Design and Development of the CDE 1.0 Test Suite

Testing a product whose parts are being developed in four different environments that have different test tools and test procedures requires setting some rigorous test goals and objectives at the beginning of the project.

by Kristann L. Orton and Paul R. Ritter

The Common Desktop Environment (CDE) test team was given the challenge of designing and organizing the development of an automated regression test suite for CDE 1.0 components. This project contained all the usual problems involved in designing and creating a test suite for a large and complex desktop environment, plus a number of challenges that came up because the development was a joint project between four different companies. ***Article 6*** provides some background about the creation of the CDE project and the four companies involved.

Several goals for the tests were developed early in the project that influenced the design of the test suite, the choice of software testing tools, and the implementation of the testing process. The rationale for some of these objectives will be developed in more detail in later parts of this article.

In setting the goals for the test suites, we determined that they had to be:

- Easy to develop. There wasn't a lot of time in the CDE schedule.
- Robust. Tests shouldn't start failing because of a minor visual change, a different font selection, and so on.
- Reliable. Tests should find the defects located in the code, but they should not report a defect when there isn't one.
- Consistent operation. Even though four companies would be developing tests, individuals at each company had to be able to run the entire test suite, not just the parts written at their own site. It was not acceptable to make someone learn four different ways to run tests.
- Consistent design and implementation. At the end of the joint development, personnel at each site would get engineering responsibility for the whole suite, including those portions written at other companies. It was important that the tests be written such that an experienced test engineer at one company could easily understand the internal workings of the tests written at other sites.
- Portable. The test suite not only had to run on each of four reference platforms (one from each company), but also had to be easily ported to other nonreference platforms.
- Maintainable. The test suite was not just for the CDE 1.0 sample implementation, but was to be the basis for company products or later versions of CDE.* It had to be relatively painless to update the tests if they had defects, enhance the test suite for new functionality, and so on.

CDE Components

The CDE components were the software under test (SUT) for this project. From a testing point of view, CDE components can be divided into three types: CDE API (application programming interface) components, CDE GUI (graphical user interface) components, and graphical API components. CDE API components have no effect on the desktop, that is, no visual impact. An example of a CDE API component is the ToolTalk® API.** CDE GUI components present desktop graphics that can be manipulated by the user, resulting in graphical changes on the desktop. Examples of CDE GUI components are the file manager and the icon editor. Graphical API components consist of a library of functions like a standard API, except that calls to these functions usually do result in visual changes on the desktop. Examples of this type of CDE component include the DtHelp library and the DtWidget library.

Tools, Utilities, and Environment

This section describes the tools selected for the test suite, the utilities created to augment the tools, and the structure of the test implementation and operating environment.

* After the sample implementation of CDE, each participating company was expected to go off and productize their own CDE desktop.

** ToolTalk is the messaging system used by CDE.

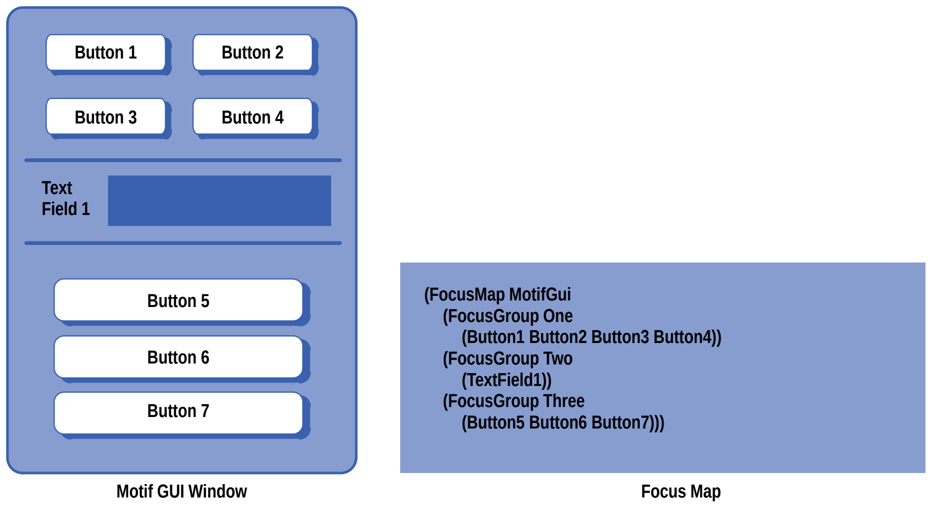
Synlib

The Synlib API was one of the most important tools used in the development of the test suite. Synlib is a C-language interface that provides a means to simulate programmatically the actions of a user with a GUI. It also contains features that allow the test program to monitor the state of the desktop, such as watching for windows to appear or disappear, checking the title of a window, or checking other desktop features.

Synlib was used to develop all the tests that required either manipulation of objects on the desktop or verification by checking the state of objects on the desktop. This turned out to be the majority of the tests. The CDE test team chose Synlib as the GUI test tool because:

- Synlib is portable. The only requirement for using Synlib is that the server be equipped with either the XTEST or the XTestExtension1 extensions,* which are used by Synlib to do such things as simulate keyboard and mouse events. Synlib was functional on all partner systems.
- Synlib is unencumbered. Synlib was developed at HP and was made available in source code form to each of the CDE partners without charge.
- Test development could begin immediately. Engineers could begin writing tests based on the components' specifications. Since Synlib is not a record-and-playback method, a functioning CDE component is not required for initial test development.
- Synlib reduces dependence upon image capture and comparison. Many of the earlier tools for testing GUI components use a method that depends heavily on capturing correct screen images during an initial phase, then comparing that image to the screen in a later test run. Experience shows that this method is quite fragile and likely to produce many false failure reports. With Synlib, other ways of checking for screen state make the need for image capture and comparison much less important.
- Synlib contains a feature that allows position independent manipulation of desktop items. A special data file called a *focus map* is created that defines the keyboard traversal for setting focus to items within a window (Fig. 1).** With the focus map, the test program can set the keyboard focus to a particular button or text field without needing to know its physical location in the window. Since the focus map is a separate data file that is read by the test program, changes in the component that result in changes in the traversal order can be incorporated into the tests by editing the focus map file. Recompiling the test is not necessary. Since the focus map file is platform independent, there only needs to be one focus map file per platform.

Fig. 1. *An example of a focus map. For keyboard traversal purposes, the Motif client can organize its objects in focus groups. In this example there are three focus groups. The focus map file allows the test designer to name the objects that can receive input focus and define the correct combination of keystrokes (tabs and up and down arrows) needed to shift the focus from one object to another. For example, the test designer can specify that the input focus should be moved to object* MotifGui.Three.Button6 *without any information about the location of the object.*

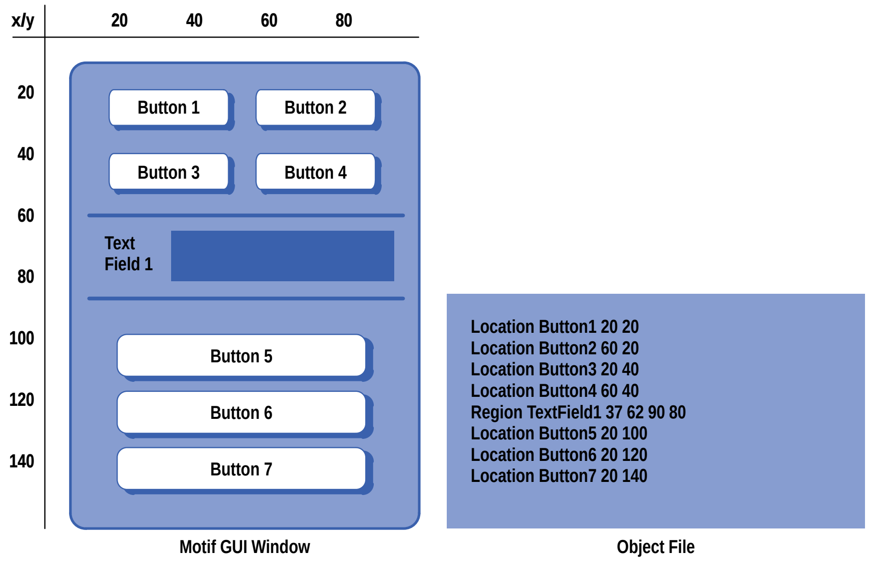


```
(FocusMap MotifGui
    (FocusGroup One
        (Button1 Button2 Button3 Button4))
    (FocusGroup Two
        (TextField1))
    (FocusGroup Three
        (Button5 Button6 Button7)))
```

* XTest and XTestExtension1 are industry-standard extensions to the X Server, which allow a client access to server information.

** Keyboard focus is the term used to describe which of possibly several objects in a window will receive the results of keystrokes. For example, a Motif window may have three different text fields, and the field that has the keyboard focus will get the characters typed by the user.

Fig. 2. *An example of an object file. The test designer can use the location in the object file to set the focus to Button6 as is done in Fig. 1. However, if the GUI is redesigned and objects are moved, the test will fail until the object file locations are updated.*



```
Location Button1 20 20
Location Button2 60 20
Location Button3 20 40
Location Button4 60 40
Region TextField1 37 62 90 80
Location Button5 20 100
Location Button6 20 120
Location Button7 20 140
```

Items that cannot be reached by keyboard traversal need a position. Synlib features the concept of an object file. This is a separate data file that defines locations (x,y pairs) or regions (rectangles) relative to a window origin (Fig. 2). The mouse pointer can be directed to an item by referring to its location defined in the object file. Any change in a component that changes locations requires that the object file be edited, but the test code remains position independent. Also, since the locations vary somewhat for each platform, there needs to be one object file for each platform.

The Synlib test tool is described in ***Article 8***.

The Test Environment Toolkit

The test environment toolkit (TET) was chosen as the test harness for the CDE test suite. C-language or shell-based tests were installed in the toolkit and run using the toolkit's utilities. TET is maintained by the X Consortium and is available in the public domain, is portable across many platforms, and has a fairly large group of users. In the X world TET is becoming a de facto standard for test suites.

TET brought two important pieces of functionality to the CDE project. First, it provided an API for journaling, which gave us a standard way to report test results. This was an important aspect in keeping the tests consistent because no matter which site wrote the test, the results journal would have the same format and could be interpreted and summarized using the same tools.

The second piece of functionality provided by TET was a mechanism whereby tests could be collected into groups (called scenarios) and run as a set with a single command. Since the complete test suite for CDE contains literally thousands of individual tests, the ability to run a batch of tests in a single step was essential.

Test Framework

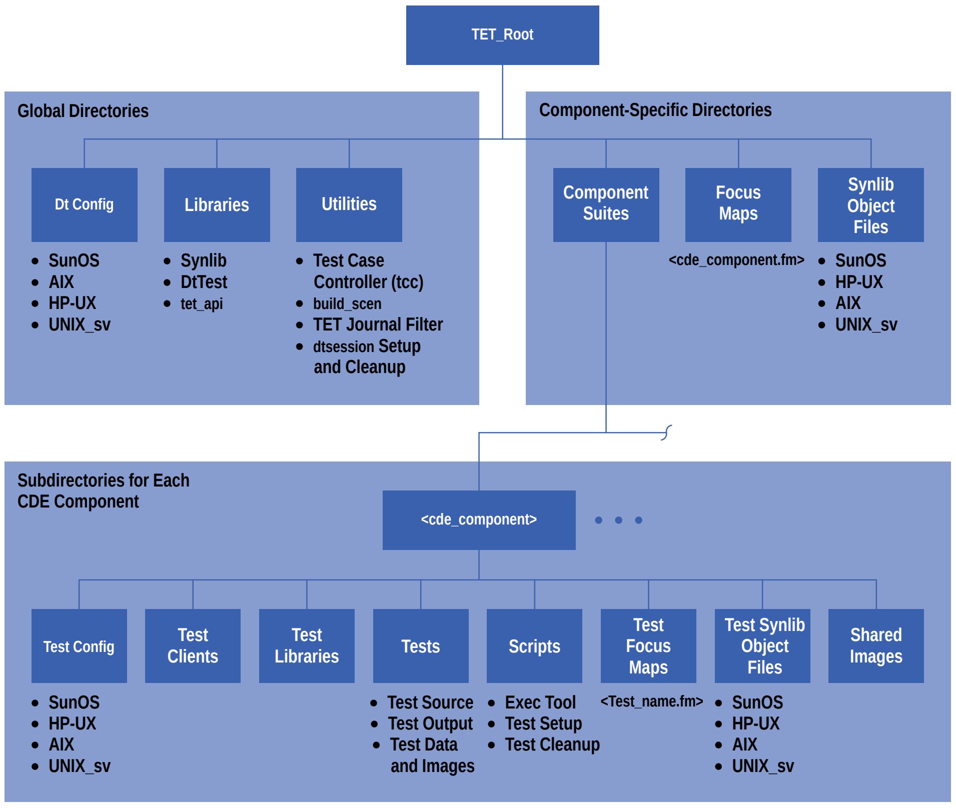
The selection of TET as the framework toolkit for our test environment meant that we had to choose a specific structure for our test tree. This gave us a starting point for building our test suite and provided a logical structure that was expandable and at the same time, robust and maintainable. Fig. 3 shows our test framework.

Located at the root of the tree were the test and environment configuration scripts and libraries containing common APIs. These utilities and libraries were available for use by all of the test suites and, with a few exceptions (Synlib and TET), were developed jointly by the CDE test team. Each function and binary had to have, at a minimum, an associated man page describing its functionality and use.

The following sections describe the test tree components in Fig. 3 as they were used for testing CDE 1.0.

Libraries. Synlib and DtTest are described in more detail in other sections of this article. The tet_api library was modified slightly for our environment, adding code that allowed the tests to be run from a build tree. The area occupied by the build tree was not writable by the test user, so any tests could not expect to use that space for creating test run data.

Fig. 3. Shared test tree framework.



Utilities. The utilities were important for providing a common way for tests to set up and clean up their environment. The build_scen script was used to create a new TET scenario file by searching through the test files for tet_testlist structures, preventing the tests and their associated scenarios from getting out of synchronization. The TET journal filter searched through the usually huge journal file and separated failures and their associated assertions, improving the efficiency of test result analysis.

Dt Config. The scripts in the Dt Config directories contained environment variables used by TET and Synlib and for satisfying platform-specific environment requirements. There was a convenience function in the DtTest library that set up global pointers to these environment variables, making them accessible from any test.

The other pieces of the tree are specific to each component test suite. A basic map of each component lives at the top of the tree, providing maximum leverage for suites testing interclient communication and doing any system testing.

Focus Maps. Each CDE component having a GUI front end was required to provide a focus map file that was accessible to all the test suites. For example, the calculator component had a file in the directory named dtcalc.fm containing the keyboard focus map for the calculator GUI.

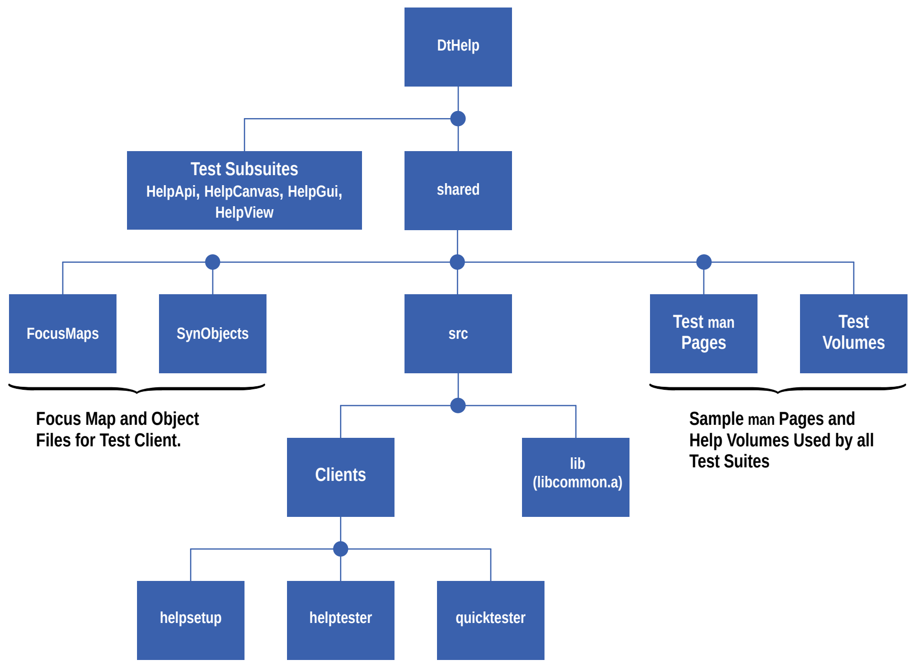
Synlib Object Files. Each CDE component provided an object file containing aliases for the interesting x,y locations in its GUI. Since these objects were platform dependent and varied greatly from platform to platform, object files had to be generated for each of the reference platforms.

Component Suites. There was a subdirectory under comp_suites for each GUI and library in CDE. For most suites, this was the top of the test tree branch. For suites having functionality that was more complex than could be encompassed in one suite, this directory contained several subdirectories for better granularity of the covered functionality. A good example of this is the CDE help component, which is made up of a help viewer and several APIs. This suite was broken into four subsuites with a top-level library and test client that was accessible by all suites (Fig. 4).

Each test suite directory had a regimented structure so that the suites could be efficiently maintained in a consistent manner. At the same time, we tried to provide some flexibility so that creative solutions could be engineered for the myriad of difficulties faced by the test-suite developers.

Test Libraries. As the test developers learned the test tools and framework, they found a lot of commonalties among the tests in their suites. These commonalities were gathered into a common library so they could be used by other tests. Some of these commonalities included the way tests were started and closed, and the verification methods used on components.

Fig. 4. *Component suite for CDE help.*


DtHelp
Test Subsuites
HelpApi, HelpCanvas, HelpGui, HelpView
shared
FocusMaps
SynObjects
src
Test man Pages
Test Volumes
Focus Map and Object Files for Test Client.
Sample man Pages and Help Volumes Used by all Test Suites
Clients
lib (libcommon.a)
helpsetup
helptester
quicktester

Test Clients. Many of the suites that tested the CDE APIs used a test client to exercise their code and verify their assertions. For example, to test the subprocess control daemon, a couple of clients were developed to send and receive messages, verifying the validity of data sent at the same time.

Test Config. Like the Dt Config scripts described above, Test Config scripts set up environment variables that were optionally platform dependent and used by the tests. An associated function was added to the test library to set up environment variables as global variables accessible within their tests in the same manner as that in the DtTest library.

Test Focus Maps. The majority of the test focus maps under this subdirectory were used for defining the keyboard traversals of test clients.

Test Synlib Object Files. Object files contain aliases for x,y locations in GUI components. Test developers used these location aliases to define areas in the GUI that needed to be tested for cut and paste and drag and drop operations and regions for image comparisons.

Shared Images. Although we encouraged test developers to limit the number of image comparisons, this method had to be used when validation was not possible by any other means. The next best thing was to reuse images so that more than one test could use the same image for verification.

Tests. The tests directory contained the tests and other files that made up a test suite. A README file in the directory described the test suite structure and any special environment considerations that needed to be taken into account when running the tests. Optionally, there was a directory at the top level that contained manual test descriptions. The tests were intended for tests in which automation was not feasible (e.g., testing the animation look when a file was dropped on a drop site).

The test source code lived in the tests directory. This directory contained as many subdirectories as needed. Developers were encouraged to divide their tests into groups based on a component's functionality, making it easy to determine which tests covered which piece of code. This directory also contained an image directory for images that applied to only one assertion and a data directory that contained validation data such as window titles and text for text comparisons. The test output directory was a place holder used by the test case controller (tcc) at run time. The tests stored information about failures in these directories and the test case controller copied them into a results directory at run time, facilitating easy results analysis. The following listing is an example of a typical test source file.

```
 #include <DtTest.h> /* DtTest library header */
 /* Forward declaration of routines */
static void tp1();       /* test purpose one*/
static void startup(),cleanup(); /*TET startup and cleanup routines* /

/*Initialize test environment toolkit data
  structures */
```

```
void(*tet_startup)()  = startup();
void (*tet_cleanup)() = cleanup();
struct tet_testlist[] = {
    { tp1, 1 },
    { NULL, 0 }
};

static void startup()
{
  DtTestStartup();
  DtTestGetTestEnv();
}
static void cleanup()
{
  DtTestCleanup();
}

static void tp1()
{
  DtTestAssert(1, DT_ACCEPT | DT_REGRESS,
           “Assertion text here.”) ;
/*Code for validation of the assertion would go
  here. */
    DtTestResult(TET_PASS,
           “Assertion passed.”);
}
```

Scripts. Many of the test suites used the test case controller mechanism that allows the tests to define their own execution tool. This script, found in the scripts subdirectory, did such things as start the component with special options and set test-specific environment variables. Additionally, a test developer could define a separate script for test-specific setup and cleanup that was called from within the exectool.

The DtTest Library

Early in the CDE test development process, it was noted that certain combinations of TET or Synlib functions were commonly used as a unit. This led to the creation of the DtTest library which provided a more efficient mechanism for accessing these units in a uniform manner.

This library is a set of convenience functions developed to supplement the basic TET and Synlib functionality. These functions provided an easier way for test developers to perform common testing practices and support the CDE testing methodology. Use of the DtTest library functions also helped enforce internal consistency among tests developed at different sites.

The convenience functions that augmented TET functionality made it easier to print messages in the journal file, enter the test result, and control the operation and execution of tests. The DtTest library provided an initialization and a cleanup routine that TET did not provide, and a way to get the values of the CDE environment variables.

Functions to supplement Synlib were focused on two areas. One was to deal with text in a GUI text field or text widget. Functions were written to place text, retrieve text, or retrieve and compare text. The second area dealt with techniques for screen image capture, storage, and comparison. These functions took care of image naming conventions, storing images, and saving image analyses when image comparisons failed.

The CMVC Defect Report Mechanism.

The tests were programs, and like any set of programs some of them contained bugs. Often the defects were found not by the originators of the tests, but by other partners during test runs. After some unsuccessful attempts to keep track of defects and defect fixes via email and phone conversations, the test team started using the Code Management and Version Control, or CMVC tool from IBM. This method of reporting, tracking, and verifying defects and fixes was used for all CDE components.

Code Coverage Tools

Code coverage tools provide coverage statistics for a set of tests. They determined how much of the tested component’s code was actually exercised during a test run. Typically these tools produce coverage statistics in terms of the number of branches taken, the number of functions called, and so on. The CDE test team needed to get coverage statistics since these metrics were one of the measures of a test suite’s completeness.

There was no single code coverage tool that was generally available to all of the CDE partners, but each had a solution that worked on their platform. Analysis of each of the different tools showed that they produced results that were comparable. The test team agreed that all partners would use their own code coverage tools to provide test coverage statistics for their own component test suites. These metrics would be provided at certain developmental milestones to ensure that adequate progress was being made toward the coverage goals.

Test Design Objectives

Early in the project, the test team adopted a set of high-level design objectives. These objectives were based on the testing experience that was brought to the project from the different partners. Although team members from each company had considerable software testing experience, the philosophy and testing methods used by each company were often quite different.

The test team did achieve agreement on a number of basic objectives and goals, which resulted in the following high-level test design goals.

Formal Test Plan and Test Cases. Each CDE component was required to have a formal test plan document, written by the test engineers who had responsibility for that component. The test team provided a template document (see Fig. 5). The test plan was written before test development began. One of the sections of the test plan contained a list of test cases for the component. These test cases were based on the contents of the component's formal specification document. The rationale for this requirement was to get test developers to plan their components' test suite carefully before writing any tests.

Assertion-Based Testing. Tests had to be written using an assertion verification methodology. With this method a test begins with a statement of functionality about the component that can be answered true or false. For example, "Clicking mouse button one on the cancel button will dismiss the dialog box," or "Calling the OpenFile function to open file foo in read mode will return error code NoSuchFile when foo does not exist." Following the statement of assertion in the test comes the code necessary to perform the action implied in the assertion (e.g., click the cancel button, call the OpenFile function). Code to verify the results of the action would report a pass or fail to the journal file.

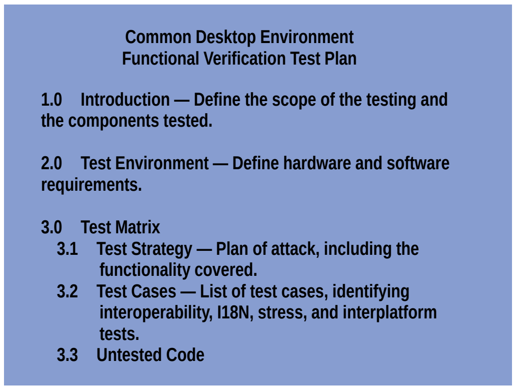
Testing Based on Component Specifications. The test team strongly encouraged that the design of the initial tests be based on a component's specifications. A component was to be treated as a black box until all the functionality described in the formal specification document was tested. This approach ensured that when a component was out of phase with the specifications, an error would be reported. Resolution of the error sometimes meant updating the specification, ensuring that changes would be captured in the documentation.

Testing Based on Code Coverage Goals. The test team realized that formal specification documents cannot cover every single detail, and that a test suite based only on the document contents would fall short of achieving the test coverage goals. Thus, the team decided that after the functionality covered in the specification document was incorporated into tests, the test suite's code coverage would be measured. If coverage fell short of the coverage goals, the code could be examined to determine how to design new tests that would exercise code branches or functions missed by the initial test set.

Test Suite Completeness Based on Coverage Statistics. A component test suite would be considered complete when it tested all the functionality described in the specifications and reached the minimum coverage goals set by the test team. The initial coverage goals for all components were that 85% of all branches would be hit at least once, and 100% of all internal functions would be called at least once. There was an additional requirement for API components that 100% of all external (developer visible) functions would be called at least once. Typically these statistics would be presented as a triplet of numbers (e.g., 85/100/100).

This was one of the goals that was revisited during the project. Although there was not total agreement in the test team, the majority felt that writing tests for GUI components was more difficult than writing tests for nongraphical

Fig. 5. Outline for a test plan.



Common Desktop Environment
Functional Verification Test Plan

1.0 Introduction — Define the scope of the testing and the components tested.

2.0 Test Environment — Define hardware and software requirements.

3.0 Test Matrix
- 3.1 Test Strategy — Plan of attack, including the functionality covered.
- 3.2 Test Cases — List of test cases, identifying interoperability, I18N, stress, and interplatform tests.
- 3.3 Untested Code

API components. As a result, the GUI coverage goal was lowered to 70% overall, with only 10% required in automated tests.

Automated Tests. The objective was to create an automated regression test suite. As the test plan overview document stated, "The expectation is that all tests are automated and that only where absolutely necessary will manual tests be acceptable." However, as noble as this goal was, we had to lower our expectations a bit. By the end of the project the test team was still requiring automated tests for the API and graphical API components, but for GUI components, the requirement was that the automated tests provide at least 40% branch flow coverage. The remaining 60% branch coverage could be obtained by manually executing tests based on either the list of test cases in the test plan or in a separate test checklist document.

Minimum Reliance on Screen Image Capture and Comparison. Experience with automated testing of desktop components showed that test suites that relied heavily on screen image capture and comparison were found to be unreliable—they would generate many false failures. The new Synlib test tool contained features that could be used to verify a correct desktop state without resorting to comparisons to previously saved "golden" images, and the test team aggressively investigated and promoted these techniques. It's estimated that the CDE test suite contains less than 10% of the "golden" images that would be required if a record-and-playback tool had been used.

Test Development Process

Once the CDE test team had straightened out the details of the testing framework and produced a set of guidelines for the test developers to follow, it was time to implement the solution. We set up test goals for each of the component development milestones. We jointly developed a training course designed to get test developers familiar with our process and test tools as soon as possible. Lastly, we set up a process for each of the members on our team to follow as we accepted the test suites for incorporation into the CDE sample implementation test suite.

Each CDE component had functionally complete, functionally stable, and code complete milestones. Each of these milestones had test coverage goals as shown in Table I. An API would be functionally stable when it had 70% coverage.

Table I
Test Coverage at Component Milestones

Milestone	API	GUI	External (Breadth)	Internal (Depth)
Functionally Complete	50%	5t*/10%	90%	70%
Functionally Stable	70%	40%	100%	90%
Code Complete	85%	70%	100%	100%

* Five tests defined or 10% coverage, whichever is greater.

When measuring the source code coverage of a test suite, besides the base number of overall coverage (lines of code hit divided by the total lines of code—the numbers in the GUI and API columns in Table I), there is also a need to look at the external or breadth coverage. For an API, this was the number of external functions called divided by the total number of external functions. For a GUI, this was the number of functional modules exercised divided by the total number of modules (i.e., Were all the buttons in the dialog box hit?). Internal or depth coverage is the number of internal (nonpublic) functions called divided by the total number of internal functions.

All components were expected to have a base level of automated acceptance tests, but GUIs could make up the rest of their test coverage through either automated tests or well-defined manual tests. For some components, such as the front panel which had to test the animation of the subpanels, creating some scripts that would be run by hand and manual verification of the results was the best way to go. APIs were expected to provide all automated tests.

As components reached one of their milestones, the test team representative from the company responsible for the component would check for more test development specific milestones. These milestones were put in place to ensure that the test team's final product was a robust, maintainable test suite.

The first item the test team checked for at each milestone was a complete test plan. It was important that the test plan thoroughly define the testing to be done since as a last resort, we would use the list of test cases in the test plan as a list of manual tests to complement the automated tests.

The second task the test team performed at the milestones was a review of the automated tests. We developed a criteria checklist that would check for the following:

- Acceptance tests. The priority for the functionally complete milestone was to have automated acceptance tests that could run in less than an hour.

- Assertions. Assertions were checked to be sure that they described the component's functionality and clearly stated what was being tested and how it was being verified. This was the hardest area for new test developers to learn and the hardest for us to judge.
- Use of the DtTest convenience functions. These functions were developed to ensure consistent journaling of tests, error handling, and naming conventions of image and data files.
- Use of copyright notices and standard header files. These tests were done manually for the first five or ten tests for a particular component. The test developers were expected to use their "blessed" test suites as a template for the rest of their tests.
- Test suites must run autonomously. TET allows for a very fine granularity of execution, down to individual test purposes within one test case. A test purpose is a function made up of an assertion, code validating the assertion, and code for reporting the result. Test purposes could be grouped together within an invocable component, ensuring that they always ran together, but beyond that, these invocable components always had to be able to run on their own.
- Test-specific libraries and clients. These were reviewed to be sure that the library calls were documented in a header file and test clients in a README file.
- Portability. Tests were reviewed for nonportable practices such as hardcoded path names and x,y locations. The total number of stored images was also expected to stay under 15 per test suite.
- Test Execution. Tests had to run using the test case controller on every platform.

As the project wore on, this checklist stretched out to 77 items and became affectionately known by the component engineers as the "77 points of pain." Our last milestone item was to check that memory integrity was being checked.

About halfway through the project, test development efforts really got into full swing at all the companies. We all used temporary test engineers from time to time, and it was necessary for the test team to get these new engineers familiar with our methodologies as soon as possible. We jointly developed a two-to-three-day training course that new test engineers took before getting started. This covered training for the tools, how to write a good assertion, and creating Imakefiles. By the third day, a test engineer would have completed at least one test and be familiar enough with the test tree structure to get around without help. We used some test suites that were good examples of the kind of tests we were looking for, and we had an example test suite as a guide for engineers to use for doing more complex functional verification. Finally, we developed a "how to" document that architecturally described the test tree design and defined all of the tools and interfaces available for use. Despite our best efforts, it still took about two to four weeks for a new test engineer to develop the ability to do a couple of test cases per day.

Test Execution Cycles

Throughout the development of CDE there were several places where we stopped and executed a common test execution cycle in which all CDE partners participated. These test cycles were driven by events such as conferences and trade shows, where the desktop was displayed, and milestone completion dates. We developed a test report form so that the results could be compiled in a consistent fashion and results reported at each company. Journal files from the test runs were checked into the tree so that it was easy to check for both component and test regressions. After each test cycle, we would do a postmortem to improve the process for the next test cycle.

Our first goal in the test execution arena was to define an execution cycle that would enable each company to execute the tests in a uniform, automated fashion. The phases of this cycle are listed below. Phases four through eight were repeated for each test suite.

Phase I Machine Setup. This included setting up the reference platforms at each company, taking into account any hardware or software requirements documented in the test plans.

Phase II Test Environment Build and Installation. This was easy for us since our test trees were set up to build on a nightly basis in the same fashion as the source trees. The more difficult part was the installation. Since the test suites would certainly run overnight, the build would interrupt the test execution, causing indeterminate results. The short-term solution was to turn off test builds during test cycles. For the long term, we wanted to create an installation process as automated as the build process.

Phase III General Test Environment Configuration. This phase included defining configuration data and executing any setup programs, including:

- Putting the general environment configuration defined in the Dt Config files into the test execution environment
- Setting up user permissions and scripts that must be run as root.

Phase IV Component Specific Test Environment Configuration. Analogous to the previous phase, during this phase the component's test environment was set up, including:

- Putting the component-specific configuration file into the test execution environment

- Starting the session using the dtsession setup script
- Running build_scen to create a current scenario file.

Phase V Run Test Cases. The tests were executed using the test case controller, specifying an output directory for saving results and a journal file.

Phase VI Test Results Evaluation. The journal files were run through the TET journal filter script to find any test failures.

Phase VII Component-Specific Shutdown. Between each test suite, the test environment was cleaned up using the same script as for setup. The session was stopped via the cleanup dtsession script to keep the previous tests run's exit state from affecting the next test.

Each company had to check their test results into the shared test source tree at the end of the test cycle. They had to state for each component the type of testing done (automated or manual), the current branch flow numbers, the number of test cases run (number of test assertions), and the pass/fail status for the total run. A postmortem was done after the test cycle, looking in particular for test suites that had different outcomes at the different companies. Defects were filed against the test suites and the components, and if a particular test cycle was targeting a release, the defects were fixed and the tests were rerun.

System testing was done more sporadically. We developed a fairly extensive system test plan covering areas of interoperability, I18N (internationalization), interplatform, and stress testing. Unfortunately, these tests were never automated, in part because of a shortage of resources. These tests were more complex, and therefore more difficult to automate than the functional tests. We did make sure that both interoperability and I18N functionality were tested with each test cycle. We usually relied on the CDE test team members to manually run through the system test plan for their company's platform. For interoperability, a matrix was developed showing the types of interclient communications that were allowed for each component. The I18N section described pieces of the component that used input methods for EUC (Extended UNIX® Code) 4-byte characters as well as sections that were expected to be localized. Our reference languages were Japanese and German, so the manual checklist was run against these two languages.

Conclusion

By the time the CDE sample was done, some of the CDE test suite was not complete. Some of the components had no automated tests and some test suites were in various states of completion. However, the existing test suites and the test framework provided a good basis for a maintainable test suite for the HP CDE product. In fact, the framework and methodologies have been expanded to encompass other HP products with a great deal of success. Work continues to be done in this area, adding further expansions such as internationalized tests and other system testing.

▶ Go to Article 8

▶ Go to Table of Contents

▶ Go to HP Journal Home Page