Getting Ahead with
Rational DevelopmentDeskTop

support@rational.com
http://www.rational.com

RATIONAL
SOFTWARE CORPORATION

IMPORTANT NOTICE

COPYRIGHT NOTICE

U.S. GOVERMENT RIGHTS NOTICE

TRADEMARK NOTICE

U.S. PATENT NOTICE

Printed in the U.S.A.

Contents

Rational DevelopmentDeskTop—working more productively

Rational DevelopmentDeskTop™ brings together five essential tools that work with each other and with Microsoft Visual Studio to help you accomplish the critical tasks involved in software development faster and more efficiently:

- **Purify**® An automatic error detection tool for finding run-time errors and memory leaks in every component of your program.
- **Visual PureCoverage**™ A code coverage tool for making sure your code is thoroughly tested before you release it.
- **Visual Quantify**™ A performance analysis tool for pinpointing performance bottlenecks so your program can run faster.
- **Rational Visual Test**® An automated testing tool for developing reusable, extensible test components. Use it with Purify, Visual PureCoverage, and Visual Quantify to maximize the value of each test run.
- **ClearQuest**™ A change request management tool for staying on top of software changes throughout the life cycle of a project.

This guide provides a brief overview of each Rational DevelopmentDeskTop tool, including tips on how the tools integrate with Visual Studio to help you work more productively.

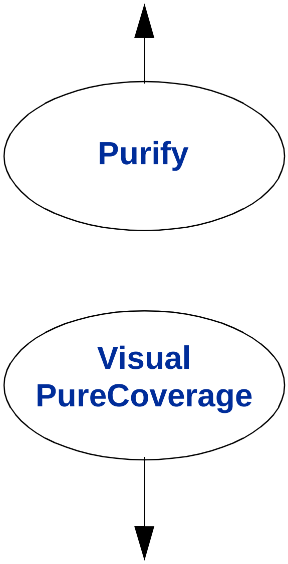
Note: Rational DevelopmentDeskTop tools integrate with your Visual Studio editor, debugger, and source-code management system. You can also use Rational DevelopmentDeskTop tools as stand-alone applications when you don’t need the resources of Visual Studio.

Tips for development engineers

Here are a few tips for using Rational DevelopmentDeskTop to develop fast, clean code.

Find memory errors early

Use Purify with Developer Studio to find the internal errors your tests don't reveal. These errors don't always show up right away, but they're the ones that will make your program crash someday!



Improve code coverage

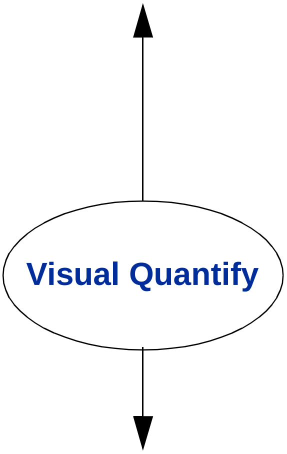
Use Visual PureCoverage to make sure you're exercising all your code during testing.

To find all the memory errors in your code, you need to exercise all your code when you use Purify. Visual PureCoverage can tell you if you're exercising your code sufficiently for Purify to find all the memory errors.

Prevent performance bottlenecks

Whenever you write new code or modify existing code, use Visual Quantify right away to catch any incremental performance losses before they turn into bottlenecks.

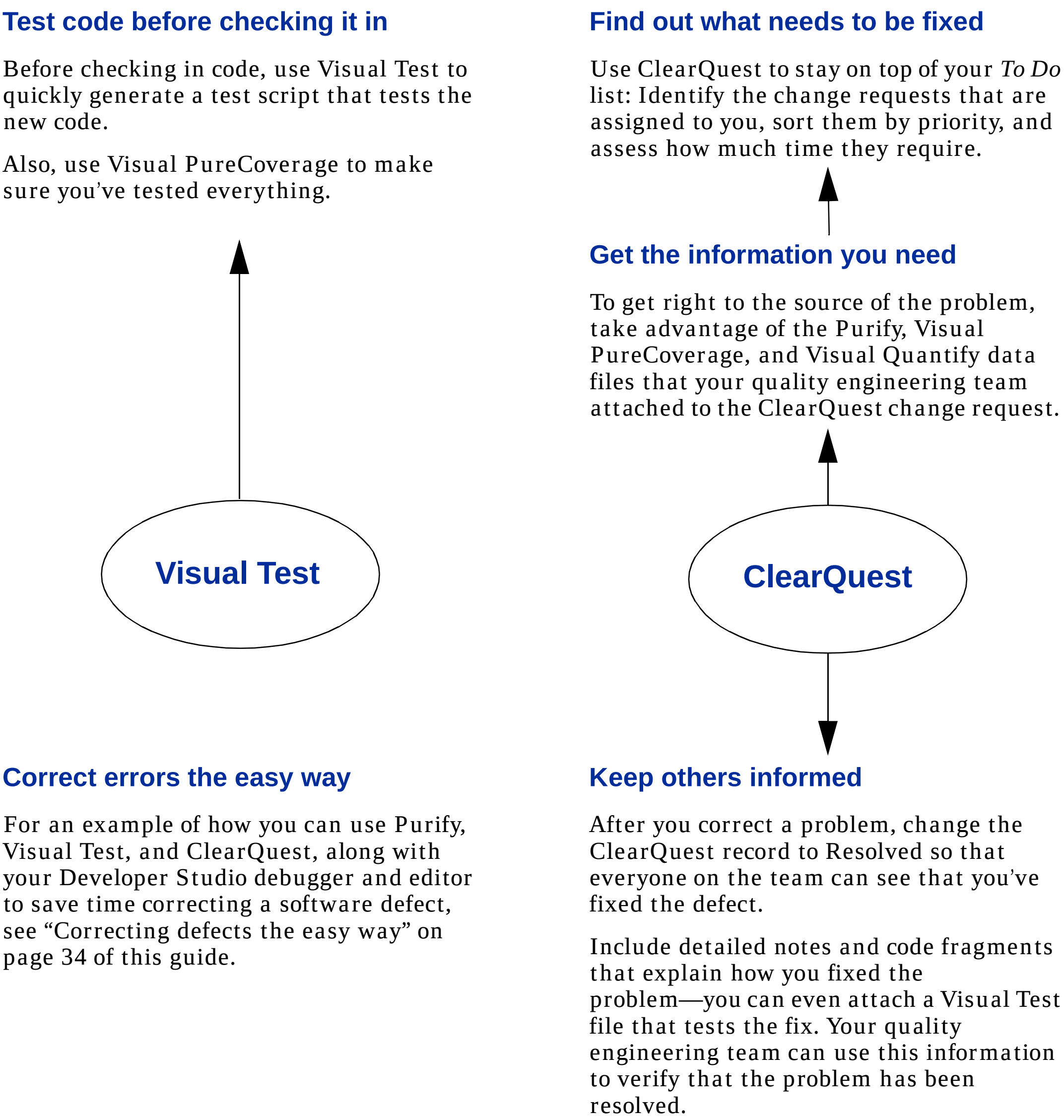
Visual Quantify gives you the insight you need to write more efficient code. It can turn everyone on your team into a performance engineer.



Improve code performance

A common reason for writing new code is to improve the performance of a program. But how can you effectively improve the performance of code that might have been developed over several years by many different people?

Use Visual Quantify not only to find performance bottlenecks, but also to learn more about how your code is structured. It will help you to make effective performance improvements.



Test code before checking it in

Before checking in code, use Visual Test to quickly generate a test script that tests the new code.

Also, use Visual PureCoverage to make sure you’ve tested everything.

Correct errors the easy way

For an example of how you can use Purify, Visual Test, and ClearQuest, along with your Developer Studio debugger and editor to save time correcting a software defect, see “Correcting defects the easy way” on page 34 of this guide.

Find out what needs to be fixed

Use ClearQuest to stay on top of your *To Do* list: Identify the change requests that are assigned to you, sort them by priority, and assess how much time they require.

Get the information you need

To get right to the source of the problem, take advantage of the Purify, Visual PureCoverage, and Visual Quantify data files that your quality engineering team attached to the ClearQuest change request.

Keep others informed

After you correct a problem, change the ClearQuest record to Resolved so that everyone on the team can see that you’ve fixed the defect.

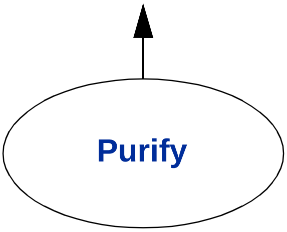
Include detailed notes and code fragments that explain how you fixed the problem—you can even attach a Visual Test file that tests the fix. Your quality engineering team can use this information to verify that the problem has been resolved.

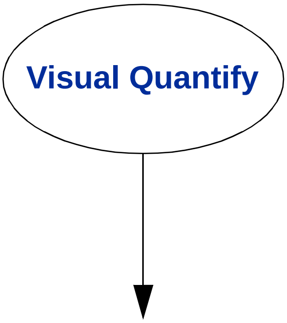
Tips for test engineers

Here are a few tips for using Rational DevelopmentDeskTop to guarantee quality software.

Find the internal errors in your code

For best results, run all your tests on a Purify'd version of your program. This will find the internal errors that your external functionality tests can't uncover.



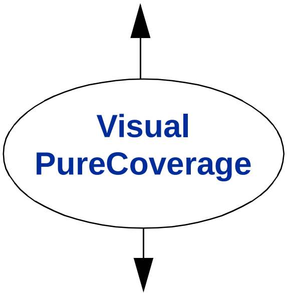


If performance suddenly drops . . .

If performance drops, this was probably caused by the most recent code checked in. Let Visual Quantify show you which parts of your program became slower compared to a previous run that had acceptable performance.

Test all your code daily

Use Visual PureCoverage every day to make sure you're testing all your code. With ongoing coverage feedback, you can be sure your tests are keeping pace with your code development.



If code coverage goes down. . .

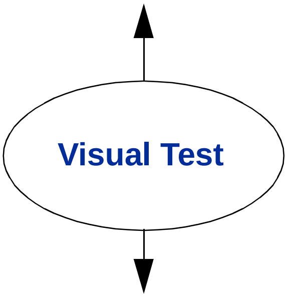
If code coverage drops, it might be an indication that new code is not being exercised by your existing tests. Or, the new code might have introduced a defect that's causing a large section of code not to be tested. Use Visual Test to write test cases that exercise the new code.

If performance suddenly increases . . .

An unexpected increase in performance can indicate that a large part of your code is no longer being exercised. Compare the most recent Visual PureCoverage results with a previous run that had acceptable performance to see if you're still getting the same amount of coverage.

Automate your tests

Successful testing requires that tests be repeatable: You need to be able to run the same tests on the same programs every night and get the same results. Use Visual Test to automate your testing.



Visual Test

Maximize the value of your tests

Run Visual Test scripts with Purify to perform external testing and look for internal errors at the same time. Purify can help you diagnose test failures.

Run your tests nightly with Visual PureCoverage to make sure you're testing all your code.

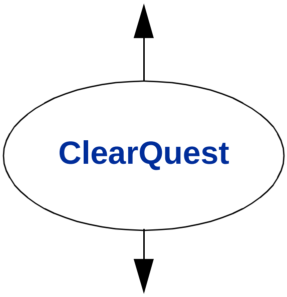
Run your tests with Visual Quantify to track performance improvement and regression.

Power-user tip: Too many tests?

If you have too many tests in your automatic test harness, you can use Visual PureCoverage to determine which tests are providing the highest level of code coverage. You can consider running these tests regularly and the others less often.

Determine release readiness

As a project leader, you can quickly evaluate the status of your project: How many high-priority defects are there; who's available to fix them; and can you meet your release date?



ClearQuest

Report defects immediately

When you find a defect, submit a change request to ClearQuest directly from Purify, Visual PureCoverage, or Visual Quantify, without interrupting your work.

You can make correcting problems easier by attaching a Visual Test script that re-creates the problem, along with Purify, Visual PureCoverage, and Visual Quantify data files.

Verify that defects are fixed

After a team member resolves a defect, a quality engineer should double-check that the problem is fixed. Then, change the ClearQuest record to Verified so everyone can see that the resolution is verified.

To make sure that the problem doesn't reappear, incorporate the Visual Test case used to verify the fix into your nightly test harness.

More information? For a complete overview of each Rational DevelopmentDeskTop tool, see the following books:

- *Getting Ahead with Purify*
- *Getting Ahead with Visual PureCoverage*
- *Getting Ahead with Visual Quantify*
- *Rational Visual Test Tour*
- *Getting Ahead with ClearQuest*

For detailed information and step-by-step instructions, see the online Help for each of the Rational DevelopmentDeskTop tools.

Purify—finding memory errors

Run-time errors and memory leaks are some of the most difficult errors to locate and the most important to correct. That's because they often remain undetected until triggered by some random event, so that a program can appear to work correctly when it's actually working only by accident.

Purify is the fastest and most comprehensive run-time error detection tool available for Visual C/C++ programs. Purify can find memory errors in every component of your program—even when you don't have the source code. With Purify, you can:

- Detect hard-to-find errors such as array bounds errors, accesses through dangling pointers, uninitialized memory reads, memory allocation errors, and memory leaks.
- Customize error detection for each component in your program.
- Use your Developer Studio debugger for just-in-time debugging.
- Refine error tracking with Purify API functions.
- Integrate Purify into Visual Test scripts, Perl scripts, makefiles, and batch files.
- Submit change requests directly to ClearQuest for errors reported by Purify, without interrupting your work in Purify.

More information? For a complete list of the errors Purify detects, select **Purify > Help > Purify Messages**.

Using Purify

To get the most out of Purify, begin using it as soon as your code is ready to run, and continue using it regularly throughout your development cycle.

1 Open your project in Developer Studio, then click 🔍 to engage Purify.

2 Build and execute your program as usual, using commands from the Developer Studio **Build** menu.

Note: To get the maximum detail in Purify messages, build your program so that debug and relocation data are available.

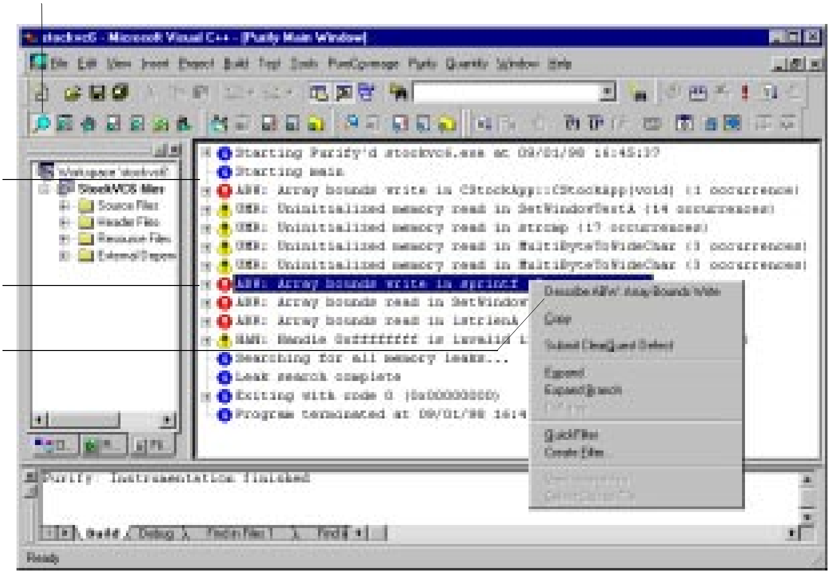
As you run your program, Purify displays run-time errors and memory leaks in the Purify window. The condensed outline format of the Purify window makes it easy to identify the critical errors in your program.

Click to engage Purify

Purify displays messages as the program runs

Acronyms like ABW identify the type of message

For a description of the message type, right-click the message, then select **Describe**



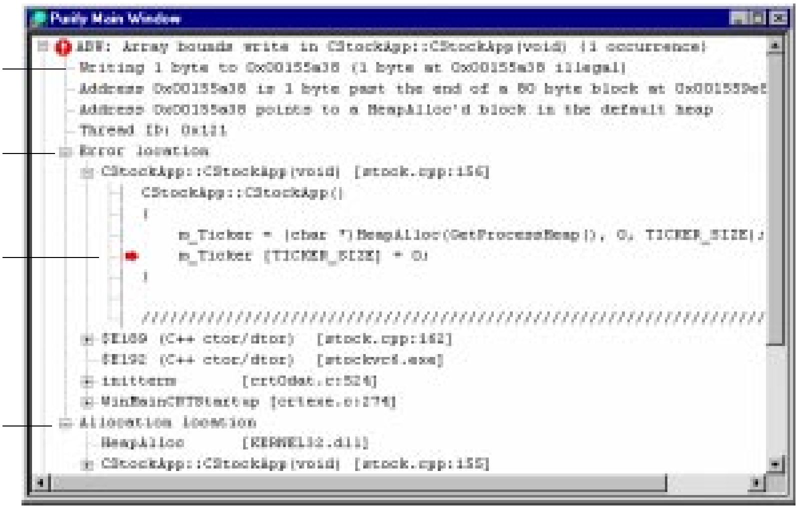
```
Starting Purify'd stockvc6.exe at 09/01/98 14:45:37
Starting main
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
UMR: Uninitialized memory read in SetWindowTextA {14 occurrences}
UMR: Uninitialized memory read in strcmp {17 occurrences}
UMR: Uninitialized memory read in MultiByteToWideChar {3 occurrences}
UMR: Uninitialized memory read in MultiByteToWideChar {3 occurrences}
ABW: Array bounds write in sprintf
ABR: Array bounds read in SetWindow
ABR: Array bounds read in lstrlenA
BAH: Handle 0xffffffff is invalid i
Searching for all memory leaks...
Leak search complete
Exiting with code 0 (0x00000000)
Program terminated at 09/01/98 14:4
```

```
Purify Instrumentation finished
```

You can filter Purify messages in order to display only the messages that are most important to you.

Analyzing and correcting errors

You can expand Purify messages to pinpoint *where* errors occur and to get the diagnostic information you need to analyze *why* they occur. Here's an example of an expanded Array Bounds Write (ABW) message.



```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
  Writing 1 byte to 0x00155a38 (1 byte at 0x00155a38 illegal)
  Address 0x00155a38 is 1 byte past the end of a 80 byte block at 0x001559e8
  Address 0x00155a38 points to a HeapAlloc'd block in the default heap
  Thread ID: 0x121
  Error location
    CStockApp::CStockApp(void) [stock.cpp:156]
        CStockApp::CStockApp()
        {
            m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_SIZE);
            m_Ticker [TICKER_SIZE] = 0;
        }

        //////////////////////////////////////////////////////////////////////
    $E189 (C++ ctor/dtor)  [stock.cpp:162]
    $E192 (C++ ctor/dtor)  [stockvc6.exe]
    initterm         [crt0dat.c:524]
    WinMainCRTStartup [crtexe.c:274]
  Allocation location
    HeapAlloc        [KERNEL32.dll]
    CStockApp::CStockApp(void) [stock.cpp:155]
```

The location in memory where the error occurs

Call stack showing the function calls leading to the error

The location of the error

Call stack showing the function calls leading to the allocation of the memory block associated with the error

Tip: When you find a critical error, you can submit a change request to ClearQuest without interrupting your work. Right-click the Purify message and select **Submit ClearQuest Defect** from the shortcut menu. You can include the entire error message, along with detailed notes to help the developer responsible for fixing the error to resolve it more easily. For more information, see page 31 of this guide.

Purify makes it easy to correct your source code: Just double-click the line where the error occurs to open your source code in the Developer Studio editor, positioned at the exact location of the error. After correcting your source code, rebuild your program, then run it again with Purify engaged in order to verify your corrections.

Tip: In order for Purify to find all the errors in your code, you have to exercise all your code. To make sure you are exercising every line of code, use Visual PureCoverage on your program.

More information? To learn more about Purify, including how to filter messages, compare program runs, set breakpoints on errors, use just-in-time debugging, and customize error detection, read *Getting Ahead with Purify*. For detailed information, see the Purify online Help.

Visual PureCoverage—checking all your code

To effectively test an application, you need to know which parts of the application were exercised during a test run and which ones were missed. Without this information, you can waste valuable time editing, compiling, and debugging your software without actually testing the critical problem areas.

With Visual PureCoverage, you can quickly and easily identify the gaps in your testing of Visual C/C++, Visual Basic, and Java programs. With Visual PureCoverage, you can:

- Identify functions, procedures, or methods that are being missed or only partially exercised during a test run, and even locate the individual lines of source code that are being missed.
- Customize data collection for individual program modules.
- Merge coverage statistics for multiple runs.
- Integrate Visual PureCoverage into Visual Test suites, Perl scripts, makefiles, and batch files for continuous coverage monitoring.
- Submit change requests directly to ClearQuest without interrupting your work in Visual PureCoverage.

Visual PureCoverage is especially useful as a companion to Purify: It can tell you whether you are exercising your code sufficiently for Purify to find all of your memory errors—and it’s essential to an automated testing environment. For more information, read “Maximizing the value of your tests” on page 25 of this guide.

Note: Visual PureCoverage is integrated into Developer Studio and Visual Basic. The examples in this chapter illustrate how to use it in Developer Studio.

Using Visual PureCoverage

Use Visual PureCoverage whenever you add new code to a program or modify existing code.

1. Open your project in Developer Studio, then click to engage Visual PureCoverage.

2. Build and execute your program as usual, using commands from the Developer Studio **Build** menu.

   Note: In order to get detailed line-by-line data, build your program so that debug and relocation data are available.

3. Run your program in such a way that it exercises your new or changed code.

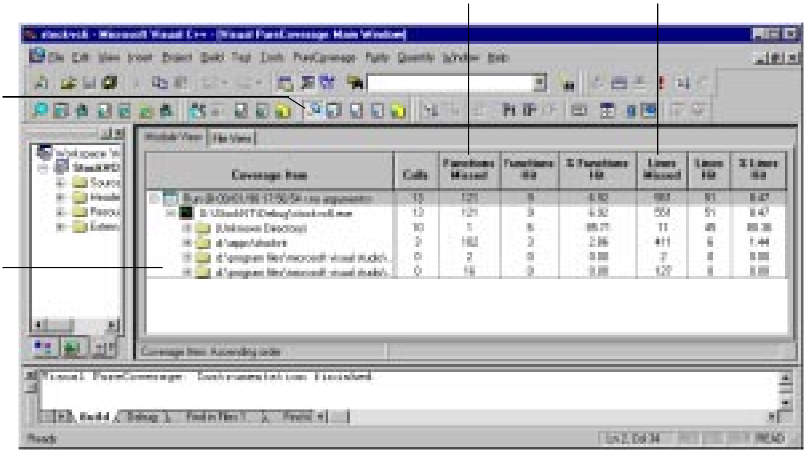
   When you exit your program, Visual PureCoverage provides an overview of the coverage for the entire run, indicating how many functions, methods, procedures, and lines were exercised and how many were missed.

Functions that were not exercised

Lines that were not exercised

Click to engage Visual PureCoverage

The Coverage Browser window shows coverage statistics for functions and lines


Microsoft Visual C++ - [Visual PureCoverage Main Window]
File Edit View Insert Project Build Tools Debug PureCoverage Purify Quantify Window Help
Workspace
Module View
File View
Build
Debug
Find in Files 1
Find in Files 2
Ready
Ln 2, Col 34
READ

Coverage Item	Calls	Functions Missed	Functions Hit	% Functions Hit	Lines Missed	Lines Hit	% Lines Hit
Run @ 00/01/98 17:50:54 (no arguments)	13	121	9	6.92	[illegible]	51	8.47
E:\Client\NT\Debug\[illegible].exe	13	121	9	6.92	551	51	8.47
[Unknown Directory]	10	1	6	85.71	11	45	80.36
d:\apps\[illegible]	3	102	3	2.86	411	6	1.44
d:\program files\microsoft visual studio\...	0	2	0	0.00	7	0	0.00
d:\program files\microsoft visual studio\...	0	16	0	0.00	122	0	0.00

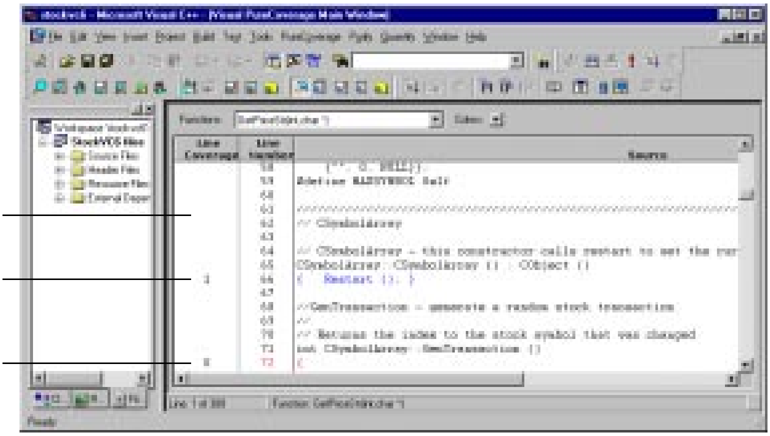
Coverage Item: Ascending order

Visual PureCoverage: Instrumentation finished

This same data can be displayed as a list of functions that you can sort to find the least-tested code in your program. Right-click in the Coverage Browser, then select **Function List** from the shortcut menu to display the Function List window.

Pinpointing untested code

The Visual PureCoverage Annotated Source window indicates the individual lines of code that were missed or only partially exercised during a run. Double-click a function in the Coverage Browser or Function List window to display the Annotated Source window.



The Annotated Source window displays a copy of your source code with line-by-line coverage annotations

This line was hit

This line was missed

Tip: To submit a change request to ClearQuest for a function that is not being adequately tested, right-click the function and select **Submit ClearQuest Defect** from the shortcut menu. For more information, see page 31 of this guide.

Once you identify the sections of your code that aren't being adequately tested, you can adjust your tests to cover your program more thoroughly. Then, rerun the program with Visual PureCoverage and compare runs to verify that coverage has improved. Visual PureCoverage automatically merges coverage data for multiple runs into an Auto Merge run that provides a valuable picture of the overall coverage for a program.

More information? To learn more about Visual PureCoverage, including how to filter coverage data, merge runs, and fine-tune data collection, read *Getting Ahead with Visual PureCoverage*. For detailed information, see the Visual PureCoverage online Help.

Visual Quantify—becoming a performance engineer

Visual Quantify quickly pinpoints performance bottlenecks in Visual C/C++, Visual Basic, and Java programs. It takes the difficulty and guesswork out of performance tuning by delivering accurate, repeatable timing data for all the components of your program, even when you don't have the source code. With Visual Quantify, you can:

- Quickly see how much time functions, procedures, or methods are costing.
- Understand the function-call architecture of your program so you can make effective performance improvements.
- Get right to the source of bottlenecks with detailed tabular views and line-by-line timing data.
- Fine-tune the depth and speed of data collection.
- Incorporate Visual Quantify into Visual Test suites, Perl scripts, makefiles, and batch files.
- Submit performance change requests directly to ClearQuest without interrupting your work in Visual Quantify.

Visual Quantify gives you the insight you need to write more efficient code and make any program run faster. It can turn everyone on your team into a performance engineer.

Note: Visual Quantify is integrated into Developer Studio and Visual Basic. The examples in this chapter illustrate how to use it in Developer Studio.

Using Visual Quantify

As soon as you add a new feature, use Visual Quantify to make sure that you haven't slowed the performance of your program.

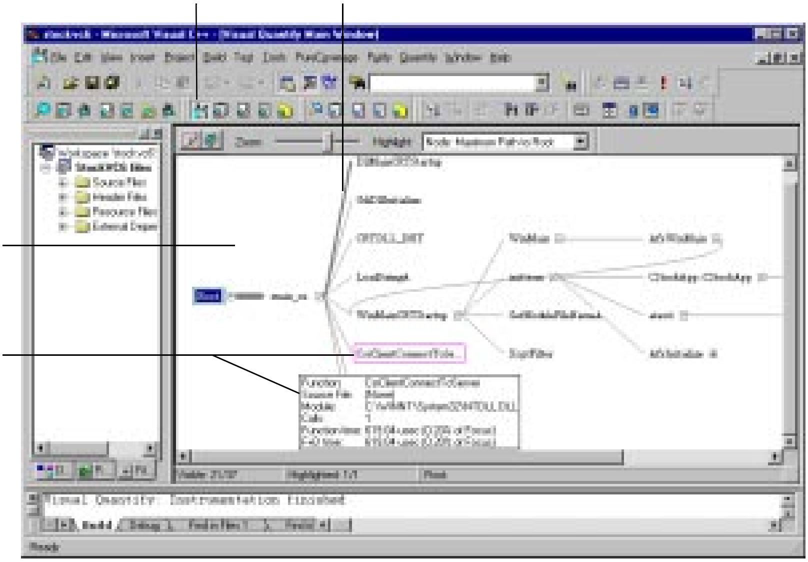
1 Open your project in Developer Studio, then click to engage Visual Quantify.

2 Build and execute your program as usual, using commands from the Developer Studio **Build** menu.

Visual Quantify's initial display is a call graph showing the 20 most time-consuming functions, procedures, or methods in your program. It shows you exactly where your code is least efficient.

Click to engage Visual Quantify

Thicker lines indicate more expensive paths



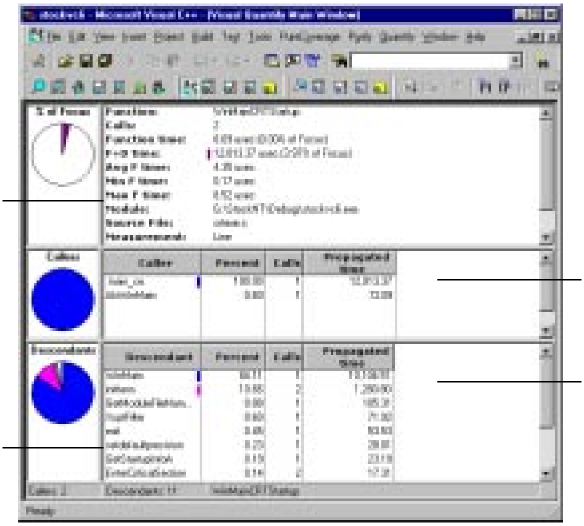
Visual Quantify's call graph provides an overview of your program's calling structure

Pause the cursor over a function to see additional information about the function

You can use the shortcut menu in the call graph to explore the calling structure of your program. For example, right-click a function, then select **Subtree > Focus on Subtree** from the shortcut menu to remove everything from the call graph except the subtree for that function. Then select **Expand** from the shortcut menu to show all the function's descendants, that is, all the functions it called.

Zeroing in on bottlenecks

After first orienting you in your program's calling structure, Visual Quantify helps you zero in on the bottlenecks in your code. To get more information about a function that appears to be too costly, double-click the function to display the Function Detail window.



Detailed data for a function

Double-click a caller or descendant function to display data for that function

Data about the calls made to the function

Data about the calls made by the function

Visual Quantify provides two additional performance analysis windows: Right-click in a Visual Quantify window, then select **Switch to > Function List** from the shortcut menu to see the numerical data for functions and their descendants, or select **Switch to > Annotated Source** to display line-by-line performance data.

Tip: When you locate a function with poor performance, you can immediately submit a change request to ClearQuest. Right-click the function and select **Submit ClearQuest Defect** from the shortcut menu. For more information, see page 31 of this guide.

Comparing runs to find performance changes

Adding new features to your code comes with the risk of slowing down your program. Before you add a new feature, run the program first with Visual Quantify to establish a *base run*, that is, a run with an acceptable standard of performance. Save the results of this run to a Visual Quantify data file (`.qfy`).

After adding the new feature, run the program again with Visual Quantify and compare the new run with the base run—you'll quickly see any performance changes. For information on how to compare runs, look up *diff* in the Visual Quantify online Help index.

More information? To learn more about Visual Quantify, including how to project performance improvements, interpret source-code annotations, compare program runs, and fine-tune data collection, read *Getting Ahead with Visual Quantify*. For detailed information, see the Visual Quantify online Help.

Rational Visual Test—automating your tests

To guarantee the quality of a complex application that consists of many components, you not only need to continually test new code, you also need to make sure that the new code doesn't break something that worked before. A thorough testing program involves functional tests (performed manually or by automated test scripts) and regression tests: comparing today's results with acceptable past standards to see what's changed.

If your tests aren't automated, you must manually test and retest every possible usage scenario for your program. When the program fails a test, you must remember the exact sequence of events leading up to the failure.

Using Rational Visual Test, you can rapidly create, manage, run, and debug tests for applications of any size, created with any development tool. With Visual Test, you can:

- Write test cases in the powerful Test language, which includes hundreds of built-in procedures that you can use to test your applications.
- Automatically generate Test language code by using the Scenario Recorder.
- Run multiple test cases and organize test cases into suites with the Suite Manager.
- Use Visual Test with Purify, Visual PureCoverage, and Visual Quantify to maximize the value of each test run.
- Attach Visual Test scripts to ClearQuest change requests.

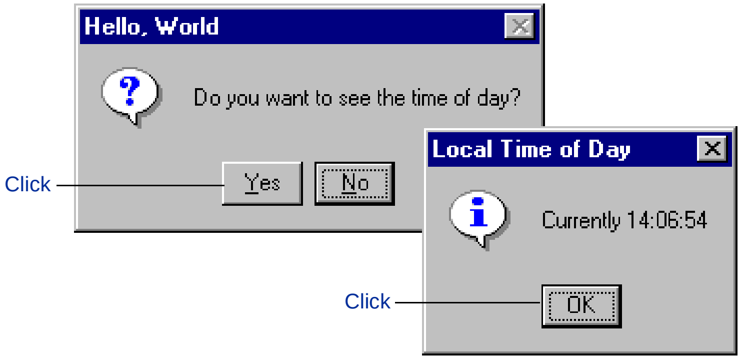
As soon as an application has passed the early-development stage—when the screen components and keystroke and mouse sequences are stable—you're ready to automate your tests with Visual Test.

Using Visual Test

A good place to begin using Visual Test is with the Scenario Recorder—it allows you to automatically generate Test language code simply by exercising your program. For example, let's say you just added a dialog to a Hello World program that displays the time of day. To quickly test this new feature:

1 In Visual Test, select **Test > Scenario Recorder**, and type a name for the scenario. For example, `hellotest`.

2 Exercise the new feature. For example:



Hello, World

Do you want to see the time of day?

Local Time of Day

Currently 14:06:54

The Visual Test Scenario Recorder records the sequence of mouse clicks and keystrokes as you exercise the feature, and automatically generates Test code that will repeat those actions.

3 Click **Scenario Recorder** in the Taskbar, then click **Stop and create scenario**.

4 To run the test script, select **Test > TestDebug > Go**.

Maximizing the value of your tests

On its own, a test script can verify the external functionality of your program. By running a Visual Test script in conjunction with Purify, Visual PureCoverage, or Visual Quantify, you can also monitor the internal behavior of your program at the same time. You can watch for memory errors, untested code, or performance changes that might otherwise go unnoticed during an external functionality check.

You can run a Visual Test script with Purify, Visual PureCoverage, or Visual Quantify from within Developer Studio or from a test harness—and you can do it without altering the format of the Visual Test script.

Working in Developer Studio

To use the Visual Test script hellotest.mst created in the previous example to exercise hello.exe and to also report Purify data during the test run:

1. Include the tools.inc file in the Visual Test script:

   ```
   ' $include 'tools.inc'
   ```

2. Add the following run statement to the Visual Test script:

   ```
   run hello.exe, nowait
   ```

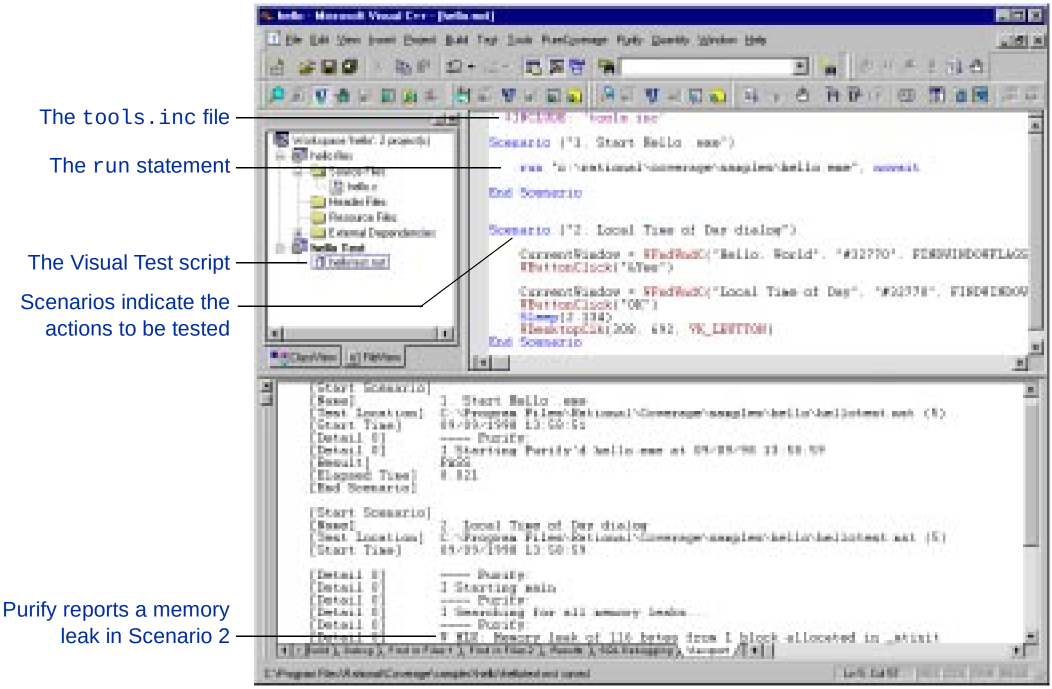
   With this run statement, you can use this same script to run Purify, Visual PureCoverage, or Visual Quantify—and at other times to run the script without these tools.

3. Select **Purify > Run Visual Test Scripts with Purify.**

   This causes Visual Test to set the environment variable VT_RUN_TOOL_OVERRIDE=PURIFY which overrides the nowait option in the run statement with the option Purify. Similar menu items are available for running Visual Test scripts with Visual PureCoverage and Visual Quantify.

4 Run the test by selecting **Test > TestDebug > Go**.

Purify automatically instruments and runs `hello.exe`, then the test script exercises the instrumented program. Below is the output from this test run showing that Purify found a Memory Leak (MLK).


The tools.inc file
The run statement
The Visual Test script
Scenarios indicate the actions to be tested
Purify reports a memory leak in Scenario 2
hello - Microsoft Visual C++ - [hello.mst]
File Edit View Insert Project Build Test Tools PureCoverage Purify Quantify Window Help
Workspace 'hello': 2 project(s)
hello files
Source Files
hello.c
Header Files
Resource Files
External Dependencies
hello Test
hellotest.mst
ClassView
FileView
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Viewport

```
'$INCLUDE: 'tools.inc'
Scenario ("1. Start Hello.exe")
    run "c:\rational\coverage\samples\hello.exe", nowait
End Scenario

Scenario ("2. Local Time of Day dialog")
    CurrentWindow = WFndWndC("Hello, World", "#32770", FINDWINDOWFLAGS
    WButtonClick("&Yes")

    CurrentWindow = WFndWndC("Local Time of Day", "#32770", FINDWINDOW
    WButtonClick("OK")
    Sleep(2.334)
    WDesktopClk(200, 692, VK_LBUTTON)
End Scenario
```

```
[Start Scenario]
[Name]            1. Start Hello.exe
[Test Location]   C:\Program Files\Rational\Coverage\samples\hello\hellotest.mst (5)
[Start Time]      09/09/1998 13:50:52
[Detail 0]        ---- Purify
[Detail 0]        I Starting Purify'd hello.exe at 09/09/98 13:50:59
[Result]          Pass
[Elapsed Time]    0.321
[End Scenario]

[Start Scenario]
[Name]            2. Local Time of Day dialog
[Test Location]   C:\Program Files\Rational\Coverage\samples\hello\hellotest.mst (5)
[Start Time]      09/09/1998 13:50:59

[Detail 0]        ---- Purify
[Detail 0]        I Starting main
[Detail 0]        ---- Purify
[Detail 0]        I Searching for all memory leaks...
[Detail 0]        ---- Purify
[Detail 0]        W MLK: Memory leak of 116 bytes from 1 block allocated in _atinit
```

Purify, Visual PureCoverage, and Visual Quantify windows are not displayed when you run these tools from a Visual Test script; however, the results of the runs are automatically saved to data files (Purify `.pfy` file, Visual PureCoverage `.cfy` file, Visual Quantify `.qfy` file). You can open a data file at any time and analyze it as you normally would. For example, select the `hello.pfy` file and drag it to Developer Studio to open it and analyze the complete error messages in Purify.

Tip: When a test uncovers a problem, you can attach the Visual Test `.mst` file along with Purify, Visual PureCoverage, and Visual

Quantify data files to a ClearQuest Submit Defect form in order to help the developer responsible for fixing the defect reproduce the problem. For more information, see “Attaching files to a change request” on page 32 of this guide.

Working with a test harness

You can run a Visual Test script with Purify, Visual PureCoverage, or Visual Quantify in a nightly test harness. For example, to run the `hellotest.mst` script with Purify:

1. Include the `tools.inc` file in the Visual Test script and use the following format for the Visual Test `run` statement:

   ```
   run hello.exe, nowait
   ```

2. Set the following environment variable at the command line:

   ```
   set VT_RUN_TOOL_OVERRIDE=PURIFY
   ```

   To set the environment variable to Visual PureCoverage or Visual Quantify, specify `COVERAGE` or `QUANTIFY`.

3. Run the Visual Test script:

   ```
   mt hellotest.mst
   ```

   Purify automatically instruments and runs `hello.exe`, then the `hellotest.mst` script exercises the instrumented program.

   As an alternative, if you want to always run a Visual Test script with Purify, Visual PureCoverage, or Visual Quantify, you can explicitly specify the option `Purify`, `Coverage`, or `Quantify` in the `run` statement. For example: `run hello.exe, Purify`. If you do this, you do not need to set an environment variable. Just type `mt hellotest.mst` to run the test script.

   Note: Purify automatically saves the results of each Visual Test run to a Purify data file (`.pfy`). The `toolsamp.inc` file contains sample Test code that you can use to delete these data files when no serious errors are reported by Purify.

Targeting specific types of Purify errors

You can edit the `MyPurifyHandler` error handler in the `toolsamp.inc` file in order to focus on specific Purify messages. For example, you can have Purify report only error messages, not informational or warning messages. See your Visual Test documentation for information on targeting specific text strings.

Tip: Purify imposes some overhead that can slow down your tests. If this is a problem, try running Purify on only part of your tests each night. For example, use Purify on a fourth of your tests on Monday, on another fourth on Tuesday, and so on. This way, you can be sure you're running Purify on all your code each week.

Making sure you're testing everything

You'll want to run your nightly tests with Visual PureCoverage in order to gauge how well your test suite is keeping pace with the evolution of your code. With ongoing feedback from Visual PureCoverage, you can guarantee that every code modification is thoroughly tested before your program is released.

Visual PureCoverage automatically saves the results of each Visual Test run to a data file (`.cfy`). To analyze this file in Visual PureCoverage, just drag it to Developer Studio to open it.

Anticipating performance changes

It's not uncommon for a program that's been performing well to become sluggish over time. You can anticipate this problem by incorporating Visual Quantify into your nightly tests. When you notice a change in performance, open the Visual Quantify data file (`.qfy`) that was automatically saved that night and use Visual Quantify's Diff feature to compare it to a data file from a previous run that had acceptable performance.

More information? To learn more about Visual Test, including how to use the Test language, build test projects, and run test suites, read *Rational Visual Test Tour*. For detailed information, see the Visual Test online Help.

ClearQuest—managing software changes

A large software development project can generate hundreds, or even thousands, of defects and change requests spread over a continually changing code base. So many change requests, each with the potential to impact multiple products, versions, and platforms, can strain even the most capable development team.

With ClearQuest, you can manage every type of change activity associated with software development, including enhancement requests, defect reports, and documentation modifications. Everyone on your development team can benefit from using ClearQuest:

- ***Development engineers*** can identify high-priority action items and get the information they need to fix problems fast.
- ***Test engineers*** can easily track the origin, status, and resolution of every change request.
- ***Project managers*** can get the metrics they need to accurately determine the overall quality and stability of a project.
- ***Database administrators*** can integrate ClearQuest with existing tools and customize it to fit your business practices.
- ***Off-site team members*** can be part of the team with ClearQuest Web.

ClearQuest includes the `defaultapp` schema and `SAMPL` database that provide a ready-to-use change-request management (CRM) system. You can easily customize this CRM system by using the ClearQuest Designer. For more information, read *Getting Ahead with ClearQuest.*

Using ClearQuest

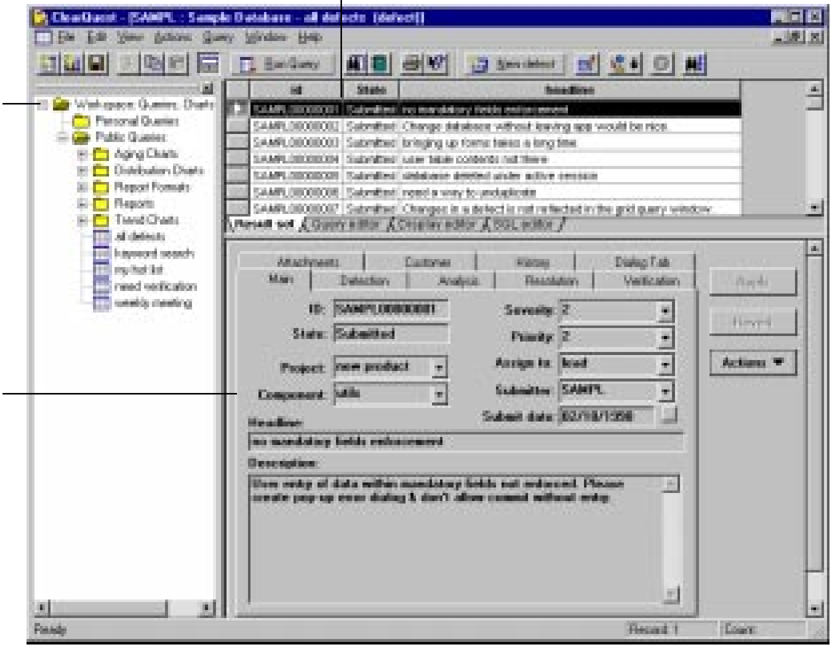
Note: Before using ClearQuest, you must install the ClearQuest server. For instructions, see *Getting Ahead with ClearQuest*.

The ClearQuest main window consists of a Workspace, a Query Builder, and a Record Form.

The Query Builder is where you create queries and view query results. Click a record to display its data in the record form below.

The Workspace lists the built-in queries, charts, and reports, along with any additional ones that you create

The Record Form displays data for the selected record



ClearQuest - [SAMPL : Sample Database - all defects (defect)]

id	State	Headline
SAMPL00000001	Submitted	no mandatory fields enforcement
SAMPL00000002	Submitted	Change database without leaving app would be nice
SAMPL00000003	Submitted	bringing up forms takes a long time
SAMPL00000004	Submitted	user table contents not there
SAMPL00000005	Submitted	database deleted under active session
SAMPL00000006	Submitted	need a way to unduplicate
SAMPL00000007	Submitted	Changes in a defect is not reflected in the grid query window

ID: SAMPL00000001
State: Submitted
Project: new product
Component: utils
Severity: 2
Priority: 2
Assign to: lead
Submitter: SAMPL
Submit date: 02/18/1998
Headline: no mandatory fields enforcement
Description: User entry of data within mandatory fields not enforced. Please create pop up error dialog & don't allow commit without entry

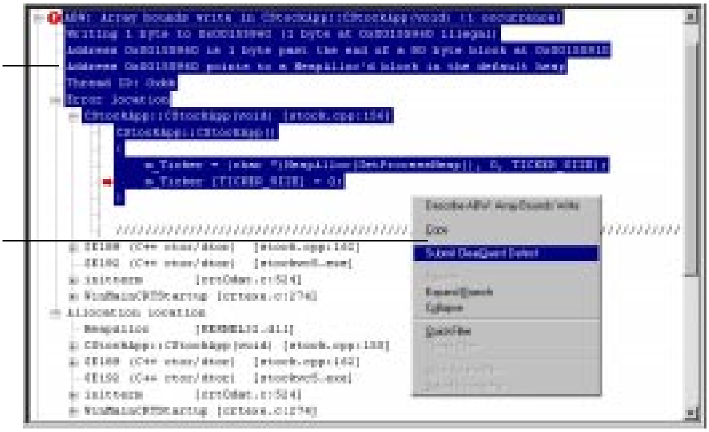
A ClearQuest change request is a record consisting of all the data related to it. ClearQuest includes many built-in queries that you can use to quickly locate records based on project or component, assigned engineer, defect severity, and so on.

You can easily create new queries or modify existing ones. For example, you might create a query called *mydefects* that you can run daily to find all the change-request records that are assigned to you.

Submitting a change request

You can submit a change request by clicking **New defect**

You can also submit a change request directly from Purify, Visual PureCoverage, and Visual Quantify. For example, to submit a change request from the Purify window in Developer Studio:



```
ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}
Writing 1 byte to [illegible] (1 byte at [illegible] illegal)
Address [illegible] is 1 byte past the end of a [illegible] byte block at [illegible]
Address [illegible] points to a HeapAlloc'd block in the default heap
Thread ID: [illegible]
Error location
  CStockApp::CStockApp(void) [stock.cpp:156]
      CStockApp::CStockApp()
      {
          m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0, TICKER_SIZE);
  =>      m_Ticker [TICKER_SIZE] = 0;
      }
  CE100 (C++ ctor/dtor) [stock.cpp:142]
  CE102 (C++ ctor/dtor) [stockvc5.exe]
  initterm        [crt0dat.c:524]
  WinMainCRTStartup [crtexe.c:274]
Allocation location
  HeapAlloc       [KERNEL32.dll]
  CStockApp::CStockApp(void) [stock.cpp:138]
  CE100 (C++ ctor/dtor) [stock.cpp:142]
  CE150 (C++ ctor/dtor) [stockvc5.exe]
  initterm        [crt0dat.c:524]
  WinMainCRTStartup [crtexe.c:274]
```

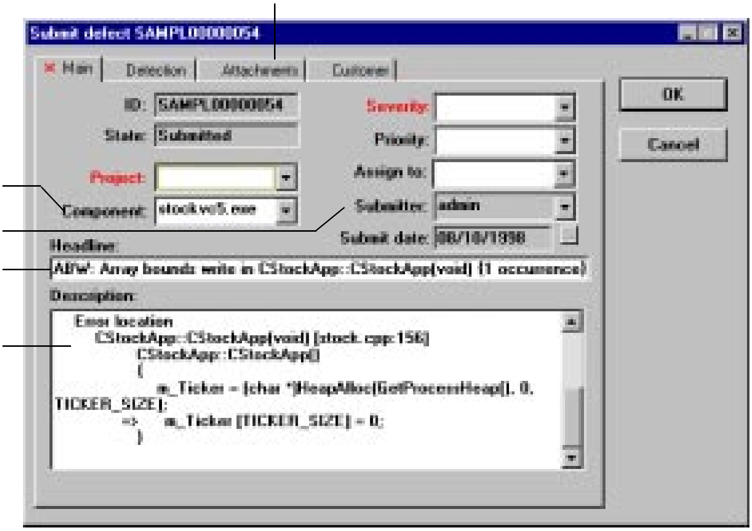
Select all or part of a Purify message

Then right-click to display the shortcut menu

Select **Submit ClearQuest Defect** to submit a change request for the error

The ClearQuest Submit Defect form appears, with fields automatically filled in with data from the selected Purify message.

Click to attach additional files



Submit defect SAMPL00000054

ID: SAMPL00000054

State: Submitted

Project:

Component: stockvc5.exe

Severity:

Priority:

Assign to:

Submitter: admin

Submit date: 08/10/1998

Headline: ABW: Array bounds write in CStockApp::CStockApp(void) {1 occurrence}

Description:

```
  Error location
      CStockApp::CStockApp(void) [stock.cpp:156]
             CStockApp::CStockApp()
             {
                 m_Ticker = (char *)HeapAlloc(GetProcessHeap(), 0,
TICKER_SIZE);
          =>      m_Ticker [TICKER_SIZE] = 0;
             }
```

The program name

Your user ID

The Purify message line

The text you highlighted

Attaching files to a change request

Select **Attachments** to attach Purify, Visual PureCoverage, Visual Quantify, and Visual Test files to the change request. These attachments help the developer assigned to fix the error reproduce the conditions that caused it.

Tip: In the Detection tab, ClearQuest automatically fills in **Detection Method** with the words *Purify*, *Visual PureCoverage*, or *Visual Quantify*. This makes it easy to locate similar types of records. For example, you can quickly locate all the memory-related errors simply by querying ClearQuest for a list of errors reported by Purify.

Working with change requests

Once a change request is submitted, everyone on your team can track it throughout the development cycle. As your software develops, you will move change requests through various "states." For example, when it's first submitted, the change request is in the Submitted state. In each state, you can perform actions such as Modify, Open, or Close that move the change request to other states, concluding with Resolved or Verified.

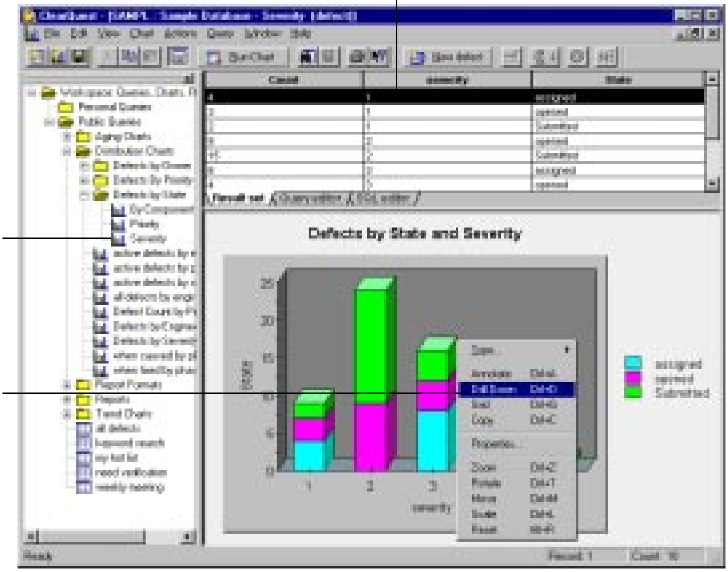
States, actions, and the fields that appear on record forms are all defined by ClearQuest's built-in CRM system, which your ClearQuest administrator can easily customize to fit your way of working.

Monitoring the status of your project

ClearQuest provides predefined charts and reports so you can see the status of your project at a glance. It's easy to modify charts and reports and to create your own in order to get the metrics you need to accurately schedule your release dates.

For example, you can see how the workload is currently distributed among the engineers on your team by running a chart that displays the defects by assigned engineer. Or, you can see the defect records graphed by their state and severity.

Chart data is also displayed in tabular form



Double-click a predefined chart

Right-click in the chart to display the shortcut menu, then select **Drill Down** to show more detail

More information? To learn how to use ClearQuest's built-in CRM system, customize the interface, and generate charts and reports from ClearQuest data, read *Getting Ahead with ClearQuest*. For detailed information, see the ClearQuest online Help.

Correcting defects the easy way

Correcting a serious defect is never easy, but using Rational DevelopmentDeskTop can make it easier. Here's an example.

Getting the information you need

Suppose you query ClearQuest for the high-priority change requests that are assigned to you and find a serious defect that must be fixed immediately. The last thing you want to do now is to waste time trying to reproduce an error that someone else has reported.

Begin by reading the ClearQuest record carefully. It contains the complete history of the reported problem. And, if the person who submitted the change request also attached a Purify data file (`.pfy`) and a Visual Test script file (`.mst`), it makes understanding and correcting the problem much easier. Now you can get to work:

1. In Developer Studio, open the workspace for your project.

2. Select the attached Purify data file (`.pfy`) and drag it to Developer Studio to open it.

3. Expand the reported Purify error message to see the exact location of the error.

4. Double-click the error call chain to open the source code in your Developer Studio editor.

5. Use your Developer Studio debugger to set a breakpoint on the error location or to enable Purify's Break On Error feature, then exercise the program to reproduce the error. When the program stops at the breakpoint, you can debug the error.

6. After you correct the error, recompile the program and rerun it with Purify to verify that the error is indeed corrected.

7. Just to make sure, rerun the original Visual Test script with Purify to verify that the corrected program no longer commits the error.

Keeping others informed

After correcting the problem, change the ClearQuest record to Resolved so that the rest of the team knows that you've fixed the defect. Another team member can now independently verify that you've fixed the defect, then close the ClearQuest record.

Making sure that defects don't reappear

To make sure that a defect doesn't reappear, write a Visual Test test case that checks the resolution of the original defect, then incorporate the test case into your nightly test harness. In this way, you can build up a library of reusable tests that will assure the ongoing quality of your software.

Where to go from here

Now that you see how Rational DevelopmentDeskTop tools can help you get ahead, give them a try. To get off to a good start, take a look at the introductory product guides listed on page 10 of this guide. They cover the basics of using each tool and provide useful pointers to information in the online Help.

Index