Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Copyright Statement

Chapter 6: Testing Code for Errors, Inefficiencies, and Performance

Testing, testing, testing. It certainly isn't the *only* way to improve the quality of your code, but testing is definitely one of the most important tools we have as developers to produce higher-quality code. Although testing can't help improve things like not following best practices and coding standards, it can help catch two of the main problems that end users perceive as poor application quality: bugs and performance. In this chapter, I'll take a look at tools and techniques that can be used to spot more code errors, detect inefficient code, and home in on performance problems.

Overview of Testing Techniques

When I began testing the first commercial-quality code I ever wrote (a retail point-of-sale system written for Windows 95), *testing* was simply running the application through as many scenarios as I could think of, checking to see whether anything broke, and checking to make sure I got the correct results for whatever input I provided. That's really still the essence of testing, but I've since learned to be a lot more formal and systematic about it.

One problem with the way many developers test—just running the application as if they were an end user—is that we developers *aren't* usually our application's end users. We tend to use an application the way it was *meant* to be used; we don't actively try to break it, we don't approach it from the same viewpoint as our end users, and we'll almost never use it in exactly the same way as the application's end users.

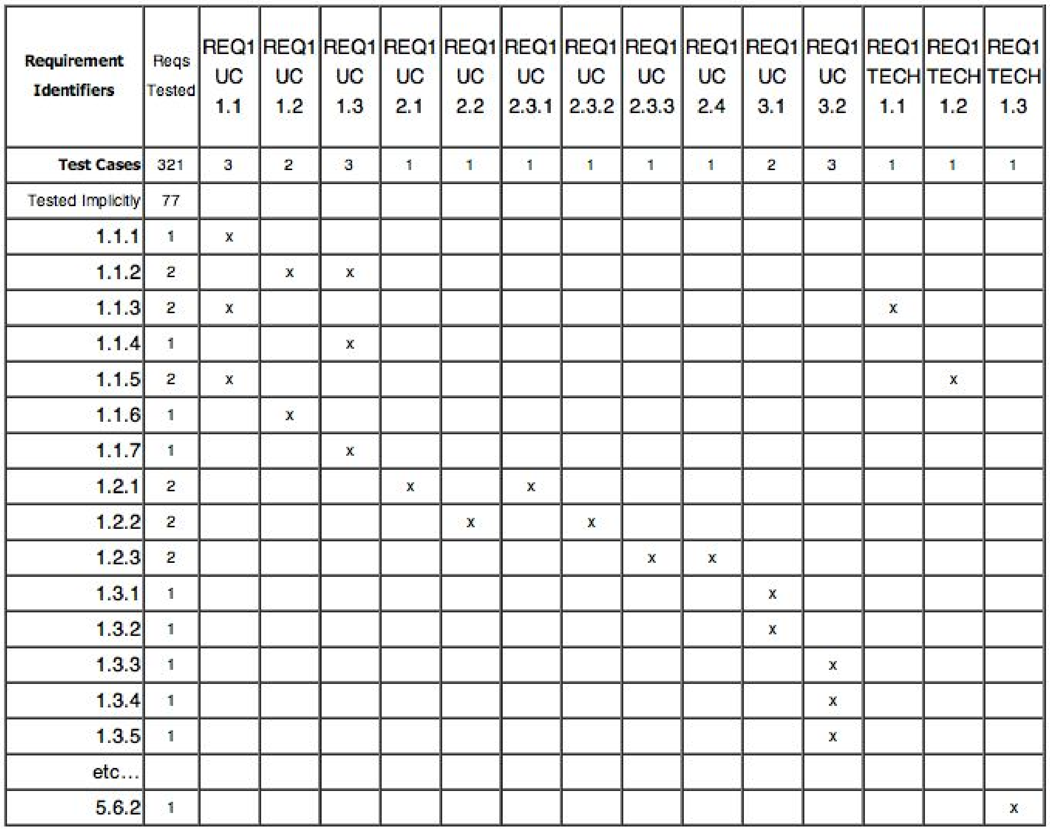
For example, in writing my retail application, I spent hours testing the software's ability to dial-in to our corporate headquarters via modem (this was 1995—none of our retail locations had a connection better than a plain old telephone line) and upload its daily transactions. With a lot of work, I had the process working smoothly every time. However, I never anticipated that one of our end users might unplug the modem—and the first time someone did, the application crashed. I came to understand that unplugging the computer's modem was actually pretty common in the field because the store used a single phone line for that modem, for the credit card authorization terminal, and in some cases for a fax machine and back-office phone line. Store personnel would often unplug one or more devices to improve line quality or to troubleshoot a line problem, then forget to plug something back in later. My viewpoint as a developer didn't include that scenario, so I didn't test for it—and so our stores had a reason to perceive my code as poor quality because they saw it crashing fairly frequently until I fixed the problem and rolled out an update.

The moral of the story is that testing techniques are important, and though they may frequently seem like adding extra work for no reason, there *is* a valid reason: to help bring more of the end users' viewpoint into the testing process. That unique viewpoint can help spot performance problems, logic errors, and other problems in code that might otherwise go unnoticed.

Construction of Test Cases

One of the chief tools for incorporating the end users' viewpoint is the construction of test cases. Test cases help ensure that software can be consistently run through every practical scenario—like a critical device being unplugged—with specific inputs so that the software's behavior, performance, and output can be analyzed and verified.

Test case definition should, as with most of software development, start in the requirements. The application's requirements should describe the conditions under which the software will be used, most of the major usage scenarios and workflows, and so forth. Formal test cases will include a workflow, a list of inputs, and a description of the expected behavior and outputs. Because these test cases are driven by the requirements, it's important to track which test case goes with each requirement, something that's often done in a *traceability matrix.* Figure 6.1 shows an example matrix, with requirements as the columns and test cases as the rows. You'll notice that a single test case is capable of testing multiple requirements, which the matrix helps to document in an easy-to-read format.



Requirement Identifiers	Reqs Tested	REQ1 UC 1.1	REQ1 UC 1.2	REQ1 UC 1.3	REQ1 UC 2.1	REQ1 UC 2.2	REQ1 UC 2.3.1	REQ1 UC 2.3.2	REQ1 UC 2.3.3	REQ1 UC 2.4	REQ1 UC 3.1	REQ1 UC 3.2	REQ1 TECH 1.1	REQ1 TECH 1.2	REQ1 TECH 1.3
Test Cases	321	3	2	3	1	1	1	1	1	1	2	3	1	1	1
Tested Implicitly	77														
1.1.1	1	x													
1.1.2	2		x	x											
1.1.3	2	x											x		
1.1.4	1			x											
1.1.5	2	x												x	
1.1.6	1		x												
1.1.7	1			x											
1.2.1	2				x		x								
1.2.2	2					x		x							
1.2.3	2								x	x					
1.3.1	1										x				
1.3.2	1										x				
1.3.3	1											x			
1.3.4	1											x			
1.3.5	1											x			
etc…															
5.6.2	1														x

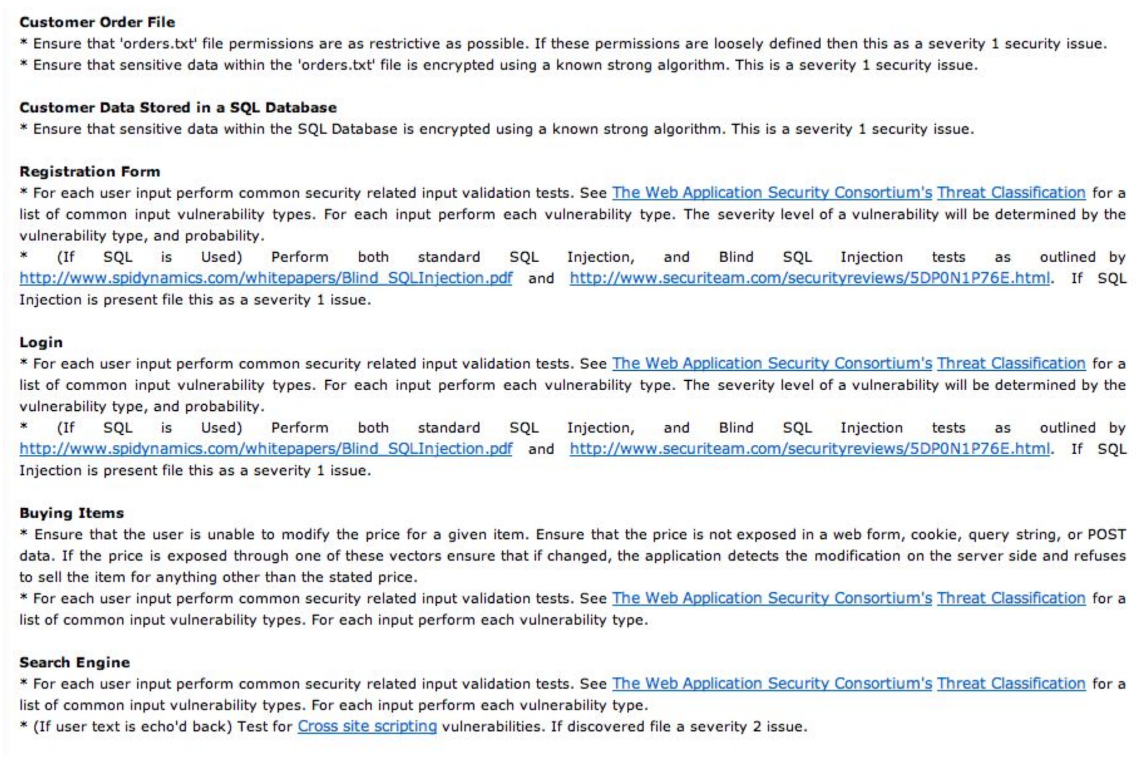
Figure 6.1: Example requirements traceability matrix.

However, test cases shouldn't *stop* with the requirements. Additional test cases can be added to the lineup by any developer who writes code that needs to be tested in a specific fashion. In my example, I should have added a specific test case for situations in which the modem is unplugged simply because I was writing code that depended upon some factor (the modem) outside my control. I obviously wouldn't expect the "unplugged" test case to result in successful operation of the software, but I *would* expect the test case to end gracefully, with the software perhaps prompting the operator to attach the modem.

Test cases are often (and should be) documented. Each test case is typically accompanied by standardized information such as:

- A test case ID, which helps everyone on the project team refer to specific test cases without ambiguity
- A description of what is being tested
- Information on the order in which the test case should be executed
- Any requirements that the test case has, such as prerequisite software, hardware, or data
- Information on the inputs that the case uses as well as the expected outputs
- Information about whether the test case has been included in an automated testing cycle or if it needs to be performed manually
- Instructions for completing the test case, including step-by-step directions

As Figure 6.2 shows, test cases may start out as simple descriptions that eventually evolve into a more fully-documented format. This simple description may derive from requirements or may be contributed by developers who want to ensure that specific conditions or dependencies are thoroughly tested.



Customer Order File
* Ensure that 'orders.txt' file permissions are as restrictive as possible. If these permissions are loosely defined then this as a severity 1 security issue.
* Ensure that sensitive data within the 'orders.txt' file is encrypted using a known strong algorithm. This is a severity 1 security issue.

Customer Data Stored in a SQL Database
* Ensure that sensitive data within the SQL Database is encrypted using a known strong algorithm. This is a severity 1 security issue.

Registration Form
* For each user input perform common security related input validation tests. See The Web Application Security Consortium's Threat Classification for a list of common input vulnerability types. For each input perform each vulnerability type. The severity level of a vulnerability will be determined by the vulnerability type, and probability.
* (If SQL is Used) Perform both standard SQL Injection, and Blind SQL Injection tests as outlined by http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf and http://www.securiteam.com/securityreviews/5DP0N1P76E.html. If SQL Injection is present file this as a severity 1 issue.

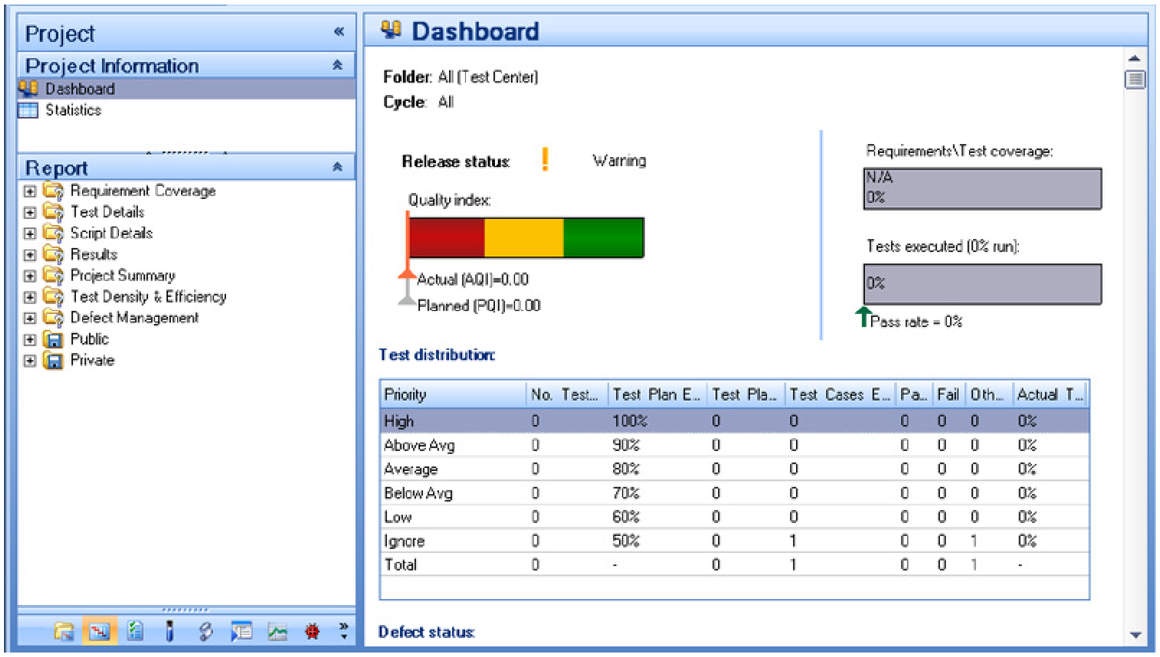
Login
* For each user input perform common security related input validation tests. See The Web Application Security Consortium's Threat Classification for a list of common input vulnerability types. For each input perform each vulnerability type. The severity level of a vulnerability will be determined by the vulnerability type, and probability.
* (If SQL is Used) Perform both standard SQL Injection, and Blind SQL Injection tests as outlined by http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf and http://www.securiteam.com/securityreviews/5DP0N1P76E.html. If SQL Injection is present file this as a severity 1 issue.

Buying Items
* Ensure that the user is unable to modify the price for a given item. Ensure that the price is not exposed in a web form, cookie, query string, or POST data. If the price is exposed through one of these vectors ensure that if changed, the application detects the modification on the server side and refuses to sell the item for anything other than the stated price.
* For each user input perform common security related input validation tests. See The Web Application Security Consortium's Threat Classification for a list of common input vulnerability types. For each input perform each vulnerability type.

Search Engine
* For each user input perform common security related input validation tests. See The Web Application Security Consortium's Threat Classification for a list of common input vulnerability types. For each input perform each vulnerability type.
* (If user text is echo'd back) Test for Cross site scripting vulnerabilities. If discovered file a severity 2 issue.

Figure 6.2: Example test case descriptions.

Project teams with a higher maturity level will often rely on tools to centrally manage test cases. For example, Figure 6.3 shows an example of Micro Focus QADirector, which provides a dashboard view that helps project members see which tests have been run and get a high-level feel for the application's overall quality—based on which tests have passed. Such tools can not only help keep track of which tests exist, which have passed, and so forth, but also provide management with useful tools for tracking the application's readiness for use and the time left to complete a higher-quality application.



Dashboard

Folder: All (Test Center)
Cycle: All

Release status: ! Warning

Quality index:
Actual (AQI)=0.00
Planned (PQI)=0.00

Requirements\Test coverage:
N/A
0%

Tests executed (0% run):
0%
Pass rate = 0%

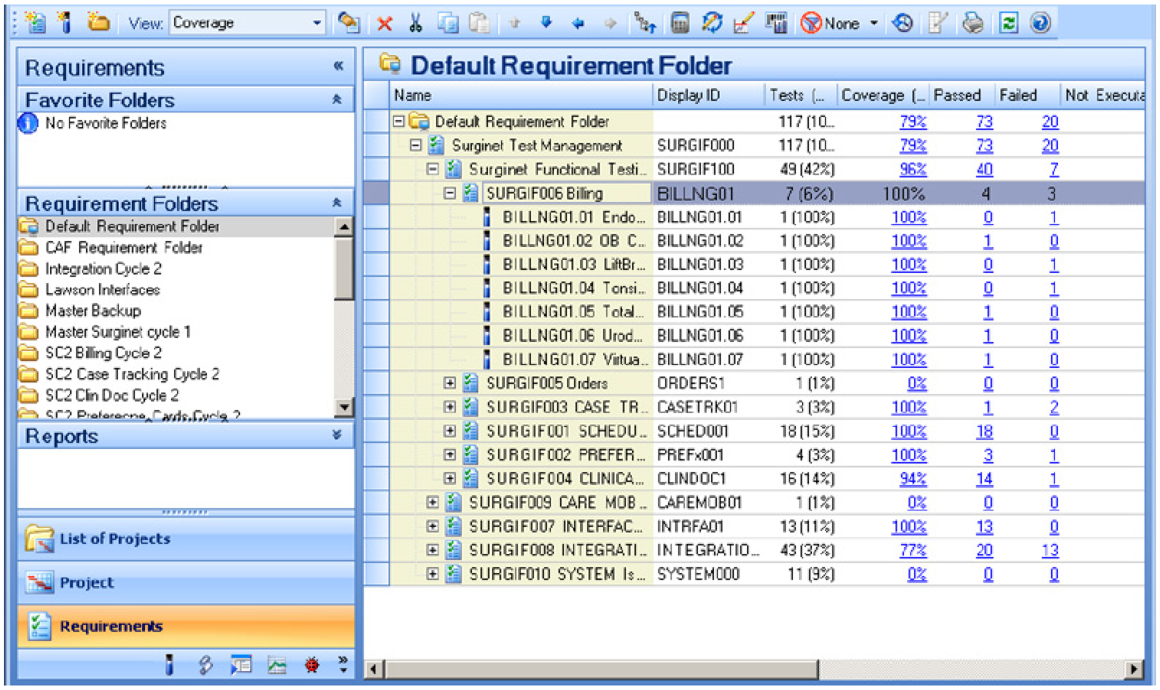
Test distribution:

Priority	No. Test...	Test Plan E...	Test Pla...	Test Cases E...	Pa...	Fail	Oth...	Actual T...
High	0	100%	0	0	0	0	0	0%
Above Avg	0	90%	0	0	0	0	0	0%
Average	0	80%	0	0	0	0	0	0%
Below Avg	0	70%	0	0	0	0	0	0%
Low	0	60%	0	0	0	0	0	0%
Ignore	0	50%	0	1	0	0	1	0%
Total	0	-	0	1	0	0	1	-

Defect status:

Figure 6.3: QADirector Dashboard.

Tools may make it easier to construct traceability matrices. Figure 6.4 shows QADirector's ability to track project requirements and connect them to specific tests, and to track which of those tests have been successfully completed. At a glance, you can see which requirements are not fully covered by existing test cases—indicating that more tests need to be created—and you can see which requirements are not yet met by the application.

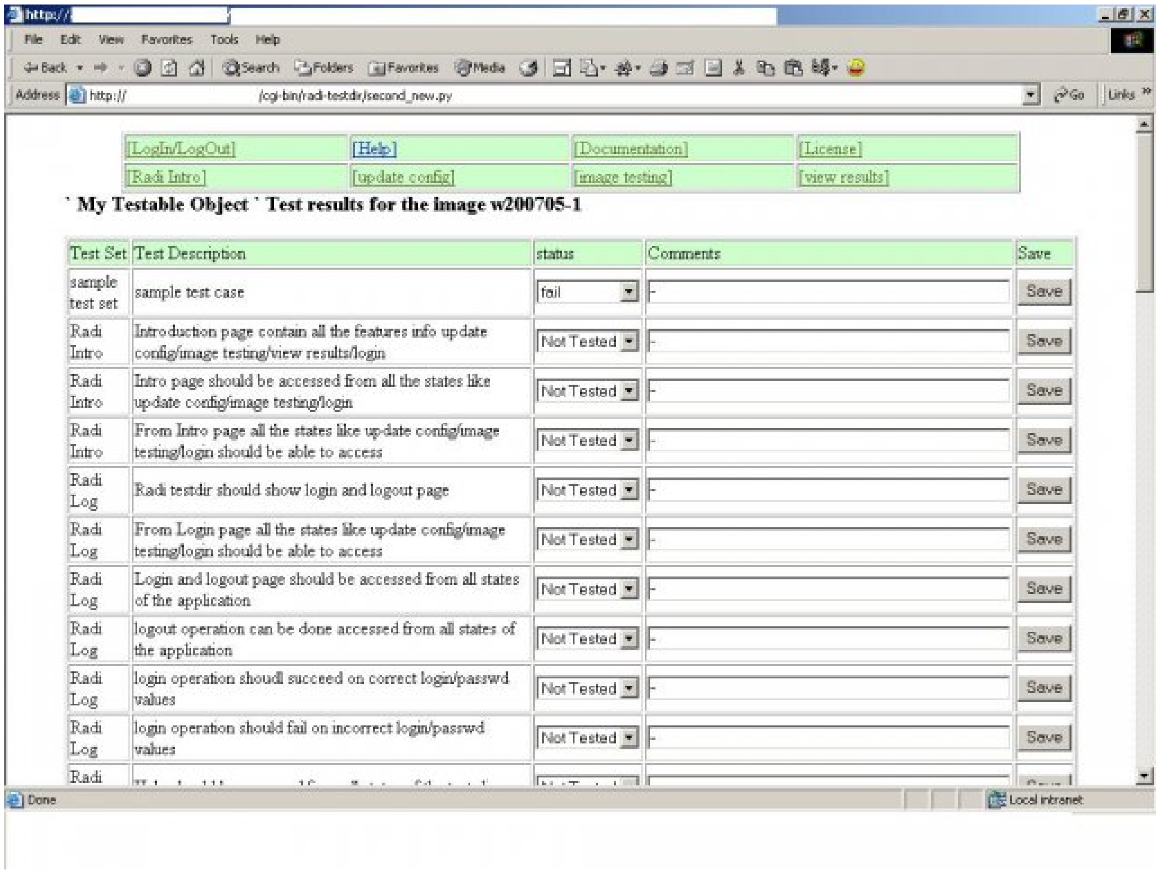


Default Requirement Folder

Name	Display ID	Tests (...	Coverage (...	Passed	Failed	Not Executa
Default Requirement Folder		117 (10...	79%	73	20	
Surginet Test Management	SURGIF000	117 (10...	79%	73	20	
Surginet Functional Testi...	SURGIF100	49 (42%)	96%	40	7	
SURGIF006 Billing	BILLNG01	7 (6%)	100%	4	3	
BILLNG01.01 Endo...	BILLNG01.01	1 (100%)	100%	0	1	
BILLNG01.02 OB C...	BILLNG01.02	1 (100%)	100%	1	0	
BILLNG01.03 LiftBr...	BILLNG01.03	1 (100%)	100%	0	1	
BILLNG01.04 Tonsi...	BILLNG01.04	1 (100%)	100%	0	1	
BILLNG01.05 Total...	BILLNG01.05	1 (100%)	100%	1	0	
BILLNG01.06 Urod...	BILLNG01.06	1 (100%)	100%	1	0	
BILLNG01.07 Virtua...	BILLNG01.07	1 (100%)	100%	1	0	
SURGIF005 Orders	ORDERS1	1 (1%)	0%	0	0	
SURGIF003 CASE TR...	CASETRK01	3 (3%)	100%	1	2	
SURGIF001 SCHEDU...	SCHED001	18 (15%)	100%	18	0	
SURGIF002 PREFER...	PREFx001	4 (3%)	100%	3	1	
SURGIF004 CLINICA...	CLINDOC1	16 (14%)	94%	14	1	
SURGIF009 CARE MOB...	CAREMOB01	1 (1%)	0%	0	0	
SURGIF007 INTERFAC...	INTRFA01	13 (11%)	100%	13	0	
SURGIF008 INTEGRATI...	INTEGRATIO...	43 (37%)	77%	20	13	
SURGIF010 SYSTEM Is...	SYSTEM000	11 (9%)	0%	0	0	

Figure 6.4: Tracing requirements to tests.

Test case management tools come in all shapes and sizes. Figure 6.5 shows a simpler open-source tool, Radi, which is designed to centralize test case definition and provide a central place for test results to be input and tracked. This Web-based interface offers a minimum level of capabilities for tracking test cases and test results but provides less management information and less comprehensive test case tracking than more full-featured, mature solutions.


http://
File Edit View Favorites Tools Help
Back Search Folders Favorites Media
Address http:// /cgi-bin/radi-testdir/second_new.py
Go Links
Done
Local intranet

[LogIn/LogOut]	[Help]	[Documentation]	[License]
[Radi Intro]	[update config]	[image testing]	[view results]

` My Testable Object ` Test results for the image w200705-1

Test Set	Test Description	status	Comments	Save
sample test set	sample test case	fail	-	Save
Radi Intro	Introduction page contain all the features info update config/image testing/view results/login	Not Tested	-	Save
Radi Intro	Intro page should be accessed from all the states like update config/image testing/login	Not Tested	-	Save
Radi Intro	From Intro page all the states like update config/image testing/login should be able to access	Not Tested	-	Save
Radi Log	Radi testdir should show login and logout page	Not Tested	-	Save
Radi Log	From Login page all the states like update config/image testing/login should be able to access	Not Tested	-	Save
Radi Log	Login and logout page should be accessed from all states of the application	Not Tested	-	Save
Radi Log	logout operation can be done accessed from all states of the application	Not Tested	-	Save
Radi Log	login operation shoudl succeed on correct login/passwd values	Not Tested	-	Save
Radi Log	login operation should fail on incorrect login/passwd values	Not Tested	-	Save
Radi				

Figure 6.5: Tracking test results in Radi.

Test cases form the basis of an overall testing plan: They define *what* must be tested, and may provide details on *how* those tests are to be conducted. Diving into the *how* brings us to the different types of tests your project may utilize.

Types of Testing

On the face of it, testing can seem complicated and even endless. The more complex an application, the more complex the associated tests become. In the end, though, most tests utilize a fairly limited set of core tests to determine whether an application passes that particular test. Although a complete list of these "test primitives" is beyond the scope of this guide, reviewing the following examples will give you an idea of what's commonly used:

- Logic and calculation testing: By providing certain known inputs to the application, you should be able to generate specific, known outputs. Checking the actual output against your test case expectation allows you to quickly determine whether all the underlying code is working properly. In my point-of-sale application, for example, the purchase of two specific products should always result in the same total amount for the transaction, inclusive of sales taxes; I could calculate that expected total on my own and compare it with what the application produced. Early on, my code was improperly rounding off the tax amount, producing a transaction total that was a penny off. Because the transaction-processing code was extensive and complicated, I needed to continually repeat tests such as this to ensure that recent changes hadn't introduced another calculation bug.
- Range checking: Given certain inputs, an application should produce specific output. Although it's important to test for the output created by specific inputs, it's also important to test across the entire range of allowed inputs—and to test outside that range. For example, if an application will accept monetary values, it's important to provide test inputs that check the entire range of expected inputs *and beyond.* Out-of-range inputs should be handled gracefully by the application, and should not result in a crash or undesired behavior. In my point-of-sale application, things worked great when product prices were less than $999.99; when the company sold its first high-end computer, however, the six-digit price threw off formatting on the receipt and resulted in "$1899.99" being printed as "$899.99," which is obviously a problem. When writing the receipt-formatting code, I should have documented my assumptions that the price field was a maximum of 5 digits, and created a test case that specifically tested for larger—*out of range*—prices.

Note

Most tests should come in pairs: One designed to check the proper operation of the software by feeding it valid inputs, and another designed to check the software's response to *improper* use—such as feeding it out-of-range inputs. Both types of tests, commonly called *positive* and *negative*, are equally important because you want to both verify the software's proper operation as well as explicitly try to break it.

- Random testing: Simply providing every possible value for a given input. This one would have helped improve the quality of my point-of-sale application. Our stock numbers when I started were four digits, but we allowed for five so that we'd have room to grow. The only in-use stock number greater than 9999 was 11111, which was a special number used when selling store gift certificates. However, by allowing five digits to be keyed, I introduced a problem: The underlying code was using a small integer type to store the number, and it could only accept values up to 32,768. That was far in excess of what we needed for normal operations, but entering 32,800—as a random test would eventually have done—crashed the application. We didn't find this problem until we'd created special numbers 22222 and 33333 for other special-handling items—and the first time someone used 33333, the system crashed.

Following this train of thought, you'll realize that there are a large number of fairly simplistic types of test that, when combined, can accommodate complex test cases and scenarios to thoroughly test the software. In fact, it's because these testing *primitives,* as I call them, are so individually simple that many such tests can be more easily automated for faster and more consistent repetition.

Test Management: Reporting, Monitoring, Tracking, and Resolution

Simply documenting and defining your tests, of course, isn't enough; you also need to retain the results of each test, monitor the progress of your testing, and resolve problems that have been uncovered by testing. These tasks drive most companies to adopt some kind of centralized test management system because without such a system test management can quickly become unwieldy.

Organizations differ in how they choose to approach test management. Let me describe the approach I worked with early in my career—an approach that's still pretty common, and which I honestly regard as a sort of worst-case scenario:

- Test cases were documented in Word documents. Each document described a specific test case, and yes, we actually printed them and worked from the hardcopies.
- Problems encountered during testing were documented in our Help desk ticketing system, under a category for whatever product was being tested. These tickets would be assigned to the lead developer, who would dole them out to his colleagues for resolution. Any anomalies from a single test case would be documented in a single ticket.
- A test case would not be re-run until any tickets related to it were closed—meaning a developer felt they had resolved the problems from previous test runs.

This approach is *functional*, in that it gets the job done, but it's not *practical.* The sheer amount of documentation lying all over the office was a testament to how much overhead we had to deal with. It was also impossible to give our managers any clear idea of how far along we were: We could point to the stack of "passed" test cases as progress, but you couldn't make any kind of meaningful measurements, unless you counted measuring the height of the stack itself ("we're 3-inches along, boss"). In *The Definitive Guide to Quality Application Delivery* (Realtime Publishers), one sign of quality maturity is described as management having "…useful metrics that give them insight into the project's current level of quality, at all times." We certainly couldn't provide that.

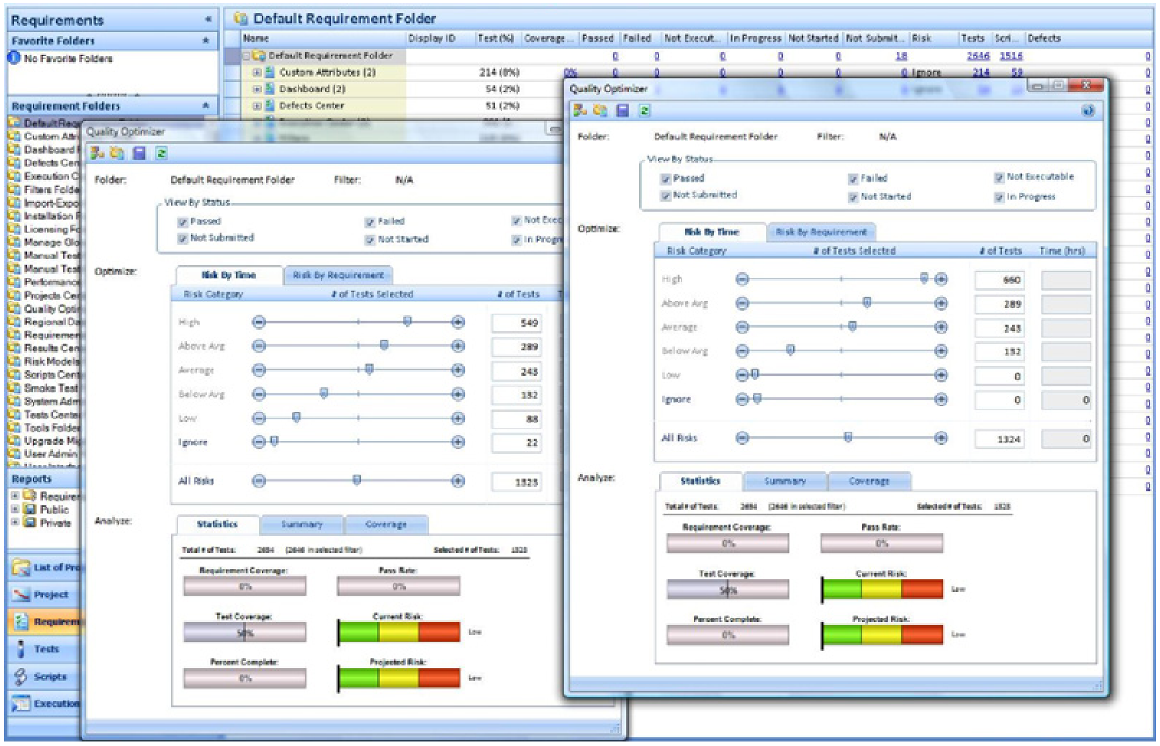
Modern test management tools bring everything together into one place—if you choose the right tool. For example, here are some subtle considerations:

- If your test management application can't directly document your application requirements and relate those to specific test cases, the tool isn't doing you much good. Application requirements will change over time; it's not enough to simply load them into the test management system once. You need to have a connection between the test management system and whatever is being used to capture business requirements. For example, Micro Focus does this with Optimal Trace for business requirements, which feeds their QADirector product for test case management; solutions from companies such as IBM and TestPine integrate in a similar fashion.
- Are you using automated testing? Many smart organizations are because automation leads to greater productivity and consistency. However, if your automated test system doesn't feed test results to your test management system, you're losing a lot of productivity by manually transcribing test results. Again, integrated solutions that can talk to one another, feeding test results into the test management system, are ideal.
- Your defect-tracking system should also integrate with your other tools. Ideally, defects from a test can be reported directly to that system so that the entire team can communicate with less effort.

Note
"Integrated" does not necessarily mean "all from one vendor;" many vendors offer integration points with popular solutions from *other* vendors.

Having all of this information available in a set of integrated solutions provides better management tools. The dashboard in Figure 6.3, for example, is a direct result of test results being fed into a central test management system that also integrates with the project's requirements-tracking system, giving managers a quick, high-level view of the project's quality progress.

More complex management information can also be made available. Figure 6.6, for example, shows how this aggregated information can be used for risk-based management decisions. Managers dealing with time and resource constraints can perform "what if" analyses, allowing them to change the number of tests scheduled for various risk-based categories. Riskier elements of the project might thus be assigned more testing, while less testing is performed for less-risky elements. The software can then help predict the overall impact, displaying test coverage information, projected project risk, and so forth. By rearranging the test plan in this fashion, fewer resources may be able to contribute the same level of quality to the project.



Folder: Default Requirement Folder Filter: N/A

View By Status: ☑ Passed ☑ Failed ☑ Not Executable ☑ Not Submitted ☑ Not Started ☑ In Progress

Risk Category	# of Tests	Time (hrs)
High	660	
Above Avg	289	
Average	243	
Below Avg	152	
Low	0	
Ignore	0	0
All Risks	1324	0

Total # of Tests: 2694 (2646 in selected filter) Selected # of Tests: 1328

Requirement Coverage: 0% Pass Rate: 0% Test Coverage: 50% Current Risk: Low Percent Complete: 0% Projected Risk: Low

Figure 6.6: Adjusting test workloads based on risk and resource availability.

Other analyses made possible by integrated test management systems can include:

- A view of the overall application quality
- Whether all requirements have been tested
- What tests have not been executed, and what the risk may be
- How much testing remains to be done, and how long it will likely take
- What each tester is currently working on
- How long it takes to correct defects discovered during testing

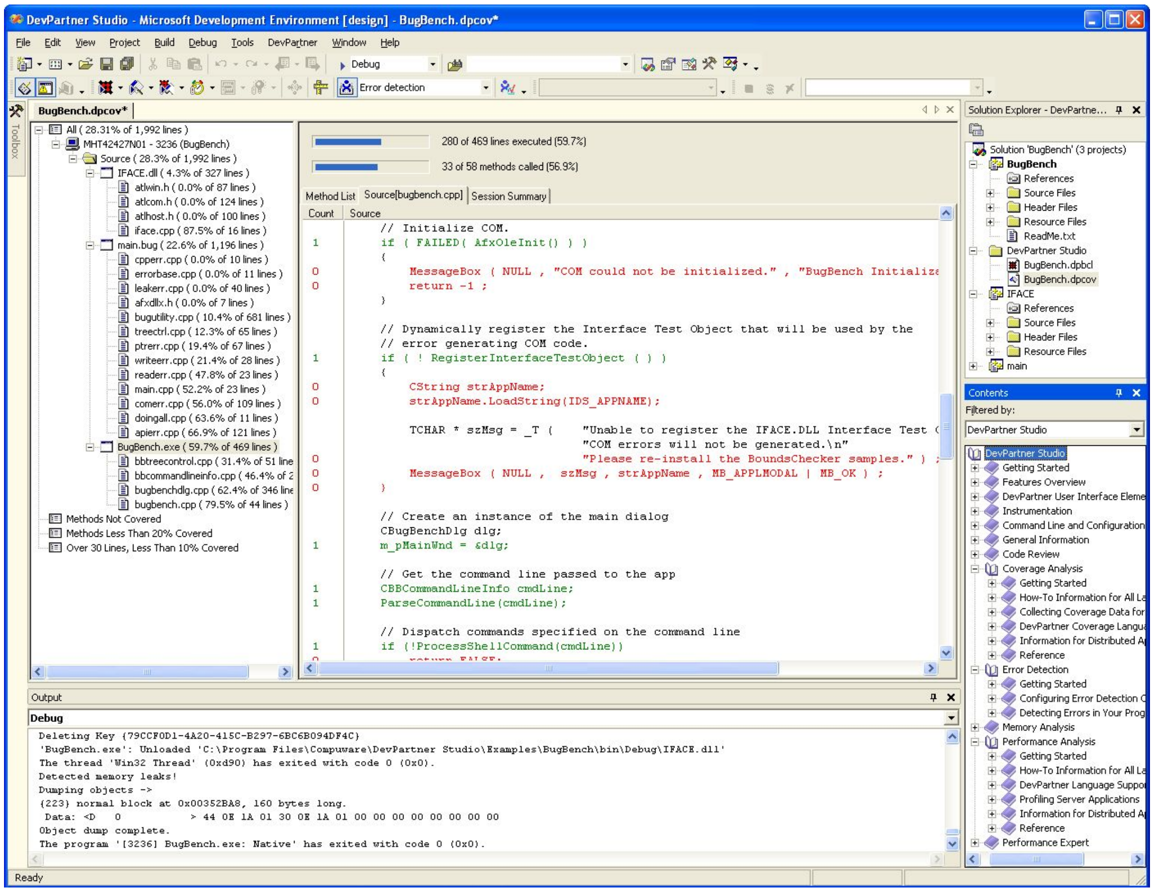
This information is invaluable to managers, and being able to provide this information *without* a lot of manual overhead is one sign of growing quality maturity within an organization.

Testing Tools

In addition to helping you better manage test cases, the right tools can make it easier to actually execute tests, determine whether you're testing the right things, and make it easier to resolve complex problems related to performance, errors, and even security. In the next several sections, I'll look at key capabilities a good testing suite should provide.

Code Coverage Analysis

There's a moment, just before a project team releases their application, where everyone asks themselves, "Did we test *everything?*" With a good code-coverage analysis, however, there's no need to ask the question: You'll know. Figure 6.7 shows what a code coverage analysis can look like.


DevPartner Studio - Microsoft Development Environment [design] - BugBench.dpcov*
File Edit View Project Build Debug Tools DevPartner Window Help
Debug
Error detection
Toolbox
BugBench.dpcov*
Method List
Source[bugbench.cpp]
Session Summary
Count
Source
Solution Explorer - DevPartne...
Contents
Filtered by:
DevPartner Studio
Output
Debug
Ready

```
All ( 28.31% of 1,992 lines )
  MHT42427N01 - 3236 (BugBench)
    Source ( 28.3% of 1,992 lines )
      IFACE.dll ( 4.3% of 327 lines )
        atlwin.h ( 0.0% of 87 lines )
        atlcom.h ( 0.0% of 124 lines )
        atlhost.h ( 0.0% of 100 lines )
        iface.cpp ( 87.5% of 16 lines )
      main.bug ( 22.6% of 1,196 lines )
        cpperr.cpp ( 0.0% of 10 lines )
        errorbase.cpp ( 0.0% of 11 lines )
        leakerr.cpp ( 0.0% of 40 lines )
        afxdllx.h ( 0.0% of 7 lines )
        bugutility.cpp ( 10.4% of 681 lines )
        treectrl.cpp ( 12.3% of 65 lines )
        ptrerr.cpp ( 19.4% of 67 lines )
        writeerr.cpp ( 21.4% of 28 lines )
        readerr.cpp ( 47.8% of 23 lines )
        main.cpp ( 52.2% of 23 lines )
        comerr.cpp ( 56.0% of 109 lines )
        doingall.cpp ( 63.6% of 11 lines )
        apierr.cpp ( 66.9% of 121 lines )
      BugBench.exe ( 59.7% of 469 lines )
        bbtreecontrol.cpp ( 31.4% of 51 line
        bbcommandlineinfo.cpp ( 46.4% of 2
        bugbenchdlg.cpp ( 62.4% of 346 line
        bugbench.cpp ( 79.5% of 44 lines )
  Methods Not Covered
  Methods Less Than 20% Covered
  Over 30 Lines, Less Than 10% Covered
```

280 of 469 lines executed (59.7%)

33 of 58 methods called (56.9%)

```
Count   Source
            // Initialize COM.
1           if ( FAILED( AfxOleInit() ) )
            {
0               MessageBox ( NULL , "COM could not be initialized." , "BugBench Initializ
0               return -1 ;
            }

            // Dynamically register the Interface Test Object that will be used by the
            // error generating COM code.
1           if ( ! RegisterInterfaceTestObject ( ) )
            {
0               CString strAppName;
0               strAppName.LoadString(IDS_APPNAME);

                TCHAR * szMsg = _T (    "Unable to register the IFACE.DLL Interface Test (
                                        "COM errors will not be generated.\n"
0                                       "Please re-install the BoundsChecker samples." ) ;
0               MessageBox ( NULL ,  szMsg , strAppName , MB_APPLMODAL | MB_OK ) ;
0           }

            // Create an instance of the main dialog
            CBugBenchDlg dlg;
1           m_pMainWnd = &dlg;

            // Get the command line passed to the app
1           CBBCommandLineInfo cmdLine;
1           ParseCommandLine(cmdLine);

            // Dispatch commands specified on the command line
1           if (!ProcessShellCommand(cmdLine))
```

```
Solution 'BugBench' (3 projects)
  BugBench
    References
    Source Files
    Header Files
    Resource Files
    ReadMe.txt
  DevPartner Studio
    BugBench.dpbcl
    BugBench.dpcov
  IFACE
    References
    Source Files
    Header Files
    Resource Files
  main
```

```
DevPartner Studio
  Getting Started
  Features Overview
  DevPartner User Interface Eleme
  Instrumentation
  Command Line and Configuration
  General Information
  Code Review
  Coverage Analysis
    Getting Started
    How-To Information for All La
    Collecting Coverage Data for
    DevPartner Coverage Langua
    Information for Distributed A
    Reference
  Error Detection
    Getting Started
    Configuring Error Detection C
    Detecting Errors in Your Prog
  Memory Analysis
  Performance Analysis
    Getting Started
    How-To Information for All La
    DevPartner Language Suppor
    Profiling Server Applications
    Information for Distributed A
    Reference
  Performance Expert
```

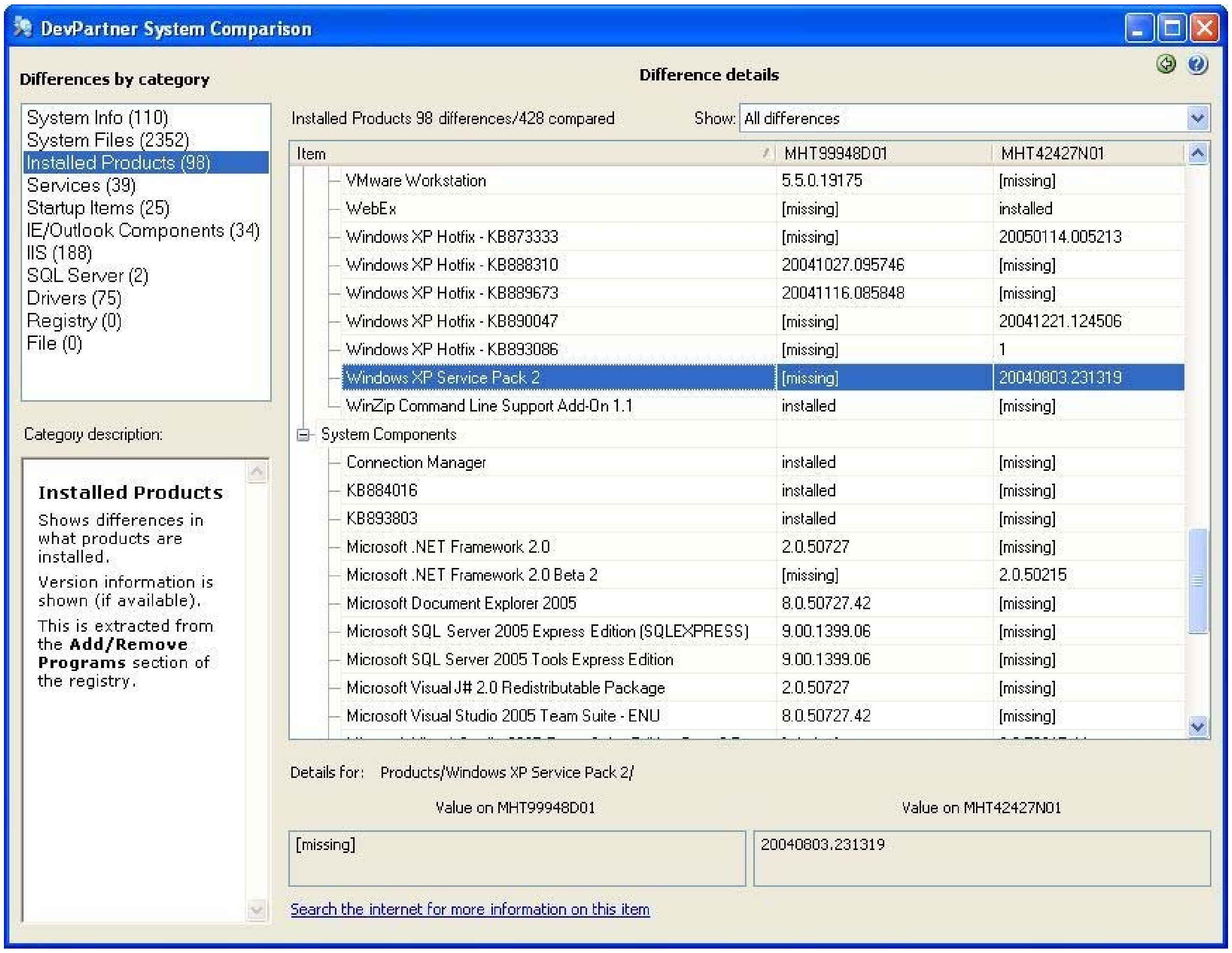
```
Deleting Key {79CCF0D1-4A20-415C-B297-6BC6B094DF4C}
'BugBench.exe': Unloaded 'C:\Program Files\Compuware\DevPartner Studio\Examples\BugBench\bin\Debug\IFACE.dll'
The thread 'Win32 Thread' (0xd90) has exited with code 0 (0x0).
Detected memory leaks!
Dumping objects ->
{223} normal block at 0x00352BA8, 160 bytes long.
 Data: <D    0            > 44 0E 1A 01 30 0E 1A 01 00 00 00 00 00 00 00 00
Object dump complete.
The program '[3236] BugBench.exe: Native' has exited with code 0 (0x0).
```

Figure 6.7: An example code coverage analysis.

Here, the tool actually analyzes the source code of the application, often examining different execution paths based on logic constructs within the code language. Each possible code path is another specific test situation, and once *every* one of those situations has actually been tested—and therefore, *every* line of code executed—then you've reach 100% coverage. These tools may highlight lines of code that have not yet been tested so that the developer can figure out what's needed in terms of input values or scenarios to complete the tests.

System Comparison

"I can't reproduce the problem." That's something no developer likes to say because it means they *know* there's a problem in the code—but if they can't reproduce it, they can't fix it. In some cases, the reason a developer can't reproduce a problem is due to differences in the system where the problem was originally identified and the developer's own system. One way to address the problem is to simply install the developer's tools on the system where the problem occurs—but that's a pretty expensive proposition in terms of time and resources (and seldom permissible on production servers). A better starting place is to get a detailed comparison of the two systems, like the one shown in Figure 6.8.



System Info (110)
System Files (2352)
Installed Products (98)
Services (39)
Startup Items (25)
IE/Outlook Components (34)
IIS (188)
SQL Server (2)
Drivers (75)
Registry (0)
File (0)

Category description:

Installed Products
Shows differences in what products are installed.
Version information is shown (if available).
This is extracted from the **Add/Remove Programs** section of the registry.

Installed Products 98 differences/428 compared   Show: All differences

Item	MHT99948D01	MHT42427N01
VMware Workstation	5.5.0.19175	[missing]
WebEx	[missing]	installed
Windows XP Hotfix - KB873333	[missing]	20050114.005213
Windows XP Hotfix - KB888310	20041027.095746	[missing]
Windows XP Hotfix - KB889673	20041116.085848	[missing]
Windows XP Hotfix - KB890047	[missing]	20041221.124506
Windows XP Hotfix - KB893086	[missing]	1
Windows XP Service Pack 2	[missing]	20040803.231319
WinZip Command Line Support Add-On 1.1	installed	[missing]
System Components		
Connection Manager	installed	[missing]
KB884016	installed	[missing]
KB893803	installed	[missing]
Microsoft .NET Framework 2.0	2.0.50727	[missing]
Microsoft .NET Framework 2.0 Beta 2	[missing]	2.0.50215
Microsoft Document Explorer 2005	8.0.50727.42	[missing]
Microsoft SQL Server 2005 Express Edition (SQLEXPRESS)	9.00.1399.06	[missing]
Microsoft SQL Server 2005 Tools Express Edition	9.00.1399.06	[missing]
Microsoft Visual J# 2.0 Redistributable Package	2.0.50727	[missing]
Microsoft Visual Studio 2005 Team Suite - ENU	8.0.50727.42	[missing]

Details for: Products/Windows XP Service Pack 2/

Value on MHT99948D01	Value on MHT42427N01
[missing]	20040803.231319

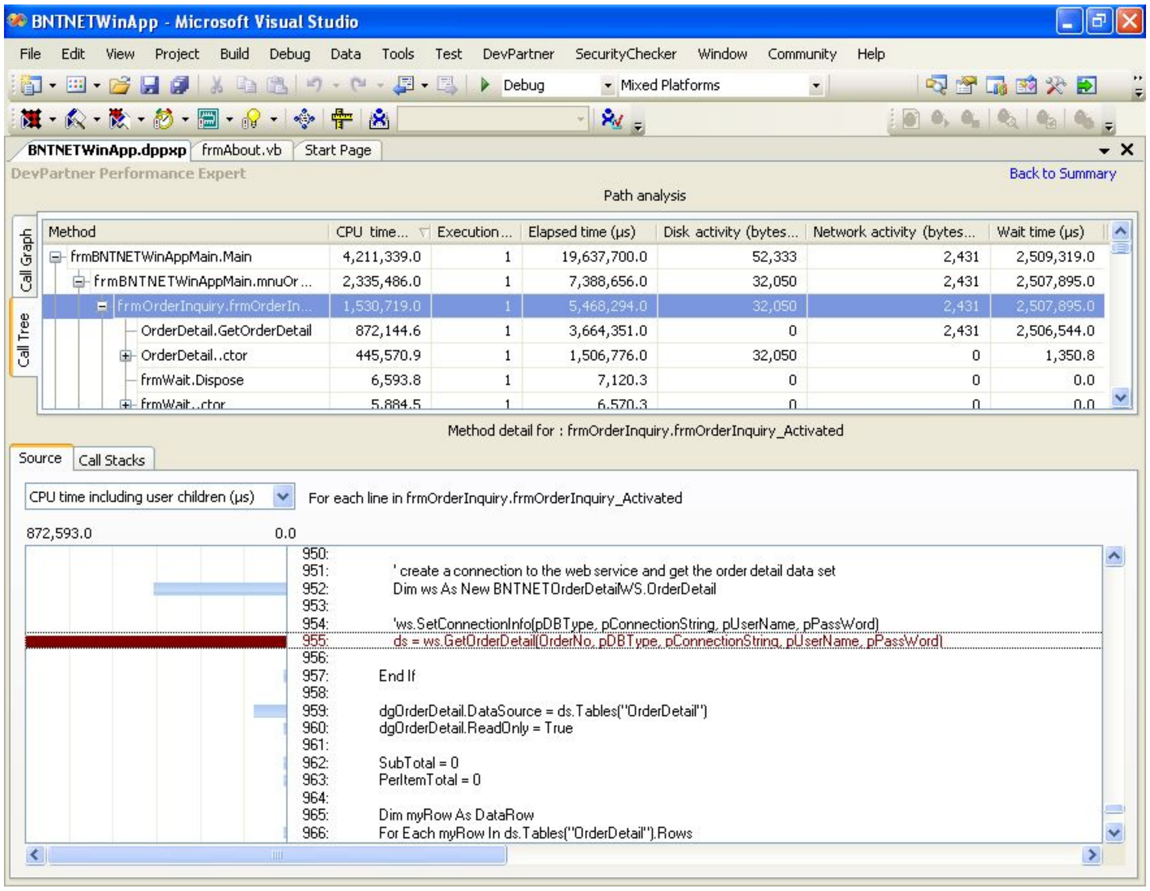
Search the internet for more information on this item

Figure 6.8: Detailed system comparison.

Often, a missing operating system (OS) hotfix, registry key, or other component will be the thing that makes the problem reproducible.

Resource Utilization and Consumption

When performance problems arise, how do you solve them? One way is to get a detailed look at how the code is actually consuming system resources such as disk, network, CPU, and memory. Testing tools can help produce that analysis right within an integrated development environment, as Figure 6.9 shows. Here, CPU time, execution time, disk activity, network activity, and more are all attributed to specific code paths within the application. In addition, top consumers are highlighted (shown in red, in this example) so that developers can focus on the areas where performance improvements may offer the biggest positive impact on the application.



BNTNETWinApp - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test DevPartner SecurityChecker Window Community Help
Debug
Mixed Platforms
BNTNETWinApp.dppxp
frmAbout.vb
Start Page
DevPartner Performance Expert
Back to Summary
Call Graph
Call Tree
Source
Call Stacks

Path analysis

Method	CPU time...	Execution...	Elapsed time (µs)	Disk activity (bytes...	Network activity (bytes...	Wait time (µs)
frmBNTNETWinAppMain.Main	4,211,339.0	1	19,637,700.0	52,333	2,431	2,509,319.0
frmBNTNETWinAppMain.mnuOr...	2,335,486.0	1	7,388,656.0	32,050	2,431	2,507,895.0
frmOrderInquiry.frmOrderIn...	1,530,719.0	1	5,468,294.0	32,050	2,431	2,507,895.0
OrderDetail.GetOrderDetail	872,144.6	1	3,664,351.0	0	2,431	2,506,544.0
OrderDetail..ctor	445,570.9	1	1,506,776.0	32,050	0	1,350.8
frmWait.Dispose	6,593.8	1	7,120.3	0	0	0.0
frmWait..ctor	5,884.5	1	6,570.3	0	0	0.0

Method detail for : frmOrderInquiry.frmOrderInquiry_Activated

CPU time including user children (µs)

For each line in frmOrderInquiry.frmOrderInquiry_Activated

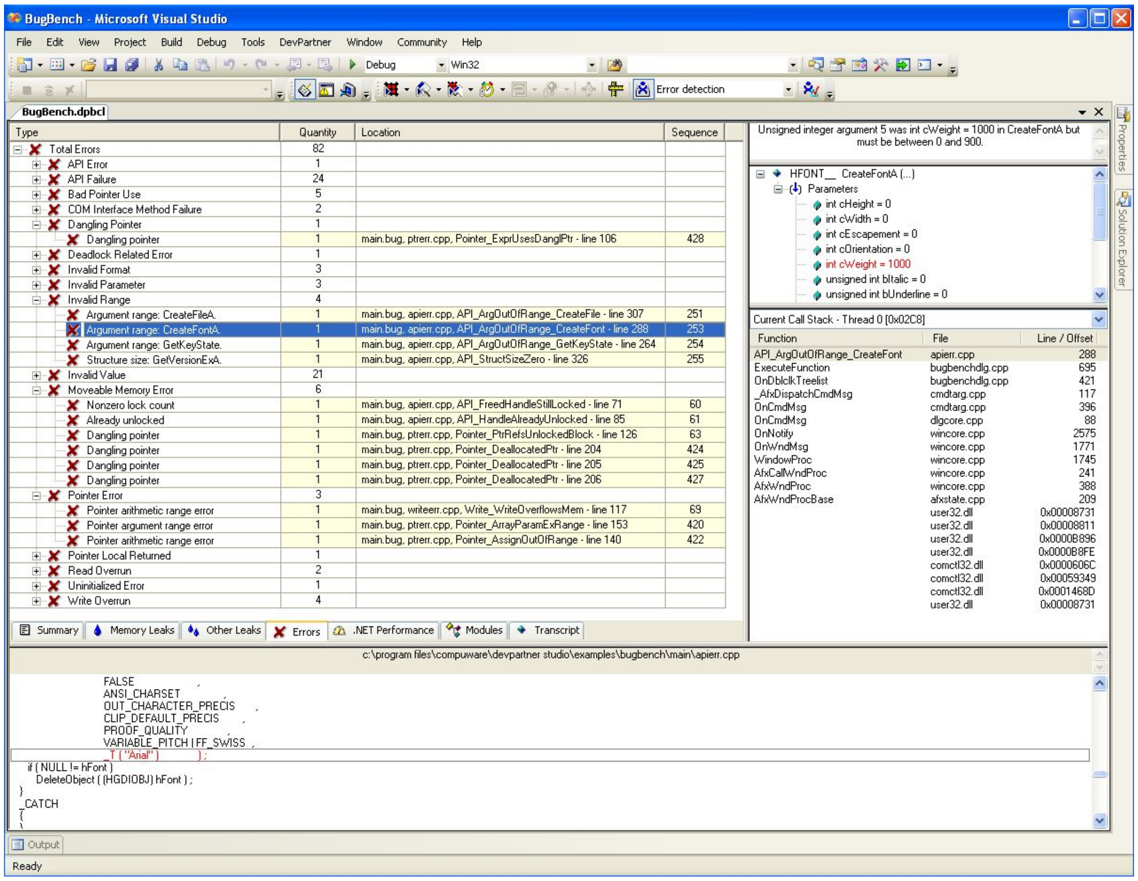
872,593.0 0.0

```
    ' create a connection to the web service and get the order detail data set
    Dim ws As New BNTNETOrderDetailWS.OrderDetail

    'ws.SetConnectionInfo(pDBType, pConnectionString, pUserName, pPassWord)
    ds = ws.GetOrderDetail(OrderNo, pDBType, pConnectionString, pUserName, pPassWord)

  End If

  dgOrderDetail.DataSource = ds.Tables("OrderDetail")
  dgOrderDetail.ReadOnly = True

  SubTotal = 0
  PerItemTotal = 0

  Dim myRow As DataRow
  For Each myRow In ds.Tables("OrderDetail").Rows
```

Figure 6.9: Performance Expert output from Micro Focus DevPartner Studio.

Error Detection

Some developer tools have the ability to detect potential run-time errors *during* design time—that is, while you're coding. Unsupportable code, runtime errors, and mishandled exceptions can therefore be addressed earlier rather than when they actually become run-time errors. Difficult or tricky code—such as Windows API calls—can be validated as you write the code, helping to avoid code that may result in an error later on down the line. Figure 6.10 shows how such tools can integrate into an IDE such as Microsoft Visual Studio, providing a breakdown of potential errors and highlighting the exact line of code that's causing the alert.



Type	Quantity	Location	Sequence
Total Errors	82		
API Error	1		
API Failure	24		
Bad Pointer Use	5		
COM Interface Method Failure	2		
Dangling Pointer	1		
Dangling pointer	1	main.bug, ptrerr.cpp, Pointer_ExprUsesDanglPtr - line 106	428
Deadlock Related Error	1		
Invalid Format	3		
Invalid Parameter	3		
Invalid Range	4		
Argument range: CreateFileA.	1	main.bug, apierr.cpp, API_ArgOutOfRange_CreateFile - line 307	251
Argument range: CreateFontA.	1	main.bug, apierr.cpp, API_ArgOutOfRange_CreateFont - line 288	253
Argument range: GetKeyState.	1	main.bug, apierr.cpp, API_ArgOutOfRange_GetKeyState - line 264	254
Structure size: GetVersionExA.	1	main.bug, apierr.cpp, API_StructSizeZero - line 326	255
Invalid Value	21		
Moveable Memory Error	6		
Nonzero lock count	1	main.bug, apierr.cpp, API_FreedHandleStillLocked - line 71	60
Already unlocked	1	main.bug, apierr.cpp, API_HandleAlreadyUnlocked - line 85	61
Dangling pointer	1	main.bug, ptrerr.cpp, Pointer_PtrRefsUnlockedBlock - line 126	63
Dangling pointer	1	main.bug, ptrerr.cpp, Pointer_DeallocatedPtr - line 204	424
Dangling pointer	1	main.bug, ptrerr.cpp, Pointer_DeallocatedPtr - line 205	425
Dangling pointer	1	main.bug, ptrerr.cpp, Pointer_DeallocatedPtr - line 206	427
Pointer Error	3		
Pointer arithmetic range error	1	main.bug, writeerr.cpp, Write_WriteOverflowsMem - line 117	69
Pointer argument range error	1	main.bug, ptrerr.cpp, Pointer_ArrayParamExRange - line 153	420
Pointer arithmetic range error	1	main.bug, ptrerr.cpp, Pointer_AssignOutOfRange - line 140	422
Pointer Local Returned	1		
Read Overrun	2		
Uninitialized Error	1		
Write Overrun	4		

Unsigned integer argument 5 was int cWeight = 1000 in CreateFontA but must be between 0 and 900.

```
HFONT__ CreateFontA (...)
    Parameters
        int cHeight = 0
        int cWidth = 0
        int cEscapement = 0
        int cOrientation = 0
        int cWeight = 1000
        unsigned int bItalic = 0
        unsigned int bUnderline = 0
```

Current Call Stack - Thread 0 [0x02C8]

Function	File	Line / Offset
API_ArgOutOfRange_CreateFont	apierr.cpp	288
ExecuteFunction	bugbenchdlg.cpp	695
OnDblclkTreelist	bugbenchdlg.cpp	421
_AfxDispatchCmdMsg	cmdtarg.cpp	117
OnCmdMsg	cmdtarg.cpp	396
OnCmdMsg	dlgcore.cpp	88
OnNotify	wincore.cpp	2575
OnWndMsg	wincore.cpp	1771
WindowProc	wincore.cpp	1745
AfxCallWndProc	wincore.cpp	241
AfxWndProc	wincore.cpp	388
AfxWndProcBase	afxstate.cpp	209
	user32.dll	0x00008731
	user32.dll	0x00008811
	user32.dll	0x0000B896
	user32.dll	0x0000B8FE
	comctl32.dll	0x0000606C
	comctl32.dll	0x00059349
	comctl32.dll	0x0001468D
	user32.dll	0x00008731

c:\program files\compuware\devpartner studio\examples\bugbench\main\apierr.cpp

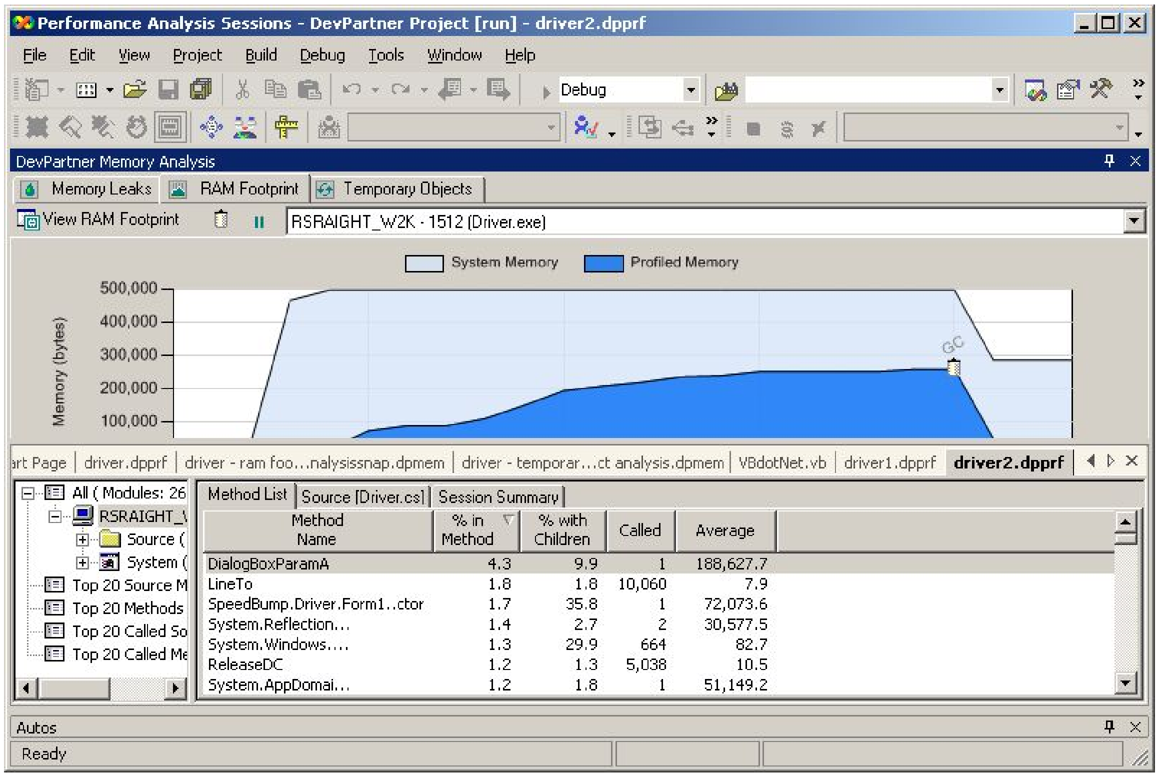
```
                    FALSE                   ,
                    ANSI_CHARSET              ,
                    OUT_CHARACTER_PRECIS        ,
                    CLIP_DEFAULT_PRECIS        ,
                    PROOF_QUALITY               ,
                    VARIABLE_PITCH | FF_SWISS ,
                    _T ( "Arial" )          ) ;
    if ( NULL != hFont )
        DeleteObject ( (HGDIOBJ) hFont ) ;
}
_CATCH
{
```

Figure 6.10: Detecting potential run-time errors while coding.

Memory and Resource Leak Detection

Poor memory utilization is a major cause for poor application performance as well as for stability issues caused by things like memory leaks. Tracking memory usage down to specific objects and classes makes it easier to detect problems early on—which is a huge benefit because these types of problems become increasingly difficult to identify and correct as the application grows and moves through its life cycle.

Tools should generally focus on three major areas: potential memory leaks, the creation of temporary objects, and the overall memory utilization (or *footprint*) of the application. This information may be displayed in a graph (as Figure 6.11 shows) and may be accompanied by a list of classes currently loaded into memory. Experienced developers can also utilize heap views, which display in-depth information about how the application is utilizing its memory and help identify lines of code responsible for the most memory use.


Performance Analysis Sessions - DevPartner Project [run] - driver2.dpprf
File Edit View Project Build Debug Tools Window Help
Debug
DevPartner Memory Analysis
Memory Leaks
RAM Footprint
Temporary Objects
View RAM Footprint
RSRAIGHT_W2K - 1512 (Driver.exe)
System Memory
Profiled Memory
Memory (bytes)
500,000
400,000
300,000
200,000
100,000
GC
art Page | driver.dpprf | driver - ram foo...nalysissnap.dpmem | driver - temporar...ct analysis.dpmem | VBdotNet.vb | driver1.dpprf | driver2.dpprf
All (Modules: 26
RSRAIGHT_\
Source (
System (
Top 20 Source M
Top 20 Methods
Top 20 Called So
Top 20 Called Me
Method List
Source [Driver.cs]
Session Summary
Autos
Ready

Method Name	% in Method	% with Children	Called	Average
DialogBoxParamA	4.3	9.9	1	188,627.7
LineTo	1.8	1.8	10,060	7.9
SpeedBump.Driver.Form1..ctor	1.7	35.8	1	72,073.6
System.Reflection...	1.4	2.7	2	30,577.5
System.Windows....	1.3	29.9	664	82.7
ReleaseDC	1.2	1.3	5,038	10.5
System.AppDomai...	1.2	1.8	1	51,149.2

Figure 6.11: Memory analysis of a running application.

This memory analysis occurs in real time, as the application is being executed on the developer's computer during development and unit testing.

Native/.NET Interoperability Analysis

Of special importance in .NET Framework applications is the ability to track performance when native (non-managed) code is being called from within .NET (managed) code. The transition from managed to non-managed code can result in several problems that, due to the way the .NET Framework itself interacts with non-managed code, can be difficult to catch. Potential problems include:

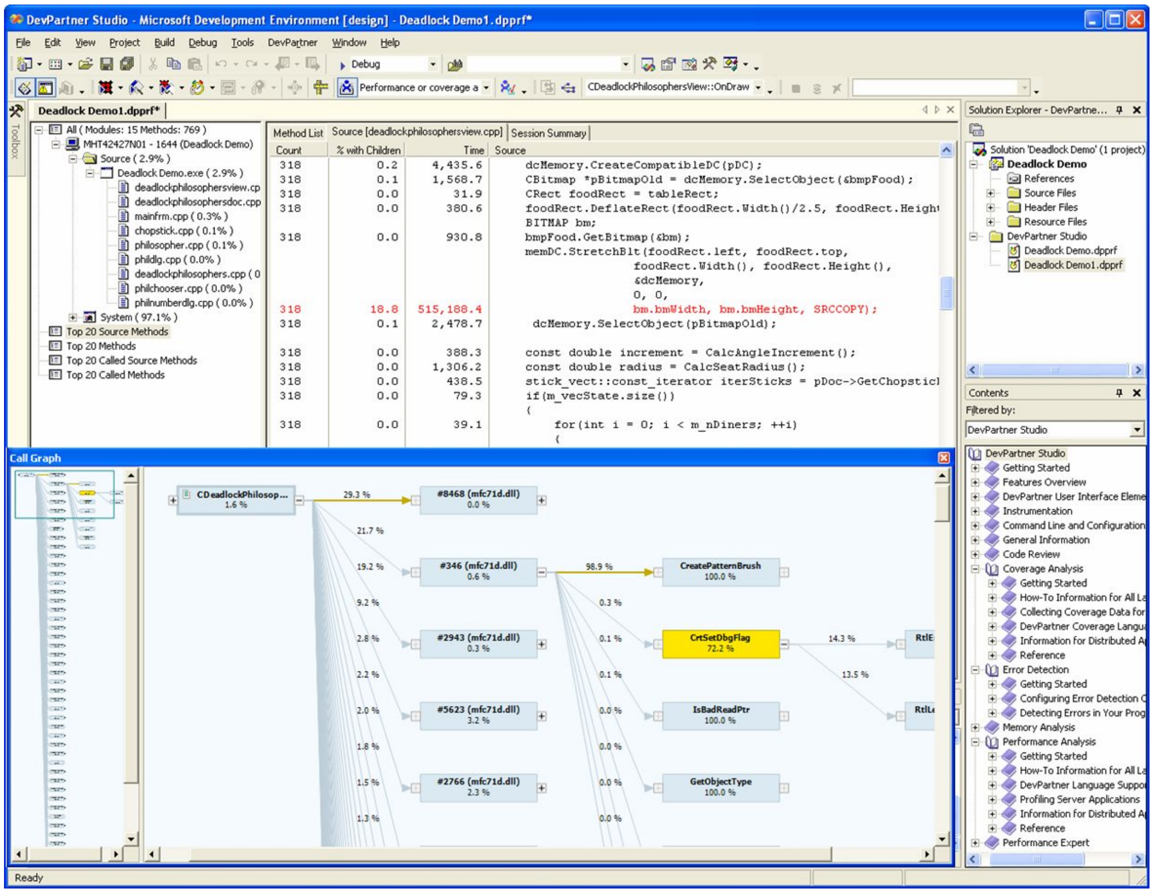
- Range and boundary errors resulting from differences in managed and non-managed data structures
- Performance issues related to memory, disk, network, or CPU consumption—with memory and CPU consumption being among the trickier aspects

- Insufficient testing, especially for tools that offer code coverage analysis but *only* cover the managed code
- Detection of potential run-time errors, especially for tools that don't understand native code and Windows API calls
- Security scanning, especially for tools that treat native code as "out of scope" and focus entirely on managed code

The key is to make sure that whatever tools you adopt fully understand, support, and exploit the managed/non-managed connection that the .NET Framework makes possible.

Code Performance Analysis

Finally, tools can provide detailed, end-to-end performance profiling of an entire application. These tools analyze the application as it runs, and can pinpoint performance problems down to specific lines of code. Displays like the one in Figure 6.12 make it easier to see where performance is lost or bottlenecked within a single-tier application (or a single tier of a multi-tier application).


DevPartner Studio - Microsoft Development Environment [design] - Deadlock Demo1.dpprf*
Deadlock Demo1.dpprf*
All (Modules: 15 Methods: 769)
MHT42427N01 - 1644 (Deadlock Demo)
Source (2.9%)
Deadlock Demo.exe (2.9%)
System (97.1%)
Top 20 Source Methods
Top 20 Methods
Top 20 Called Source Methods
Top 20 Called Methods
Method List
Source [deadlockphilosophersview.cpp]
Session Summary
CDeadlockPhilosophersView::OnDraw
Call Graph
Solution Explorer - DevPartner...
Contents
Ready

```
Count   % with Children        Time   Source
318                0.2      4,435.6   dcMemory.CreateCompatibleDC(pDC);
318                0.1      1,568.7   CBitmap *pBitmapOld = dcMemory.SelectObject(&bmpFood);
318                0.0         31.9   CRect foodRect = tableRect;
318                0.0        380.6   foodRect.DeflateRect(foodRect.Width()/2.5, foodRect.Heigh
                                      BITMAP bm;
318                0.0        930.8   bmpFood.GetBitmap(&bm);
                                      memDC.StretchBlt(foodRect.left, foodRect.top,
                                                       foodRect.Width(), foodRect.Height(),
                                                       &dcMemory,
                                                       0, 0,
318               18.8    515,188.4                    bm.bmWidth, bm.bmHeight, SRCCOPY);
318                0.1      2,478.7    dcMemory.SelectObject(pBitmapOld);

318                0.0        388.3   const double increment = CalcAngleIncrement();
318                0.0      1,306.2   const double radius = CalcSeatRadius();
318                0.0        438.5   stick_vect::const_iterator iterSticks = pDoc->GetChopsticl
318                0.0         79.3   if(m_vecState.size())
                                      {
318                0.0         39.1       for(int i = 0; i < m_nDiners; ++i)
                                          {
```

Figure 6.12: Application performance profiling.

In the example, overall performance is broken down and attributed to each element of the application, allowing the tool to highlight—in yellow, in this display—elements that are consuming an above-average amount of time or other resources. For *multi-tier* applications, more complex tools are required. These typically include some kind of agent that runs on middle- or backend-tiers and communicates performance information to a master tool running on the developer's computer. This setup allows the application's entire performance to be traced more accurately: With it, you can see that what appears to be a long execution time in a client module may in fact be due to performance problems in a remote tier that the client is simply waiting for.

Testing Phases and Efforts

Testing isn't something that occurs only during a specific phase of a project's life. Rather, it's a continuous activity, with different types of testing during different portions of the life cycle, each designed to accomplish a slightly different purpose.

Unit Test

A unit test allows a developer to gain confidence that their code is fit for use. Unit testing is commonly performed by developers on an ad-hoc basis as they code, and may include ad-hoc tests as well as tests from formally defined test cases.

Developers should ideally work from test cases that are completely independent from one another. To this end, they may create substitutes such as method stubs and mock objects so that the code they are testing has as few external dependencies as possible—so that, in other words, only a single *unit* is being tested. For example, suppose that Method A calls upon Method B; to test Method A, a mock Method B might be provided that delivers static output to Method A—thus eliminating any code in Method B from impacting the test on Method A. The test case for Method A, then, tests *only* Method A, and can be re-used as often as is necessary to ensure Method A continues to behave properly. In theory, if every unit of code operates perfectly as a set of independent units, they are more likely to operate correctly as a complete application.

Integration Test

Integration testing focuses on entire portions of an application rather than individual units of code. Integration testing is often performed by dedicated quality assurance (QA) testers, rather than by developers, and most commonly focuses on making sure the tested portions behave properly and meet their requirements. Developers typically participate in integration testing to some degree, and in some organizations, may be responsible for integration testing.

For unit testing, I said that, "In theory, if every unit of code operates perfectly as a set of independent units, they are more likely to operate correctly as a complete application." Integration testing starts to prove that theory by aggregating related units of code and testing them together. Essentially, it is testing the *integration* between those units because the units have (ideally) already been tested individually. Test cases focus on the interaction between units, ensuring that data is properly passed between them.

Integration testing does not necessarily seek to test the entire application; thus, test assets such as test harnesses and other "fakes and stubs" may still be needed. Once a group of units has passed integration testing, more groups may be added, building an increasingly-larger compilation of tested code until the entire application is represented.

System Test

System testing focuses on the entire application, with an emphasis on testing against requirements. It should require no knowledge of the internal code and is usually performed by QA testers and not by developers. System testing is the final level of formal, pre-release testing that an application receives.

Alpha and Beta Tests

No matter how well an application is tested prior to release, you'll usually encounter more than a few defects once real users get their hands on it. That's where alpha and beta tests come into play.

Note

There's a joke around the term *beta test*: It means you're using software that's "*beta* (better) than nothing." It's a tacit acknowledgement that beta code isn't expected to be perfect.

Alpha-test software is generally complete (although different organizations have different standards and definitions in this regard), but is usually not ready for end-user consumption. In some cases, alpha software is what is used in system testing, and software that passes system testing is, by definition, no longer "alpha."

Beta software, by contrast, may be released to a limited audience to get that "real world" effect and to flush out more defects. Typically, beta software is the first release that is seen outside the development team; it may contain known (documented) problems that the team is still working on.

Maintenance/Fix Tests

After the software is released, defects will likely be discovered and they will need to be fixed. It's impractical to perform a complete system test for every fix that comes along, and so maintenance testing—sometimes called *regression testing*, although that term is also applied by some to other types of tests—is used. Maintenance tests typically focus just on the units of code that have been modified, and may include some major system-level tests that exercise those modified units.

Load Testing

This type of testing is designed to run against the entire application and to automatically perform key operations that would normally be performed by an end user. The trick is that load testing uses tools that simulate dozens, hundreds, or even thousands of users, to place the application under a more realistic production workload—or even to test it to the breaking point to see what kind of workload the infrastructure can handle. This kind of testing is often the most difficult, simply because it can often only be performed—if you desire accurate results—against a full-scale production environment. Testing against a smaller environment and extrapolating the results can provide insight on overall load capability, but the answers won't be definitive because computers don't scale perfectly evenly. In other words, a test environment with half the capacity of the production environment might be able to handle 60% of the production environment's load; just because the production environment is twice as powerful doesn't necessarily mean it will support twice the workload.

Coming Up Next

The next chapter is the last in this guide, and I'll use it as an opportunity to bring together everything that I've discussed so far, but from a slightly different perspective: the business and developer benefits. I'll look at how automated debugging, analysis, and testing tools can help boost productivity, shorten development cycles, and improve overall code quality.