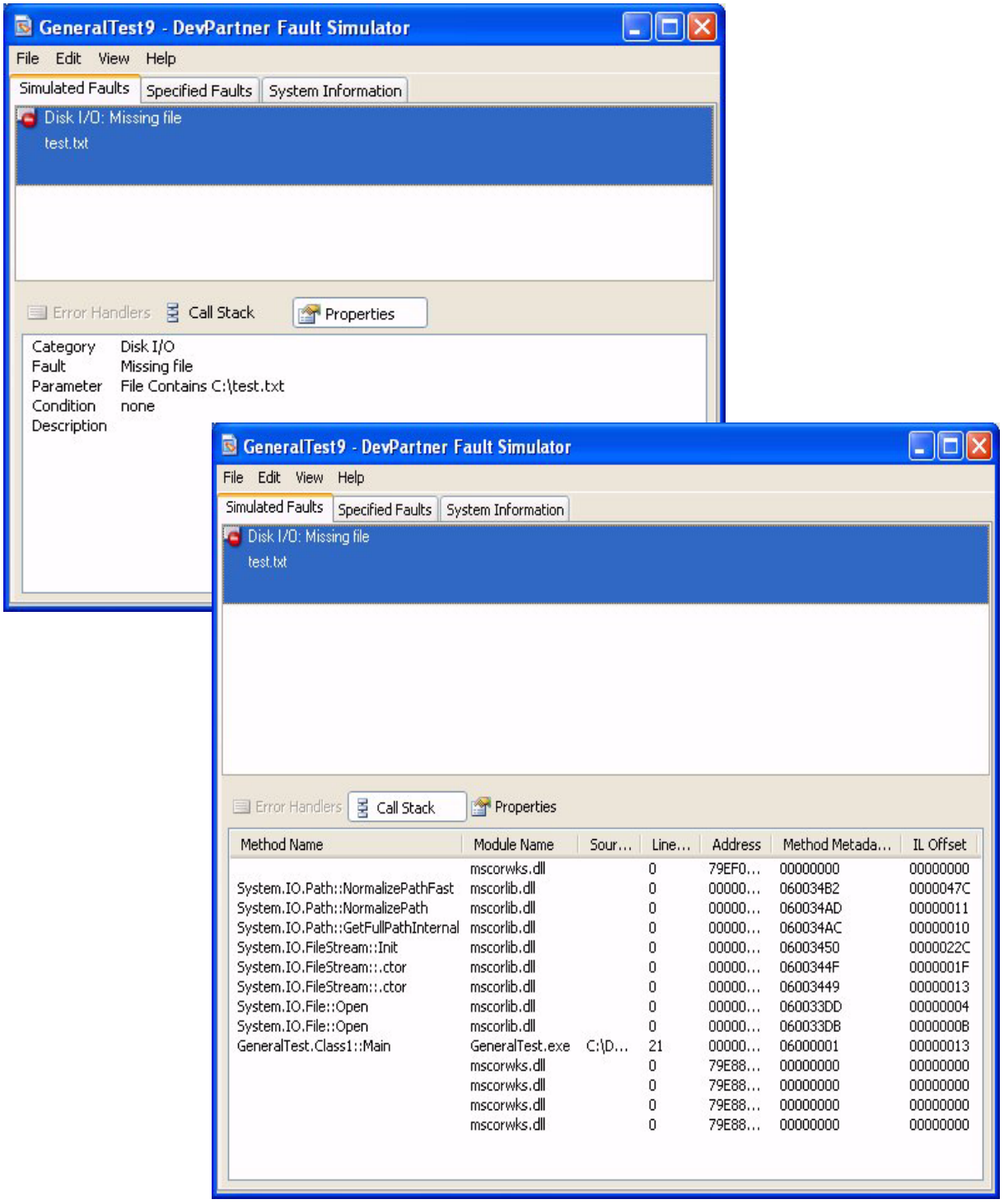


Figure 8-8. Properties for the Missing File Fault along with the Call Stack Details

You conduct additional tests on other environmental failure conditions. For example, you might run a fault simulation on a **Network offline** fault condition to test how the application reacts without physically disconnecting the network cable. You might 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. You save the fault data for each of these tests to individual fault set files for future reference.

Note: Refer to Chapter 7, "Evaluating Error Handlers" for an explanation of error handler and call stack results.

Automated Testing Using Fault Simulator from the Command Line

In automated testing, you create scripts that test how the application handles a variety environmental fault conditions. You can run tests that concentrate on weaknesses in the application code.

You plan to run automated testing on applications that you are currently developing. To that end, you have two options, as described next.

Option 1 You can create a script using the following template as a model.

```
<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 8-9. Template to Automate Fault Simulator From the Command Line

Note: See Appendix B, "Command Line Quick Reference" for an overview of the command line interface. Refer to the DevPartner Fault Simulator Command Line online help from the InfoCenter for detailed usage information.

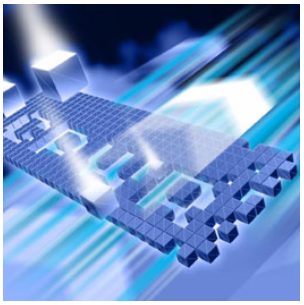
Option 2 You can use the Fault Simulator standalone application to create a batch script for you based on your preferences.

Note: See "Automatically Generating a Batch Script" on page 41 for more information.

Using either option, you can run nightly fault simulation tests on your application to check for any regression.

Appendix A

Troubleshooting



- Analyzing Environmental Issues
- Resolving Issues While Testing in Visual Studio
- Encountering General Issues

This chapter provides assistance resolving issues you might encounter using Fault Simulator either in Visual Studio or the standalone application, along with other general issues.

Analyzing Environmental Issues

This section helps you analyze environmental problems as you test your application.

Having Fault Simulator Automatically Generate Environmental Faults

Fault Simulator provides relevant advice to quality assurance professionals. The Fault Simulator standalone application can watch your target application as you use it normally and then generate environmental faults from the program activities it observed. Program activities might include those you initiate as well as those executed by the program itself. You can apply one or more environmental faults to a subsequent fault simulation. See "Walk Through to Generate Environmental Faults" on page 30.

Switching from Watch My Target to Configuring Faults Myself

Choose **Switch to Fault Editor** on the **Edit** menu in the standalone application to display the window where you can configure environmental faults and start a fault simulation. To see the original *Watch My Target* window, exit and restart the standalone application.

Simulating A Single Environmental Fault Multiple Times

As a monitored program executes, it invokes calls multiple times. As a result, Fault Simulator might simulate a single environmental fault more than once during monitoring. For example, using Notepad, you configure the Disk I/O *Insufficient read-file privileges* fault to occur based on the file designation you set for the parameter. Notepad will generate two instances of the fault even though you only expect a single attempt to access the defined file. Other applications might exhibit similar behavior.

Testing a File That Resides on a Network Path

Fault Simulator can simulate network failures associated with the Microsoft Winsock implementation. However, Fault Simulator does not directly support the universal naming convention (UNC) file and registry resources as network problems.

To target a file-related fault on a network path, choose a disk I/O fault. However, if you want to target a registry key failure on a network path, choose a registry fault. In either case, you should specify the fully qualified file designation.

Seeing Multiple Instances of Simulated Registry Faults

It is possible that when you simulate a registry fault you might see multiple fault instances, rather than a single fault instance following a fault simulation. This scenario could result from calls to a single .NET method that result in multiple queries to access information about a particular registry key or value.

For example, if you configure the registry fault, *Missing value*, you would provide the key name and value name parameters for the registry value you want to target. During monitoring, the .NET Framework potentially makes several calls:

- Requesting the value
- Checking the size of the value

Fault Simulator records these calls during monitoring, and displays final results following the completion of the fault simulation.

Simulating Heap Memory Allocation Faults

The *Heap allocation limits* fault can have serious consequences for the monitored program if the program does not have enough time to initialize. Heap allocation is fundamental to most programs. Failed allocation errors often occur outside the user code (such as, program initialization). As a result, programs that fail heap-allocation requests might behave erratically.

To bypass program initialization errors and prevent unrelated faults from firing prematurely, you can use the suspend simulation feature in Fault Simulator (see "Why Would I Suspend a Fault Simulation Session?" on page 20). Pausing a fault simulation gives the target application an opportunity to execute to a point where you would actually want to start simulating faults (such as, giving it time to load system DLLs first). Once the target application has reached an acceptable point of execution, you can resume the fault simulation activity.

Simulating a Fault on a Missing Image File in a Web Application

How you configure an environmental fault will determine if Fault Simulator can successfully simulate it during monitoring. Consider how the steps you take to configure the simulation might affect the outcome.

1. You created a Web application that generates a Web page with the following image tag in it:

   ```
   <img id=Image1 src=file:///
   c:\Inetpub\wwwroot\MyTestApp\imagemapare>
   ```

2. You configured a new environment fault, choosing the disk I/O fault, *Missing File*.

3. You designated the missing file parameter as:

   `c:\Inetpub\wwwroot\MyTestApp\imagemaparea.gif`

4. You start a simulation on your Web application at:

   `http://localhost/MyTestApp`

5. You see that Fault Simulator does not report the image file as missing.

When you simulate faults in a Web application, Fault Simulator will watch every instance of **`aspnet_wp.exe`** and **`inetinfo.exe`**. Since you specified the ASP.NET application, **`MyTestApp`**, Fault Simulator watches for the ASP.NET worker process (`aspnet_wp`) to load this Web application. This instance of ASP.NET becomes the active process. However, the **`aspnet_wp`** process will not access your configured parameter of:

`c:\Inetpub\wwwroot\MyTestApp\imagemaparea.gif`

As a result, no simulations will occur.

Because Internet Explorer (IE) treats the uniform resource identifier (URI) as a file moniker, IE will attempt to access the file directly. Since Fault Simulator is not monitoring IE, the image file, configured for the environmental fault, is successfully detected and retrieved. Even if this URI made a request through the Web server by employing an http scheme, the file would still be detected and retrieved because Fault Simulator is not monitoring the `Inetinfo` process or any resources associated with it.

As a workaround, ensure that the process is the same one that uses the file specification you configured for that environmental fault. You should configure the environmental fault with the file specification from within the IE process, rather than `MyTestApp`, as in this example. Fault Simulator then will successfully simulate the missing file fault condition.

Encountering Zero-Length Files Created After a Disk Full Fault Is Simulated

During the simulation of a *Disk full* environmental fault, the target application will attempt to create a new file, and Fault Simulator will then simulate this fault. However, in this scenario, the operating system creates a zero-length file before Fault Simulator takes control. You should clean up any zero-length files that remain from the disk full simulations.

Resolving Issues While Testing in Visual Studio

This section deals with possible situations you might encounter while testing error handlers in Visual Studio.

Simulating Faults in a Visual C++ Project

Fault Simulator supports managed or unmanaged C++ projects for environmental faults only, but not for .NET faults. However, you can simulate .NET faults in Visual C# and Visual Basic .NET source files.

Adding a .NET Fault to a Source Statement

You want to add a .NET fault to a source line but you do not see any icons in the source window (see "Adding a .NET Fault to a Source Location" on page 68). Several reasons could apply:

- Ensure that you are only adding a .NET fault to a Visual C# or Visual Basic .NET source file.
- Verify the project type is supported in Fault Simulator and that all project requirements have been met. Refer to the DevPartner Fault Simulator online help in Visual Studio for more information.
- Ensure that the target executable is current with the source. If not, perform a clean build.
- Check that the source statement contains at least one supported method. Refer to the DevPartner Fault Simulator online help in Visual Studio for a list of supported namespaces.

Missing Show Fault and Handler Exception Indicators

If you do not see the  icon in the source window to advise you where to improve your exception code, ensure that:

- **Show Fault and Exception Handler Indicators** is selected from the **Fault Simulator** menu (enabled by default).
- The modified date for a source file is current with the assembly file. Otherwise, the icons will not appear for that source file. As a workaround, modify the source file, save, and rebuild the project.
- Ensure that all other project requirements are met, and especially those specific to using this feature (see the DevPartner Fault Simulator online help in Visual Studio for more information).

Note: See "Walk Through Focusing on Exception Handlers in Your Code" on page 63 for more information on using this feature.

Locating a Previously Added Source-Based .NET Fault

If you originally added a .NET fault to a source location but Fault Simulator detected a code change, Fault Simulator will remove the original .NET fault configured there. Other reasons for removal include:

- Changes to the contents in the project file
- Removal or renaming of the project, source file, or specific source statement in question

You should add a new .NET fault to the source location again. To assist you, Fault Simulator identifies an eligible source statement that contains at least one supported method.

Determining Why a .NET Fault Fails to Fire as Expected

If you add a source-based .NET fault where the code construction includes compound or nested source statements on a single line, Fault Simulator might not be able to properly evaluate the designated method call. As a result, Fault Simulator might not fire the corresponding fault as expected during the fault simulation.

As a workaround, break out the source statement into multiple lines of code. Consider the following examples:

Example 1:

Change from this construction:

```
DateTime[] dates = new DateTime[]{
    new DateTime(2000, 1, 1),
    new DateTime(2001, 1, 1, 0, 0, 0)};
```

To this construction:

```
DateTime dt1 = new DateTime(2000, 1, 1);
DateTime dt2 = new DateTime(2001, 1, 1, 0, 0, 0);
DateTime[] dates = new DateTime[]{dt1, dt2);
```

Example 2:

Change from this construction:

```
Label1.Text = Environment.GetEnvironmentVariable("TEMP");
```

To this construction:

```
String text = Environment.GetEnvironmentVariable("TEMP");
Label2.Text = text;
```

Simulating Faults in a Project Dependent on Others in the Solution

The projects and the target executable in a given solution should be current and synchronized with each other. Otherwise, Fault Simulator might not simulate a source-based fault. Consider the next example:

You have a solution with several projects. Projects A and B produce assembly DLLs, while project C produces an executable that references classes and methods in A and B. Because you must build the assemblies in projects A and B before you build the executable in project C, you set the dependencies in the project properties to specify that build sequence.

You intend to use Fault Simulator to add a .NET fault to a source statement in project C. If you have not saved changes since the last solution build, Fault Simulator will fail in its attempt to build project C.

In addition, Fault Simulator will not simulate the source-level fault in question. Fault Simulator will display this status in the output window.

As a workaround, ensure that the target executable is current with the source, or at minimum — all the projects that project C depends on. Rebuild if necessary. Ensure that the assemblies for these projects are present in their appropriate locations according to the project properties.

Seeing an Unexpected Exception Thrown

Sometimes when Fault Simulator simulates an exception, it appears that Fault Simulator throws a different exception. This could occur because the operating system intercepts the original exception being simulated and then throws another exception in its place. This scenario is normal behavior for the operating system.

Simulating Source-Based Faults on Virtual Methods

Fault Simulator can simulate faults on a wide range of managed classes and methods. Many of these classes have virtual methods that can easily be overridden in user code. When a supported class has one or more of its virtual methods overridden in derived classes (user code), Fault Simulator cannot simulate faults on overridden methods of .NET methods.

The following example clarifies the scenario:

```
1:  class MyArray : ArrayList
2:  {
3:     public override int Add(object value)
4:     {
5:        return base.Add(value);
6:     }
7:  }
8:
9:  class Program
10: {
11:    private MyArray m_array1 = new MyArray();
12:    private ArrayList m_array2 = new ArrayList();
13:
14:    static Main(string[] args)
15:    {
16:       m_array1.Add(new object());
17:       m_array2.Add(new object());
18:    }
19: }
```

The `MyArray` class overrides the Add virtual method on line #3. The `Program::Main` method adds a new object to both of its internal arrays on line 16 and 17. In this case, Fault Simulator visibly identifies the source code (with icons) where it can simulate a source-based fault on

both lines 16 and 17. However, since line 16 is really calling the `MyArray::Add()` method instead of the `ArrayList::Add()` method, a fault will not be simulated there. Line 17 will be able to simulate a fault since that is a call directly to `ArrayList::Add()` method.

Using Signal Modules for Processes That Host Multiple Applications

Fault Simulator supports the use of signal modules for processes that host multiple applications, such as **`asp_net.exe`** and **`w3wp.exe`**. Once you start the fault simulation, Fault Simulator will watch for the target application (such as, the signal module). The fault simulation will not actually commence until it detects the signal module. Once it does and then begins the session, the specified fault set becomes active for the entire process including all hosted applications. If the fault set contains environmental faults, they will affect all hosted applications within the process instance.

For example, assume you have two Web applications called `WebApplication1` and `WebApplication2`. (Application names do not matter, and can even be the same, as uniqueness is based on the fully-qualified path.):

1. You configure a fault on `System.IO.DirectoryInfo.GetDirectories()` for `WebApplication1`, and start the fault simulation.
2. You launch Internet Explorer, and start up `WebApplication2` (which also utilizes `System.IO.DirectoryInfo.GetDirectories()`.

   Fault Simulator will not begin simulation because `WebApplication1` has not signaled.
3. You then start up `WebApplication1`, and it loads into the same `asp_net.net` worker process as `WebApplication2`.

   Fault Simulator detects the signaled application and begins simulation on the hosting ASP.NET process.
4. You use `WebApplication2` again. Fault Simulator will simulate a fault on `System.IO.DirectoryInfo.GetDirectories()` now, because the signaled application has activated the fault set for the entire hosting process.

Note: Fault Simulator does not support application scoping inside a single process. It does support application signaling as described above. Once a host process detects the signaled application's presence, the fault set applies to the entire process and not just the signaled application. As a remedy, use scoped source-based .NET faults.

Simulating Against a Web Application in a Debug Session

If you choose **Fault Simulator** > **Start with Fault Simulator without Debugging** but did not debug a Web application, and then you subsequently choose **Debug** > **Start Debugging** without ending the fault simulation you just started, the debugger will attach to the currently running Web server. This might result in faults being simulated in your debug session even though you did not choose **Fault Simulator** > **Start with Debugging**.

Encountering General Issues

This section explains general issues you might encounter.

Seeing No Simulation Activity Occurring on a Web Application

If you start the Web application before you start monitoring, Fault Simulator does not know about that instance. When monitoring a Web service or application, Fault Simulator looks for the *next* instance.

Submitting a TrackRecord Defect

Fault Simulator is compatible with Compuware TrackRecord 6.2 or later. If you have an earlier version of TrackRecord on your system, you must uninstall it and reinstall the correct version.

Inability to Submit a Work Item to Team System

Several reasons might explain why you cannot submit a bug to Visual Studio Team System. See "Visual Studio 2005 Team Foundation Server Integration Requirements" on page 4.

Attempting to Run Another DevPartner Process on the Same Target

Only one DevPartner Studio feature (such as, DevPartner performance analysis or memory analysis) or DevPartner product (such as, DevPartner SecurityChecker or Fault Simulator) can monitor a single program at any given time on the same machine. For example, if the DevPartner error detection feature is monitoring the same executable file, Fault Simulator will indicate that it must wait until the other operation concludes.

Note: The exception is DevPartner coverage analysis. You can start a session with Fault Simulator and coverage analysis, as long as DevPartner Studio is also present. See "Can I Collect Coverage Analysis During a Fault Simulation?" on page 23 for more information.

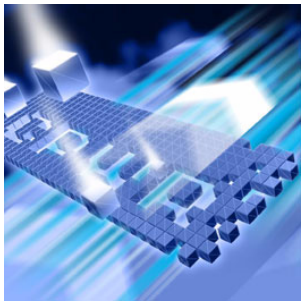
Determining if Fault Simulator Encountered an Internal Error

Fault Simulator writes to the operating system's event viewer if it encounters an internal error. For more information, go to **Administrative Tools** > **Event Viewer** > **Application** from the Control Panel. You can use the data in this log to troubleshoot unexpected errors with Compuware Technical Support.

Note: If your user account does not have permission to write to the log and Fault Simulator encounters an internal error, a user-visible error might occur.

Appendix B

Command Line Quick Reference



This appendix introduces the Fault Simulator command line interface. It also includes a quick reference of the Fault Simulator commands, as well as the return codes that the command line interface might generate.

Introducing the Command Line Interface

Fault Simulator extends its fault simulation capabilities to the command line. With Fault Simulator, you can automate fault simulations on projects that do not require user intervention. You can use this functionality to enhance the unit testing, functional testing, and regression testing you perform on applications under development. For example, you can write scripts that you can run repeatedly from the command line to augment regression testing.

The command line uses fault sets previously configured in Fault Simulator. Results generated from the command line are available for viewing in the Fault Simulator user interface.

The remaining sections provide an overview of the command line interface. For more in-depth usage information, refer to the DevPartner Fault Simulator Command Line online help from the InfoCenter.

Fault Simulator Commands

Table B-1 provides a brief summary of the Fault Simulator commands. For more information, consult the online help.

Table B-1. Fault Simulator Commands

Command	Required	Function	Syntax
/?		Accesses the command line console help; can be used independently or with another option for additional assistance	/? /?:*keyword*
/a	yes (see Note on page 119)	Specifies the path and file name of the application	/a:*<executable path>*
/c	yes (see Note on page 119)	Specifies a COM+ component program name (not the DLL name)	/c:*<component>*
/d		If present, outputs the faults in a fault set (as specified by /f) to the console window	/d
/e		Performs individual simulations, enabling one fault at a time; cannot be used in conjunction with /n:x	/e
/f	yes	Specifies the path and file name of the designated fault set; if missing, Fault Simulator will return an error code	/f:*<file path>*
/g		Specifies the command line arguments for the launched process	/g:"arg arg" /g:*argument*
/l		Specifies an application to launch (only required when /g is used)	/l:*<executable path>*
/n:x		Simulates a fault depending on its numeric position (represented by x) in the fault set file; cannot be used in conjunction with /e	/n:x
/p		If present, pipes the return code of the launched application (as specified by /l) to the console, rather than using an error code returned by Fault Simulator	/p
/r		Specifies the results file path and file name	/r:*<file path>*
/s		Specified the working directory for the process defined by /l	/s:*<folder path>*
/u	yes (see Note on page 119)	Specifies the vroot of the local URL to watch Fault Simulator for in a Web application	/u:*<vroot>*
/v		Enables the collection of managed code coverage information	/v

Note: The /a, /c, and /u command line options are mutually exclusive. Consequently, only one of these switches can appear on the command line. If all of these commands are missing, Fault Simulator will return an error code and will also display the help output to the console.

If all of the required arguments are missing with no help option specified but you included other optional arguments, Fault Simulator will return an error code

Command Line Return Codes

When an error condition occurs from the command line, Fault Simulator will echo the error text back to the console output and exit with the applicable return code. You must check any return code (other than 0 for success) and determine the proper course of action.

Note: Fault Simulator will generate the return code piped from the launched process when the optional `/p` switch is specified.

Table B-2 lists the return codes that Fault Simulator generates.

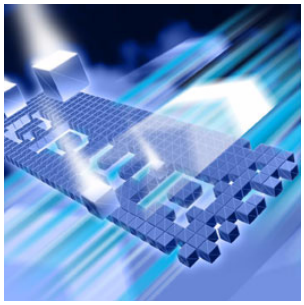
Table B-2. Fault Simulator Return Codes

Return Code	Description
0	Indicates success
1	The fault set file you specified could not be found. Please verify that the fault set file exists, and fully specify it on the command line.
2	The application executable could not be found. Please verify that the executable exists, and fully specify it on the command line.
3	The results file could not be created. Please verify that there is sufficient disk space and that you have permission to write to the specified folder.
4	Only one type of application may be specified for a test run. Please specify either `/a`, `/c`, or `/u` and then restart.
5	An unrecognized option was found on the command line. Please review the list of options in the console help, correct the unrecognized option, and restart.
6	One of the options was used more than once on a single command line. Review the command line, remove the duplicate option, and then restart.
7	A valid DevPartner Fault Simulator license could not be obtained.
8	An error was encountered. Open the System Event Log by running the **Control Panel** > **Administrative Tools** > **Event Viewer** for a full explanation of the error. Errors will be logged under the category *DevPartnerFaultSimulator*.

Table B-2. Fault Simulator Return Codes (Continued)

Return Code	Description
9	The results file specified is currently in use by another process. Please choose a different results file name and restart.
10	The required `/f` option was not specified. Please specify a fault set and then restart.
11	The required target application was not specified. Please specify either `/a:<exepath>`, `/c:<component>`, or `/u:<URL>` and then restart.
12	None of the required arguments were specified. Please specify a fault set file, `/f:<filespec>` and either an application `/a:<exepath>`, a COM+ component name, `/c:<component>`, or a URL `/u:<URL>` and then restart.
13	`/l` requires the use of the `/a` switch or a launch executable.
14	DevPartner Studio was not detected. The `/v` option cannot be used.
15	The process could not be launched (with reason indicated).
16	The COM+ application specified for `/c` is invalid.
17	The fault set file specified for `/f` is invalid.
18	The `/e` and `/n` options are mutually exclusive and cannot be used on the same command line.

Glossary



activity

Program function that Fault Simulator observes as it watches your target application; subsequently associated with environmental fault that you can use in your next fault simulation (only available in the standalone application); see watch my target; see also "Automatically Generating Environmental Faults" on page 29

argument

Input that is passed to a function; optionally used when configuring a .NET fault

call stack

Path the code took to the point where a function failed; view in Fault Simulator that represents this path; code logic that analyzes where the exception was thrown in the code, and traces the steps through that code leading to the thrown exception

CLSID

Parameter pertaining to COM environmental faults that designates the Class ID for a COM method call; see parameter

COM

Category of environmental faults based on the Component Object Model; see "Using Environmental Faults to Simulate Application Failures" on page 14

command line interface

Allows you to use scripts to automate the testing of applications; 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 Appendix B, "Command Line Quick Reference"

coverage analysis

Analysis feature in DevPartner Studio that lets developers and quality assurance engineers automatically locate untested code in managed applications and components; the command line and user interfaces allow you to integrate fault simulation with coverage analysis (integrated feature only applicable to managed code); see "Can I Collect Coverage Analysis During a Fault Simulation?" on page 23

delay time

Optional condition you set when configuring a fault descriptor; refers to the time interval (in seconds) Fault Simulator waits before simulating the associated fault

environmental fault

Represents the emulation of failure conditions applied to several classes of methods that deal with runtime environments; cannot be tied to a source location in managed code; see "Using Environmental Faults to Simulate Application Failures" on page 14

error handler

Log of unwind events or a log of Win32 return values applicable to the selected fault instance; see "Error Handling for Function Calls" on page 74

exception handler

Code construct that handles the occurrence of a fault condition that alters normal program execution; term used interchangeably with error handler; see "What is an Exception" on page 75

failure count

Integer value representing how many times a failure should occur before the fault is simulated (ideally after a delay time has been met); see parameter

fault

Represents a condition that occurs in an application that can have unforeseen consequences; results from a failure of some method called by an application

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 "How Do I Use Fault Descriptors to Simulate Fault Conditions?" on page 14

fault instance

Simulated fault that occurred during the execution of the target application; appears in the **DevPartner Fault Simulator** window during monitoring and following the completion of the fault simulation; see "What Does a Fault Instance Represent?" on page 20

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 "Can I Reuse Fault Sets?" on page 22

fault simulation

Alternative to traditional testing methodology; validates the robustness of the software application code; functional basis for DevPartner Fault Simulator

InfoCenter

Repository for DevPartner Fault Simulator online and PDF documentation; available from the **Start** menu on the Windows desktop; choose **Programs > Compuware DevPartner Fault Simulator > InfoCenter** (if you have the QA Edition, choose **InfoCenter** from **Compuware DevPartner Fault Simulator QA Edition**; also available from the **Help** menu in the standalone application)

IID

Parameter pertaining to COM environmental faults that designates the interface for a COM interface; see parameter

.NET 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 "Using .NET Faults to Simulate Thrown Managed Exceptions" on page 19

parameter

Variable you define that determines whether an environmental fault will be simulated during a session

ProgID

Parameter pertaining to COM environmental faults that designates the programmatic identifier for a COM method call; see parameter

QA Edition

New product edition available in DevPartner Fault Simulator; includes the standalone application and the command line interface; see "DevPartner Fault Simulator QA Edition" on page 11

resource

String that is associated with an observed activity; correlates to an environmental fault that can be used in a subsequent fault simulation (such as, `C:temp.txt` designates a file name resource); see activity, target application, and watch my target

skip count

Optional condition you set when configuring a fault descriptor; refers to the number of instances Fault Simulator will wait before it will simulate the associated fault

standalone application

Separate application and accompanying user interface available in DevPartner Fault Simulator; targeted to quality assurance engineers to help them validate applications under development; see "Available as a Standalone Application" on page 10

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 75

target application

Represents the application selected for fault simulation; could also represent the application that Fault Simulator watches in order to generate environmental faults; see "Automatically Generating Environmental Faults" on page 29

TestPartner

Compuware product that can be used in conjunction with Fault Simulator to automate the testing of user interface-based applications; go to the TestPartner Web site for more information

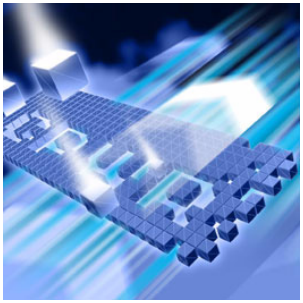
vroot

Used to identify the IIS virtual root part of the URL for a Web application or service

watch my target

Represents a new feature in the standalone application where Fault Simulator can watch your target application and record program activities that it observes as you use the program normally; from that data Fault Simulator will generate environmental faults that you can use in a subsequent fault simulation; see "Automatically Generating Environmental Faults" on page 29

Index



E

F

G

I

L

M

N

W

X