U S E R ' S G U I D E

TestWorks for Windows

Version 3

Software TestWorks Test Tool Suite

SOFTWARE RESEARCH, INC.

This document property of:

Name:_____

Company:_____

Address:_____

_____

Phone_____

SOFTWARE RESEARCH, INC.
625 Third Street
San Francisco, CA 94107-1997
Tel: (415) 957-1441
Toll Free: (800) 942-SOFT
Fax: (415) 957-0730
E-mail: support@soft.com
http://www.soft.com

(Last Update January 22, 1999)

/home/l1/wu/win-testword/msw-testworks

Table of Contents

List of Figures

Preface

Congratulations!

By choosing the TestWorks integrated suite of testing tools, you have taken the first step in bringing your application to the highest possible level of quality.

Software testing and quality assurance, while becoming more important in today's competitive marketplace, can dominate your resources and delay your product release. By automating the testing process, you can assure the quality of your product without needlessly depleting your resources.

Software Research, Inc. believes strongly in automated software testing. It is our goal to bring your product as close to flawlessness as possible. Our leading-edge testing techniques and coverage assurance methods are designed to give you the greatest insight into your source code.

TestWorks is the most complete solution available, with full-featured regression testing, coverage analyzers, and metric tools.

Audience

This manual is intended for software testers who are using *TestWorks for Windows*. You should be familiar with the Microsoft Windows System and your workstation.

Contents of Chapters

Chapter 1	*INTRODUCTION TO TESTWORKS FOR WINDOWS* introduces the concepts of automated testing.
Chapter 2	*FREQUENTLY ASKED QUESTIONS* answers a wide range of questions about TestWorks and software testing.
Chapter 3	*UNDERSTANDING THE USER INTERFACE* provides a brief overview of the **TestWorks** window and its commands.
Chapter 4	*USING THE TESTWORKS WINDOW* explains how to use the **TestWorks** window.
Chapter 5	*A NOTE ABOUT THE INITIALIZATION FILES* provides information about the default settings for **TestWorks** GUIs.
Chapter 6	*GLOSSARY* lists terms concerning software testing that are commonly used in the SQ community and includes specifics to the **TestWorks** tool suite.
Chapter 7	*TESTWORKS INDEX* provides an approach to assess the relative quality of a software system.
Chapter 8	*TESTWORKING SCRIBBLE* explores the entire suite of Software Research testing tools using the sample application Scribble.

.

Typefaces

The following typographical conventions are used throughout this manual.

boldface — Introduces or emphasizes a term that refers to **STW**'s window, its sub-menus and its options.

italics — Indicates the names of files, directories, pathnames, variables, and attributes. Italics is also used for manual, book, and chapter titles.

"Double Quotation Marks" — Indicates chapter titles and sections. Words with special meanings may also be set apart with double quotation marks the first time they are used.

`courier` — Indicates system output such as error messages, system hints, file output, and *CAPBAK/MSW*'s keysave file language.

`Boldface Courier` — Indicates any command or data input that you are directed to type. For example, prompts and invocation commands are in this text. (**`stw`**, for instance, invokes *STW*.)

CHAPTER 1

Introduction to TestWorks for Windows

This chapter introduces regression and coverage tools for Windows.

1.1 STW/Regression Overview

STW/Regression™ is designed to overcome the tedious and error-prone process of manual testing by automating the execution, management and verification of a suite of tests. Five components are included in *STW/Regression*:

❒ *CAPBAK*™ for automated capture and playback of user sessions.

❒ *SMARTS*™ for test organization and management.

❒ *CBDIFF*™ for test verification.

❒ *CBVIEW*™ for viewing captured images.

CAPBAK records all user activities during the testing process including keystrokes and mouse movements; it captures bitmap images and ASCII values. The captured images and characters provide baselines against which future test runs are compared. *CAPBAK*'s automatic synchronization ensures reliable playback of these test sessions, allowing tests to be run unsupervised as many times as the tester wants.

SMARTS organizes *CAPBAK*'s test scripts into a hierarchy for execution individually or as a part of a test suite, and then evaluates each test according to the verification method selected.

CBDIFF compares bitmap images, while discarding extraneous discrepancies during the differencing process.

CBVIEW displays images captured during recording and playback sessions.

NOTE: For certain product sets, we have included postscript (**.pdf**) files of the user manuals. Check your media to see if these files are available.

1.2 TCAT C/C++/Java for Windows Overview

1.2.1 The QA Problem

It is a sad fact of the software engineering world that on average, without coverage analysis tools, only around 50% of source code is actually tested before release. With little more than half of the logic covered, many bugs go unnoticed until after release. Worse still, the actual percentage of logic covered is unknown to SQA management, making any informed decisions impossible.

Questions such as when to stop testing or how much more testing is required are answered not on the basis of data but on ad hoc comments and sketchy impressions. Software developers are forced to gamble with the quality of the released software and to make plans based on inadequate data.

1.2.2 SR's Solution

Software Research, Inc. offers a solution: **TCAT C/C++/Java for Windows** is a coverage analyzer tool that gives a numerical value to the completeness of a set of tests. It also shows what parts of an application have been tested, so that effort can be focused on creating test cases that will exercise the parts that were not previously tested.

This product ensures tests that are more diverse than those chosen by reference to functional specification alone, or those based on a programmer's intuition. It ensures that they are as complete as possible by measuring against a range of high-quality test metrics. TCAT C / C++ / Java measures runtime coverage at the following levels:

- Coverage at the logical branch (or segment) level and the call-graph level, employing the *C1* metric. You can choose to test a single module, multiple modules, or the entire program using the *C1* metric.
- Coverage at the call-pair level employing the *S1* metric. After individual modules have been tested, you can test all the interfaces of the system using the *S1* metric.
- Dynamic visualization of test attainment during unit testing and system integration. This visually demonstrates, in real time, such things as segments and call-pairs hit / not hit.

1.2.3 Testing and TCAT C/C++/Java for Windows

TCAT C/C++/Java for Windows instruments your program. During instrumentation, **TCAT C/C++/Java for Windows** inserts function calls (special markers) at every logical branch (segment) in each program module. Instrumentation also creates a reference listing file, which is a version of your program which has logical branch marking comments added to it in a manner similar to the code added to the instrumented version. Extensive logical branch notation and sequence numbers are also listed.

This instrumented program is then compiled and run. By running it, you are exercising logical branches in the program. The more tests in your test suite, the higher the coverage. This test information is then written to a trace file. From the information stored in the trace file, you can generate coverage reports. In general, the reports give the following information:

- Reports included in the current iteration.
- A summary of past coverage runs.
- Current and cumulative coverage statistics.
- A list of logical branches that have been hit.

Recommended coverage is >85%. If reports indicate that you have less than this amount, you can identify unexercised logical branches by studying the coverage reports, and looking at the source code associated with the untested functions. When you identify the troubled areas, you can then create new test cases and re-execute the program.

TCAT C/C++/Java for Windows can help you reach your goal of creating the most extensive test cases possible.

1.3 TestWorks Supported Platforms

TestWorks products are also available on the following platforms: DEC Alpha using OSF/1; HP9000/7xx-8xx under HP-UX; IMB RS-6000 under AIX; NCR 3000 under SVr4; SGI under IRIX; Sun SPARC under SunOS and Solaris; 80x86 under SCO/ODT, Solaris, MS/Windows 3.1, Windows NT and MS/Windows 95 MS/Windows 98.

In addition to the most complete line of software testing products on the market, **Software Research, Inc.** offers extensive seminars, training, and high-quality technical support.

TestWorks Supported Platforms

Hardware Platforms	OS	Graphic Interface
DEC Alpha	**OSF/1**	**X11/Motif**
HP 9000/700,800	**HP-UX**	**X11/Motif**
IBM RS/6000	**AIX**	**X11/Motif**
NCR 3000	**SVR4**	**X11/Motif**
Silicon Graphics	**IRIX**	**X11/Motif**
Sun SPARC	**SunOS, Solaris**	**X11/Motif**
X86/Pentium	**SCO/ODT,Solaris**	**X11/Motif**
X86/Pentium	**MS-DOS, Windows**	**Microsoft Windows 3.1** **Windows 95** **Windows 98** **Windows NT**

CHAPTER 2

Frequently Asked Questions About Testworks

2.1 About Testing

1. Why is it necessary to test software? Shouldn't software be built so that it doesn't have any errors?

- Software testing is essential to building high-quality software because testing is effective in reducing defects in software products.
- Testing saves money. Estimates of savings are as much as 100:1 over field-discovered.
- Testing is insurance that high-quality software is built.

2. Why are automated tools needed?

Because the size and complexity of applications have grown so rapidly, "old style" methods of testing are no longer as effective. In the late 1990s, the economic choice is automated testing or no testing at all.

3. Why not continue doing manual testing?

Manual testing is difficult, costly, and not terribly reliable or effective. Complicated programs virtually require some kind of automated testing. Also, some kinds of analyses, such as code coverage analysis, cannot be done manually.

2.2 About TestWorks

1. What is TestWorks?

TestWorks is a collection of software test tools that supports all of the major functions of most software test project including the following:

- Static analysis
- Metrics
- Test file generation
- GUI testing
- Test management
- Test validation
- Branch and call-pair coverage analysis

2. Why is TestWorks organized into bundles?

There are two reasons for organizing TestWorks into bundles.

1. Lower license fees can be provided.
2. The use of one tool, like CAPBAK/X, generally leads to the use of a companion tool, like SMARTS.

3. Why is it important to purchase automated testing tools from Software Research, the long-established technical leader of the software testing industry?

Experience is the best teacher, and Software Research has put years of real-world test experience into TestWorks.

4. How will TestWorks save time and money?

Savings can be as high as 10:1 to 50:1 when a defect is found and repaired before the product goes to the field. Automated testing does have some setup cost, but most organizations receive a return on their investment after two or three months of TestWorks use.

5. How do TestWorks products work together?

Each of the TestWorks products have the same “look and feel,” and they work well together because they share many common features and approaches.

6. Do the TestWorks tools work across platforms?

Yes, but you have to be very cautious. No tool works perfectly across platforms regardless of what everyone thinks.

2.3 Regression Testing

1. What is the benefit of regression testing?

Regression tests confirm that an application under test works on the test run the same way it worked on the previous run. The lack of confirmation indicates a problem.

2. Can tests be moved from one platform to another? How much work is required to accomplish this task?

With careful planning and organization of tests, tests can usually be moved from one platform to another. It is important that the platforms are not vastly different.

3. Why is GUI testing important?

GUI testing has two advantages.

- GUI testing is easy for a user to understand.
- GUI testing makes an excellent testbed for running tests that have a high likelihood of revealing defects.

4. What are the main modes of GUI test capture/playback?

TestWorks' CAPBAK test capture / playback engine has three main operating modes:

- TrueTime Mode

  In TrueTime Mode, the test is played back exactly the way it was recorded. TestWorks' TrueTime Mode includes the powerful Automatic Output Synchronization capability, so tests are very reliable without much extra work.

  - PROS: You keep the user's real timing the way it was recorded.
  - CONS: Test may be excessively sensitive to changes in the GUI.

- Character Mode

  The recorded test uses the built-in OCR engine to synchronize or to capture essential test validation data.

  - PROS: You can base test on what is written on the screen, independent of font and type size.
  - CONS: There is a bit of local testware that has to be written to get the most out of the scripts.

- Object Mode

The test is recorded in such a way that playback is less sensitive to the locations of buttons and other "widgets" on the GUI.

- PROS: Test are insensitive to GUI layouts, other "noncritical" formatting details.
- CONS: Test may miss out on catching missing or invisible buttons, and they don't preserve any actual-user timing information.

5. How many defects are actually detected by regression testing?

It is estimated that a 5% functionality change in a 1000 suite test will reveal 5-25 new defects.

6. Is it true that capture/playback tools are dangerously invasive? Exactly how do these tools work?

Every kind of recording mechanism is to some extent "invasive," but most users do not consider this low level of interaction with the underlying operating system software much of a risk.

CAPBAK/X works with X windows on UNIX by using the XtestExtension1 or Xtrap extension to the X11 server (display driver). The "xdpyinfo" command on the UNIX machine tells which extension(s) are available. CAPBAK/X also uses a special version of the Xt toolkit for ObjectMode recording from X/Motif GUIs.

On Windows machines, CAPBACK/MSW uses built-in features of the Windows 3.x, Windows '95, Windows '98 or Windows NT operating systems.

2.4 Coverage Testing

1. What is the benefit of coverage testing?

Test coverage indicates whether a test missed something. Good test coverage is necessary for thorough testing.

Manual testing seldom exercises more than one-third of the overall structure of code. Therefore, unless coverage testing is used, applications can go to the field with major sections never executed.

2. What is the difference between "BlackBox" testing and "WhiteBox" testing?

With "BlackBox" testing, the tester cannot see what's going on inside the application under test whereas with "WhiteBox" testing, the tester can see what's going on.

Most of the time, BlackBox testing means that functional testing is being executed. WhiteBox testing usually means that coverage analysis is being executed.

3. What is a segment? What is a [logical] branch?

A segment, sometimes called a [logical] branch, is a piece of code that is always executed as a unit after some piece of program logic happens. For example, an "IF" statement has two segments: the "true" and the "false" outcomes.

4. Why not just use "statement coverage?"

Statement coverage turn out to be 100% identical to branch coverage if every piece of logic in your program starts on a new line. Obviously, this never happens, so statement coverage tends to overstate the actual coverage by 50% or more. You can 100% statement coverage and only 50% branch coverage. CAUTION: This overstatement may be dangerous to the health of your software!

5. Why do we need automated tools at all?

Because the size and complexity of applications have grown so rapidly, "old style" methods of testing just don't work anymore.

6. What is a call-pair?

When two functions (programs) call one another, they make a call-pair.

The caller function could call the callee many times, and TestWorks' view of coverage analysis holds that all call-pairs must be checked.

7. Do all these coverage measures have names?

Yes, because it helps to keep track of the facts.

- Statement coverage is called C0. (SR does not support C0 because it's too misleading.
- Segment / branch is called C1.
- Module coverage is called S0.
- Call-pair coverage is called S1.

8. Why not just do module coverage?

You can, and we recommend it as a minimum step. But remember, errors in interfaces happen most often when the calling sequence doesn't match up with the called-function's definition. Call-pair coverage overcomes this by requiring every caller-callee pair to be tested at least once.

9. What is a test path?

TestWorks treats test paths as sequences of segments, counted up to a repetition count for a loop. (See the TCAT-PATH manual for full details.)

10. How many defects are actually detected by coverage testing?

It is estimated that increasing branch coverage from 50% to around 90% will expose 5-10 defects per 1000 lines of code (KLOC).

11. Can I set up my makefiles to do coverage analysis automatically?

Yes. Make a one or two line modification to the makefile to tell it how to call the TestWorks' Instrumentor when you want to "make" an instrumented target. Then you **`make instrumented version`** rather than **`make normal version`**.

Some users make instrumentation the normal and only take out the instrumentation prior to shipping.

2.5 Static Testing

1. Why should I do static analysis of my source code?

Using a simple mechanical check that finds an error inexpensively saves money. The alternative is finding errors from the field, a more costly method in terms of money and reputation.

2. How many defects are actually detected by advisor/static testing?

It is estimated that static and metric analyses can yield as many as 2-8 defects per 1000 lines of code (KLOC).

3. About how many defects are there in code?

There are approximately 30-50 / KLOC (1000 Lines of Code) in new software. QA testers aim to get defects down to 1 / KLOC; some critical aerospace applications must get below 0.1 / KLOC.

4. Why are software metrics so important?

Experience shows that the most complicated pieces of a software product often contribute to most of the errors. The best use of resources is knowing how to identify the most complex pieces of software and then concentrating efforts there.

2.6 Integration Testing

1. What is integration testing and how is it beneficial?

Most of the time, two or more pieces of a large system are put together after they have been tested separately. Testing two or more parts of a program together is called integration testing.

Many defects are discovered in integration testing, and TestWorks is effective by insisting on high levels of call-pair coverage during the integration test process.

2. What must I do to make TestWorks integrate into my software process smoothly?

Make sure that your software process runs smoothly in the first place. If it does, then TestWorks adds to the existing processes quite nicely.

2.7 About TestWorks Licensing

1. What platforms does TestWorks run on?

TestWorks runs on the following:

- UNIX platforms (SUN SPARC Solaris, HP-9000, SGI, DEC-Alpha, etc.)
- Windows (3.1x, '95, '98 and NT)

2. How is TestWorks licensed?

On UNIX products, Software Research, Inc. offers LAN-based floating licenses.

On Windows products, Software Research, Inc. offers group licenses, department licenses, and site licenses.

3. What is the TestWorks warranty agreement?

Software Research, Inc. has a standard 30-day warranty. Check your license agreement for complete details and limitations.

4. What are the benefits of keeping my software maintenance contract current?

There are two benefits to keeping your software maintenance contract current:

- Continuous technical support
- The option of receiving upgraded products at no cost

Once your maintenance contract lapses, it is more expensive to restore it than it would be to continue the contract. Maintenance contracts lapse 60 days after the end of prior maintenance.

5. What if I have problems with TestWorks after 30 days?

If you are on maintenance, help is guaranteed. However, if you are not on maintenance, Software Research, Inc. will do its best to help.

2.8 About TestWorks Internal Architecture

1. Why does TestWorks use "C" as its basic command language?

"C" is the language most universally understood by programmers and testers alike, and it is compact and easy to use.

2. Do I have to be highly skilled in "C" programming to use TestWorks?

No. Most of the time it is not necessary to edit scripts, and if it is, the syntax rules are very simple to follow.

3. Why are TestWorks license prices so high?

Compared with other testing products, license fees for TestWorks' products, bundles, and the entire TestWorks Suite are very low on a value per function basis.

TestWorks is the most cost-effective way to automate testing.

4. Why is there a price difference between UNIX and Windows?

Licensing creates the price difference between UNIX and Windows. A single floating license on UNIX can serve 2-4 testers, but on Windows, each tester for each machine needs a license.

5. How difficult is it to install TestWorks

It is easy to install TestWorks.

On UNIX version there is an **`install.stw`** that does all the instalation**.**(If a systems administrator installs TestWorks, "root/superuser" permission may be required.)

On Windows, products are installed using the supplied hands-off installation script.

6. How long does it take to install TestWorks?

If the install script is used, installation takes approximately twenty minutes or less. There are not guarantees if the install script is not used.

7. How much memory do TestWorks products use?

On UNIX, the distribution tapes can range in size from 20 MB to 75 MB, but that includes online documentation and several utilities.

The UNIX products take minimum 16 MB of RAM to execute.

The Windows products run from 20 MB to 30 MB depending on the version. They take at least 8 MB of RAM to execute.

2.9 Embedded and Cross-Testing

1. What does cross-development and cross-testing mean?

Cross-development means that developing is taking place on one machine (the host) and product runs are taking place on another machine.

Cross-testing is TestWorks' way of supporting cross-development.

2. Does TestWorks test embedded code?

Yes. For cross-development host/target type development, Software Research, Inc. has special kits to test embedded code.

3. How many changes to my code do I have to make for TestWorks to be effective?

None.

The coverage analysis tools work with source code modification, but they make "throw away" versions of your code, then feed that directly to your compiler. You never see the intermediate versions.

4. My frequently asked question isn't answered above. What should I do?

Send your question to *info@soft.com* and we'll answer it immediately.

2.10 Technical Support

1. What kind of technical support can I expect?

We respond to ALL incoming calls, Emails, FAXes, etc., within 24 hours.

2. Is training available on the TestWorks products?

Standard 2-day, 3-day, 4-day, and 4-1/2 day trainings exist for TestWorks. The length varies with how many products are being learned.

3. Is a demo available?

A "demo disk" is not available, but a trial or evaluation of the fully-functioning TestWorks products can be arranged.

Contact our sales group to arrange a trial/evaluation.

4. Can we evaluate the software in our own environment?

We strongly advocate *in situ* trials/evaluations so that you can see TestWorks working in your environment on your product.

5. What if we are not satisfied after the purchase?

If you are dissatisfied for any reason within the 30-day guarantee period, you can return the product, no questions asked. For longer periods, contact sales. Our guideline is that we want satisfied customers, and we will do the best that we can to achieve that goal.

6. Do the TestWorks' tools work across platforms?

When used with caution, TestWorks' tools can work across platforms.

CHAPTER 3

Understanding the User Interface

This chapter summarizes *TestWorks'* windows, menus and commands. Individual application of commands is describedin detail in the relevant chapters of this guide.

3.1 Basic MS-Windows User Interface

This section demonstrates using file selection dialog boxes, help menus, message dialog boxes, option menus, and pull-down menus. If you are familiar with the basic MS-Windows graphical user interface (GUI) style, you can go on to Section 3.2 on page 25.

3.1.1 File Selection Windows

File selection windows allow you to select or specify test file names or select saved image files.

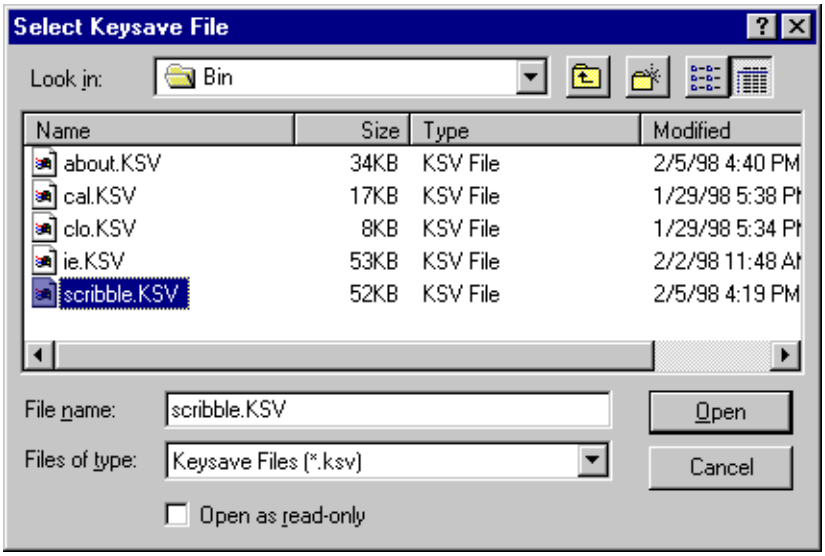

FIGURE 1 File Selection Window for Windows 95/NT

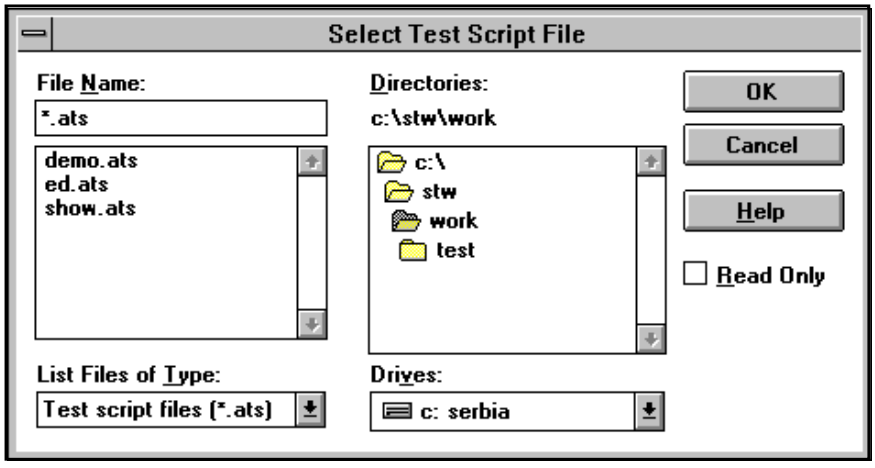

FIGURE 2 File Selection Window for Windows 3.1x and NT 3.5

File Name entry box

Selects and enters a file name.

File Name list box — Lists files in the path defined in the **List Files of Type** area.

List Files of Type — Specifies which files are listed in the **File Name** area. The current type of file and its extension are displayed.

Directories list box

Lists directories in path defined in the **Filter** entry box. Use it to locate the desired directory.

Drives — Selects your system's current drive.

Scroll bars — Move up / down and side / side in the **Directories** and **File Name** list boxes. You use them to search for the appropriate directory or file.

File Name entry box

Selects and enters a file name.

Use the three buttons at the right of the dialog box to issue commands:

OK — Accepts the directory and file in the **File Name** entry box as the new file or the file to be opened and then exits the dialog box.

Help — Supplies on-line help.

Cancel — Cancels any selections made and then exits the dialog box. No file is selected as a result.

To use a file selection dialog box:

1. Click the directory name where an existing file is located or where you want a new file to be placed.
2. Select an already existing file name in the **File Name** list box or type in the name of the new file name in the **File Name** entry box, with no limit on character length.
3. The convention for naming test files, or keysave files, is *basename.ksv*, where *ksv* represents a keysave file. Captured images take the form of *basename.bxx*, *basename.sxx*, *basename.rxx*, where *b* represents a baseline image, *s* identifies a image captured for synchronization, *r* represents a response file, and *xx* represents the original sequence in which the image was captured.
4. To select a keysave file name, do one of these three things:
   - Double click on the file in the **File Name** list box.
   - Highlight the file in the **File Name** list box or type in the file name in the **File Name** entry box and click on **OK**.
   - Highlight or type in the file name and press the **Enter** key.

3.1.2 Help Windows

On-line help is available for all the **TestWorks/Regression** products. Help automatically brings up the text corresponding to the topic you choose.

Here's how to use the help.

1. From the main window, click on the **Help** button, or from any other window, click on the **Help** pull-down menu.
2. The **Help** window pops up with the contents of the help information.
3. Simply click on the topic you want information for and the **Help** window automatically displays it.

Help: If this is the first time you've used on-line help, you might want to choose **How To Use Help** from the **Help** menu. You can also refer to your *Microsoft Windows User's Guide* for complete information on using **Help** menus.

FIGURE 3 Help Window

3.1.3 Pull-Down Menus

Pull-down menus are located within the menu bar of *CAPBAK/MSW*'s windows. They often contain several options. To use pull-down menus and their options, follow these steps:

1. Move the mouse pointer to the menu bar and over the menu containing the item.
2. Hold the left mouse button down. This displays the items on the menu.
3. While holding down the left mouse button, slide the mouse pointer to the menu item you want to select. The menu item is highlighted in reverse shadow.

An ellipsis (...) following an option indicates that selecting the item will bring up a pop-up window, such as a file selection window.

4. To choose an item from a selected menu, click the item, or type the letter that is underlined in the item name, or use the arrow keys until you reach the item you want to select, and then press the **Enter** key.

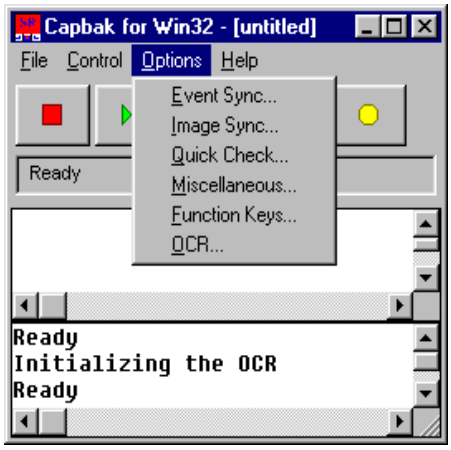

FIGURE 4 Pull-Down Menu

3.2 The TestWorks Window

The **TestWorks** window displays all the commands and menus to operate *TestWorks* including verifying installation, invoking various product lines, using the supplied demos, and viewing the on-line glossary.

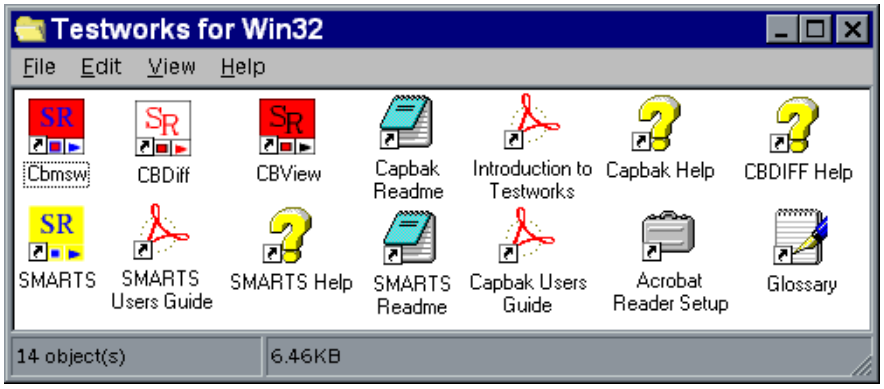

FIGURE 5 TestWorks Window

The window is divided into the following parts:

1. The *CAPBAK MS-Windows* icon brings up the capture / playback utility.
2. The *CBDIFF* icon brings up the image differencing utility.
3. The *SMARTS* icon brings up the test management utility. (Only if SMARTS is installed)
4. The *CBVIEW* icon brings up the image-viewing utility.
5. The *GLOSSARY* icon brings up a list of terms relevant to software testing in general and the *TestWorks* product set in particular.

To bring up any of the utilities, double-click on the appropriate icon.

Using the TestWorks Window

This chapter explains how to use the TestWorks window, its commands and options.

4.1 Invoking the TestWorks Window

For Windows NT, Windows 95 or Windows 98, go to your program's menu from your start button.

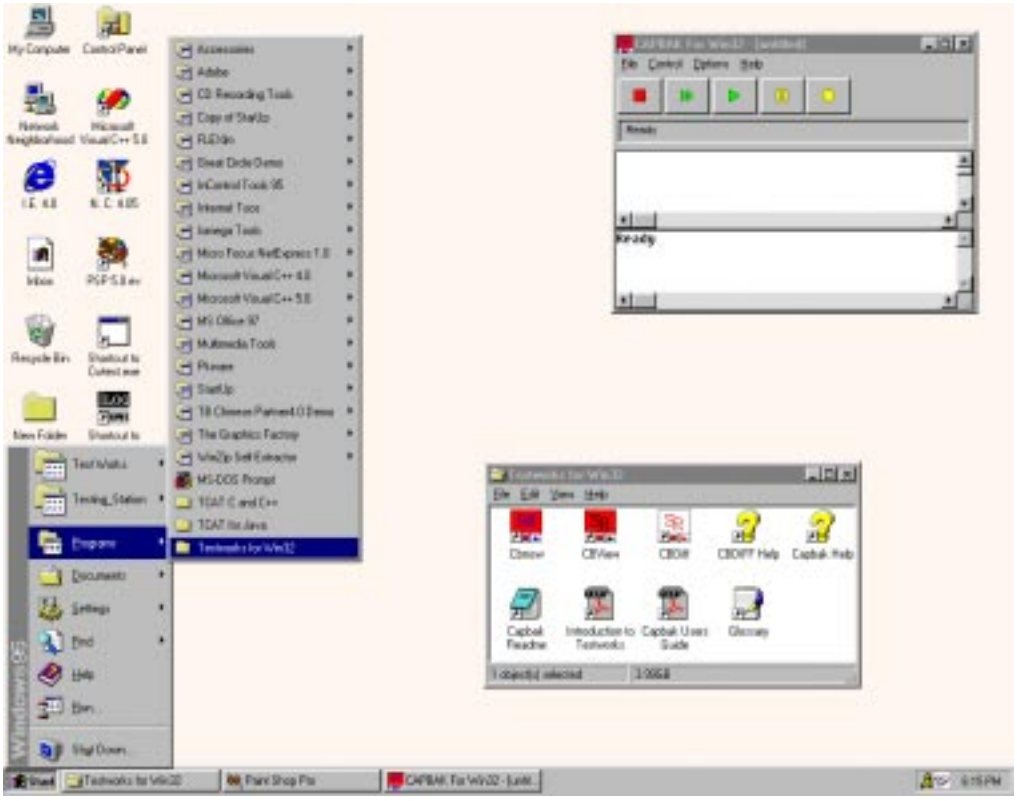

FIGURE 6 Typical Program Manager with TestWorks icon displayed.

The **TestWorks** window pops up.

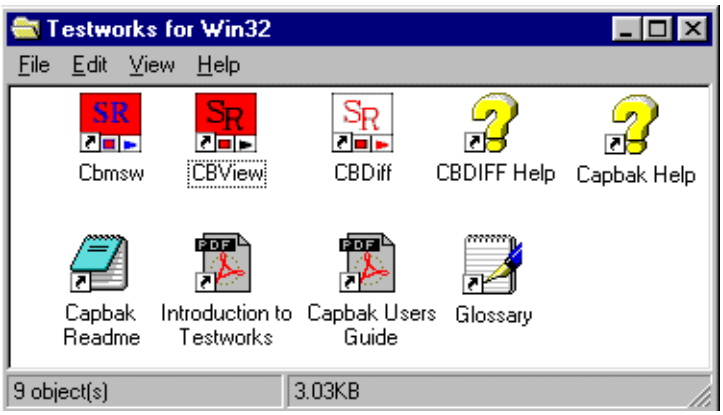

FIGURE 7 TestWorks Window

Before you invoke any of the utilities, make sure you are already in Windows. *C:\Program Files\Software Research\Regression* or whatever directory you installed the SR executables in, must be in your DOS *$PATH* (see *Installation Instructions* for further details).

4.2 Cbmsw

The Cbmsw icon brings up the capture/playback utility

4.3 CBDiff

The CBDiff icon brings up the image differencing utility.

4.4 CBView

The CBView icon brings up the image viewing utility.

4.5 Introduction to TestWorks

The Introduction to TestWorks icon brings up an overview of the TestWorks suite of testing tools.

4.6 Capbak Help

The Capbak Help icon brings up information to help you use the capture/playback utility.

4.7 CBDIFF Help

The CBDIFF Help icon brings up information to help you use the image differencing utility.

4.8 Smarts

The Smarts icon brings up the test management utility

4.9 Smarts Users Guide

The Smarts Users Guide icon brings up a quick start for the Smarts utility.

4.10 Smarts Help

The Smarts Help icon brings up information to help you use the Smarts utility.

4.11 Capbak Users Guide

The Capbak Users Guide icon brings up a manual for the capture / playback utility.

4.12 Acrobat Reader Setup

The Acrobat Reader Setup icon launches the utility to configure the Acrobat Reader.

4.13 Glossary

The Glossary icon brings up a list of terms relevant to software testing in general and the STW product set in particular.

CHAPTER 5

A Note about the Initialization Files

5.1 Locations for Defaults

Initialization files hold of all the default settings for STW product GUIs. For *CAPBAK/MSW* the initialization file is called *cbmsw.ini* and for *SMARTS/MSW* it is called *smarts.ini*. These files follow the Microsoft Windows file format conventions, and should be located in *C:\windows*.

These are text files and can be amended using any standard text editor. For complete listings of the *cbmsw.ini* and *smarts.ini* files, please refer to the *CAPBAK/MSW* and *SMARTS/MSW* manuals.

CHAPTER 6

Glossary

This glossary includes terms that are commonly used in the Automated Software Test and Software Safety and Reliability community, as well as terms which pertain to SR's Software TestWorks (STW) system. Definitions given to a term are used in one or more industry-sponsored standard technical vocabularies. When the term is specific to SR, the product with which it is most closely associated is named in the definition.

acceptance tests — Formal tests conducted to (1) determine whether or not a system satisfies its acceptance criteria and (2) to enable the customer to determine whether or not to accept a system. This kind of testing is performed with the STW/Regression suite of tools. {Regression}

action statement — A non-decision statement in a program that results in executable code. {Coverage}

activation clause — A clause in the ATS file, composed of a sequence of system commands which perform actions for the test case execution. {SMARTS}

Ada — The DoD standard programming language. Also, Ada9X refers to the 1990's update of this language. {Regression}

alpha testing — Testing of a software product of system conducted at the developer's site by the customer. [Ref. 1]

ALT-M — The CAPBAK/DOS hotkey menu trigger character. {Regression}

ALT-S — The CAPBAK/DOS screen save hotkey character. {Regression}

ancestor node — A node in a directed graph that lies on some path (i.e., sequence of segments) leading to the specified node. {TCAT-PATH}

apg — **All Paths Generator**. A TCAT-PATH facility which generates equivalence classes that include all program paths from a directed graph. {TCAT-PATH}

arc — In a directed graph, the oriented connection between two nodes. Also called an edge. {Coverage}

archive file — A file containing test trace information in reduced form. {Coverage}

ASCII synchronization — The process by which a playback (e.g. from CAPBAK) holds back execution until a character string is located.

ATS — **Automated Test Script**. A SMARTS user-designed description file which references a test suite. Test cases are referenced in a hierarchical structure and can be supplemented with activation commands, comparison arguments, PASS/FAIL evaluation criteria, and system commands. When SMARTS is run on either an X Window or UNIX system, the ATS is written in SMARTS' Description Language (which is similar to C language in syntax). The ATS file is written in SMARTS C-Interpreter Language when SMARTS is run on an MS Windows system.

AUT — Application-under-test.

automatic flow control — When CAPBAK is being run in terminal emulation record mode, a record of the manual flow control is stored in the keysave file and response file. When CAPBAK is transmitting keys in playback mode the flow control is maintained by using the information saved in these files. See manual flow control. {CAPBAK/UNIX}

Automated Test Script — See **ATS**. {SMARTS}

axis — A subset of the nodes in a digraph used as a basis for digraph display. {Coverage}

back-to-back testing — For software subject to parallel implementation, back-to-back testing is the execution of a test on the software's similar implementations and a comparison of the results.

baseline file — A text or image file created during initial testing. Baseline files provide expected program output for comparison against future test runs. These kinds of files are created during STW/Regression testing.

basis paths — The set of non-iterative paths. {TCAT-PATH}

beta-testing — Testing conducted at one or more customer sites by the end-user of a delivered software product or system. This is usually a "friendly" user and the testing is conducted before general release for distribution. {Software Technology Support Center}

black-box testing — See **closed-box testing**. {Regression}

bottom-up testing — Testing starts with lower level units. Each time a new higher-level unit is added to those already tested, driver units must be created for units not yet completed. Again, a set of units may be added to the software system at that time, and for enhancements the software system may be complete before the bottom- up test starts. The test plan must reflect the approach, though. {Coverage}

branch — See **segment**.

branch testing — A test method satisfying coverage criteria that requires that for each decision point, each possible branch be executed at least once. {Software Technology Support Center}

built-in testing — Any hardware or software device which is part of a piece of equipment, a subsystem or system, which is used for the purpose of testing that equipment, subsystem or system.

byte mask — A differencing mask used by EXDIFF that specifies to disregard differences based on byte counts.

"**C++**" — The "C++" object-oriented programming language. The current standard is ANSI C++ and/or AT&T C++. Both are supported by TCAT/C++. {TCAT/C}

"**C**"	The programming language "C". ANSI standard and K&R "C" are normally grouped as one language. Certain extensions supported by popular "C" compilers are also included as normal "C".
C0 coverage	C0 is the percentage of the total number of statements in a module that are exercised, divided by the total number of statements present in the module. {Coverage}
C1 coverage	The percentage of segments exercised in a test as compared with the total number of segments in a program. {Coverage}
call graph	The function call tree capability of S-TCAT. This utility shows the caller-callee relationship of a program. It helps the user to determine which function calls need to be further tested. {Coverage}
call pair	A connection between two functions in which one function "calls" (references) the other function, in a call tree. {Coverage}
capbak	This command invokes CAPBAK/DOS or CAPBAK/UNIX.
CAPBAK	The test Capture and Playback component of the STW/Regression product set. It has the capability of capturing keystrokes, mouse movements, partial screens, windows, or the whole screen and putting them into a test script language which can be played back later.
capbak	This CAPBAK/UNIX command allows the user to capture keystrokes in a keysave file with or without actually being attached to an application. No response file is created. This provides a way to generate keysave files independent of the user doing anything.
capset	A utility to control CAPBAK/DOS operation from a command line and from the SMARTS ATS file.

CBDIFF CAPBAK/MSW's image differencing utility. This utility offers general differencing and masking capabilities.

CBVIEW CAPBAK/MSW's image viewing utility.

certification report This report summarizes the total number and percentage of tests that have passed and failed, providing a brief overview of testing status. {SMARTS}

character recognition See **OCR**.

clear-box testing See **glass-box testing**.

closed-box testing A method where the tester views the program as a closed box; i.e. the test is completely unconcerned with the internal behavior and structure of the program. The tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from these specifications, without taking advantage of knowledge of the internal structure of the program. Also known as **black-box testing**.{Regression}

COBOL The COBOL programming language.

coding rule A rule that specifies a particular way in which a program is to be expressed.

coding style A general measure of the programming nature of a system; abstractly, the way the programming language is used in a real system.

collateral metrics Secondary metrics gathered as an unexpected by-product of the gathering of primary metrics. These may not be needed or even useful, but then again, may prove to be of value later. Consider saving, even if costly. {Software Technology Support Center}

collateral testing Collateral testing is that testing coverage which is achieved indirectly, rather than as the direct object of a test case generation activity. {Coverage}

combinational flow Combinational flow is represented by a sequence of segments, with the property that no segment is repeated within the flow. {Coverage}

command mode This mode allows the user to program the key-save file for conditional execution based on system calls. The other mode of execution is data mode. Command mode is supported by {CAPBAK, CAPBAK/UNIX}

compilers Compilers are included here as a reminder of how much static code checking is done by compilers. These are valuable automatic test tools. {Software Technology Support Center} 2. A computer program that translates instructions, other programs, etc. (from)...a high-level language into a machine language. {Webster's New World Dictionary}

complexity A relative measurement of the ‘‘degree of internal complexity" of a software system, expressed possibly in terms of some algorithmic complexity measure. {METRIC}

complexity report This report lists all of a source code program's encountered procedures and lists Software Science metrics (which are concerned with the ‘‘size" of software) and Cyclomatic Complexity measures (which are concerned with the flow of control within the program's code). {METRIC}

component A part of a software system smaller than the entire system but larger than an element.

conditional playback See also **playback programming**. Certain STW components incorporate a language that provides for logical operations to control behavior during test execution; e.g. a SMARTS test can involve use of the if or while constructs, as can a CAPBAK script.

configuration file A file used to declare start-up time parameter values. Usually suffixed as ***.rc**.

connected digraph A directed graph is connected if there is at least one path from every entry node to terms of all the possible sub-trees that can be executed for that program. A TCAT-PATH component used to measure Ct coverage against a path file.

ctcover A TCAT-PATH utility used to assess Ct coverage.

cumulative coverage The test coverage attained by a set of several test runs. {Coverage}

cumulative report This report charts branch and / or call-pair coverage for the current test cumulatively, and also for each module in the total system. {Coverage}

current position The current position of the screen's cursor, expressed in x,y coordinates. NOTE x=0, y=0 is the upper left corner of the screen. On some machines this same pixel may be called x=1, y=1.

cycle A sequence of segments that forms a closed loop, so that at least one node is repeated. {Coverage}

cyclomatic number A number which assesses program complexity according to a program's flow of control. A program's flow of control is based on the number and arrangement of decision statements within the code. The cyclomatic number of a flowgraph can be calculated as follows:

e - n + 2

where n is the number of nodes in the graph, and e is the number of edges or lines connecting each node. {METRIC, TCAT, TCAT-PATH}

data flow graph A graph of a variable name's uses along a fixed path within a module or software system, expressed in terms of the legal and illegal transitions within the system for the variable. {Coverage}

data sensitivity fault A fault that causes a failure in response to some particular pattern of data. {Software Technology Support Center}

data mode — In this execution mode for keysave files, text is interpreted as saved keystrokes, to be played back along with timing information which is enclosed in brackets. {CAPBAK, CAPBAK/UNIX}

DD-path — See **segment**.

de-instrumentation — When certain parts of your code have already been tested, you can use TCAT's and S-TCAT's de-instrumentations utilities to exclude those parts from instrumentation. For large programs, this can save time.

debug — After testing has identified a defect, one "debugs" the software by making certain changes that repair the defect.

decision-to-decision path — See **logical branch**.

decisional depth — The number of decisions that must take on a particular value prior to arriving at a specified logical branch. “The decisional depth for this logical branch is..." {Coverage}

defect — A difference between program specifications and actual program text of any kind, whether critical or not. What is reported as causing any kind of software problem.

defect analysis — Using defects as data for continuous quality improvement. {Software Technology Support Center}

defect density — Ratio of the number of defects to program length (a relative number). {Software Technology Support Center}

deficiency — See **defect**.

delay multiplier — The multiplier used to expand or contract playback rates.

desk checking

A form of manual static analysis, usually performed by the originator. Source code, documentation, etc. is visually checked against standards. It is cheap, effective, and usually underestimated and under-applied. (Maybe if we called it an *individual design review*, it would get more respect. This is where pride in workmanship and individual empowerment are exhibited.)
{Software Technology Support Center}

development test and evaluation (D T & E)

Testing conducted throughout the acquisition process to ensure an effective and supportable system by assisting in design and development and verifying specifications, objectives, and supportability. {Software Technology Support Center}

digraph

Short name for a directed graph, a graph which displays all of a program's **nodes** and **edges** and their relationships. The **Xdigraph** utility within STW's TCAT and S-TCAT set of tools draws digraphs and has options for displaying them in many different ways.

direct metric

A metric that represents and defines a software quality factor and which is valid by definition, e.g. mean time to software failure for the factor reliability. {Software Technology Support Center}

dump

A display of some aspect of a computer's execution state, usually the contents of memory, registers, etc. Is used as a diagnostic aid. Some examples are a postmortem dump (taken after a failure), and a snapshot dump (taken during execution).
{Software Technology Support Center}

dynamic analysis

A process of demonstrating a program's properties dynamically, by a series of constructed executions. {Coverage}

dynamic call-tree display — An organic diagram showing modules and their call-pair structure, where the call-pairs are "animated" based on behavior of the instrumented program being tested. {Coverage}

dynamic digraph display — An organic diagram showing the connection between segments in a program, where the segments are "animated" based on behavior of the instrumented program being tested. {Coverage}

edge — In a directed graph, the oriented connection between two nodes. {Coverage}

emulator — From Webster's, emulate, 'to strive or equal or excel.' Therefore, a machine that strives to equal or exceed the performance characteristics of another, very often through software. Similar to a simulator. Example: Software in a PC that causes it to emulate a data terminal. {Software Technology Support Center}

end-to-end testing — Test activity aimed at proving the correct implementation of a required function at a level where the entire hardware/software chain involved in the execution of the function is available.

entry node — In a program-directed graph, a node which has more than one outway and zero inways. An entry node has an in-degree of zero and a non-zero out-degree. {Coverage}

entry segment — An entry segment, or logical branch is one which has no predecessors, a situation which can occur only at the entrance (i.e., invocation point) of a module. {Coverage}

environment clause — A clause in the ATS file that defines local environment variables that can be used as variables in the activation and evaluation clauses. {SMARTS}

equivalence classes partitioning — This involves identifying a finite set of representative input values that help to minimize the number of necessary test cases. {Software Technology Support Center}

error — A difference between program behavior and specification that renders the program results unacceptable. See **defect.**

error-based testing — Testing where information about programming style, error-prone language constructs, and other programming knowledge is applied to select test data capable of detecting faults, either a specified class of faults or all possible faults. {Software Technology Support Center}

error model — A model used to estimate the number of remaining errors, time to find these errors and similar characteristics of a program. {Software Technology Support Center}

error tolerance — See **robustness**. {STSC}

essential complexity — A measure of the level of 'structuredness' of a program.
{Software Technology Support Center}

essential edges — The set of paths that first includes each of the edges only on one of the original set of paths. {TCAT-PATH}

essential logical branch — A logical branch of a program that exists only on one path. Hence, execution of an essential logical branch is required to obtain complete segment (branch) coverage.

essential paths — The set of paths that include one essential edge; that is, an edge that lies on no other path. {TCAT-PATH}

essential segment — A segment of a program that exists on only one path. Hence, execution of an essential segment is required to obtain complete segment (branch) coverage.

evaluation clause — A clause in the ATS file that specifies how to assess the correctness of a test. {SMARTS}

evaluation — The process of examining a system or system component to determine the extent to which specified properties are present. {Software Technology Support Center}

exception report — A METRIC report which identifies source code procedures that exceed a user-defined metric threshold.

EXDIFF — The Extended Differencing System, a component of STW/Regression. EXDIFF compares two files and reports the difference between them, and it ignores differences that lie within a user-defined masked area.

executable statement — A statement in a module which is executable in the sense that it produces object code instructions. A non-executable statement is not the opposite; it may be a declaration. Only comments can be left out without affecting program behavior.

execution history report — This report shows how many times individual test cases have been run after they have been passed. This shows if test cases are being run needlessly. Identifies "spinning of the wheels."
{Software Technology Support Center}

execution verifier — A system to analyze the execution-time behavior of a test object in terms of the level of testing coverage attained.

exit logical branch — An exit logical branch is one for which there are no successor logical branches. This occurs only when the consequence of the logical branch is an exit from the module. {Coverage}

exit node — In a directed graph, a node which has more than one inway, but has zero outways. An exit node has an out-degree of zero and a non-zero in-degree. {Coverage}

exit structure — The exit structure of a program-directed graph is the set of segments which, if executed, lead unalterably to termination of program flow without involving subsequent repetition of any logical branches. {Regression}

explicit predicate

A program predicate whose formula is displayed explicitly in the program text. For example, a single conditional always involves an explicit program predicate.

A predicate is implicit when it is not visible in the source code of the program. An example is a program exception, which can occur at any time.

failure

The inability of a system or component to perform its required functions within specified performance requirements. A failure may result when a fault is encountered. {Software Technology Support Center}

faithful time recording

The capability of CAPBAK to record complete timing information about the CAPBAK session in such a way that it can be played back at identically the same rate it was recorded.

fault

An incorrect step, process, or data definition in a computer program. {Software Technology Support Center}

fault-based testing

Testing that employs a test data selection strategy designed to generate test data capable of demonstrating the absence of the pre-specified set of faults; typically, frequently-occurring faults. {Software Technology Support Center}

fault dictionary

A list of the faults that have occurred in a system and the tests that will detect them. {Software Technology Support Center}

fault masking

A condition in which one fault prevents the detection of another. {Software Technology Support Center}

fault tolerance

See **robustness**.

fault tree analysis

A form of "safety analysis" that assesses system safety to provide failure statistics and sensitivity analyses that indicate the possible effect of critical failures. {Software Technology Support Center}

feasible path A sequence of logical branches is logically possible if there is a setting for the input space relative to the first logical branch in the sequence, which permits the sequence to execute. {TCAT-PATH}

flow control When a terminal emulation program establishes communications with a mainframe application, it establishes flow control to prevent characters being lost. In some cases the mainframe application (or cluster controller) locks out the keyboard. This prevents the user from typing ahead; however, when CAPBAK is being used to record terminal sessions, the user is expected to wait for a response from the mainframe. The user thus imposes manual flow control to prevent data from being lost in cases where the keyboard is not locked.

When CAPBAK is being run in terminal emulation mode, a record of the manual flow control is stored in the keysave and response files. When CAPBAK is transmitting keys in playback, flow control is maintained by using this item.
{Software Technology Support Center}

full report A METRIC report which indicates a set of metrics for each of the modules in a given source file.

function call A reference by one program to another through the use of an independent procedure-call or functional-call method. Each function call is the “tail" of a caller-callee callpair.

hotkey window

When recording or playing back a test session, you can issue commands via function keys (i.e. your **F1** to **F10** keyboard functionkeys). During a recording session, you can use the function keys to bring up the hotkey window; mark the keysave file; select an image or window to synchronize during playback; save a partial image or window; save the root; and pause, resume or terminate the session.

During playback the function keys allow you to slow or to quicken the speed of playback; insert or append new keysave records into a keysave file; pause, resume or terminate a playback session. {CAPBAK}

function points

A measure of software size. Most appropriate for MIS applications. A product of five defined data components (inputs, outputs, inquiries, files, external interfaces) and 14 weighted environmental characteristics (data comm, performance, reusability, etc.).

Example from *Computer World*, March 8, 1993: A 1,000-line Cobol program would typically have about 10 function points, while a 1,000-line C program would have about eight. {Software Technology Support Center}

functional test cases

A set of test case data sets for software which are derived from structural test cases.

functional specifications

A set of behavioral and performance requirements which, in aggregate, determine the functional properties of a software system.

glass-box testing

A test method where the tester views the internal behavior and structure of the program. In using this strategy, the tester derives test data from an examination of the program's logic without neglecting the requirements in the specification. The goal of this method is to achieve a high test coverage examination of as many of the statements, branches, and paths.

grammar-based test — A testing method that generates test cases from a formal specification of a system or system component. {Software Technology Support Center}

Halstead metric — A measure of the complexity of computer software that is computed as **n * log n** where n is the product of the number of operators and the number of operands in a program. {METRIC}

history report — This SMARTS report shows a summary of all the test history entries stored in the designed test-log file. The display is always relative to a given node (group or test case). {SMARTS}

hit report — This report is used by TCAT, S-TCAT, and TCAT-PATH to identify all of the segments or call-pairs which were exercised in present and past tests. It analyzes both the trace file and archive file. {Coverage}

homogenous redundancy — In fault tolerance, realization of the same function with identical means; for example, use of two identical processors. {Software Technology Support Center}

hotkey window — When recording a session, this window pops up when the hotkey function key is pressed (defaulted to F1). It allows you to issue commands, including inserting comments, command, or conditional statements into the keysave file, to save an image for synchronization, save a partial image, mouse window, or the root window, to resume or end a recording session. {CAPBAK}

ICCM — Inter-Client Communications Conventions used by X-Windows.

image synchronization — The process by which a playback (e.g. from CAPBAK) is forced to wait until an image is completed.

in-degree — In a directed graph, the number of inways for a node. {Coverage}

incompatible segment — Two segments in one program are said to be incompatible if there is no logically feasible execution of the program which will permit

both to be executed in the same test. See also **essential logical branch**.

incremental analysis The partial analysis of an incomplete product to allow early feedback on the development of that product. {Software Technology Support Center}

independent logical branch pair A pair of logical branches is (sequentially) independent when there are no assignment actions along the first branch. This changes any of the variables used in the predicate of the second statement. {Coverage}

independent verification and validation Verification and validation performed by an individual or organization that is technically, managerially, and financially independent of the development organization. {Software Technology Support Center}

infeasible path 1. A logical branch sequence is logically impossible if there is no collection of input data relative to the first branch in the sequence, which permits the sequence to execute. {Coverage}

2. A sequence of program statements that can never be executed. {Software Technology Support Center}

inherited error An error that has been carried forward from a previous step in a sequential process. {Software Technology Support Center}

inspection/review A process of systematically studying and inspecting programs in order to identify certain types of errors, usually accomplished by human rather than mechanical means.

instrumentation The first step in analyzing test coverage is to instrument the source code. Instrumentation modifies the source code so that special markers are positioned at every logical branch or call-pair or path. Later, during program execution of the instrumented source code, these markers will be tracked and counted to provide data for coverage reports. {Coverage}

Integration Testing

Exposes faults during the process of integration of software components or software units and it is specifically aimed at exposing faults in their interactions.

The integration approach could be either bottom-up (using drivers), top-down (using stubs) or a mixture of the two. The bottom-up is the recommended approach. {Coverage}

interface

The informational boundary between two software systems, software system components, elements, or modules.

interface testing

Testing conducted to evaluate whether systems or components pass data and control correctly to one another. {Software Technology Support Center}

invocation point

The invocation point of a module is normally the first statement in the module.

invocation structure

The tree-like hierarchy that contains a link for invocation of one module by another within a software system.

ISO (International Organization for Standardization) 9126

ISO 9126 defines a set of six quality characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) and provides a framework for software quality assessments. ISO 9126 is a product of the ISO / International Electrotechnical Committee / Joint Technical Committee No. 1 Subcommittee on Software Engineering.
{Software Technology Support Center}

iteration level

The level of iteration relative to the invocation of a module. A zero-level iteration characterizes flows with no iteration. A one-level iteration characterizes program flow which involves repetition of a zero-level flow.

junction node

A junction node within a program-directed graph is a node which has an in-degree of two or greater and an out-degree of exactly one. {Coverage}

keycvt

A utility program for keystroke editing. **keycvt** transforms a keysave file into an editable ASCII version. {CAPBAK/UNIX, CAPBAK/DOS}

keypla

This command is used to read a keysave file and emit the characters to the screen. {CAPBAK/UNIX}

keysave file

See **ksv** {CAPBAK}.

keysave mode

The mode that enables the user to save every keystroke, and the time spent before each is typed in. {CAPBAK/DOS}

Kiviat chart

Kiviat charts provide a graphical means to view the impact of multiple metrics on a source code file or multiple files. In its summary report, each metric is represented by an axis and results are plotted with reference to user-definable upper and lower bounds. The Kiviat chart quickly identifies the metrics to focus on for a particular program. {METRIC}

ksv

A test script file automatically generated during the CAPBAK's recording session. A keysave file contains a sequence of event statements (including keystrokes, mouse movements and screen captures), which represent user input directed to the AUT. {CAPBAK, CAPBAK/UNIX, CAPBAK/DOS, and CAPBAK/MSW}

When a test is played back, the event statements in the keysave file are regenerated and the AUT executes the previously-recorded statements exactly as before.

length

Maurice Halstead defined the length of a program to be

N = N1 + N2

where N1 is the total number of operators and N2 is the total number of operands. This measure is used with METRIC to identify error-prone modules. {METRIC}

lifecycle The period that starts when a software product is conceived and ends when the product is no longer available for use. Test development, execution, and analysis involve the entire life-cycle. {Software Technology Support Center}

line mask An EXDIFF statement that permits masking a line or group of lines.

linear histogram A dynamically-updated linear-style histogram showing accumulating C1 or S1 coverage for a selected module. {Coverage}

logarithmic histogram A dynamically-updated logarithmic-style histogram showing each logical branch or call-pair hit in logarithmic form. {Coverage}

log file 1. A file used by SMARTS to record test history information.

2. An established or default SMARTS file where all test information is automatically accumulated.

logical block See **segment**.

logical trace An execution trace that records only branch or jump instructions.
{Software Technology Support Center}

logical units A logical unit is a concept used for synchronization when differencing two files with the EXDIFF system. A logical unit may be a line of text, a page of text, a CAPBAK screen dump, or the keys (or responses) between marker records in a keysave file. {Regression}

loop A sequence of segments in a program that repeats at least one node. See **cycle**.

loopback testing Testing in which signals or data from a test device are output to a system or component, and results are returned to the test device unaltered for measurement or comparison.
{Software Technology Support Center}

(M,N)-cycle

An M-entry, N-exit cycle in a flowgraph. A program is perfectly structured ("pure-structured") if it is composed of loops that involve only (1,1)-cycles. Most real-world programs contain many multiexit cycles, however. Some studies show that over 99% of programs are non-pure-structured. {TCAT-PATH}

make file

Most often, TCAT, S-TCAT and TCAT-PATH will be used to develop test suites for systems that are created with make files. make files cut the time of constructing systems, by automating the various steps necessary to build systems, including preprocessing, instrumenting, compiling and linking. All these steps can be written in a **make** file. {Coverage}

makeats

A SMARTS utility which, based on minimal information, generates the initial hierarchical test structure for an ATS file, as well as basic source, activation, and evaluation clauses.

manual analysis

The process of analyzing a program for conformance to in-house rules of style, format, and content as well as for correctly producing the anticipated output and results. This process is sometimes called code inspection, structured review, or formal inspection.

marker text

When CAPBAK/UNIX creates a marker record, it prompts the user for text to place in the marker record. This text is intended to be used for terminal emulator data flow synchronization and special differencing evaluation.

marker trigger key

When CAPBAK/UNIX is activated in marker trigger mode, it reads in a set of special keys from the marker trigger file, determining which keys should be used to impose flow control. Flow control is maintained by creating the marker records in the keysave and response files.

marker trigger mode

When CAPBAK/UNIX is activated in this mode, each time a marker trigger key is pressed, a marker record is recorded in the keysave and response files.

McCabe metric See **cyclomatic number**.

measure To ascertain or appraise by comparing to a standard; to apply a metric. {Software Technology Support Center}

menu trigger character Alt-M is typed to invoke the CAPBAK/DOS menu at any time.

metric A quantitative measure of the degree to which a system, component, or process possesses a given attribute (Maybe we should think of metrics as the clues to the scene of a crime. Gather them now or lose them forever, and who knows what clues will crack the case?). {Software Technology Support Center}

METRIC The Software Metrics Processor/Generator component of STW/Advisor. METRIC computes several software measures to help you determine the complexity properties of your software.

metric validation The act or process of ensuring that a metric correctly predicts or assesses a quality factor. {Software Technology Support Center}

mkarchive The TCAT utility creates null archive files.

mksarchive The S-TCAT utility creates null archive files. These utilities ensure that the coverage utility reports on all modules on your system whether or not they have been executed. Sometimes, when testing a subsystem, the initial tests do not touch every module in the program. When this occurs, the C1 of S1 measure will start at an artificially high level and, as the tests touch more modules, the C1 or S1 value will decrease. Although no logical branches or call-pairs are being hit, more modules are included in the percentage calculation, so the result value is lower. {Coverage}

module A module is a separately invocable element of a software system. Similar terms are procedure, function, or program.

mouse save file — The file of mouse movements (and associated timing information) captured during a CAPBAK/DOS session.

multi-unit test — A multi-unit test consists of a unit test of a single module in the presence of other modules. It includes (1) a collection of settings for the input space of the module and all the other modules invoked by it and (2) precisely one invocation of the module under test.

mutation testing — A method whereby errors are purposely inserted into a program under test to verify that the test can detect the error. Also known as "error seeding."
{Software Technology Support Center}

newly hit report — This report is used for TCAT and S-TCAT and identifies all the segments or call-pairs that are hit in the present test and which were not hit in any prior test.

newly missed report — This report is used for TCAT and S-TCAT and identifies what the current test "lost".

node — 1. A position in a program assigned to represent a particular state in the execution space of that program. {Coverage}

2. Group or test case in a test tree. {SMARTS}

node number — A unique node number assigned at various critical places within each module. The node number is used to describe potential and/or actual program flow. {Coverage}

non-executable statement — A declaration or directive within a module which does not produce (during compilation) object code instructions directly.

not hit report — A TCAT or S-TCAT report giving the names of logical branches or call-pairs "not hit" yet by any test.

object under test — See **test object**.

operational test and evaluation (OT&E)

The field test, under realistic conditions, of an item or component to determine effectiveness and suitability, and the evaluation of the results of the tests. {Software Technology Support Center}

operator interface analysis

1. A form of interface analysis that examines the usage of operators applied to data structures.

2. An analysis of the machine-human (operator) interface. {Software Technology Support Center}

out-degree

In a directed graph, the number of outways of a node. {Coverage}

output synchronization

The process by which a playback (e.g. from CAPBAK) is forced to wait until an expected window opening is completed.

outway

In a directed graph, an arc (edge) leaving a node. {Coverage}

P1 Coverage

Paragraph coverage, measured by TCAT/COBOL.

partition analysis

A program testing-and-verification technique that employs symbolic evaluation to provide common representations of a program's specification and implementation. {Software Technology Support Center}

partition analysis verification

The verification process used in partition analysis that attempts to determine the consistency properties that hold between a program specification and its implementation. {Software Technology Support Center}

Pascal

The ISO and/or ANSI standard Pascal programming language.

past test report This report lists information from the stored archive file for TCAT and S-TCAT. It summarizes the percentage of logical branches / call-pairs hit in each module listed, giving the C1 / S1 value for each module and the program as a whole. {Coverage}

path, path class An ordered sequence of logical branches representing one or more categories of program flow. {Coverage}

path predicate The predicate that describes the legal condition under which a particular sequence of logical branches will be executed. {Coverage}

path testing A test method satisfying coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes; one path from each class is tested.
{Software Technology Support Center}

pathcon A TCAT-PATH utility which generates a path's conditions.

pattern mask A pattern mask specifies one or more rectangular areas which are to be excluded from file comparison. (EXDIFF)

perturbation testing A test path adequacy measurement technique that proposes using the reduction of the space of undetectable faults as a criterion for test path selection and is intended to reveal faults in arithmetic expressions.
{Software Technology Support Center}

playback counter The time interval between two keystrokes recorded or played back by CAPBAK.

playback delay Minimum interval between keystrokes at playback time with CAPBAK.

playback mode The CAPBAK mode that enables the user to play back a file that contains all the keystrokes.

playback programming — A technique in which playback behavior is controlled by the use of various system calls placed in the keysave file. This provides an easy way for a user to playback a keysave file as a script that modifies behavior on the basis of system and environmental factors. {CAPBAK, CAPBAK/UNIX}

predecessor logical branches — One of many logical branches that precede a specified logical branch in normal (structurally-implied) program flow. {Coverage}

predicate — A logical formula involving variables/constants known to a module.

predicted length — Maurice Halstead theorizes that a well-written program with n1 unique operators and n2 unique operands should have a length of

N^ = [n1 x log2(n1)] + [n2 x log2(n2)]

{METRIC}

preview — A CAPBAK utility which simulates keysave file activity. The simulation shows the recording session's mouse movements, button and keyboard activities, and captured images.

program — See **module**.

program digraph — See **digraph**.

program predicate — See **predicate**.

program-sensitive fault — A fault that occurs when a particular sequence of instructions is executed. {Software Technology Support Center}

proof checker — A program that checks formal proofs of program properties for logical correctness. {Software Technology Support Center}

pseudocode — A form of software design in which programming actions are described in a program-like structure; not necessarily executable, but generally held to be humanly readable.

purity ratio

Maurice Halstead suggested that programs which are not the same length as predicted by N^ (see predicted length) are victims of impurities. The purity ratio is the ratio of N^ to N (predicted length / length). This measurement is used by METRIC to determine error-prone parts of code. {METRIC}

qualification

The process ensuring that a given software component, at the end of its development, is compliant with the requirements. The qualification shall be performed with appropriate and defined software components and subsoftware systems, before integrating the software to the next-higher level. The techniques for qualification are testing, inspection and reviewing.

quality assurance

A planned and systematic use of metrics to provide adequate confidence that an item or product conforms to established requirements. {Software Technology Support Center}

quick check mode

The CAPBAK and CAPBAK / MSW playback mode that replays a test in order to generate a new set of AUT responses. The new responses, the actual results, are compared with earlier results; that is, the expected results of the test.

This mode verifies an application's behavior by automatically comparing any currently-captured actual images, windows or ASCII characters with the image, window or characters that were captured and stored as the expected results.

record

This command is a program that records keystrokes being entered at a terminal and saves them in a keysave file format. It records and displays the responses from the remote machine, and saves them in a baseline file which can be used to synchronize playback. {CAPBAK / UNIX}

reference analysis

A form of static-error analysis that can detect reference anomalies; for example, when a variable is referenced along a program path before it is assigned a value along that path. {Software Technology Support Center}

reference listing report

A report produced by TCAT and S-TCAT which shows the coverage level achieved for all modules that are named in the specified reference listing.

regression report

This report shows only those tests whose outcomes have changed, thereby identifying bugs which have been fixed or introduced since the last time the test cases were activated. It lists test name, outcome, and activation date. {SMARTS}

regression testing

Testing which is performed after making a functional improvement or repair of the software. Its purpose is to determine if the change has regressed other aspects of the software.

As a general principle, software unit tests are fully repeated if a module is modified, and additional tests which expose the removed fault are added to the test set. The software unit will then be re-integrated and integration testing repeated.

resource file

For X Windows applications only, a file that contains a set of pre-determined values for parameters.

response file

CAPBAK captures images from the server and stores them in a response file. This file can be compared against the baseline file. {Regression}

return variable

A return variable is an actual or formal parameter for a module, which is modified within the module.

review

A planned activity during which a work product (strategy, budgets, requirements, design, code, test, support, training, etc.) is reviewed by the author and others involved in an attempt to gain an objective, varied, and complete perspective of the product.

Review is commonly referred to as code review, technical review, walk-through, inspection, etc. Walk-throughs are generally less formal and led by the author, while inspections are more formal and led by a more independent party.
{Software Technology Support Center}

robustness

1. The degree to which a system or component can still function in the presence of partial failures or other adverse, invalid, or abnormal conditions.

2. This is a characteristic of a product that enables it to more than meet minimum requirements. Customers really expect more than just minimum requirements, and although "satisfied" with minimum requirements, will look elsewhere next time if not "delighted" by the product.This is more a function of product design than design process. {Software Technology Support Center}

S-TCAT

The System Test Coverage Analysis Tool of the STW / Coverage tool group. S-TCAT measures the structural completeness of a test suite by reporting on the percentage of function call-pairs exercised.

S0 coverage

The percentage of modules that are invoked at least once during a test or during a set of tests. Measured by S-TCAT.

S1 coverage

The percentage of call-pairs exercised in a test as compared with the total number of call-pairs known in a program. This metric is calculated by S-TCAT. By definition the S1 value for a module which has no call pairs is 100% if the module has been called at least once, and 0% otherwise.

scover — An S-TCAT utility used to assess the value of S1 coverage.

screensave file — The file of screen images saved each time a trigger keystroke was hit during a CAPBAK/DOS session.

screensave mode — The CAPBAK/DOS mode that enables the user to save screens exactly as they appear immediately before a trigger key is pressed.

screensave trigger character(s) — Characters that invoke CAPBAK/DOS to save a screen of data, starting from the time the last keystroke was typed prior to when the current trigger was typed.

screensave trigger mode — When screensave trigger mode is on, any time a trigger character is pressed the system records a copy of the current screen contents.

segment — A [logical branch] segment or decision-to-decision path is the set of statements in a module which are executed as a result of the evaluation of some predicate (conditional) within the module. The segment should be thought of as including the outcome of a conditional operation and the subsequent statement execution (up to and including the computation of the value of the next predicate, but not including its evaluation in determining program flow). {Coverage}

segment instrumentation — The process which results in an altered version of a module, logically equivalent to the unmodified module but containing calls to a special data collection subroutine. This subroutine accepts information as to the specific segment sequence incurred in an invocation of the module. {Coverage}

semantic error — An error resulting from a misunderstanding of the relationship of symbols or groups of symbols to their meanings in a given language. {Software Technology Support Center}

sensitivity analysis — In safety analysis, analysis that assesses the potential impact of a potentially-critical failure on the ability of the system to perform its mission. {Software Technology Support Center}

Shift-PrtSc — This key terminates playback (abnormally) during playback mode, before the end of the session. {CAPBAK/DOS}

simulator — From Webster's, simulate 'to create the effect or appearance of.' Therefore, a machine that creates the effect or appearance of another. Similar to an emulator. Examples peripheral or network simulators.
{Software Technology Support Center}

smarts — This command invokes the ASCII version of SMARTS for UNIX and MS-DOS.

SMARTS — The Software Maintenance and Regression Test System of the STW/Regression tool set. SMARTS reads a user-designed test description file to find out what actions to take for each test or group of tests. This description file, called the Automated Test Script (ATS), is written in SMARTS' description language (similar to C language in syntax). In this file, the user can specify test commands to dispatch and test outcome evaluation methods.

At the user's command, SMARTS performs the pre-stated actions, runs a difference check on the outputs against the baseline, and accumulates a detailed record of the test results.

software subsystem — A part of a software system, but one which includes many modules. Intermediate between module and system.

software system — A collection of modules, possibly organized into components and sub-systems, which solves some problem or performs some task.

source clause

A clause in the ATS file that contains comments which may give some explanation to the origin of the test(s) invoked in each particular case. Most commonly the source clause is used to specify the purpose of a test.

The comments in a source clause are displayed by SMARTS when a test case activation is evaluated as a test failure this allows you to note which files need to be inspected. {SMARTS}

spaghetti code

A program whose control structure is so entangled by a surfeit of GOTO's that its flowgraph resembles a bowl of spaghetti.

statement complexity

A complexity value assigned to each statement which is based on (1) the statement type, and (2) the total length of postfix representations of expressions within the statement (if any). The statement complexity values are intended to represent an approximation to potential execution time.

statement testing

Testing designed to execute each statement of a computer program. See **test coverage**. {Software Technology Support Center}

static analysis

The process of analyzing a program without executing it. This may involve a wide range of analyses. The STW / Advisor suite of tools performs static analyses. {STATIC}

static frequency

Forced constant CAPBAK playback rate.

STATIC

The Static Analyzer for C reports on source code errors and inconsistencies that otherwise may go undetected. STATIC does a more detailed check than your compiler, including locating nonportable constructs. It also looks across multiple modules for bugs and so enjoys a perspective that your compiler does not have.

status report

The report presents the most recent information about executed tests. It contains test case name, outcome (pass / fail), activation date, execution time (seconds), and error number. {SMARTS}

stress testing Testing conducted to evaluate a system or component near, at, or beyond the limits of its specified requirements. {Software Technology Support Center}

strong typing Strong typing refers to typedef-based type checking for STATIC. {STATIC}

stw The command that invokes the GUI for STW.

subtest A part of a test that occurs between passing control to the test object and the return of control to the test environment.

successor logical branch One or more logical branches that (structurally) follow a given logical branch.

successor segment One or more segments that (structurally) follow a given segment. {Coverage}

Summary report This report is an accumulated account of the complexity measures for the entire program. {METRIC}

symbolic evaluation A technique of analyzing program behavior without executing the program. This generally results in the generation of a series of formulas that describe the input/output relationships in a software system.

symbolic testing A method of examining the path computation and path condition to ascertain the correctness of a program path. {Software Technology Support Center}

synchronization Synchronization is the process of maintaining coherence between a recording and the resulting system-under-test's responses. During playback of a test script, e.g. with CAPBAK, it is possible, due to many factors, for the playback process to "de-synchronize" with the synthetic input being reproduced by CAPBAK. Synchronization schemes are used to control the playback so that synchronization is not lost. CAPBAK has several ways to prevent loss of synchronization, among them "automatic output synchronization" and "image synchronization".

syntax error — A violation of the structural rules defined for a language. {Software Technology Support Center}

system testing — Verifies that the total software system satisfies all of its functional, quality attribute and operational requirements in simulated or real hardware environment.

It primarily demonstrates that the software system does fulfill requirements specified in the requirements specification during exposure to the anticipated environmental conditions. All testing objectives relevant to specific requirements should be included during the software system testing. Software system testing is mainly based on covered-box methods. {Coverage}

TCAT-PATH — The Path Test Coverage Analysis Tool of the STW / Coverage tool group. TCAT- PATH measures the thoroughness of your test case coverage by reporting on the paths exercised.

TCAT — The Test Coverage Analysis Tool of the STW / Coverage tool group. TCAT measures the thoroughness of your test case coverage by reporting on the percentage of logical branches exercised.

TDGEN — The Test Data Generator System which is a component of the STW / Advisor product line. TDGEN produces test data files in a user-designed format by replacing variable fields in a template file with random or sequential data values from a values file.

template file — A user-designed TDGEN file which indicates where selected values are to be placed within an existing test file. A template file provides a format for the generation of additional tests.

termination clause — A clause in the ATS file that allows for execution of concurrent processes in order to test the timing of specific test cases and terminate them if necessary. It is executed when a special termination command fails to complete normally. {SMARTS}

test

A [unit] test of a single module consists of (1) a collection of settings for the inputs of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the current testing. The intent of a test is to find faults in the module.

test case

Information about observable states, conditions, events, and data: all the causes (stimuli, inputs) that compel or allow software under test to perform one separately definable and measurable function. It should be possible to identify and track individual test cases. See test failure report. {Software Technology Support Center}

test coverage measure

1. A measure of the testing coverage achieved as the result of one unit test, usually expressed as a percentage of the number of segments within a module traversed in the test. {Coverage}

2. The degree to which a given test or set of tests addresses all specified requirements for a given system or component. Components are depth of coverage and breadth of coverage. Test coverage can also refer to code coverage, such as branch and statement test coverage, the results of which will be realized as a metric. {Software Technology Support Center}

test data set

A specific set of values for variables in the communication space of a module which are used in a test.

test development — The development of anything required to conduct testing. This may include test requirements, strategies, processes, plans, hardware, software, procedures, cases, documentation, and maintenance strategies (essentially, the same or similar efforts as that of any product development, except that usually, but not always, the test products are used by internal customers rather than external. The people involved in the test development effort should use a lifecycle approach; essentially, the same as in the product development effort, and the test products should be treated as assets to be managed rather than expenses to be pared). {Software Technology Support Center}

test failure report — A report containing a unique identifier (ID) for each failure, an ID of the software under test, an ID of the test case, the date and time of the failure, symptoms of the failure, and a classification of the failure (criticality, priority, etc.). {Software Technology Support Center}

test harness — A tool that supports automated testing of a module or small group of modules.

test object — The central object on which testing attention is focused. Also known as **object under tes**t.

test path — A test path is a specific (sequence) set of segments which is traversed as the result of a unit test operation on a set of testcase data. A module can have many test paths. {Coverage}

test procedure — The formal or informal procedure that will be followed to execute the test in question. This is usually a written document that will allow others to carry out the test with a minimum of training and confusion. There will be a separate test procedure for each case, which will be noted in the test plan. {Software Technology Support Center}

test purpose — The free-text description of the purpose of a test, normally included in the source clause of an ATS file that is processed by SMARTS.

test readiness review

In *Technical Review and Audits For Systems, Equipments, and Computer Software,* the test readiness review comes after the critical design review and before the functional configuration audit. By default, one test readiness review is conducted for each CSCI. This is to verify that the item is ready for formal testing and approval. {Software Technology Support Center}

test status report

Shows metrics for work products and work processes. Shows quantity for information at a glance; e.g. total test cases, total run, total passed. Generally, shows little or nothing about quality of tests.
{Software Technology Support Center}

test stub

A test stub is a module simulating the operations of another module invoked within a test. The test stub can replace the real module for testing purposes.

test target

The current module (system testing) or the current segment (unit testing) upon which testing effort is focused.

test target selector

A function which identifies a recommended next testing target.

testing techniques

Can be used in order to obtain a structured and efficient testing which covers the testing objectives during the different phases in the software life cycle.

testability

A design characteristic which allows the status (operable, inoperable, or degrade) of a system or any of its subsystems to be confidently determined in a timely fashion. Testability attempts to qualify those attributes of system design which facilitate detection and isolation of faults affecting system performance.

top-down testing

The testing starts with the main program, which becomes the test harness. The subordinated units are added as they are completed, and testing continues. Stubs must be created for units not yet completed.

This strategy results in re-testing of higher level units when more lower level units are added. The adding of new units one by one should not be taken too literally. Sometimes a collection of units will be included simultaneously, and the whole set will serve as test harness for each unit test. Each unit is tested according to a unit test plan, with a top-down strategy.

trace file

A file containing the most recent test run of trace coverage information. {Coverage}

trigger key

These are user-defined keys that CAPBAK/DOS uses to record screens or to write marker records to the keysave file.

true-time recording

The capability of CAPBAK to record complete timing information about the CAPBAK session in such a way that it can be played back at the same rate it was recorded.

T-SCOPE

The Test Data Observation and Analysis System provides dynamic visualization of test attainment during unit testing and system integration. It is a companion tool for TCAT, S-TCAT and TCAT-PATH.

unconstrained paths

The set of edges that will imply execution of other edges in the program. {TCAT-PATH}

unit test

See **test**.

Unit Testing

This procedure is meant to expose faults in each software unit as soon as the unit is available, regardless of its interaction with other units. The unit is exercised against its detailed design, and by ensuring that a defined logic coverage is performed.

Informal tests on module level which will be done by the software development team are necessary to check that the coded software modules reflect the requirements and design for that module. Clear-box (glass-box) oriented testing, in combination with at least one closed-box method, is used.

unreachability

A statement (or segment) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or segment) to be traversed.

validation

The evaluation at the end of the development process to ensure compliance with software requirements. The techniques for validation are testing, inspection and reviewing.

values file

A user-designed TDGEN file which indicates the actual test values, test value ranges or test value generation rules for the creation of additional test files.

verification

The process of determining whether or not the products of a given phase of the software development cycle meet the implementation steps, and can be traced to the objectives established during the previous phase. The techniques for verification are testing, inspection and reviewing.

vertex

See **node**.

white-box testing

See **glass-box testing**.

Xcalltree

An S-TCAT utility displaying a software system's caller-callee dependence structure (may be called **Xcgpic** in older versions of STW).

Xcapbak

Command to invoke the GUI version of CAPBAK/X. See also **Xrecord, Xplabak,** and **Xdemo.**

Xdemo	A variation of **Xplabak** that does not require access to licensing, but does check to assure that the keysave file played back has been processed by **Xdemo.key**.
Xdemo.key	A command that is part of the CAPBAK/X package that authorizes a keysave file for playback by **Xdemo**.
Xdigraph	A TCAT or TCAT-PATH utility used to create a picture of a directed graph (may be called **Xdigpic** in older versions of STW.)
Xexdiff	The EXDIFF command to perform a pixel-by-pixel comparison of two saved images.
Xkiviat	The command to invoke the Kiviat chart generator supplied with the METRIC product.
Xmask	The EXDIFF command to mask out regions. This is useful whenever there are differences between two files that are inconsequential, such as a date, header, footer, or path name.
Xplabak	A CAPBAK/X command which reads a keysave file and plays back the captured keystrokes, mouse movements and images.
Xrecord	A CAPBAK/X command which allows you to record keystrokes, screen captures, and mouse movements and save them to a keysave file.
Xsmarts	The command to invoke the GUI version of SMARTS.
Xstatic	The command to invoke the STATIC component of the STW system.
Xtcat	The command to invoke the GUI version of Xtcat.
Xtcatpath	The command to invoke the GUI version of TCAT-PATH.
Xtdgen	The command to invoke the TDGEN system within the STW package.
Xtscope	The command invoking the GUI for T-SCOPE.

CHAPTER 7

The TestWorks Index

A Quantitative Quality Index for Your Application

7.1 Abstract

Assessing the relative quality of a software system is a complex but important matter in software engineering. To make rational decisions about complex software requires an approach that combines analysis of product properties with analysis of the underlying software construction process. A weighted figure of merit software quality index — The TestWorks Index™ — offers an attractive approach because it takes into account software quality metrics, process assessments, and practical considerations.

7.2 Introduction

This paper attacks a common problem in software development: How good is the quality of a specific software application? How do I know it? How can I make decisions about it? (For example, should it be released yet?) How can I estimate what to do next on my product based on where I am now?

The approach to this problem is to assess the quality of a particular application by weighing the answers to questions that address BOTH the properties of the application itself and the characteristics of the process used to produce it.

7.3 Quality Process Assessment Methods

The SEI CMM and the ISO-9000 type quality process models are based on examining the process that produces the product. This approach is based on the well-documented fact that a better industrial process tends to produce a better product, and that continual incremental improvements to that process tend to lead to continual incremental improvements in its product. This simple method can account for spectacular quality — and consumer acceptance — gains.

While this technique is clearly valid in general terms, sometimes good processes produce bad products and bad processes produce good products. This happens annoyingly often in software products, perhaps because some of the intermediate elements of the process can be very difficult to measure.

So Quality Process Models accept such exceptions, focusing on the main point: Improving the process improves the product. And the exceptions are anomalies.

7.4 Product Methods

The Product Analysis approach, often called the metrics approach or the static analysis approach, takes the opposite tack: look at the final product only, and base decisions about its quality on what is actually there, regardless of how it got there. After all, the final source code itself completely determines what an application can do. Regardless of how it was produced, regardless of the methodology or tools or process used to make it, the actual quality of a software product is determined directly by its own internal, intrinsic properties.

So, even if it is junky, spaghetti code hacked together by rank amateurs, if it works well then it works well. Who needs a fancy software process, anyway?

Simply put, quality is determined in the contest of the marketplace.

Of course, given that quality is implicit in the as-built product, we still have to find a way to measure it if we want some measure of control over the result. To measure the quality of an application by its structural properties or content, we use software metrics (e.g., cyclomatic complexity, size metrics — there are hundreds of possible metrics).

Yet we know from experience that sometimes applications with very poor software quality metrics, e.g., with **E(n)** in the 100's and Halstead weights in the 1,000,000's, are perfectly good, perfectly reliable code and don't have any field problems. At the same time, some products that are high-scoring by every metric one can find are complete disasters.

Automated static analysis attempts to mechanize code inspection methods, but most implementations tend to find far too many things wrong. Long lists of product features that are "dangerous" but not "fatal" suggest not that a product will fail in the field, but that the builders don't mind living on the edge.

For example, if you always fixed everything that /bin/lint said ought to be fixed, assuming you could afford to do that, your software quality would be sure to go up, but there would be no guarantee that your delivered functionality would change for the better. It might change for the worse! You could be spending money to improve the product and changing it for the worse.

The paradox is that just having measurable high-quality code that meets small-scale and large-scale quality guidelines is no more a measure of field product success than having a perfect manufacturing process that is building a "Monday Morning Car."

7.5 Process/Application Assessments

A combined process / application assessment is a way out of this mess. A software manager needs to take into account the following factors: HOW a product was built; WHAT its characteristics are; and WHY better quality is important; and what the producers and their management — the team — FEEL about how good the team / product combination is.

A multi-faceted assessment method can be fooled too, of course, but its strength is that it focuses on perceived quality-key aspects of both process assessment and product assessment.

- HOW a product was built is addressed by questioning whether certain basic quality-oriented processes are used in its construction. For example: Was coverage analysis done? What coverage metrics, and how thoroughly? (How well was it tested?) Was automated regression done? How thoroughly? (How well was it tested?)
- WHAT its characteristics are is addressed by identifying certain basic metrics that pertain to its actual functional content. For example: What is the average **E(n)** for functions? (How complex is the code?) What is the calltree aspect ratio? (How is the code shaped?)
- WHY better quality is important is, simply put, an assessment of how critical the product is to the producers. If product quality isn't important, then quality shouldn't be of any concern. But if a product is intended for a life-critical application, and therefore has to have very high quality, then the need for quality has to be taken into account.
- HOW the team FEELS about the product they've built affects a lot of things — and will be controversial to measure. The very best software development efforts have often been fielded by dedicated, talented, teams who believe in their work.

7.6 The Methodology

The TestWorks Index™ is a balanced, weighted, experience-determined estimate of selected factors and uses a combination of estimates, measurements, and process-characterizations to come up with a quality figure that can be used — within a single organization — to compare products. The TestWorks Index™ is the average score obtained on a simple question list, where specific quantitative responses based on current engineering experience assign "points". The more points scored, the better the product. The average point count (the total points scored divided by number of questions) is the TestWorks Index™.

In engineering this has been called a "Figure of Merit (FOM)" and the notion of using FOMs has a long tradition of use in comparing complex things. From assessing competitive proposals (which are scored according to weighted averages), to determining plant efficiency, engineers take the practical approach even when it is known there is no theoretical solution.

Some benefits the TestWorks Index™ offers in assessing *your* products are:

- You can compare two products from your production shop in a quantitative way.
- You can adjust the point values, if you like, to match your own experience. Or you can add quality-related factors if you like (but you can't take them away or you may destroy the integrity of the index).
- You can use the evidence internally for your own purposes. You don't have to expose yourself to SEI or ISO-9000 audit processes.
- You CAN have or seek a high SEI rating or obtain ISO-9000 approval. The TestWorks Index™ assesses product / producer / process quality in a way that combines many of the features of these two important quality-related standards.
- You can use The TestWorks Index™ now, with data that you probably already have available, to come up with meaningful comparative estimates.
- By adopting The TestWorks Index™ you create an internal culture in which achieving practical goals for a software product becomes a common goal. You might even say that the product had been "TestWorked" to a certain level in your advertising.

7.6.1 Caveats

If you read this, you may get the impression that this is a foil to encourage users to buy the TestWorks™ product line. True, you may want to use some TestWorks™ products in your software production line to make sure you obtain a better TestWorks Index™ score. But you could use our competitors' products as well. The TestWorks Index™ is NOT tied use of TestWorks™ products.

7.6.2 How It Works

The TestWorks Index works as shown on the following chart. The factors on the chart are metrics that you can measure, or are assessments you can make, in a straightforward way. Detailed explanations of the terms follow the chart.

As you read the explanations, think of a specific project that you're working on, and try to calculate its The TestWorks Index™ score as you go along.

TestWorks™ PRODUCT QUALITY INDEX™ — DEFINITION AND EXPLANATION

This table shows how to compute the TestWorks Product Quality Index™ — the "TestWorks Index." The index gives an organization the chance to assess how well their internal software quality process is actually used on a particular product level and to compare their indicated product quality against current likely industry standards.

Each factor in the TestWorks Index is evaluated so that a simple point — scored between 0 and 100 — can be assigned depending on the answers. The overall TestWorks Index is the arithmetic average of the individual scores on each factor (you add them up and divide by the number of factors used).

WORKSHEET TestWorks Index™ EVALUATION FACTOR	50 Points	60 Points	70 Points	80 Points	90 Points	100 Points	My Score
F1 Cumulative C1 (Branch Coverage) Value for All Tests	>25%	>40%	>60%	>85%	>90%	>95%	
F2 Cumulative S1 (Callpair Coverage) Value for All Tests	>50%	>65%	>80%	90%	95%	98%	
F3 Percent of Functions with E(n) < 20	<25%	>25%	>50%	>75%	>90%	>95%	
F4 Percent of Functions with Clean Static Analysis	<20%	>20%	>30%	>40%	>50%	>60%	
F5 Last PASS / FAIL Percentage	>25%	<25%	>50%	>75%	>90%	>95%	
F6 Total Number of Test Cases / KLOC	>10	<10	>15	>20	>30	>40	
F7 Calling Tree Aspect Ratio (Width/Height)	>1.0	<1.25	<1.5	<1.75	<2.0	>2.0	
F8 Current Number of OPEN Defects / KLOC	>5	<5	<3	<2	<1	<0.5	
F9 Path Coverage Performed for % of Functions	<1%	>2%	>5%	>10%	>15%	>25%	
F10 Cost Impact / Defect:	>$100K	>$50K	>$25K	>$10K	>$1K	<$1K	
Total Points	—>	—>	—>	—>	—>	—>	

7.7 Index Criteria

Not just any list of scored questions should quality as a valid comparative index. To qualify as an effective indicator some constraints have to be put on The TestWorks Index™ or its home-brewed derivatives to make sure that it isn't manipulated to favor a particular process or product feature or quality assurance approach. The constraints that make sense are the following:

- There can be no more than 10 (ten) factors. This keeps the arithmetic simple (managers need this, technical people think).
- At least half of the criteria must be completely quantitative, objective, measurable, repeatable; i.e. not subject to any kind of judgment **[Q]**.
- At least three of them have to deal with something about the static (non-dynamic) characteristics of the product **[S]**.
- At least three of the criteria have to somehow involve actual tests of (experiments on, examinations of, executions of) the as-built software product — you can't design quality into a product without *some* kind of checking, and you can't score high on the index without something that works and does something **[T]**.
- At least three of the factors have to deal with something about the dynamic behavior of the product, how it actually works when you run it **[B]**.
- At least one of them have to deal with something about how the product was put together, i.e., about the process used in constructing it **[P]**.
- At least one of the factors has to deal with a measure of the quality "need" imposed from some outside force, e.g. the cost of repairing a defect in the field or the life-criticality of the application **[$]**.
- Some of the factors can address several of these criteria: they don't have to be unique to each area.
- No more than one of the qualifying factors can be a "wild card" and need not meet any of the above criteria (The error will be still no worse than 10%).
- At least eight of the ten factors have to have some non-zero value. (You can leave out two if you have to!).

7.8 Explanation Of Terms

Here are short explanations of the above indicated measures. (The keys **[B, S, T, P, $]** are explained on the preceding page.)

F1 *Cumulative C1 (Branch Coverage) Value for All Tests* **[B, S, Q]**

This is the total C1 value achieved for this product on all tests. E.g. as measured by TCAT. Note that statement coverage is NOT usable because it understands results by half or more. Statement coverage is accepted as inadequate.

F2 *Cumulative S1 (Callpair Coverage) Value for All Tests [***B, S, Q]**

This is the total C1 value achieved for this product on all tests. E.g. as measured by TCAT. Note that we are counting the connects between caller and callee, not just whether a function was ever called (which is called module testing). Module testing is accepted as inadequate.

F3 *Percent of Functions with* ***[E(n)*** *< 20* **[S, Q]**

This measures the structural complexity for all functions or modules or methods in the current application. Experience shows that the cyclomatic complexity **E(n)** = E - N + 2 > 20 implies a "complex function" — not necessarily bad, but a potentially troublesome problem if a high percentage of the individual functions has this value. Though not necessarily harmful too high a percentage of "too complex" functions can be a serious warning sign of trouble ahead.

F4 *Percent of Functions with Clean Static Analysis* **[S, Q]**

Static analysis finds a broad class of defects that may cause trouble in the future. Many errors found by static analysis are non-critical, but too many static analysis detections is an indicator of poor quality. The measurement made here requires that a certain percentage of functions be subjected to some form of static analysis.

F5 *Last PASS / FAIL Percentage* **[B, P, Q]**

This is the total number of tests that PASS vs. the total number of tests available, as would be measured by the test controller, e.g. SMARTS. Tests PASS if they run as expected, and produce output close enough (as determined by the programmable differencer) to the baseline to be acceptable.

F6 *Total Number of Test Cases / KLOC* **[T, Q]**

This is a measure of the degree to which you have thoroughly tested the software relative to its size measured in 1000's of lines of code (KLOC). Most software is very poorly tested, so it may not take a great many tests to score high on this measure.

F7 *Call Tree Aspect Ratio* **[S, Q]**

This is a measure of the "verticalness" of the call-tree of the package, with packages that have a less vertical structure (and thus more independently testable) viewed as superior. The vertical height is the maximum depth calling tree (this is shown in Xcalltree / WINcalltree), and the horizontal width is the largest number of functions on any level in the tree. If the tree has multiple roots (as it will likely have in most modern applications) then the average values for all possible roots is taken.

F8 *Current Number of OPEN Defects / KLOC* **[T, J]**

No software product / project is perfect; this metric indicates how many defects per KLOC are open that are critical. An open defect typically means it is reported, reproduced, but unresolved with no work-around available to the user.

F9 *Path Coverage Performed for% of Functions* **[P, J]**

For almost all packages some critical functions or modules require full path coverage, but not all. This measures the percentage of all functions for which some form of path coverage has been performed. Remember, path coverage is NOT the same as branch coverage; path coverage would be measured by something like TCAT-PATH.

F10 *Cost Impact / Defect* **[$, J]**

This is an indication of how critical a serious software defect might be, expressed in monetary terms, i.e. in terms of the direct cost of any defect. Note that the scale used tends to take points *away* for the most-critical kinds of projects; this is done so that the more critical projects receive the greatest attention. Any product which is "life critical" gets 0 points.

7.9 Examples

Here's how the The TestWorks Index™ works when applied to some (admittedly, 100% fictitious) example projects.

Quick-And-Dirty Order Tracker

This project was done by one programmer and involved putting together an order tracking system for his company. Done in C and using an ASCII interface to a standard library of C++ functions...

Here's a summary of the scores that resulted:

```
Cumulative C1 (Branch Coverage) Value for All Tests  25
Cumulative S1 (Callpair Coverage) Value for All Tests70
Percent of Functions with E(n) < 20                  50
Percent of Functions with Clean Static Analysis      50
Last PASS / FAIL Percentage                          80
Total Number of Test Cases / KLOC                    50
Calling Tree Aspect Ratio (Width/Height)             60
Current Number of OPEN Defects / KLOC                50
Path Coverage Performed for % of Functions           50
Cost Impact / Defect:                               100
TOTAL POINTS SCORED                                 535

TestWorks Quality Index                            53.5
```

Test Tool Vendor's Coverage Analyzer

This coverage analyzer, a very popular one, is a sophisticated compiler-based product...

```
Cumulative C1 (Branch Coverage) Value for All Tests  70
Cumulative S1 (Callpair Coverage) Value for All Tests85
Percent of Functions with E(n) < 20                  80
Percent of Functions with Clean Static Analysis      60
Last PASS / FAIL Percentage                          50
Total Number of Test Cases / KLOC                    80
Calling Tree Aspect Ratio (Width/Height)             60
Current Number of OPEN Defects / KLOC                95
Path Coverage Performed for % of Functions           50
Cost Impact / Defect:                                80

TOTAL POINTS SCORED                                 710

TestWorks Quality Index                              71
```

Bedside Cardiac Monitor

Think of this as a life-critical application...

```
Cumulative C1 (Branch Coverage) Value for All Tests  100
Cumulative S1 (Callpair Coverage) Value for All Tests100
Percent of Functions with E(n) < 20                   80
Percent of Functions with Clean Static Analysis       80
Last PASS / FAIL Percentage                           90
Total Number of Test Cases / KLOC                     85
Calling Tree Aspect Ratio (Width/Height)              60
Current Number of OPEN Defects / KLOC                 95
Path Coverage Performed for % of Functions            60
Cost Impact / Defect:                                 50

TOTAL POINTS SCORED                                  800

TestWorks Quality Index                               80
```

7.10 Connecting To Reality

The hard part comes when trying to connect with reality. The main question everyone asks is, ``**How reliable will my application be in the field?**''

As students of software quality know very well, this is a very deep question to which there are few definitive or even suggestive answers. Instead, about the best we can do is associate a particular process's TestWorks Index score with a likely estimate of reliability based on judgment and experience.

An initial experimental *estimate* of this is done in the attached chart.

Time will tell whether the numbers are too high or too low. Time will tell if the reliability values correspond to the SEI/CMM levels, or if the achieved reliability is too low or too high. And, time will tell whether that application of relatively simple quality filters will achieve, or won't achieve, the expected effect often enough to be relied upon.

But in any case, making the attempt to tie these essential ingredients together is totally essential.

TestWorks™ — PRODUCT APPLICATION PROFILE

This table shows the components of TestWorks and how they are applied in an increasingly sophisticated sequence of quality processes, all calibrated with the TestWorks Index™ minimum value. For comparison, the corresponding CMM value is shown as a range (because of uncertainties in the CMM definitions).

PROCESS LEVEL	ADVISOR	PLANNING	REGRESSION	COVERAGE	PROBABLE DETECTION EFFICIENCY
Introductory Process TestWorks Index: > 40 (CMM Level 0-1)			CAPBAK/Lite	TCAT/Lite	10%-40%, 2 filters
Basic Process TestWorks Index: > 50 (CMM Level 1)		Manual	CAPBAK/Standard	TCAT/Lite	25%-50%, 2.5 filters
Standard Process TestWorks Index: > 60 (CMM Level 1-2)	METRIC	SMARTS	CAPBAK/Standard	TCAT/Standard	50%-90%, 3+ filters
Advanced Process TestWorks Index: > 70 (CMM Level 2-3)	METRIC	Informal	CAPBAK/Professional	TCAT/Professional	75%-90%, 4 filters
Critical Process TestWorks Index: > 80 (CMM Level 3-4)	STW/Advisor	Semi-Formal Specification-based	STW/Regression	STW/Coverage	80%-95%, 5 filters
Life-Critical Process TestWorks Index: > 90 (CMM Level 4-5)	STW/Advisor	Formal Specification-Based	STW/Regression	STW/Coverage	90%95%, 5 filters

CHAPTER 8

TestWorking Scribble Using TestWorks™ for Windows

This comprehensive application note explores how the entire suite of Software Research's testing tools interacts with the sample application **Scribble**.

8.1 Sample Application: Scribble

Scribble employs many features of Microsoft Foundation Classes (MFC). There are several versions of **Scribble**, which become increasingly complex in each chapter. MSVC++ 5.0 has eight chapters; The present example uses Chapter 8. MSVC++6.0's **Scribble** example is in Chapter 7.

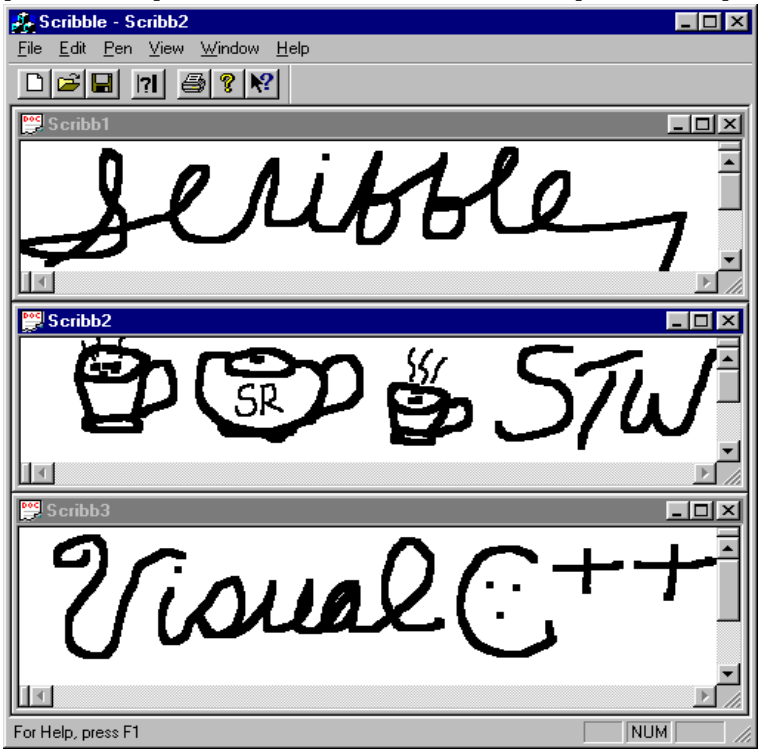

FIGURE 8 Scribble, Chapter 8

8.2 Overview

This Application Note shows results from TestWorks for Windows' comprehensive treatment of **Scribble**:

- **CAPBAK/MSW**™, TestWorks' capture/playback tool for MS Windows, records all user activities during a test of **Scribble**, including keystrokes and mouse movements and capturing baseline images for comparison (against future tests).
- **CBDIFF**™, TestWorks' extended file differencing system for MS Windows, performs graphical file comparisons while discarding extraneous discrepancies during the differencing process.
- **CBVIEW**™ displays images captured during recording of **Scribble** and playback sessions.
- **SMARTS/MSW**™, TestWorks' software maintenance and regression test system, runs a hierarchy of tests of **Scribble** and displaying current and cumulative results.
- **TCAT**™, TestWorks' test coverage analysis tool, displays its analysis of test coverage of **Scribble** in graphical displays, including the following:
    - Call-trees that show the caller-callee structure of the program
    - Directed graphs that show the control-flow structure of program modules
    - Coverage charts that display numerical results of the amount of coverage provided by a given test

8.3 CAPBAK™ for Windows and Scribble

CAPBAK™ helps you to design and develop tests that automate the testing process. It captures bitmap images and ASCII values, as well as all user activities during the testing process including keystrokes, mouse movements, and verification information into a "C" language script that is easily understood. The captured images provide baselines against which future reruns of the tests are compared. Future tests then entail the playing back of these test sessions. **CAPBAK**'s automatic synchronization ensures reliable playback of these test sessions, allowing tests to be run unsupervised as many times as the tester wants.

When a test is played back, the same input statements are regenerated and sent to your program, the application under test (AUT). The AUT executes the previously recorded statements, exactly as before. Therefore, all keystrokes and mouse movements corresponding to the recorded input are played back. **CAPBAK/MSW**™ ensures reliable playback because it has a built-in synchronization feature, and it permits user-defined synchronization points.

For comparing the behavior of two AUT versions, **CAPBAK/MSW**™ recaptures the same images during playback as you captured during the recording session. You can look at these corresponding images and automatically or manually compare them to determine if there are any discrepancies during the two sessions.

Optical Character Recognition capabilities are also available for use in text comparisons.

8.3.1 Record/Playback Modes

CAPBAK™ uses multiple modes for capture / playback: TrueTime Object and Character Recognition.

With TrueTime, the keyboard and mouse inputs are replayed exactly as recorded by the tester. Playback timing is duplicated from the server's own timing mechanism, allowing tests to be run as if executed by a real user. The results of the tests indicate any variances from the baseline cases, permitting the tester to determine the implication of those differences. Therefore, if a button were moved to a different location in the window, it would be flagged as change. This TrueTime user-level testing has been augmented with the inclusion of Optical Character Recognition Mode (OCR Mode).

Character recognition allows the test to search for items that may have moved or changed fonts since testing a previous version of the application. Test activities reflect the contents of the screen as processed through a built-in OCR engine.

The character recognition software also allows for pixel images to be converted to ASCII character for later analysis to determine the success of a particular test. By using the OCR technology, the same technology used for document scanners, **CAPBAK**™ now has the ability to recognize any font without special training regardless of size or other characteristics.

8.3.2 Multiple Synchronization Modes

Software Research has licensed OCR technology from Xerox Imaging Systems (XIS) to provide generalized character recognition capabilities. **CAPBAK**™ offers more synchronization modes than any other capture/playback tool.

OCR implementation with **CAPBAK/MSW**™ allows the user to do the following:

- Automatic Event Synchronization automatically synchronizes on event-sensitive environment differences, such as new windows popping up in varying locations.
- Image and Window Synchronization waits for the contents of a screen fragment or window to update and match the baseline image.
- Timing Synchronization Playback adjusts timing to different values, allowing overall playback to be slowed down or adjusted after events such as mouse clicks and carriage returns.

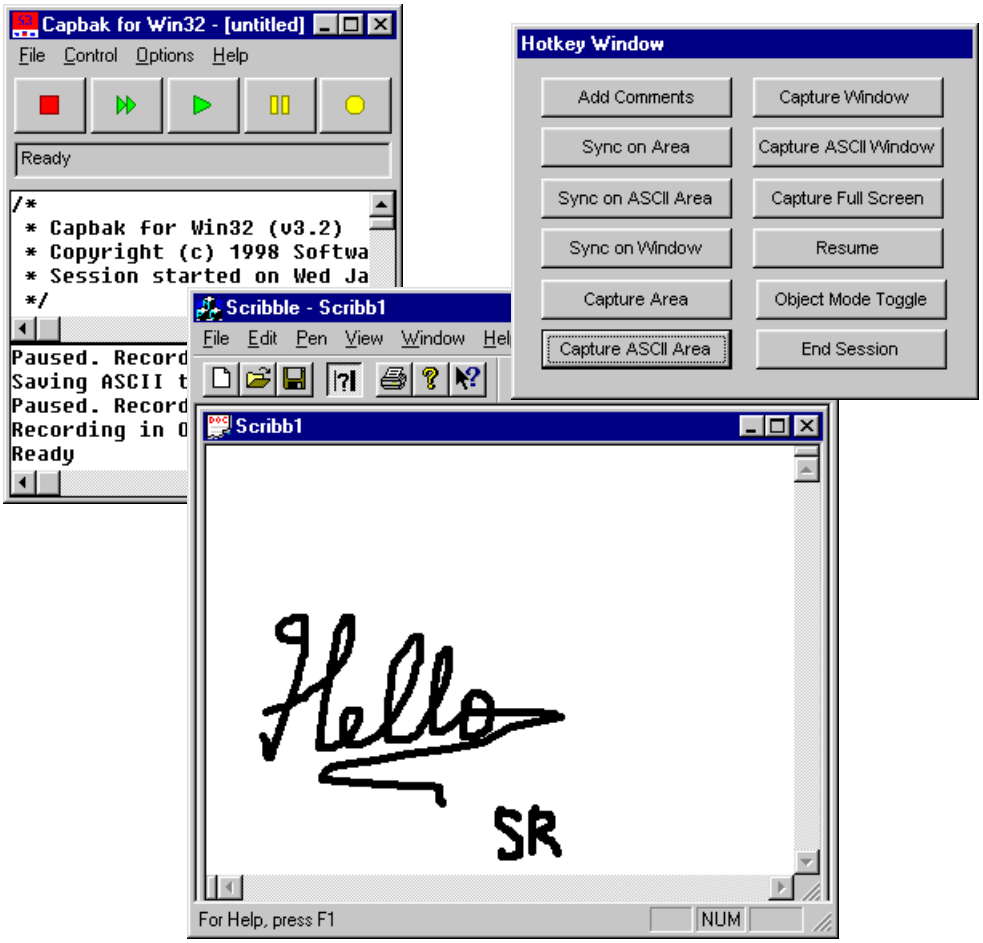

FIGURE 9 Illustrates capture and playback commands recording Scribble in Truetime mode

CAPBAK's multiple synchronization modes ensure a reliable playback, so tests can be run unattended. Response images corresponding to baseline images are automatically captured. Comparison of the baseline and response images is done automatically and results are written to a log file. This log file allows the tester to quickly identify where tests have failed.

The user has complete control over the tests. Both recording and playback sessions can be sped up, slowed down, paused, or aborted so that the user can process other commands.

The following components complement **CAPBAK**'s recording features:

SMARTS™ organizes **CAPBAK**'s test scripts into a hierarchy for execution individually or as a part of a test suite, and then evaluates each test according to the verification method selected.

CBDIFF™ compares bitmap images, while discarding extraneous discrepancies during the differencing process.

CBVIEW™ displays images captured during recording and playback sessions.

To verify that tests have successfully played back, **CAPBAK**'s **CBVIEW**™ utility displays captured images of test sessions. Tests are further verified with the **CBDIFF**™ utility, which compares baseline and response file images for differences. **CBDIFF**'s masking capability disregards those areas of images that are not necessary for comparison, such as time or date changes.

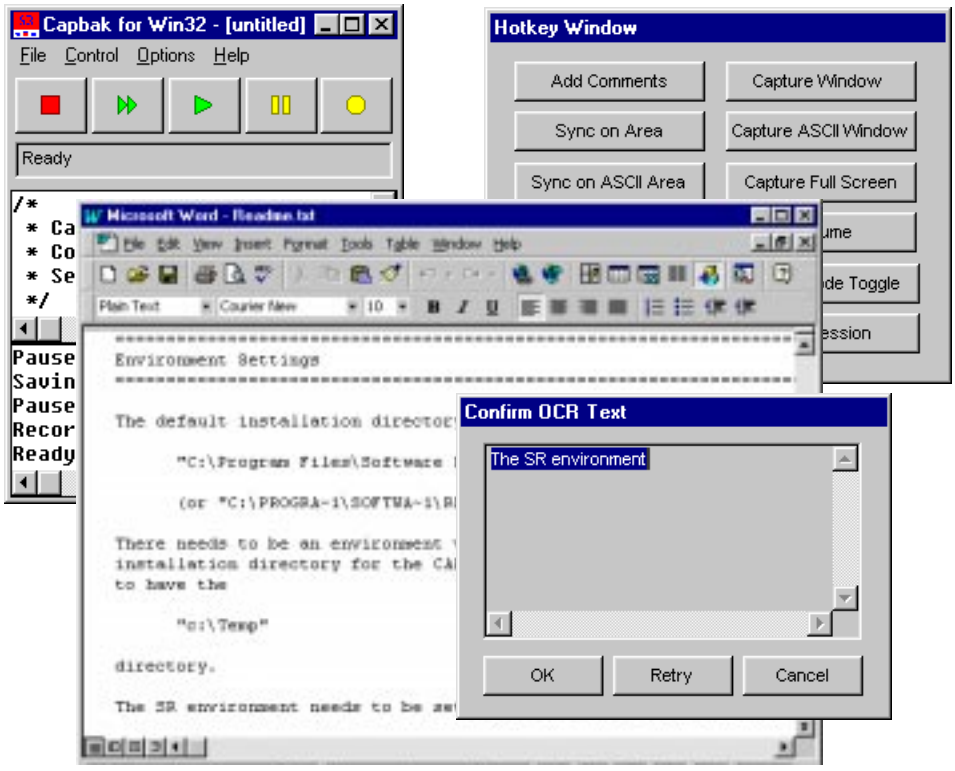

FIGURE 10 **CAPBACK'S** Hotkey Window and Optical Character Recognition (OCR) Technology from Application Under Test (AUT)

8.4 The CBDIFF Utility

CBDIFF™, TestWorks' extended file differencing system, is a test evaluation facility that extends commonly available file differencing facilities. **CBDIFF**™ provides masking options that allow the user to specify areas within ASCII or image files to be ignored during the differencing process.

CBDIFF™, an advanced differencing utility, has the following capabilities:

- Pixel-by-pixel comparison of image files
- Detection of color differences
- Line and byte comparisons for ASCII files
- Advanced masking capabilities
- Combines with ***EXDIFF***™ utility for differencing ASCII files

You can display and then compare the differences between expected AUT images that were captured during a recording session to the actual images that were captured during playback with the **CBDIFF**™ utility. Differences such as changed dates, times, and file list differences can be masked out. Likewise, comparisons between text files can also be conducted.

FIGURE 11 **CBDIFF** illustrates the differences between the two screen shots, shown in **CBView** (BO1) and **CBView** (BO2).

8.5 SMARTS/MSW™: Streamlining the Testing Process

SMARTS/MSW™ automatically executes tests, thereby saving time. It automates the testing process by reading a user-designed test description file, referred to as an Automated Test Script (ATS). The ATS is written in **SMARTS/MSW**™ code, which is a subset of the C programming language.

SMARTS™ organizes and manages an extensive number of test scripts into an efficient hierarchy for the purpose of automating the testing process. The test script "test tree hierarchy" emulates the modularity and functionality of the tested application. It allows test cases to be supplemented with activation commands, comparison arguments, system calls, evaluation methods, and control structures (`for`, `while`, `if, break`, `return`, `expressions,` and compound statements).

SMARTS™ allows the user to create a hierarchical tree of test cases and to execute those tests individually or in groups. **SMARTS** also captures results from the tests and allows reports to be created based on the most current run of tests, historical reports of all test runs, or summary information on overall test success / fail rates.

SMARTS™ has the flexibility to perform setup and cleanup activities prior to each test and to call any application or script to perform the verification in order to determine pass / fail results.

All **SMARTS**™ commands are context sensitive. Test execution and reporting are based on the selection, either of an individual test or group of tests, from the displayed test script "test free hierarchy". When executed, **SMARTS**™

- Performs the prestated actions.
- Runs a difference check on the application outputs against the baseline.
- Accumulates a detailed record of the test results.

Using the STW / Regression comparison utility **CBDIFF**™, differencing capabilities can be extended to ignore specified character strings and text differences in ASCII files and masked areas in image files.

8.5.1 SMARTS™ Reports

SMARTS™ saves a detailed record of test outcomes and timing statistics to a default log file and generates the following comprehensive reports:

- Latest reports
- All reports
- Regression reports
- Summary reports
- Time reports
- Failed reports

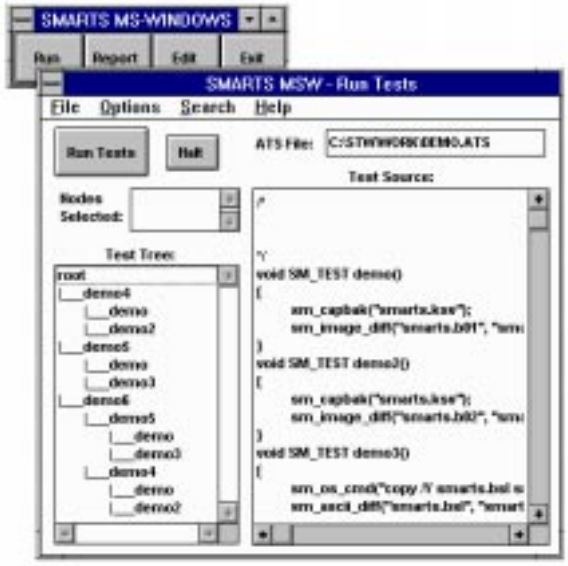

FIGURE 12 Run Tests Window

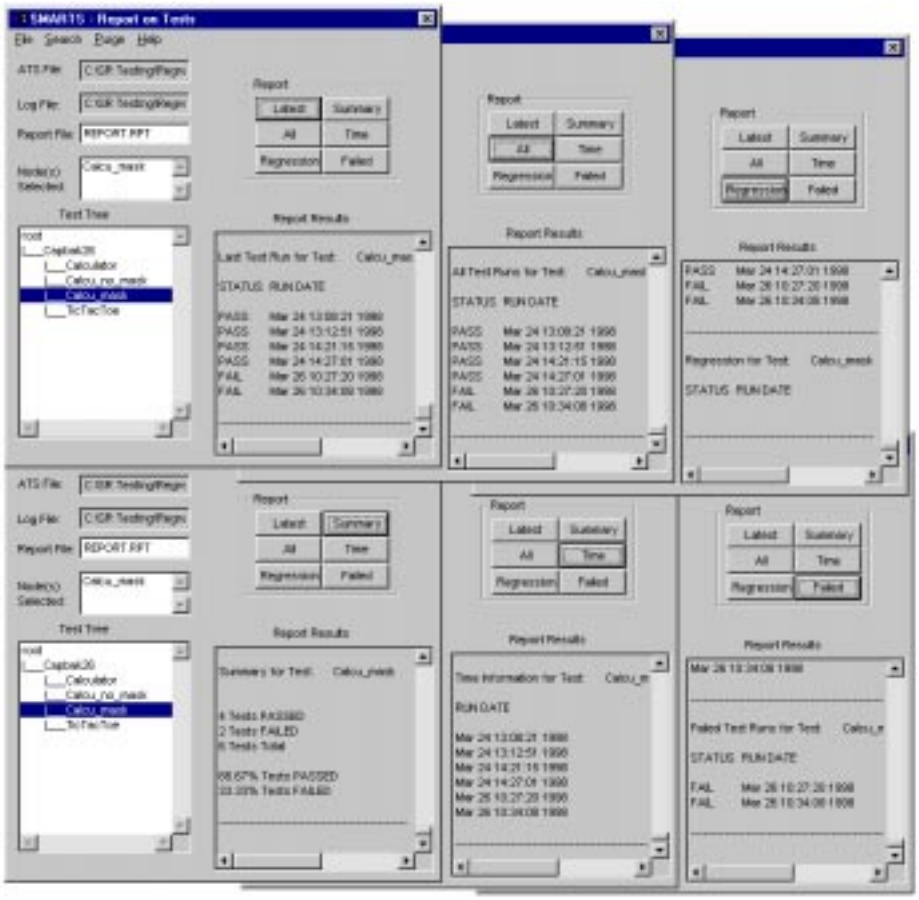

FIGURE 13 SMARTS' Report Windows: Latest, All, Regression, Summary, Time, and Failed

8.6 TCAT C/C++ and Scribble

TestWork's test coverage analysis tool (**TCAT C/C++**™) measures the completeness of test cases and identifies unexcised code. It ensures tests that are more diverse than those chosen by reference to functional specification alone or those based on a programmer's intuition.**TCAT C/C++**™ allows the user to create and view the coverage reports, calltrees, and directed graphs of the trace files that **TCAT C/C++**™ for Windows creates when an instrumented application is tested. It ensures that they are as complete as possible by measuring them against a range of high quality test metrics such as the following:

- Coverage at the logical branch (or segment) level and the call-graph level, employing the C1 metric

  You can choose to test a single module, multiple modules, or the entire program using C1 metric.

- Coverage at the call-pair level employing the S1 metric

  After individual modules have been tested, you can test all the interfaces of the system using the S1 metric.

- Dynamic visualization of test attainment during unit testing and system integration

  This test visually demonstrates, in realtime, such things as segments and call-pairs hit / not hit.

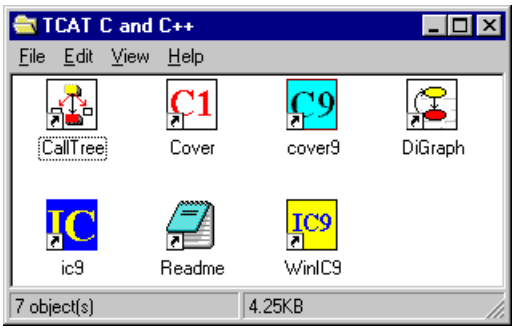

FIGURE 14 **TCAT C/C++** Program Group

8.6.1 Instrument Using WinIC9

WinIC9 instruments the application under test in order to produce trace files of the test.

During instrumentation, **TCAT C/C++**™ for Windows inserts function cells (special markers) at every logical branch (segment) in each program module. Instrumentation also creates a reference listing file, which is a version of your program that has logical branch-marking comments added to it in a manner similar to the code added to the instrumented version. Extensive logical branch notation and sequence numbers are also added.

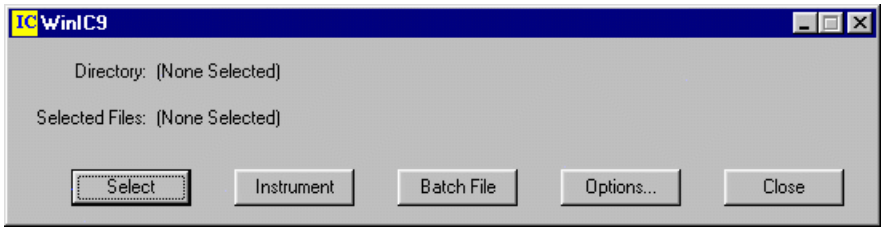

FIGURE 15 WinIC9 Window

Instrumenting Scribble will not change its functionality. When compiled, linked, and executed, the instrumented application will behave as it normally does, except that it will write coverage data to a trace file.

By runningWinIC9, you are exercising logical branches in the program. The more tests in your test suite, the higher the coverage. This test information is then written to a trace file. From the information stored in the trace file, you can generate coverage reports. In general, the reports give the following information:

- Reports included in the current iteration
- A summary of past coverage runs
- Current and cumulative coverage statistics
- A list of logical branches that have been hit

8.6.2 Viewing Coverage Reports with Cover

Cover displays trace and coverage information on your development project in a treelike list. **TCAT C/C++**™ does the following:

- Measures the completeness of test cases
- Improves quality by focusing the creation of additional tests
- Saves time by not creating tests for code already exercised
- Improves process by providing metrics measurements

You can click on a branch of the list to expand it, show its content, and contract it. The several fields in the report have the following meanings:

- Hits: the number of times the segment and call pair were executed during the test
- Count: the number of segments and call pairs within the function
- C1: the percentage of branch coverage for each function
- S1: the percentage of call pair coverage for the function

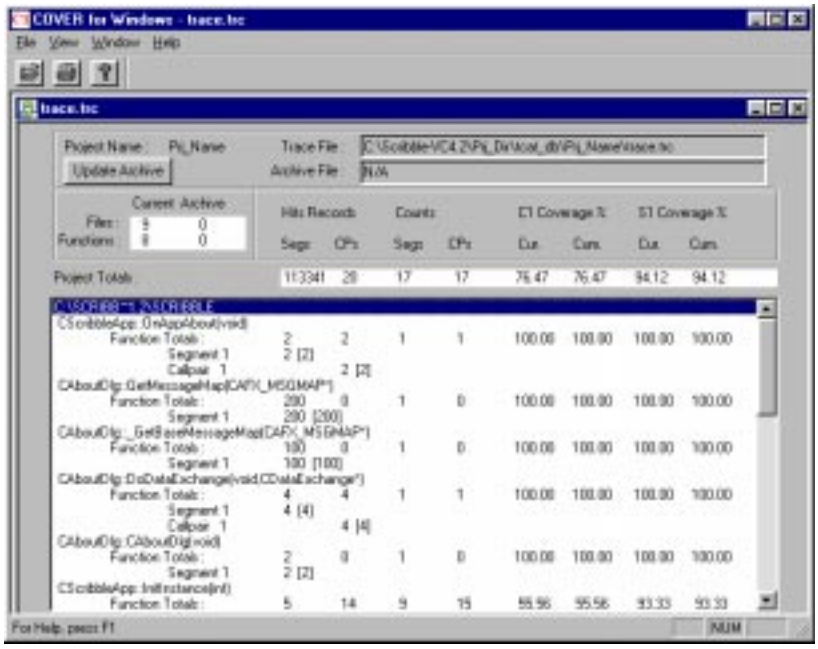

FIGURE 16 Coverage Report on Scribble, with One Function Expanded to Show Segments.

TCAT C/C++ for Windows draws digraphs based on archive files that are created during instrumentation.

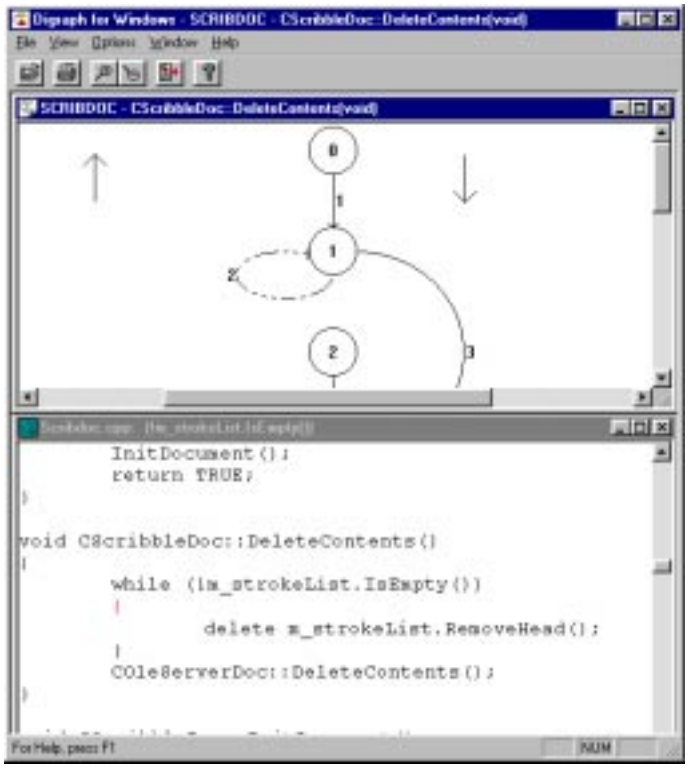

FIGURE 17 Digraph Main Window

8.6.3 Viewing A Calltree

TCAT C/C++™ for Windows generates a calltree graph for each segment of your executable during instrumentation and stores it in a separate archive file. Once the instrumented application has been exercised, you can display a calltree window for a specified program segment.

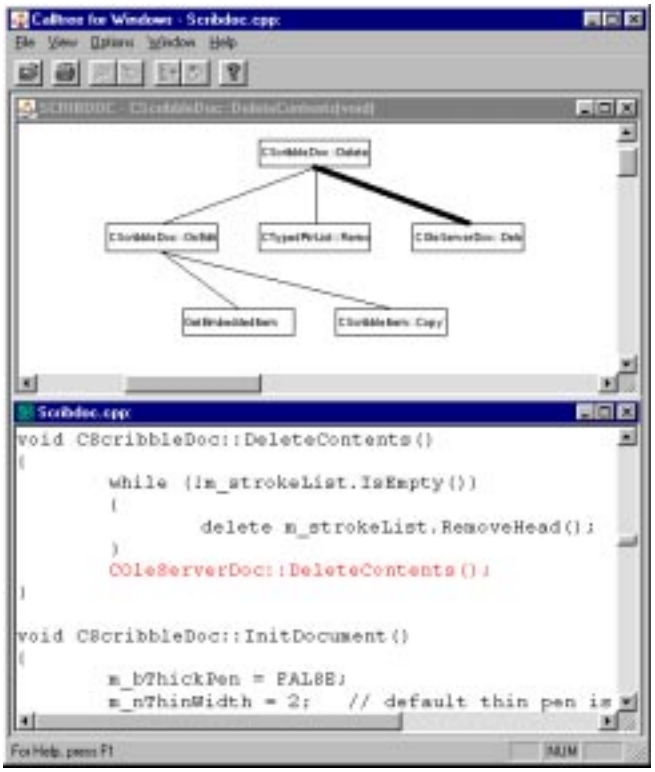

FIGURE 18 Displaying a Calltree

For each node in your calltree, you can easily display an associated directed graph.

TCAT C/C++™ for Windows allows quick navigation from graphs to source code.

Index

S

T

W