8.1 Developing Automated Test Scripts

Automated test scripts can be either recorded or programmed using a test tools test-scripting language, or if an automated test tool is not available, they can be recorded using the Windows macro recorder function. Regardless of which method you choose, following a few simple test case design rules will result in more robust and easier-to-maintain test cases. Adhikari [1] cites four test case design principles that are in use at Charles Schwab:

1. Design independent test cases
2. Design self-contained tests
3. Design home-based test cases
4. Design nonoverlapping test cases

Test case independence assures that one test case does not depend on the successful completion of another test case in order to run. It also assures that automated test suites will run to conclusion when left unattended.

The Scientific Method was founded on testing. As scientists conduct experiments, they test only one condition for each repetition. This ensures that the results of each experimental trial are a direct function of the condition that was varied. If two or more variables are changed simultaneously, it is very difficult to pinpoint the one that is responsible for the observed outcome of the experiment. Of course, in science, as in applied endeavors, it is sometimes not possible to create ideal experimental conditions.

In testing software, if two test cases are not independent, two things may happen. First, the subsequent test case may fail to execute, and second, isolating the cause of any failures becomes extremely difficult. It is possible to
design independent test cases, but constructing independent test cases is sometimes very difficult.

The purpose of software testing is to identify new errors while the purpose of debugging is to locate and remove known errors [25,27]. “Fat” test cases (those that cover many test conditions) are used to identify new errors. “Lean” test cases (those that cover only a single test condition) are used to locate and remove known errors. So, test cases that are not independent are better and more economical for finding errors because fewer are required, but additional independent test cases are frequently required to find and remove the errors.

It becomes a trade-off. If your purpose is to have automated test cases that are executed in batch mode via shell procedures, you want independent test cases. The major drawback is that more test cases may be required to find the same number of errors than fewer and fatter manual test cases would discover. Another drawback is that more test cases require more maintenance of the test suite. In short, the more test cases, the more testing costs.

Self-contained tests use test cases that have testing requirements implemented in baseline databases. Maggio [21] describes the need for base states in test scripts. Base states eliminate linear dependence among test cases. He argues that because an initial precondition surrounds the verification process within each test script and ends with a postcondition, it is isolated from other test scripts. Furthermore, it will pass or fail regardless of previous or future application conditions. Setting up base states requires using home-based test cases.

All home-based test cases start from the same point in the application. According to Maggio, the application must be in a specific state such as its initial state when it is first executed (the precondition). This means that the application should be open, with no child windows or dialog boxes open, and all menus should be enabled. The corollary to this is that when the test cases finish they must return the application to home base (the postcondition). When I use SQA TeamTest, I start from the standard desktop and use SQA to open and close the AUT.

Constructing home-based test cases is very important because there is less chance of test case execution failures when each test case begins from a known point and cleans up after itself. In an automated suite of test scripts, this assures that each script begins under the same set of conditions as its predecessors, helping assure that the test scripts are independent.

No gap and no overlap means that test cases should cover all aspects of the system functions and that test case redundancy should be eliminated. Testers sometimes have a tendency to test everything; they will even test for conditions that can never occur. Comprehensive testing is not possible [26,28]. So, the tester is charged with testing as much as is possible, given the available time and resources.

Testing economics dictate the use of fat test cases to identify new software errors. These test cases are not just randomly selected. They are
designed and constructed according to guidelines that ensure adequate testing coverage with a limited number of test cases. Techniques such as those discussed below (cause-effect graphing, equivalence partitioning, boundary analysis, error guessing, decision logic tables, and basis testing), when used in combination, result in a test database that covers all system functions and minimizes test case redundancy.

Automated test cases should test:

1. That the application does what it is intended to do (known as either constructive or positive testing)

2. That the application does not do anything that it is not supposed to do (this is known as either destructive or negative testing)

3. That the application is robust (i.e., can handle spurious data without crashing); when possible, specific positive and negative results should be identified as expected outputs of specific test scripts

Fuchs [13], who prefers writing automated test scripts as opposed to recording them, offers a three-step approach for creating automated test scripts. The first step is to design the test case. He says each test script should contain from one to ten closely related scenarios. Test scripts with more than ten scenarios should be separated into multiple test scripts. Each scenario should be associated with a unique expected result.

The second step is to run the tests manually. He argues that automated tests are best for finding errors during regression tests and that manual testing is better for the first test cycle. The first set of tests should be the most productive at finding new errors whereas the regression tests should find more errors related to the evolution of the software. Regression tests do, however, find errors that were missed originally. I have seen regression testing identify errors that were in software systems since their inception (in one case, this amounted to more than 20 years of use with the error present in the system until regression testing led to its discovery).

The third step is to automate the test cases. For each test scenario, the test script should contain sections that

Perform setup
Perform the test
Verify the result
Log the results
Handle unpredictable situations
Decide to stop or continue the test case
Perform cleanup

The setup activities the test script must perform include defining common variables, constants, functions, and procedures the test will use; starting the application under test; creating any required directories and data files; and updating reference tables.
The testing activities should simulate how a user would use the application under test. An important point to make here is that scenarios must execute in the order in which they are written because this represents how the user will perform the work. The functional requirements document can be a place to start when identifying typical business scenarios. A better way, however, is to observe someone actually using the system or doing the tasks manually, but in many cases this will not be possible until the system is already under test.

Verifying results involves checking the initial and final states for the controls involved in each user action and checking the database for expected and unexpected changes. The only way to verify application functionality with respect to business rules and business rule interactions is to confirm that changes have actually occurred to associated data values.

Logging results means keeping a record of the pass/fail status of each test case. The results can be viewed after the tests are finished, and test logs from different test executions can be compared.

Handling unpredictable situations involves trapping unexpected events and recovering from them. Unexpected events can include unexpected keystrokes, unexpected windows, expected windows that are not present, and system-level interrupts and automated test script runtime errors.

Deciding to stop or continue should happen at the end of each scenario, and depends on the pass/fail status of the scenario just executed. In some instances, the subsequent scenario can still run even if the preceding one failed. In other situations, a later scenario would provide invalid test results if an earlier one or more have failed.

Performing the cleanup involves closing the application under test, deleting any no-longer-needed directories and data files, resetting reference data and usually returning the test case to home base.

Fuchs also suggests organizing the test scripts into test suites that are executed from shell scripts. The MS Visual Test on-line documentation suggests designing a functional area test suite, a regression test suite, a benchmark test suite, a stress test suite, and an acceptance test suite.

One functional area test suite should be developed for each functional area you can identify in the application under test.

The regression suite should test previously tested system features to determine if they continue to function correctly as corrections, tuning, and enhancements are added to the system during its maintenance phase.

The benchmark suite tests system performance under varying conditions such as different hardware and software platforms, and contrasting system loads.

The stress suite tests the system under such extreme conditions as heavy system loads during peak usage times.

The acceptance suite tests the software to determine if it meets the minimum standards for user acceptance of the system. The purpose is to determine if the system is ready to withstand the rigors of beta testing and/or production use.
8.2 RECORDING VS. PROGRAMMING TEST SCRIPTS

Recorded test scripts have their limitations and frequently must be edited before they work properly. Additionally, many test conditions represent functional variations of one test case. Therefore, many variations of the same test script must be constructed, and one way to do this is to record the first script and edit it to produce the variant scripts or to write a test script template and modify it for each variation. In either method, to do any meaningful testing you must become proficient in the test scripting language.

Many testers prefer to write test scripts rather than record them [13]. In fact, Segue’s QA Partner tool did not have test script recording capabilities until release 2.01. Don Felgar, a frontline tester, who has experience with several automated testing tools, says in a USENET comp.software.testing article:

I don’t see any good reason to record tests. If you can record it then it works, right? What are we supposed to do, guess what will be broken in the next release. Hummm, I have a feeling that the developers will gray out this menu item erroneously during the next release. I better record a test.... I believe that automated software testing is complex enough that it must at some level involve coding, so folks may as well admit that and bite the bullet.

Fuchs [13] suggests creating a test case script template. The template file could contain common header information, setup routines, and include files. The tester would use the template as a starting point to write test cases, keeping the portions of the template that are relevant and deleting the portions not required for a particular testing circumstance.

Of course, not everyone is a programmer and many testers are individuals who have business domain experience but not technical expertise. Often, people who know the business make much better testers than people who have technical expertise because they have a different perspective, an extended perspective and more knowledge of the business processes. Software developers have an internal view of the system under development while users have an external view.

Thus, a typical scenario for automated testing tool use should include all of the above: developing some test scripts via the recording facility, creating functional variations of template test scripts through the tool’s editor, and, possibly, writing test scripts from scratch via the tool’s script editor.

The crux of this situation is that test scripts that have been recorded using the test tool’s recording facility are somewhat appropriate for testing the GUI but not for testing an application’s functionality.

If you intend to test business objects, database objects, and their associated rules with your test tool, then simply recording test scripts and editing them will not suffice. In client-server systems rule-based validations can occur at many levels, but they commonly involve GUI-level validations and business rule-level validations. The business-rule validations occur at the level of the server through application logic or through stored procedures and triggers in
the database. The GUI validation occurs in the interface code that executes on the desktop. To test at either of these levels you must develop test data that exercises the application’s ability to enforce the validation rules.

The problem is that if you were to record all of the specific data input scenarios (functional variations of the recorded test scripts) that might be required to test the application’s functionality, you would be sitting at the computer and inputting data for years. The solution is to develop test data files and create specialized test scripts to read in the data they contain. This type of test script contains some recorded portions, but most of it has to be written by a tester who is competent in the tool’s test scripting language.

This approach has been generically dubbed “data-driven” [18] and is extremely effective for testing 2- and 3-tier client-server applications. The data drives what is tested and how it is tested. This is the only manner in which you can effectively test application functionality with an automated test tool.

A modular test script design technique known as framework-based testing [18] has evolved as a complement to data-driven testing. Framework-based testing is really structured scripting [20] for test script design and construction as we once had structured programming for program design and construction [5, 19, 25, 34, 35]. Structured scripting follows different rules from those of structured programming, however, and is not nearly as rigorous.

An excellent review of the evolution of software testing automation was published in the February 1999 issue of Software Development [20]. In it, Kit discusses the path from the first uses of capture/playback tools to the third generation and on to future generations of automated software testing.

The most important point that Kit discusses is that naive test tool users believe the hype that recording and playing back user interactions with the application under test is sufficient to test the application. This is a continuing problem. Based on the e-mails I receive and the questions and comments that are posted to tool user news groups, I estimate that 70% of the current crop of automated test tool users are inexperienced.

Individuals who attend the data-driven scripting seminar that Bruce Posey, the Archer Group, and I give develop a complete suite of recorded test scripts during the seminar. Many then abandon the recorded scripts when they return to work, and they develop a completely new set of data-driven scripts that are modular and function-driven.

Kit very clearly describes the key lesson we have learned. Treat test case design as separate from test script design. This idea must be extended further to separate test data construction from the test script construction. Do not embed test data in the test script.

### 8.3 Managing Manual and Automated Test Scripts

Because test scripts are fairly complicated and many may be required during the test, it is best to organize them in *test case folios*. The test cases in the folio
can be grouped according to the design objectives or requirements of the system they test. Based on Hetzel’s [15] earlier work, scripted test case folios should include at least five objectives:

1. Detail the objects to be tested
2. Describe restrictions and limitations on script use
3. Give an overall description of the test scenario
4. Organize groups of scripts according to purpose
5. Describe the expected behaviors for each script

The test case folios can be constructed and maintained with many different desktop tools. At a minimum a word processor and/or a spreadsheet is required. Of course, it is much better if the test case folio can be created with the testing management components of automated software testing tools. For example, SQA’s Manager component does not embed items such as the test plan or test case folio, but it allows the user to set internal document locators to these items.

In addition, GUI testing tools should include test management components that can organize test scripts and link them to test requirements. Two products that already have these capabilities are Rational’s TeamTest (SQA Manager component) and Mercury Interactive’s WinRunner (TestDirector component).

8.4 Test Suite Maintenance

A suite of automated tests is itself a software system, containing the same problems that the system it is designed to test contains. It is prone to errors and extremely sensitive to changes. So, when you test a C-S system with automated test scripts you are dealing with two systems that have to be maintained. This doubles the maintenance problem.

Steve Fuchs [13], program manager for Microsoft’s MS Test product (a component in Rational Software’s arsenal of software development and testing products), believes the macro recording capabilities of the major test tools cause several maintenance problems because:

1. The software product will change during its life cycle
2. If it is successful it will be reproduced in other languages
3. The next version of the product will have a better user interface
4. There will be less time to test subsequent versions of the product

He argues that because the product will change scripts will contain invalid events. The effort needed to isolate and rerecord those portions of the script will be substantial. He says that the scripts will contain very little context information about the events, which makes maintenance harder. The scripts
will also contain hard-coded function calls, which will require extensive updating for even minor user-interface changes. He concludes that even simple changes can affect 50–90% of the test scripts.

Fuchs’s arguments are valid because new system releases do impact automated test suites. Recorded test procedures will most likely require some updating when new versions of the system include changes to any of the system layers (interface layer, data layer, or function layer). Consider that when we change a software module, we must determine which other modules are affected by this change. Now when we change a software module we must also determine which test scripts will be affected by the change.

Automated test suites add a new constraint on the system itself. James Bach of Software Testing Laboratories (in the USENET news group comp.software.testing article) says, “automation systems constructed from scripts are very complex and hard to maintain. The more complex a system is, the more likely it is to fail.” The implication here is that the test scripts themselves can be error prone. So do we write test scripts to test the test scripts?

Another concern is the impact of software and/or hardware platform changes: As Bach says, “Any change in such a platform will cause widespread failure of test automation, unless the change was specifically anticipated.” Bach has experienced problems with test automation stemming from DOS updates and the change to a Novell network. He was forced to add platform-related processes to the test automation. He found that only subtle configuration differences could cause automated test suites to fail.

To be fair to the automated test tool vendors, I must add that the test suites are much more robust than they used to be. Many of the conditions that caused earlier test suites to fail are now trapped and logged, and the testing process continues.

Test suite maintenance begins far in advance of the actual testing. It is always advisable to start designing and constructing test scripts and test data as soon as you have enough information to begin. The problem that you will encounter is that the test scripts and test data must be revised each time the developers change the software. If you have little or no control over how developers insert changes during the analysis and design processes, it will drive you crazy trying to keep the automated test scripts and test data up to date.

A compromise solution is to begin designing, but not constructing, test scripts and test cases, as soon as possible. This eliminates 50% of the maintenance burden; you only have to update the design and not rerecord or reprogram the test scripts. At some point, as the actual test date nears, you have to begin construction, but the longer you can put it off, the less pretest maintenance you will have to do. This approach can be enhanced by also constructing paper-and-pencil test scripts that can be easily modified.

Ultimately, data-driven testing can reduce the level of test script and test data maintenance by 70–80% and is a better approach than all of the others.
8.5 DATA-DRIVEN TESTING

Data-driven testing uses archived test data, usually in the form of simple comma separated values (CSV) text files, to drive the automated testing process. It can be expanded to include control data, as well as test data. Both GUI- and server-level data validation rules, representing an application’s functionality, are tested. The control data drives the test script by directing it to the appropriate location in the application to execute the test and by indicating what type of test to execute.

Data-driven test scripts use simple text files, are highly maintainable, can be used by nonprogrammers, document what tests are being executed, allow dynamic data input via “placeholders.” The input data controls test execution.

The content of data-driven tests can include [18,26]:

1. Control parameters that you can input to the program
2. Sequences of operations or commands that you use to make the program execute
3. Sequences of test data that you drive through the program
4. Placeholders that cause the test script to create a dynamic data value at runtime
5. Documents that you have the program read and process

The test data should be designed using the techniques described in the sections, “Black Box” and “White Box” approaches. The test conditions and actual test data values are entered into spreadsheets that are used to develop the CSV files the scripts use.

Kit [20] has proposed designing test cases at a higher level of abstraction. He suggests that a test case preprocessor be developed that interprets the test case design in the spreadsheet and creates the data executed by the scripts. To do this, he suggests the need for a high-level test case design language. He has developed an “interpreter script,” which contains functions that read and interpret commands contained in the spreadsheet. The commands include instructions that control the test scripts’ navigation of the application under test, as well as commands that direct the input of data into GUI fields.

What Kit is suggesting we have already accomplished but in a slightly different manner. By embedding control codes in the data the script reads, we direct and control the test script’s behavior with respect to the application under test (AUT). Furthermore, when the application changes, we only have to change a control code in the test data. We do not have to maintain the code in an interpreter script. Our approach is more economical in terms of test script maintenance. However, our approach is somewhat more laborious in that we export the spreadsheet data to the CSV files before the test script is executed (we use SQA TeamTest Suite). However, Andrew Tinkham, who regularly posts to the SQA users’ group (http://www.dundee.net/sqa.htm), has
developed a set of SQA TeamTest script functions that read the MS Excel spreadsheets and directly pull the data into the AUT. This users’ group is an excellent resource for tips and tricks to use when developing automated tests. Even if you use a product other than SQA TeamTest, you will find useful information in this forum.

The advantages of data-driven test scripts are

1. It is not necessary to modify the test script when the test data require changes.
2. It is not necessary to modify the test script when it is necessary to add additional test data as the test data is appended to the existing text file.
3. It is not necessary to modify the test script when the application under test is modified because you merely reset the control data values by changing or adding new ones.
4. It is easy to modify the data records with a text editor.
5. Multiple input data files can be created and used when required.

8.6 STRUCTURED TEST SCRIPTING

The major advantage of structured test scripting is that it isolates the application under test from the test scripts. It also provides a set of functions in a shared function library. The functions are treated as if they were basic commands of the test tool’s programming language. Structured test scripts can be programmed independent of the user interface.

Structured scripting can occur at multiple levels [18]: the menu/command level, executing simple commands; the object level, performing actions on specific things; and the task level, taking care of specific, commonly repeated tasks.

8.6.1 Developing Framework-Driven Structured Test Scripts

The following hints [26] will help you develop structured test scripts. Write

1. Functions for all features of the application under test
2. Functions for custom controls
3. Functions around language-specific commands
4. Functions for tasks that are used repeatedly
5. Functions for large complex tasks that are used across test scripts

8.7 WRITING EFFECTIVE TEST SCRIPTS AND TEST DATA

The following list of do’s for writing effective automated test scripts is taken from the workbook [26] of the advanced SQA TeamTest script-writing seminar offered by CSST Technologies and the Archer group.
DO
Use structured test script design
Implement data-driven controls
Develop and use script-writing guidelines
Limit script sizes
Break scripts down by functionality
Document scripts well
Organize test scripts into related groups
Use shell scripts
Include test parameters in data files, such as .ini files, settings files, and configuration files, rather than as constants embedded into the test script
Prompt users for input specifics with preset defaults
Create error traps and provide the user with feedback

The following are do’s and don’ts for creating effective test data.

DO
Use the test data design techniques discussed in the section “Functional Test Case Design”
Place the data in simple text files
Document what tests are being executed
Allow dynamic data input via “placeholders”
Use input data to control test execution

DON’T
Use capture/playback as the principal means of creating test scripts
Use test scripts that individuals code on their own, without following common standards and without building shared libraries
Use poorly designed scripting frameworks

Here are some additional tips and tricks taken from our advanced scripting class.

1. Construct separate test scripts for adding, deleting, and updating data records and for verifying the edits.
2. Create a single test script for general menu properties, system menu properties, keyboard shortcuts, and tool bars.
3. Create an additional script for object properties tests for all major GUI screens.
4. Develop and use test script templates for adding, editing, and deleting test data records.

5. Avoid hard coding items such as data paths, file names, and constants. Instead, use global include files as header files (for example, in SQA use .sbh—SQABasic Header) for constants and definitions.

6. Use Source (for example, in SQA use .sbl—SQABasic library) for executable code such as functions.

Avoid letting the main script become too complex. Break complex testing activities/tasks into small pieces. Use subroutines, functions, and additional procedures when necessary. Convert subroutines that require input variables to functions. Functions are better than subroutines because they return a pass/fail code that indicates whether the procedure executed correctly. Back up scripts before making major modifications. Use a configuration management process/tool for test script version control.

8.7.1 Improving Test Script Maintainability
Use plenty of comments. Insert a comment heading at the beginning of the script for complex test procedures. Use a product such as Cyrano for documenting test suites.

8.8 Functional Test Data Design
There are three basic approaches to design: requirements based, code based, and a hybrid of the two.

8.8.1 Black Box (Requirements-Based) Approaches
8.8.1.1 Cause-Effect Graphing This approach involves identifying specific causes and effects that are outlined in the requirements document. Causes are conditions that exist in the system and account for specific system behaviors known as effects. Effects can be either states that exist temporarily during the processing or system outputs that are the result of the processing. The causes and effects are entered into a cause-effect diagram that can be used to create test cases.

8.8.1.2 Equivalence Partitioning This approach uses the system requirements to identify different types of system inputs. Each input type is defined as an equivalence class and rules are devised to govern each class. The rules become a basis for creating test cases.

8.8.1.3 Boundary Analysis This approach strives to identify boundary conditions for each equivalence class. The conditions are used to create test cases
8.9 REQUIREMENTS-BASED APPROACHES

8.9.1 Requirements-Driven Cause-Effect Testing

Elmendorf [8,9] describes the cause-effect graphing method as “disciplined specification-based testing.” Based on Elmendorf’s work, Myers [27] defines a cause-effect graph as “a formal language into which a natural-language specification is translated.” The graph is a “combinatorial logic network” using notation similar to, but simpler than, standard electronics notation. More pre-
cisely, it is a Boolean graph describing the semantic content of a written functional specification as logical relationships between causes (inputs) and effects (outputs).

As a Black Box technique, cause-effect graphing can be used early in the development process in conjunction with review procedures such as desk checking and walkthroughs. It is a versatile approach because the test cases generated can be used during all subsequent levels of testing.

**8.9.1.1 The Cause-Effect Graph** Cause-effect graphs are models of complex narrative software descriptions as digital logic circuits that can easily be used to develop functional test cases [8]. Each circuit is a pictorial representation of the semantics portrayed in the written specifications. The semantic information in the cause-effect graphs is translated into Limited-Entry Decision Tables (LEDT) that are used to construct the actual test cases. An LEDT is a binary truth table in which each rule represents a logical path through a program segment.

The only requirement for using and understanding cause-effect graphs is knowledge of Boolean logical operators: AND, OR, and NOT are the most commonly encountered operators, but NAND and NOR may be required in some instances. Tables 8.1, 8.2, and 8.3 are truth tables for the AND, OR, and NOT operators, respectively. Tables 8.4 and 8.5 represent the NAND and NOR operators, respectively.

**Table 8.1 Truth Table for Logical AND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Note: The ANDed variables are “A” and “B.”*

**Table 8.2 Truth Table for Logical OR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Note: The ORed variables are “A” and “B.”*

**Table 8.3 Truth Table for Logical NOT**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*Note: The NOTed variables are “A” and “B.”*
The basic cause-effect graph notation is illustrated in Figure 8.1. There are four fundamental configurations and two infrequently used negative forms.

- **Identity** defines a situation in which node Y is true if node X is true. In Boolean terms, if X = 1, Y = 1, else Y = 0.
- **AND** defines a circumstance where X and Y must be true for Z to be true. Again, in Boolean logic, Z = 1 only if X = 1 and Y = 1, else Z = 0.
- **OR** defines a condition in which either X or Y must be true if Z is to be true. In Boolean format, Z = 1 if X = 1 or Y = 1, else Z = 0.
- **NOT** defines the instance where Y is true only if X is false. In Boolean logic, Y = 1, if X = 0, else Y = 1.
- **NAND** defines the situation where, if both X and Y are false, Z is true. In Boolean, Z = 1, if X = 0 and Y = 0, else Z = 0.
- **NOR** defines the condition where, if neither X nor Y is true, Z is true. In Boolean notation, if neither X = 1 nor Y = 1, Z = 1, else Z = 0.

Here is a word of caution about the use of negative logic: It can lead to unnecessarily complex logical combinations and should be purposely avoided when possible. If the situation is a NAND or NOR, try to restate the logic in the positive before developing the cause-effect graph.

From a cause-effect perspective, some combinations of causes may be impossible because of semantic or syntactic constraints. In addition, certain effects may mask other effects and when this occurs it must be indicated on the graph. Consequently, the notation for constraints (shown in Figure 8.2) must be used in conjunction with the basic cause-effect notation.
Fig. 8.1 Cause-Effect Graphing—Basic Notation
8.9 Requirements-Based Approaches

Causes

EXCLUSIVE

ONE AND ONLY ONE

INCLUSIVE

REQUIRES

Effects

MASKS

Fig. 8.2 Cause-Effect Graphing—Constraint Notation
Constraints on Causes

- **Exclusive** constraints define the situation where cause X and cause Y cannot simultaneously be true. If $X = 1$, $Y = 0$; if $Y = 1$, $X = 0$, however, both causes X and Y can simultaneously be equal to 0.

- **Inclusive** constraints define the situation in which either X or Y must always be true. If $X = 0$, $Y = 1$; if $Y = 0$, $X = 1$. Causes X and Y may simultaneously be equal to 1, but the state where $X = 0$ and $Y = 0$ is not a possibility.

- **Requires** constraint defines the circumstance where Y must be true if X is to be true. If $X = 1$, $Y = 1$. The states where $X = 0$ and $Y = 0$ simultaneously and where $Y = 1$ but $X = 0$ are also possible.

- **Only** defines the instance where one and only one of X and Y must be true. If $X = 1$, $Y = 0$; if $Y = 1$, $X = 0$. Causes X and Y cannot both be simultaneously equal to 1 or simultaneously equal to 0.

The Inclusive, Only, and Requires constraints are used with the logical and operator. The Exclusive constraint is used with the logical or operator.

Constraints on Effects

- **Masks** define the circumstance where, if effect V is true, effect Z is forced to be false. If $V = 1$, $Z = 0$.

8.9.1.2 Developing Test Cases The following procedure for deriving test cases using the cause-effect method is adapted from Myers's work [27].

1. Divide the specification into “workable” pieces. Do not attempt to create a single graph for the entire specification. Large specifications are too complex and must be taken in smaller (less complex) segments that are more understandable.

2. Identify the causes and effects in each specification segment. A cause is a unique input condition or class of input conditions (an equivalence class). An effect is an output condition or a system transformation (an alteration of the system database).

3. Translate the semantic relationships in each segment into Boolean relationships linking the causes and effects in a cause-effect graph.

4. Annotate each graph with the constraints affecting the causes and effects.

5. Trace the binary condition states (which can be perceived as true-false or 1 – 0 at each node in the graph) and identify each unique combination of binary states that link a cause to an effect.

6. Draw an LEDT summarizing all of the possible condition-state (rule) combinations (see Figure 8.3).
7. List the causes in the Condition Stub of the table and the effects in the Action Stub. Describe each combination of condition states (causes) in the condition-entry quadrant of the table. Divide the entry side of the table into Rules, one for each unique combination of condition states in the cause-effect graph. Finally, indicate which state combinations are associated with specific effects by placing an X in the column that represents the condition-state combination (rule) next to the invoked effect.

8. Convert each column (rule) in the decision table into a test case.

Identifying the unique combinations of condition states in the cause-effect graph (see step 7 above) is a reasonably difficult task. Furthermore, there is no check for completeness as in the DLT procedure described below. However, tracing backward from an effect through each of the possible combinations of intermediate input values using the primary input values will yield a set of distinct condition states. As before, the following procedure is adapted from Myers [27].

1. Work with a single effect (output) at a time.
2. Set that effect to the true (1) state.
3. Work backward through the graph and identify all the combinations of causes (inputs) that will force this effect to the true state. The number of possible combinations may be reduced because of the constraints on the causes.
4. As stated in the guidelines above, create a column in the decision table for each combination.
5. Determine the true or false (1 or 0) states for all other effects for each combination.

8.9.2 Equivalence Partitioning, Boundary Analysis, and Error Guessing

Equivalence partitioning and boundary analysis are complementary Black Box test case design strategies that are very useful early in the development life cycle. They are techniques that translate written specifications into function-based test data. Restrictions upon input that are described in the functional specification document are used to define classes of input and output. Equivalence partitioning describes categories of input only, while boundary analysis can define both classes of input and classes of output.

Both techniques result in two basic kinds of input classes: valid and invalid input classes. A single class will describe the valid input and one or more classes will describe invalid types of input. Test cases representative of each class are created and added to the test data set.

Myers [27] has established a set of guidelines for identifying equivalence classes and a set of rules for constructing test cases that cover each class. An additional set of rules governs the creation of test cases for the boundaries of the equivalence classes.

8.9.2.1 Defining Equivalence Classes

Defining equivalence classes is to some extent a trial-and-error process. It is based largely upon intuition and experience and it may be something you have been or are currently doing as part of your testing activities. There are, however, general guidelines that can expedite the process. As adapted from Myers, they are:

1. For input descriptions that specify a range of possible values (continuous input), identify one valid equivalence class which is representative of the values included in the range, and identify two invalid equivalence classes: one for values that lie above and one for those below the range.
2. For input descriptions that define a set of values, each of which is processed differently (discrete input), identify one valid equivalence class for each value and one additional equivalence class that represents a value not included in the set.
3. For data typed input (e.g., the data types numeric and alphabetic), create one valid equivalence class representing the correct data type and at least one equivalence class representing a data type that would be considered incorrect.
4. For mixed data types (e.g., the data type alphanumeric), with specific mandatory conditions (e.g., as in part-number where the first position in part-number must always be an alphabetic character), identify one
equivalence class in which the conditions are met and one equivalence class in which the conditions are not met.

5. Review the equivalence classes looking for instances where the classes may be further subdivided. The classes are subdivided only if values discovered in a class are not all processed in the same manner.

Myers says it is helpful to create a table with three columns—a left column where each external input restriction is separately listed, a center column where the valid equivalence classes are described, and a right column where the invalid equivalence classes are placed. Arbitrarily number all equivalence classes in the table. Organizing and numbering the equivalence classes in this manner simplifies the test case construction. Figure 8.4 illustrates the tabled information from equivalence partitioning and boundary analysis as it was applied to a control table used by a C language program from an application I helped to test.

<table>
<thead>
<tr>
<th>Input Conditions</th>
<th>Valid Equivalence Classes</th>
<th>Invalid Equivalence Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect_Type</td>
<td>Character “P,” “I,” “F,” “D”</td>
<td>Not “P,” “I,” “F,” or “D”</td>
</tr>
<tr>
<td>Start_Time</td>
<td>HHMM Value 24 Hour Clock</td>
<td>Empty</td>
</tr>
<tr>
<td>End_Time</td>
<td>HHMM Value 24 Hour Clock</td>
<td>Empty</td>
</tr>
<tr>
<td>Pause_All</td>
<td>“N,” “Y,” or Space</td>
<td>Not “N” or “Y,” or Space</td>
</tr>
<tr>
<td>Stop_All</td>
<td>“N,” “Y,” or Space</td>
<td>Not “N,” “Y,” or Space</td>
</tr>
<tr>
<td>Poll_Increment</td>
<td>Integer Value in Seconds</td>
<td>Non-Integer Value</td>
</tr>
<tr>
<td>Redial_Increment</td>
<td>Integer Value Set to 900 Seconds</td>
<td>Value Less than 900 Seconds</td>
</tr>
<tr>
<td>Site_Loop_Time</td>
<td>Integer Value Set to 60 Seconds</td>
<td>Value Greater than 60 Seconds</td>
</tr>
<tr>
<td>Connection_Loop_Time</td>
<td>Integer Value Set to 60 Seconds</td>
<td>Value Greater than 60 Seconds</td>
</tr>
<tr>
<td>Connection_Wait_Time</td>
<td>Integer Value Set to 600 Seconds</td>
<td>Value Greater than 600 Seconds</td>
</tr>
</tbody>
</table>

Fig. 8.4 An Example Black Box Test Case Design Diagram
8.9.2.2 Constructing Test Cases from the Tabled Equivalence Classes

The rules that follow for building test data are predicated on testing economy. They allow for the least number of test cases that can be created and executed with a reasonable level of confidence that we are testing effectively. It is possible to create fewer test cases, but these collapsed test cases will be less effective because of the error-masking phenomenon.

Error masking can occur when a test case contains more than one invalid value. The system will evaluate each value separately and the evaluation process may terminate when the first invalid value is discovered. If this happens, the remaining invalid values are not processed, which means we don’t know what would have happened had they been processed. In some computing languages, the ANDing operation (a Boolean operation) found in conditional actions exemplifies this. If the first condition is in the false or off state, the state of the second condition is not checked. We cannot know what the system’s behavior regarding the second condition will be unless its condition state is evaluated.

An abbreviated version of the test case construction guidelines Myers proposed is:

1. All of the valid equivalence classes can be incorporated into a single test case.
2. Each invalid equivalence class must be represented by a separate test case.

8.9.2.3 Defining Boundary Conditions for Equivalence Classes

Myers defines boundary conditions as values that fall on, above, or below the edges of equivalence classes. We can conclude that those values above and below the edges will have already been identified if the rules defined above have been used to develop the equivalence classes. So, the only new test cases we will create are the ones representing the values on each of the bounds. Consequently, boundary analysis is secondary to equivalence partitioning in implementation order. In addition, there is one major difference between the two techniques: Boundary analysis can also be applied to the output domain.

The guidelines presented below are once again adapted from Myers’ work.

**Input Domain**

1. For continuous input, write test cases that represent the lowest and highest valid values within the range.
2. For discrete ordered sets of input, construct test cases that represent the first and last elements in the set (e.g., a sequential input file).

**Output Domain**

1. For continuous output, construct input test cases that will cause output values to be generated for the highest and lowest value in the output range.
2. For discrete ordered sets of output, write input test cases that ensure that the first and last output elements will be processed (e.g., ensure that the first and last detail lines in an output report are printed).

8.9.3 Error Guessing

Error guessing is the process of using intuition and past experience to fill in gaps in the test data set. There are no rules to follow. The tester must review the test records with an eye to recognizing missing conditions. Two familiar examples of error-prone situations are dividing by zero and calculating the square root of a negative number. Either of these will result in system errors and garbled output.

Other cases where experience has demonstrated errors may occur are the processing of variable-length tables, the calculation of median values for odd- and even-numbered populations, cyclic master file/database updates (improper handling of duplicate keys, unmatched keys, etc.), overlapping storage areas, overwriting of buffers, forgetting to initialize buffer areas, and so forth. I am sure you can think of plenty of circumstances unique to your hardware/software environments and use of specific programming languages.

Error guessing is as important as equivalence partitioning and boundary analysis because it is intended to compensate for their inherent weaknesses. As equivalence partitioning and boundary analysis complement one another, error guessing complements both of these techniques.

8.10 HYBRID APPROACHES

8.10.1 Decision Logic Tables

Decision logic tables (DLTs) are unique in the fact that they can be constructed from segments of the system’s functional specification, or may be based on program flow charts or source code listings. From a testing perspective, this approach could be classed as either White Box or Black Box—hence the Gray Box classification.

If you are already familiar with DLTs you may wish to skip this section. But the DLT is a design tool, so why should it be included in a discussion of test case design strategies? Because the DLT can be viewed as a path coverage approach that offers an advantage basis testing does not: The DLT format allows a test for completeness that ensures no path in any module is omitted. In addition, DLT looks at combinations of equivalence classes (testing their interaction).

The structural complexity of the module is computed as the product of the number of possible states for each condition for each decision. This produces a much larger complexity value than the value of C that would be obtained if McCabe’s metric (explained in the "Basis Testing" section) were used. The value that is produced represents the total possible logical combina-
tions of condition states, not the number of independent combinations. Thus, the value of the total table complexity will differ from the Cyclomatic complexity. The advantage in knowing the total table complexity is that logical completeness can be verified. Thus, the DLT approach addresses the major criticism of White Box strategies that they cannot account for missing paths.

Developing one test case for each rule in the embedded DLT can create a complete set of test cases for a specific program module. One DLT diagram is constructed for each program module. Test cases from each DLT are subsequently merged to form a test data set that may in turn be merged with other test data sets created using other White Box and/or Black Box methods.

From a testing perspective, the DLT is an important tool for software reliability [2,26]. The DLT is a tabular diagram that is used to clarify complex logic that has previously been specified in a design narrative [14]. DLTs deal only with conditional logic, allowing the designer to easily understand a situation that contains many decision steps. Such decisions will ultimately end up in If/Then/Else form in the final program. A distinct advantage is that the decision-making logic of DLTs is void of the If/Else nesting, which often occurs in the narrative description and is perpetuated in the program code. Even structured English descriptions may contain If/Else nesting.

The problem with If/Else nesting is that of increasing structural complexity. Our ability to deal with complexity falls off rapidly after a certain level has been reached [24,32]. When this happens, we begin to introduce errors into our work. The goal of a decision table is to reduce a narrative to a set of conditions and actions that can be easily implemented in If/Then/Else form.

Conditions and the actions dependent on those conditions represent everything except simple sequence. In the selection construct the actions are dependent upon the state of the condition being evaluated. Most selections are limited to two mutually exclusive alternatives; however, some decisions involve evaluating conditions with more than two mutually exclusive alternatives. The DLT format does not distinguish between limited-entry and extended-entry conditions. It merely organizes the conditions and actions so the proper actions are associated with the right condition-state combinations.

The iteration construct, as well, can be modeled using conditions and actions. A loop is a repeated set of actions. In some instances, the set is repeated if a condition that represents the exit criterion is true (the While or pretest loop) and in others the set is repeated until the condition that represents the exit criterion is false (the Until or posttest loop). Any loop can be specified if two things are known: what actions are going to be repeated (conditional on the exit criterion) and the condition state required in order to exit the loop (exit criterion). Consequently, iteration can be portrayed with a DLT.

**8.10.1.1 DLT Format**  A DLT is drawn with four quadrants, Condition Stub, Condition Entry, Action Stub, and Action Entry (refer to Figure 8.3). Conditions are listed in top-down fashion in the Condition Stub in order of their impact on the processing logic. More comprehensive conditions are listed first.
For instance, end-of-file is the most comprehensive condition of all because processing stops when it is reached. Consequently, it would be listed first. In Control Break processing, the innermost processing level would be represented in a DLT as the last condition listed in the Condition Stub and the condition representing the outermost level would be listed first.

Condition state names are placed in the Condition Entry quadrant at a level corresponding to the condition they define. The number of condition states for a specific condition defines the condition complexity of that condition.

Actions are entered in the Action Stub, listed in the order in which they will execute from top to bottom. Which actions are dependent on which condition state combinations is specified in the Action Entry quadrant, which is divided into rules.

A rule is a vertical column through the entry side (both condition and action) of the table. It represents a unique combination of condition states and the actions that are executed when that combination occurs. The number of rules in a DLT is a function of the product of the condition complexities for all of the conditions in the Condition Stub. A DLT containing 3 conditions with each condition having 2 possible states would contain 8 rules ($2 \times 2 \times 2 = 8$). The value 8 also represents the table complexity, which is a measure of the total number of logic paths through the table.

### 8.10.1.2 Enumerating the Rules

**Step 1.** The rules in a DLT are determined by dividing the entry side of the table into the same number of rules as there are condition states for the first (most comprehensive) condition in the Condition Stub (see Table 8.6).

**Step 2.** Divide each of the previously created subdivisions into the same number of rules as there are condition states in the second (next most comprehensive) condition (see Table 8.7).

| Table 8.6 | Enumerating the Rules |
|-----------|
| **End of File?** | No | Yes |
| Process | | |
| Transaction | | |
| Exit | | |

| Table 8.7 | Enumerating the Rules |
|-----------|
| **End of File?** | No | Yes |
| **Transaction Type?** | A | C | D | O | A | C | D | O |
| Process | | | | | | | | |
| Transaction | | | | | | | | |
| Exit | | | | | | | | |
Step 3. Repeat Step 2 for each of the remaining conditions in the condition stub.

Developing the table in this manner ensures that all possible condition-state combinations are present.

8.10.1.3 Specifying Actions  When all of the rules have been enumerated, the actions dependent on each rule may be specified. More than one action can occur as a consequence of a particular rule. If several actions are dependent on a single rule, they should be listed in top-down sequence in the Action Stub. Placing an X in the column that represents the rule across from the action indicates which actions are executed (see Table 8.8).

8.10.1.4 Condition-State Indifference  Certain condition states, when they occur, overrule other condition states. When this happens the dominant condition state is indifferent to the values of subsequent condition states that may finish out the logical combination of states defining the rule.

Indifference is important because when it occurs the table can be collapsed. A collapsed table is one that is stated more precisely: Rules that have different logical significance, but which result in the same set of actions, are combined into a single rule. From a software-testing standpoint, one test case must be generated for each rule in the table, and when rules can be collapsed because of condition-state indifference, fewer test cases have to be constructed (see Table 8.9).

Table 8.8 Indicating the Actions

<table>
<thead>
<tr>
<th>End of File?</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Type</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>Process Transaction Exit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Rule Complexity =</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: X in column specifies action taken

Table 8.9 Rule Complexity in Collapsed Table

<table>
<thead>
<tr>
<th>End of File?</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Type</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>Process Transaction Exit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Rule Complexity =</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: X in column specifies action taken
Collapsing the table in no way changes the total table complexity. Collapsed rules account for more than one logic path. Rule complexity is defined as the number of logical combinations of condition states (paths) for which the rule accounts.

Gane and Sarson [14] have defined a set of guidelines for collapsing DLTs.

1. Find a pair of rules for which the action(s) is (are) the same and the condition-state values are the same, except for one condition in which they differ.
2. Replace that pair of rules with a single rule using the indifference symbol (~) for the condition that was different.
3. Repeat guidelines 1 and 2 for any other pair of rules meeting the indifference criteria.

8.10.1.5 Proof of Completeness  Proof of completeness for a given DLT lies in the fact that the total table complexity can be computed as the product of the condition complexities, or as the sum of the rule complexities. If the product of the condition complexities equals the sum of the rule complexities, the table is proved complete: No logical combinations of condition states are missing.

The reason completeness can be proven is because of the fundamental premise of DLTs that “given a finite number of conditions with a finite number of condition states, a known number of combinations exist” [14].

8.10.1.6 The DLT as a Software Testing Tool  DLTs are an excellent tool for developing test cases based on path coverage criteria because the number of paths is dependent on the number of decisions in a module (see the discussion of basis testing below). They are also excellent because the tables incorporate all the decisions in a module in one format, with every possible combination of condition states (each combination is logic path) covered [11,12,26]. Moreover, the completeness check discussed previously ensures that no paths are forgotten.

Thus, each table results in a finite number of test cases being added to the test data set. Because a test case is generated for each of the rules in the table, the tester can be confident of completely testing the module’s logic from a path coverage perspective. Furthermore, if the embedded DLT can be collapsed, complete coverage can be attained with even fewer test cases. From the standpoint of testing economics, the smaller the number of test cases required for adequate coverage, the better.

Using the conventions set forth by Myers [27] concerning the development of test data records using equivalence classes, we can generate heuristics that we can use to fill in the test data set.

1. For all DLTs, assign an arbitrary number to each rule in the embedded DLT, continuing until all of the rules across tableaux are uniquely identified.
2. Until each rule representing a valid input value is covered, write a single test record covering as many of the valid rules as possible.

3. Until each rule representing invalid input value is covered, write a single test record covering one and only one of the rules.

Because the rules set forth by Myers [27] for creating test cases for equivalence classes are also applicable to the tabular format of the DLT diagram, a set of test data that are similar and in some instances redundant with the test data from equivalence classes is produced. The advantage of the DLT method over the equivalence class method is primarily the ability to prove completeness. In equivalence, partitioning has no inherent way to determine that a set of all possible classes has been identified. From a Black Box perspective, each equivalence class identifies a unique type of input to the module and each unique kind of input will invoke a distinct pathway through the module. A rule in a DLT identifies a unique set of circumstances (which, in turn, defines a unique kind of input). In this sense, the two test case design methods are equivalent, with the difference residing in the guidelines for establishing equivalence classes vs. the guidelines for enumerating the rules in the DLT.

For a complete discussion of DLTs as a tool for generating test data that includes comprehensive examples, refer to Mosley [26]. You might also want to investigate an extension of the DLT term structured tableau methodology [1,6,11,12] as well.

8.10.1.7 An Automated DLT Design Tool Logic Technologies offers LogiCASE, an automated reengineering and design tool based on DLTs. The Designer module allows a user to construct DLTs from requirements and design specifications. The Designer is fully functional, including a completeness check that will add missing rules to the table, and a Reduce command that will collapse the table to its most concise form. The Designer can translate the finished decision tables into C, Basic, FORTRAN, xBASE, COBOL, and English language statements.

The Reengineering module allows DLTs to be developed from selected C source code segments. It does not support reengineering for any other languages. The reengineering component automatically removes contradictions and redundancies from the source code and the table, using a process called “disambiguation.”

There is a limit of 20 conditions, 20 actions, and 10,240 rules per table. However, LogiCASE supports a nested table structure, which allows users to decompose larger tables into a set of smaller, less complex tables. It accomplishes this by allowing include files that contain other tables to replace actions and conditions.

The LogiCASE product can be obtained from Logic Technologies, 56925 Yucca Trail, Suite 254-A, Yucca Valley, CA 92284, Phone: 800-776-3818, Fax: 619-228-9653.
### 8.11 Code-Based Approaches

#### 8.11.1 Basis Testing Overview

Basis testing is a White Box test case technique employing control flow graph representations of program module logic. Control flow graphs are network diagrams graphically depicting the logical pathways through a module. Test cases are created based on a set of independent paths enumerated from the graph. Thomas J. McCabe developed this method, formally known as “structured testing,” and informally as basis testing. I will discuss McCabe’s approach as a White Box test case development strategy and not as a testing methodology. For a comprehensive discussion of McCabe’s methodology, see his original work [22,23].

The major advantage of McCabe’s approach is that it incorporates the Cyclomatic number as a measure of program/module complexity. The disadvantage is that the control flow charts are generated from either a flow chart before a module is coded or from the source listing after construction is completed. There are two problems with this approach. First, control flow graphs are logical in nature and flow charts and source listings are inherently physical or implementation dependent. Second, there is not a set of explicit guidelines for systematically converting the physical information contained in flow charts and source listings into the logical detail depicted in control flow graphs.

The second approach has several advantages. Control flow graph construction occurs earlier in the design process, and test case design happens before physical design (via flow charts) and coding. This facilitates design reviews, walkthroughs, and inspections, but more important, program and module testing are placed within the framework of a formal structured systems design methodology.

#### 8.11.1.1 The Basis Testing Technique

Basis testing, as described by McCabe [23], uses source code or flow charts to generate control flow graphs of program modules. The flow graphs enumerate the independent control paths and calculate the Cyclomatic number $V(G)$, a measure of procedural complexity (see Myers’s extension to the Cyclomatic complexity measure [28]). A basis set of test cases, covering the independent paths, is then created.

A control flow graph is a network diagram in which nodes represent program segments and edges are connectors from given segments to other program segments. Edges depict a transfer of control from one node to another and can represent any form of conditional or unconditional transfer. A node represents a decision with multiple emanating edges. A loop is a node with an emanating edge that returns to the node. A region is an area within a graph that is completely bounded by nodes and edges. Each graph has an entry and exit node. A path is a route through a graph that traverses edges from node to node, beginning with the entry node and ending with the exit node.

A node is a block of sequentially executed (imperative) actions and an
edge is a transfer of control from one block to another. The If/Then/Else/Endif and nested If/Then/Else/Endif are also examples of statements that conditionally transfer control to specific groups of actions.

Structured programming doctrine advocates that program logic be designed using only the logical constructs of sequence, selection, and iteration (e.g., see Boehm and Jacopini [5], King [19], and Orr [29,30]).

Internal structural complexity in program modules is a consequence of the number of functions the module implements, the number of inputs to the module and outputs from the module, and the number of decisions in the module. Structured programming doctrine dictates that a module should implement one and only one function [34,35]. If this basic rule of thumb is followed, complexity due to the number of functions is minimized. Structured programming dogma also stands on the premise that each module has but a single entry point and a single exit point. If such is the case, complexity due to the number of inputs and outputs can be controlled because the module interface is simplified. This leaves complexity because of the number of decisions as the major structural dimension contributing to module complexity.

McCabe’s complexity measure is an indication of a module’s decision structure. It measures procedural complexity as a function of the number of decisions in the graph. Cyclomatic complexity is a useful metric because it is also representative perceived complexity. Miller [24] found that the maximum amount of information the human mind can simultaneously process is three “bits.” He defined a bit as the amount of information required in order to discriminate between two equally likely alternatives. The total number of alternatives is two raised to a power equal to the number of discriminations that are involved in a complex decision.

Based on his findings, Miller formulated the $7 \pm 2$ rule which is why your telephone number is 7 digits long. A 7-number string is optimal for memorization of phone numbers as each digit is a discrimination alternative. Miller determined that the maximum number of alternatives humans can handle simultaneously is 9. If this principle is applied to the decision structure of program modules, the number of bits a programmer must process to understand a complex decision is a function of 2 raised to a power that represents the number of conditions contained in the decision. An If/Else nest three levels deep has $2^3$ (8) alternatives to comprehend—below the limit established by Miller. But adding another nested decision raises the number of alternatives to 16—well beyond the inherent limit. Consequently, a module with a Cyclomatic complexity greater than 10 is too complex for the human short-term memory to comprehend at one time and must be decomposed into several less complex modules. If it is not made less complex it will be error prone and difficult if not impossible to test.

If the Cyclomatic number ($C$) is greater than 10, a module should be reconstructed or it may be untestable. King [19] suggests such modules can be rendered less complex through decomposition into two or more subordinate modules. Decomposition necessitates redesign and/or recoding depending on
whether the control flow graphs were constructed from program listings or flow charts. The earlier structural complexity can be better measured from the standpoint of module design.

The Cyclomatic number can be calculated using any of three simple equations. The first equation representing the Cyclomatic number is:

\[ C = E - N + 2 \]

where

- \( C \) is the Cyclomatic number (\( C \) has been substituted for \( V[G] \))
- 2 is used instead of \( 2P \) (\( P \) is usually 0) to simplify and make the notation more meaningful
- \( E \) is the number of edges
- \( N \) is the number of nodes

The Cyclomatic complexity is computed as a function of the relationship of edges to nodes.

Complexity can also be computed as a function the regions in the control flow graph. Edges cannot cross one another, and regions formed by violations of this rule are not “legitimate” regions. Only legitimate regions can be included in the calculation of complexity.

The equation based on the number of regions is:

\[ C = R + 1 \]

where

- \( C \) is again the Cyclomatic number
- \( R \) is the number of legitimate regions

The Cyclomatic number can also be computed based upon the number of primitive decisions in the graph. A primitive decision is one evaluating the condition states associated with a single condition. Nested conditions and conditions connected by logical operators should be treated as though they were completely separate decisions.

The equation is:

\[ C = D + 1 \]

where

- \( D \) is the number of primitive decisions

The advantage of the latter two equations is that they are easier to understand and use. In fact, the control flow graph does not have to be constructed to use the third equation. The number of decisions in the program flow chart or source listing can be counted and substituted for \( D \) in the equation.
These equations are applied to individual program modules. A measure of the total program complexity can be derived from the sum of the individual module complexities with a factor subtracted out for redundant nodes. A redundant node is a node in a high-level module, such as the mainline module, that would be replaced by a subdiagram if the called module were called in-line as part of the superordinate module.

The equation is:

$$C_t = C_i + 2 - (N - 1)$$

where

- $C_t$ represents the total program complexity
- $C_i$ represents the individual module complexities
- $N$ is the number of modules in the program

The value of $C$ in each module represents the upper boundary for the maximum number of independent paths through a given program unit. If a set of control paths is constructed equal to $C$, test cases that exercise those paths will adequately test the module and constitute the basis set of test cases. The basis set does not necessarily execute all possible paths through a segment, but rather a subset from which all other paths can be fabricated.

Once the Cyclomatic number is known, the independent paths can be enumerated. To enumerate the basis set of paths:

1. Identify all nodes with either a unique letter or number.
2. Begin at the entry node and travel the network using the left-most path to exit node. List the nodes contained in the path and indicate on the diagram the edges traversed.
3. Follow the previous path backward until a node is encountered that has one or more unmarked emanating edges. Begin at the entry node and follow the preceding path to the node with an unmarked edge. Continue from that point to the exit node using the left-most unmarked edge.
4. If the new path at any time intersects a previous path follow the latter path to the exit node.

When no unmarked edges remain, the basis set of paths is complete. The total number of paths must equal $C$. If a set of paths that equals $C$ cannot be created, then the module is poorly designed and overly complex. It is best to redesign such modules.

It is extremely helpful when constructing the test cases to annotate the decision nodes with the condition being tested and to label the true and false branching edges (see figure 8.5). A test case is considered to be created when it will execute each independent path at least once. The basis set of test cases must also exercise each conditional branch at least once. A test case is an input data record containing either a valid or an invalid value for each data field.
Figure 8.5 is a control flow graph for the C program logic that processed the Global Table illustrated in Figure 8.4.

**Fig. 8.5** Arc Polling Software Main Module Control Flow Graph
8.11.2 Testing Object-Oriented Applications

Although this is not a book about object-oriented (OO) testing, many client-server software development efforts follow that development approach. They employ OO design and programming during the development process, and developers use automated object-oriented development tools. Thus, it is important to include these methods among the weapons in your testing armory.

Siegel [33] says the major issue in testing object-oriented systems is how to assure the integrity of a system that is designed to allow the user bounded but relatively unlimited flexibility.

Isn't this a question that applies to client-server systems in general? In fact, this is the goal of any C-S system. Thus, we can benefit by approaching the functional testing of C-S systems as the testing of highly flexible complex systems. Siegel states that OO systems are exactly of this complex and flexible character. So, the testing of client-server and object-oriented systems offers a common basis for comparison.

In fact, the PD/CTS approach discussed in chapter 3 was developed for use with OO programming environments such as C++ and Smalltalk. It is structured so that OO systems can be developed and tested very rapidly (Siegel speaks of two weeks for each OO component).

Object-oriented systems are designed in a layered, hierarchical set of abstractions (object classes). Object-oriented design (OOD) is a formal method that is iterative and incremental. The argument for OOD is that it approximates the way people work, allowing developers to analyze and design a little, code a little, test a little, and then do it all again, in iterative cycles.

Siegel suggests that object classes must be tested for complete code coverage within each class and should also include comprehensive testing of the methods or processes for all classes; the usage relationships within and between classes; any new and modified methods for derived or instantiated classes; and, finally, combinations of usage relationships between methods defined for the class, including inherited methods using a method redefined by the class.

8.11.2.1 Object-Oriented Testing Strategies  OOD requires early testing as part of a parallel design and development approach. Siegel suggests that the major impact of OOD is not on the types of testing needed, but on the amount of testing required. He argues that traditional White Box and Black Box testing strategies are still useful in an OO environment. He believes that the analysis of class structure in well-designed OO systems, with respect to usage and visibility, yields all testable combinations of object classes.

In fact, he suggests that testers must test the traditional minimum and maximum values, values above and below range, zero values, null values, and other special cases for each object class. This is exactly what the equivalence-partitioning, boundary-analysis, and error-guessing testing approaches have
always accomplished. Siegel’s work suggests that these three test case design
techniques should be employed to design test cases for each object class.

Furthermore, DLTs that can model the relations among equivalence
classes can be used to design test cases for relationships among object classes.
Using DLTs to test class relationships addresses the object visibility issue.
DLTs can be used to model interobject relationships and the indifference crite-
ron can be used to determine an object’s visibility to other objects. When
indifference occurs, one object determines the system’s behavior, and the other
objects’ effects are not expressed by the system. When indifference occurs, the
number of test cases is reduced. Thus, when one object dominates the system
response, that object is the only one that must be tested.

8.11.2.2 White Box OO Tests At a minimum, White Box test strategy for OO
software should include: provision for designing and managing a library of
testing history; allocation of resources to perform, document, and review class
and method analysis; development of a test apparatus comprising stubs, driv-
ers, and object libraries; use of standard procedures, naming conventions, and
libraries; maintenance of regression test suites and procedures; and produc-
tion or purchase of tools to automate capture/replay/comparison, execution of
the test suite, and verification and documentation of results [34].

8.11.2.3 Black Box OO Tests Black Box test strategy for OO software should
include: purchase of tools and training early in the development period; provi-
sion for analyzing application-domain relationships (using control flow graphs
is helpful); preparation and review of test scenarios during requirements anal-
ysis and system design; use of prototyping and reviews with users; use of auto-
mated regression testing; consideration of the extent of flexibility and
abstraction that should be built into the system (the more there is, the more
the techniques above may be needed) [34].

8.11.3 OO Testing Resources
There are many good books and articles on OO testing [4,7,10,17,31,33]. Two
excellent OO testing bibliographies are available on the World Wide Web. Rob-
ert Binder Systems Consulting (RSBC), Inc., maintains a comprehensive bibli-
maintains an equally useful bibliography at http://www.toa.com/pub/html/
testbib.html/.

8.12 Code-Level Error Checking
In addition to the test case design approaches discussed above, testing MS
Windows client-server applications requires the tester to deal with several
new areas. Dynamic Link Libraries (DLLs), Application Programming Inter-
face (API) calls, and memory leaks require consideration. Areas of concern at the programming-language level include read and write checks, DLL checks, API checks, dynamic memory checks, stack memory checks, static memory checks, type mismatches, locked and freed file handles, null and out-of-range pointers, invalid arguments, and conflicting flags. These types of errors are typically found in programs written in the most popular client-server development languages: Visual Basic (VB) and C/C++.

A point of concern with VB is that it is both a compiled and an interpreted language. The reason for this concern is the way developers use VB when writing their code. Frequently they do not compile the source code until just before it is given to the test team. They do all of the development and any testing (which is not usually very much) in interpreted VB. This has caused many problems, particularly in cross-Windows environments.

In one instance a developer was developing VB in Windows 95 and testing interpreted code in Windows 95. The system, however, was targeted to run on Windows NT, Windows 95, and Windows for Workgroups 3.11. The testers frequently reported problems in the other Windows environments that could not be reproduced in the development environment.

I puzzled over this until I realized the developer was using the interpreter and not compiling his code until it was time to create the executable file that was included in the build.

8.12.1 Test Drivers to Test Hidden Objects

Is structured capture/playback just for GUI testing? The GUI is a shell and structured capture/playback approaches are excellent for assuring that the user interface works, but can you test application functionality with a client-based GUI approach? Eric Schurr of Rational Software, Inc., says that any function that returns a value to the interface can be tested. He is essentially correct, but in many instances the test of the function is at best indirect. An input value may undergo many transformations before it is used by the function that is under test. If testers understand all of the transformations, they probably can devise and input values that will test the application function in question. Many times, though, testers don’t have access to such in-depth knowledge of the application’s internal specifications or the specifications are incomplete or incorrect with respect to the requirements.

What about testing hidden objects—objects that do not appear on the GUI screen? Certain application functions such as operating system behaviors, server functions, stored procedures, and data transfers do not return any output via the GUI. These functions result in the setting of internal states when the application runs and, in many instances, the setting, variable values, flags, etc. disappear when the application stops running. These activities occur in the application layer and in the data layer, not in the user interface layer. So, how can testers test these functions?
Front-line client-server testers are finding that software modules known as “drivers” are an effective way to test hidden objects. The testers use the drivers to feed contrived data directly to the object that implements the function under test. This makes the design and construction of test data simple and straightforward. The driver accepts the input and displays the result outside of the GUI.

The advantage of this approach is that it bypasses other intermediate levels of processing and is particularly productive when testing functions that are normally invoked by remote procedure calls. In some instances, it is the only way to test a hidden object.

The scripts that run the driver are written using the scripting language or recorded via capture-playback tools such as SQA TeamTest, Mercury Interactive’s WinRunner, Segue’s QA Partner or manually typed into the GUI text boxes, etc. In this manner, the tester performs a direct test of the hidden objects’ functionality.

As an alternative to constructing a separate GUI for each driver, testers can choose to call the drivers from test scripts and return a pass/fail code that the test tool recognizes. This method eliminates the need to create the GUI for the driver because the test scripts that call the driver can run against the application GUI. The two approaches are identical in the level of testing that can be achieved.

SQA TeamTest allows the tester to create user-defined test cases that can run executables or make calls directly to DLLs to test remote procedure calls. The latest version of SQA TeamTest supports calling most DLL functions and has rich support for logging events. This approach eliminates the need for specially written test DLLs. Figure 8.6 illustrates a user-defined call from an SQA TeamTest test script to an executable file. Figure 8.7 illustrates a user-defined call to a DLL from an SQA TeamTest test script.

**Fig. 8.6 User-Defined EXE Call Example from SQA TeamTest**

```c
#include <windows.h>
#include "rbudf.h"

int PASCAL WinMain (hInstance, hPrevInstance, lpCmdLine, nCmdLine, nCmdShow)
HANDLE hInstance;        /* current instance*/
HANDLE hPrevInstance;    /* previous instance*/
LPSTR lpCmdLine;         /*command line*/
int nCmdShow ;        /*Show-window type (open/icon)*/
{
    //Define Local Variable
    int iStatus;
    /* Assume RunTest returns 1 if SUCCESSFUL and 0 if an error occurs*/
    iStatus=RunTest ();
    if (iStatus==1)
        UDPSetResult ((HANDLE)NULL, RBUDF_PASS);
    else
        UDPSetResult ((HANDLE)NULL, RBUDF_FAIL);
}
```
Segue Software has developed an extension kit (EK) for QA Partner, a component of the Quality Works tool suite, that allows testers to develop drivers (in C code) and test nondisplayable application logic through DLL calls from its 4Test scripting language. The EK is also the mechanism Segue has chosen to address testing in cross-platform environments. The EK provides the physical (environment-specific) hooks to logical objects in test scripts written in 4Test.

Coding and/or recording the test procedures that feed the driver is tedious and time consuming. So expect a lot of work up front. Some testers see this as a disadvantage of capture-playback tools, however, the payoff once the test procedures are recorded and stored in the test repository makes it worthwhile. Stored test procedures can be supplemented and replayed to regression test the software each time a correction or enhancement is made. This can save up to 50% over the testing resources for manual testing.

There are many newer test tools (Rational PerformanceStudio, etc.) that capture http protocols, SQL calls, etc., and then use the recorded call to test an application using a proxy server. This is a common approach used by the major performance/load test products. Although designed as performance testing tools, they could also be used to test single-user functionality outside of the GUI for a more direct test of a specific function or group of functions.

In addition, products such as SQA Suite are able to detect and examine objects such as hidden PowerBuilder DataWindows. SQA is extremely adept at handling PowerBuilder code and recognizes PowerBuilder Objects by their internal names. It is also very good at doing this with VB code, but the SQA OCX must be loaded in order for SQA to recognize the VB internal object names.

**8.12.2 Memory Leaks**

Memory leaks have received a lot of press lately. What is a memory leak? Memory leaks occur because many Windows programs use memory ineffi-
ciently. Frequently, they do not release memory when terminated and continue to accumulate and hold even more memory when they are opened and closed more than once during a Windows session (as you can see, “memory leak” is a bit of a misnomer). Eventually you run out of memory.

You have probably experienced a Windows message stating that there is not enough memory to run the program you are trying to start. So, you close all the other applications and try to start it again, but you still receive the same error message. Finally you exit Windows and restart Windows, which frees up all of the trapped memory. Now your program runs.

8.12.3 Testing APIs

Commercially available operating systems and software and hardware products have defined software interfaces known as APIs. Any software that interacts with one of these products must conform to the API rules. APIs can define rules for passing information to be contained in external function calls, external variables, and symbolic names that represent numeric values or collections of variables [16]. API requirements include specification of Parameters, Function Return Values, Feature Test Macros, Variable Types, and Headers.

According to Jones, API function calls are susceptible to errors in the values they accept as arguments and in the values they return. Type-mismatch errors in function calls will be flagged at compile time, but errors in the logic that makes the function calls and the logic that checks the return values will not. Frequently, parameters and return values that have specific value limits need to be checked by logic with multiple relational operators. This complicates the logic and makes this an error-prone area.

DLTs can be invaluable in clarifying the complex logic that checks the return codes of API calls. As with functions, objects that are defined within an API have specified limits that indicate the values they can contain. Both objects and functions can be tested using the equivalence partitioning and boundary analysis approaches.

Another area where API problems can occur is in headers used to identify function calls. The headers sometimes do not reflect the API requirements. Common problems include incorrectly specified arithmetic types, syntax violations, and incorrect number values for macros (symbolic names) [16].

Jones describes two things API tests must do: scan the application source code to identify all uses of external identifiers and be able to verify identifiers in the application against a database that describes the APIs and the identifiers they define.

Jones also summarizes the information API tests should produce. They should uncover violations of the defined interface (number and type) and identifiers referenced that are not known in the API, including headers, functions, external identifiers, and macros.
With the capacities of the current generation of automated testing tools, it should be relatively easy to create API test drivers that can be called from the test scripts and return a pass/fail value to the test scripts. This method has been used to test DLLs via SQA TeamTest test scripts in previous product versions. Segue's EK could also be used to create API-checking test scripts.


Nu-Mega Technologies, Inc. (currently part of Compuware, Inc.), developed two powerful tools for error-checking and debugging MS Windows applications. The first, BoundsChecker, is an automatic error detection and analysis tool for C/C++ development. It inserts code into the programs using a compile-time instrumentation (CTI) technology. CTI detects errors that can remain unnoticed by regular compilers and symbolic debuggers. BoundsChecker detects 12 different types of errors:

- Invalid Windows API parameters
- Invalid Windows API return codes
- Invalid ANSI C parameters
- Invalid ANSI C return codes
- Memory leaks
- Resource leaks
- Dynamic memory overruns
- Stack memory overruns
- Data and heap corruption
- Memory-locking problems
- Null-pointer manipulations
- Processor faults

BoundsChecker is available in both 16- and 32-bit environments for MS-DOS, Windows 3.x, Windows 95, and Windows NT. Figure 8.8 specifically details the kinds of errors BoundsChecker can identify.

**Fig. 8.8** BoundsChecker C++ Error Categories (adapted from a table found at the NuMega website, http://www.numega.com)

<table>
<thead>
<tr>
<th>Error-Checking Technology</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Checking</td>
<td>Dynamic Memory Overrun</td>
</tr>
<tr>
<td></td>
<td>Stack Memory Overrun</td>
</tr>
<tr>
<td></td>
<td>Static Memory Overrun</td>
</tr>
<tr>
<td></td>
<td>Writing Array Out Of Range</td>
</tr>
<tr>
<td></td>
<td>Writing Dangling Pointer</td>
</tr>
</tbody>
</table>
### Fig. 8.8 BoundsChecker C++ Error Categories (adapted from a table found at the NuMega website, http://www.numega.com) (Continued)

<table>
<thead>
<tr>
<th>Error-Checking Technology</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing Null Pointer</td>
<td></td>
</tr>
<tr>
<td>Writing Uninitialized Pointer</td>
<td></td>
</tr>
<tr>
<td>Writing Overflows Memory</td>
<td></td>
</tr>
<tr>
<td>API Checking</td>
<td></td>
</tr>
<tr>
<td>API Failure: Windows Function Failed</td>
<td></td>
</tr>
<tr>
<td>API Failure: Windows Function Not Implemented</td>
<td></td>
</tr>
<tr>
<td>Format String Is Inconsistent</td>
<td></td>
</tr>
<tr>
<td>Freed Handle Is Still Locked</td>
<td></td>
</tr>
<tr>
<td>Handle Is Already Locked</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Illegal Format Specifier</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Bad Source Pointer</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Conflicting Combination Of Flags</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Format String Is Not Followed By Valid Arguments</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: General</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Invalid Pointer To Format String</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Not Enough Arguments For Format String</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Out Of Range</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Not Enough Arguments For Format String</td>
<td></td>
</tr>
<tr>
<td>Invalid Argument: Illegal Flags</td>
<td></td>
</tr>
<tr>
<td>C++ Checking</td>
<td></td>
</tr>
<tr>
<td>Inconsistent Use Of Delete Operator</td>
<td></td>
</tr>
<tr>
<td>Memory Allocation Conflict</td>
<td></td>
</tr>
<tr>
<td>Virtual Function Table Is Invalid</td>
<td></td>
</tr>
<tr>
<td>Compile Checking</td>
<td></td>
</tr>
<tr>
<td>Cast Of Pointer Loses Precision</td>
<td></td>
</tr>
<tr>
<td>Function Has Inconsistent Return Type</td>
<td></td>
</tr>
<tr>
<td>Global Declarations Are Inconsistent</td>
<td></td>
</tr>
<tr>
<td>Mismatch In Argument Type</td>
<td></td>
</tr>
<tr>
<td>Reading Array Out Of Range</td>
<td></td>
</tr>
<tr>
<td>Returning A Pointer To A Local Variable</td>
<td></td>
</tr>
<tr>
<td>Writing Array Out Of Range</td>
<td></td>
</tr>
<tr>
<td>Writing Overflows Memory</td>
<td></td>
</tr>
</tbody>
</table>
Soft-ICE is a powerful tool for debugging Windows system crashes. (Since I have had several major problems with virtual device drivers in Windows 95 that resulted in system crashes, I see the value of such a tool.) Soft-ICE provides source- and machine-level debugging, and it can debug the following system crashes.

- Virtual device drivers (VxDs) and dynamically loadable VxDs
- DLLs
- Windows device drivers
- Interrupt-service routines
- 16- and 32-bit Windows applications
- DOS device drivers
- DOS applications running under Windows

8.13 DATA-DRIVEN TESTING AND FRAMEWORK-BASED TEST SCRIPT EXAMPLES

SQA’s version of the BASIC programming language is almost as powerful as the real thing yet is designed to enhance testing capabilities. It can be used to read test data from input files and to write the results to output files. It can launch sophisticated SQL queries allowing testers to capture before and after snapshots of database tables.

An example of a real-world tax exemption application will be used to illustrate the data-driven approach to automated testing. Figure 8.9 describes an operational set of test objectives that were used to develop the test conditions that Figure 8.10 illustrates. This figure is excerpted from an MS Excel worksheet that defines test conditions for the sample application. The test conditions are Black Box and include equivalence partitions, boundary conditions, and error-guessing conditions.

**Fig. 8.9** Operational Testing Objectives for Creating Test Conditions in Excel Spreadsheet

<table>
<thead>
<tr>
<th>For Each GUI Screen Include Data Input Records That Cover the Following Conditions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. At least one GOOD record where all fields contain valid data (Passes all GUI and Server-level edits)</td>
</tr>
<tr>
<td>2. Include at least one GOOD Duplicate record</td>
</tr>
<tr>
<td>3. Include one INVALID record for GUI edit defined in the test requirements</td>
</tr>
<tr>
<td>4. Include one INVALID record for each server edit (Business Rule) defined in the test requirements</td>
</tr>
<tr>
<td>5. Include one or more records for each type of special processing described in the test requirements</td>
</tr>
<tr>
<td>6. Include one or more records for each type of Y2K date processing described in the test requirements</td>
</tr>
</tbody>
</table>
Fig. 8.9 Operational Testing Objectives for Creating Test Conditions in Excel Spreadsheet (Continued)

**For Each Alpha Field**
- Character Strings longer than stated field size
- Spaces
- Nulls
- Special Characters
- Numeric

**For Each Numeric Field**
- + and – Values
- Leading Sign Values
- Trailing Sign Values
- Zeroes
- Nulls
- Spaces
- Special Characters
- Alpha

**For Ranges**
- Value less than lower bound
- In range value
- Value greater than upper bound

**For Range Boundaries**
- Value equal to lower bound
- Value equal to upper bound

**For Each Date Field**
- Months 00 and 13
- Days 00 and 32
- Feb. 28, 29, 30
- Dates equal to the current date
- Dates greater than the current date
- Dates less than the current date

**US Format**
- M/D/Y
- MM/DD/YYYY
- MM-DD-YY
- MM-DD-YYYY

**Invalid Format**
- YYYY/MM/DD
- MM/YYYY/DD
- DD/YYYY/MM
### Y2K Dates

**Formats**
- MM/DD/YY (Implicit Century)
- MM/DD/YYYY (Explicit Century)

**Dates and Date Ranges**
- 01/01/1900 to 12/31/2050

**Two-digit user input**
- 12/31/1999
- 01/01/2000
- 01/03/2000
- 01/02/2000 to 01/07/2000

**Two-digit system date input**
- 12/31/1999
- 01/01/2000
- 01/03/2000
- 01/02/2000 to 01/07/2000

**Two-digit date input from files**
- 12/31/1999
- 01/01/2000
- 01/03/2000
- 01/02/2000 to 01/07/2000

**Leap Year Dates**
- 02/28/2000
- 02/29/2000
- 02/28/2004
- 02/29/2004
- 02/28/2008
- 02/29/2008

For the following situations test general date integrity
- Date Arithmetic
- Date Conversion
- Sort Dates
- Search Dates
- Dates in Variables
- Dates in Assignment Statements
- Date Values as Constants
- Dates Used in Indexing
- Dates Used in Linked Lists
- Dates Used in Internal Tables
### Fig. 8.10 Excerpted Test Conditions for Sample Application

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Detail/License List</td>
<td>Numeric—can be edited</td>
<td>Not Numeric</td>
<td>Will accept only numeric</td>
<td>May be controlled via key press</td>
</tr>
<tr>
<td>Screen GUI Fields FEIN</td>
<td></td>
<td>Zeroes</td>
<td>characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>May be controlled via</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>key press</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer Name</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Will accept only alpha</td>
<td>May be controlled via key press</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td>characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>May be controlled via</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>key press</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address1</td>
<td>Alpha—can be edited</td>
<td>Blank spaces</td>
<td>Will accept only alpha</td>
<td>May be controlled via key press</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>May be controlled via</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>key press</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address2</td>
<td>Alpha—can be edited</td>
<td>Blank spaces</td>
<td>Will accept only alpha</td>
<td>May be controlled via key press</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>May be controlled via</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>key press</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item/Event</td>
<td>Valid Conditions</td>
<td>Invalid Conditions</td>
<td>Expected Behavior</td>
<td>Comments</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>----------</td>
</tr>
<tr>
<td>City</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Will accept only alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Will accept only alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zip</td>
<td>Alpha—can be edited</td>
<td>Blank spaces</td>
<td>Will accept only alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>Numeric—can be edited</td>
<td>Non-numeric</td>
<td>Will accept only numeric characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zeroes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contact</td>
<td>Alpha—can be edited</td>
<td>Blank spaces</td>
<td>Will accept only alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td>Alpha—can be edited, 255 characters max</td>
<td>Blank spaces</td>
<td>Will accept only alpha characters</td>
<td></td>
</tr>
</tbody>
</table>
Fig. 8.10 Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustID</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Branded</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>License ID Number</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Federal Type</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Country Code</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>State Code</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>County Code</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>City Code</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Tax type</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Product Code</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Tran Type</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Exempt Pct</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Effective Date</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Expiration Date</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td>Alpha—noneditable</td>
<td>Can be edited</td>
<td>Display Only</td>
<td></td>
</tr>
<tr>
<td>License Detail Screen GUI Fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Fig. 8.10** Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exempt License ID</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Should accept only valid alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tax Type ID</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Should accept only valid alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction Type ID</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Should accept only valid alpha characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exempt %</td>
<td>Numeric—can be edited</td>
<td>Non-numeric</td>
<td>Should accept only valid numeric characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cert Effective Date</td>
<td>Standard Date Format, MM/DD/ YYYY</td>
<td>Not in standard date format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zeroes in month</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zeroes in day</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zeroes in year</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Fig. 8.10 Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
</table>
| Cert Expiration Date | Standard Date Format, MM/DD/ YYYY | Accepts two-digit year  
                        | Does not handle Feb 28  
                        | Does not handle Feb 29  
                        | Does not handle Feb 30  
                        | Does not handle month 13  
                        | Does not handle day 32  
                        | Does not handle year 1999  
                        | Does not handle year 2000  
                        | Does not handle year 2001  
                        | Accepts wrong format e.g. DD/MM/ YYYY or YYYY/MM/ DD  
                        | Accepts blanks  
                        | Accepts spaces  
                        | Accepts special characters  
                        | Not in standard date format  
                        | Zeroes in month  
                        | Zeroes in day  
                        | Zeroes in year |
### Fig. 8.10 Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment</td>
<td>Alpha—can be edited, 255 characters max</td>
<td>Accepts two-digit year</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Federal Type</td>
<td>Alpha—can be edited</td>
<td>Does not handle Feb 28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle Feb 29</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle Feb 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle month 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle day 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle year 1999</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle year 2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does not handle year 2001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts wrong format e.g. DD/MM/ YYYY or YYYY/MM/ DD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts blanks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Federal Type</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts only valid alpha characters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Fig. 8.10** Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country Code</td>
<td>Alpha—can be edited</td>
<td>Special characters Cannot be edited Non-alpha Blank spaces Special characters Accepts invalid Country codes Cannot be edited</td>
<td>Accepts only alpha characters that are valid Country codes</td>
<td></td>
</tr>
<tr>
<td>State Code</td>
<td>Alpha—can be edited</td>
<td>Special characters Accepts invalid state codes Cannot be edited Non-alpha Blank spaces</td>
<td>Accepts only alpha characters that are valid state codes</td>
<td></td>
</tr>
<tr>
<td>County Codes</td>
<td>Alpha—can be edited</td>
<td>Special characters Accepts invalid County codes Cannot be edited Non-alpha Blank spaces Special characters</td>
<td>Accepts only alpha characters that are valid County codes</td>
<td></td>
</tr>
</tbody>
</table>
The test conditions were used to develop the test data that SQA TeamTest test scripts executed. The test data was designed in the same Excel workbook on a different sheet. Figures 8.11 and 8.12 illuminate two portions of the total test data set that was created—test data for creating a new customer and for adding licenses for the new customer.

### Fig. 8.10 Excerpted Test Conditions for Sample Application (Continued)

<table>
<thead>
<tr>
<th>Item/Event</th>
<th>Valid Conditions</th>
<th>Invalid Conditions</th>
<th>Expected Behavior</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>City Code</td>
<td>Alpha—can be edited</td>
<td>Non-alpha</td>
<td>Accepts only alpha characters that are valid City codes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts invalid City codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-alpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blank spaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Special characters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accepts invalid FEIN values?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be edited</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 8.11 Example Test Data for Adding New Customer

<table>
<thead>
<tr>
<th>ID</th>
<th>Record Code</th>
<th>FEIN</th>
<th>Customer Name</th>
<th>Address 1</th>
<th>Address 2</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>Comments</th>
<th>Phone</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>123 North St.</td>
<td>Suite 1</td>
<td>Clayton</td>
<td>MO</td>
<td>63105</td>
<td>Good test record</td>
<td>314-555-9876</td>
<td>Joe Smith</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>00-123456789</td>
<td>Acme Company</td>
<td>124 North St.</td>
<td>Suite 2</td>
<td>Clayton</td>
<td>MO</td>
<td>63106</td>
<td>Too many FEIN #</td>
<td>314-555-9877</td>
<td>Joe Smith</td>
</tr>
</tbody>
</table>
### Fig. 8.11 Example Test Data for Adding New Customer (Continued)

<table>
<thead>
<tr>
<th>ID</th>
<th>Code</th>
<th>FEIN</th>
<th>Customer Name</th>
<th>Address 1</th>
<th>Address 2</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>Comments</th>
<th>Phone</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>E 00-123</td>
<td>Acme Company</td>
<td>125 North South St.</td>
<td>Suite 103</td>
<td>314-9878</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E 00-abcdefg</td>
<td>Acme Company</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E 00-0000000</td>
<td>Acme Company</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>E 00-###%^$&amp;</td>
<td>Acme Company</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E Acme Company</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>E Acme Company</td>
<td>124 North South St.</td>
<td>Suite 102</td>
<td>314-9877</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>E 00-1234567</td>
<td>1234456789</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>E 00-1234567</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>E 00-1234567</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>E 00-1234567</td>
<td>123 North South St.</td>
<td>Suite 101</td>
<td>314-9876</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>Suite 102</td>
<td>314-9877</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>Suite 103</td>
<td>314-9878</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>Suite 104</td>
<td>314-9879</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>127 North South St.</td>
<td>Suite 104</td>
<td>314-9879</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>128 North South St.</td>
<td>Suite 104</td>
<td>314-9880</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>129 North South St.</td>
<td>Suite 104</td>
<td>314-9881</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>130 North South St.</td>
<td>Suite 104</td>
<td>314-9882</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>E 00-1234567</td>
<td>Acme Company</td>
<td>131 North South St.</td>
<td>Suite 104</td>
<td>314-9883</td>
<td>Joe Smith</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 8.11 Example Test Data for Adding New Customer (Continued)

<table>
<thead>
<tr>
<th>ID</th>
<th>Record Code</th>
<th>FEIN</th>
<th>Customer Name</th>
<th>Address 1</th>
<th>Address 2</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>Comments</th>
<th>Phone</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>132 North South St.</td>
<td>Suite 110</td>
<td>MO</td>
<td>63105</td>
<td>Special Characters</td>
<td>314-555-9885</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>133 North South St.</td>
<td>Suite 111</td>
<td>Clayton</td>
<td>63105</td>
<td>Spaces for State</td>
<td>314-555-9886</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>63105</td>
<td>Blank State</td>
<td>314-555-9887</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>63105</td>
<td>Special Characters</td>
<td>314-555-9888</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>63105</td>
<td>Illegal State</td>
<td>314-555-9887</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>MO</td>
<td>Good test record</td>
<td>314-555-9888</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>MO</td>
<td>Spaces for Zip</td>
<td>314-555-9887</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>MO</td>
<td>Blank Zip</td>
<td>314-555-9888</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>MO</td>
<td>Special Characters</td>
<td>314-555-9887</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>MO</td>
<td>Special Characters</td>
<td>314-555-9888</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>MO</td>
<td>Spaces for phone number</td>
<td>-</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>MO</td>
<td>Blank phone number</td>
<td>-</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>134 North South St.</td>
<td>Suite 112</td>
<td>Clayton</td>
<td>MO</td>
<td>Special Characters</td>
<td>314-555-9887</td>
<td>Joe Smith</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>E</td>
<td>00-1234567</td>
<td>Acme Company</td>
<td>135 North South St.</td>
<td>Suite 113</td>
<td>Clayton</td>
<td>MO</td>
<td>numeric contact name</td>
<td>314-555-9888</td>
<td>123 456789</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8.12 Sample Data for Adding Customer License

<table>
<thead>
<tr>
<th>Record Code</th>
<th>Exempt License ID</th>
<th>Federal Type</th>
<th>Trans Type ID</th>
<th>Product Code</th>
<th>Exempt %</th>
<th>Cert Effective Date</th>
<th>Cert Expiration Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00-1234569</td>
<td>123 ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Good Test Record</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>AA-1234569</td>
<td>123 ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Good Test Record</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>ABCDEFG</td>
<td>123 ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Good Test Record</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>123 ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for ELI</td>
<td></td>
</tr>
</tbody>
</table>
### Fig. 8.12 Sample Data for Adding Customer License (Continued)

<table>
<thead>
<tr>
<th>Record Code</th>
<th>Exempt License ID</th>
<th>Federal Type</th>
<th>Trans Type ID</th>
<th>Product Code</th>
<th>Exempt %</th>
<th>Cert Effective Date</th>
<th>Cert Expiration Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank ELI</td>
</tr>
<tr>
<td>5 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for ELI</td>
</tr>
<tr>
<td>6 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Fed Type</td>
</tr>
<tr>
<td>7 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for Fed Type</td>
</tr>
<tr>
<td>8 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for Fed Type</td>
</tr>
<tr>
<td>9 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for TTI</td>
</tr>
<tr>
<td>10 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank TTI</td>
</tr>
<tr>
<td>11 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special TTI</td>
</tr>
<tr>
<td>12 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Numeric TTI</td>
</tr>
<tr>
<td>13 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Product Code</td>
</tr>
<tr>
<td>14 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank PC</td>
</tr>
<tr>
<td>15 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for PC</td>
</tr>
<tr>
<td>16 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters PC</td>
</tr>
<tr>
<td>17 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for Exempt %</td>
</tr>
<tr>
<td>18 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Exempt %</td>
</tr>
<tr>
<td>19 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for Exempt %</td>
</tr>
<tr>
<td>20 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for Exempt %</td>
</tr>
<tr>
<td>21 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for Cert Effective</td>
</tr>
<tr>
<td>22 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Cert Effective</td>
</tr>
<tr>
<td>23 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for Cert Effective</td>
</tr>
<tr>
<td>24 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Character for Cert Effective</td>
</tr>
<tr>
<td>25 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for Cert Expiration Date</td>
</tr>
<tr>
<td>26 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Cert Expiration Date</td>
</tr>
<tr>
<td>27 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for Cert Expiration Date</td>
</tr>
<tr>
<td>28 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for Cert Expiration Date</td>
</tr>
<tr>
<td>29 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for Tax Type ID</td>
</tr>
<tr>
<td>30 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Tax Type ID</td>
</tr>
<tr>
<td>31 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Numeric Characters for Tax Type ID</td>
</tr>
<tr>
<td>32 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters</td>
</tr>
<tr>
<td>33 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for Country Code</td>
</tr>
<tr>
<td>34 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank Country Code</td>
</tr>
<tr>
<td>35 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for Country Code</td>
</tr>
<tr>
<td>36 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for Country Code</td>
</tr>
<tr>
<td>37 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for State Code</td>
</tr>
<tr>
<td>38 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank State Code</td>
</tr>
<tr>
<td>39 E</td>
<td>00-1234569</td>
<td>ST</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Numeric State Code</td>
</tr>
</tbody>
</table>

### Table Notes:
- **Cert Effective Date**
- **Cert Expiration Date**
- **Comments**
- **Spaces** represent spaces in data fields.
- **Special Characters** include various symbols and may represent specific data formats or values.
- **Alpha Characters** include letters (A-Z, a-z).
- **Numeric Characters** include digits (0-9).
- **Spaces for Fed Type**
- **Blank Fed Type**
- **Spaces for TTI**
- **Blank TTI**
- **Spaces for Product Code**
- **Blank Product Code**
- **Spaces for PC**
- **Blank PC**
- **Spaces for Exempt %**
- **Blank Exempt %**
- **Spaces for Cert Effective**
- **Blank Cert Effective**
- **Spaces for Cert Expiration Date**
- **Blank Cert Expiration Date**
- **Spaces for Tax Type ID**
- **Blank Tax Type ID**
- **Spaces for Country Code**
- **Blank Country Code**
- **Spaces for State Code**
- **Blank State Code**
- **Spaces for Cert Expiration Date**
- **Blank Cert Expiration Date**
Fig. 8.12 Sample Data for Adding Customer License (Continued)

<table>
<thead>
<tr>
<th>Record Code</th>
<th>Exempt License ID</th>
<th>Federal Type</th>
<th>Trans Type ID</th>
<th>Product Code</th>
<th>Exempt %</th>
<th>Cert Effective Date</th>
<th>Cert Expiration Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for State Code</td>
</tr>
<tr>
<td>41 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for County Code</td>
</tr>
<tr>
<td>42 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank County Code</td>
</tr>
<tr>
<td>43 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for County Code</td>
</tr>
<tr>
<td>44 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for County Code</td>
</tr>
<tr>
<td>45 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Spaces for City Code</td>
</tr>
<tr>
<td>46 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Blank City Code</td>
</tr>
<tr>
<td>47 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Alpha Characters for City Code</td>
</tr>
<tr>
<td>48 E</td>
<td>00-1234569</td>
<td>123</td>
<td>ST</td>
<td>9876</td>
<td>1.2345</td>
<td>1/1/97</td>
<td>12/31/99</td>
<td>Special Characters for City Code</td>
</tr>
</tbody>
</table>

The approach used to create the test data is described extensively in my previous work [25]. To reiterate briefly, it is only necessary to create one good record that will be added. All other records contain at least one bad data value that should be rejected by either the GUI validation processes or by the business rules validation processes. Remember, only one invalid value should be present in each input record. Putting more than one error value in a single test record can cause errors [27]. The test data values are exported to a sequential CSV file that an SQA test script reads as input during test execution.

Figure 8.13 is an example of an SQA test script that reads the input data from two text files and places the data values in text boxes on the GUI. The script adds new customer records and new licenses for the customers. Figures 8.14 and 8.15 illustrate the contents of the sequential CSV input files (Reccustl.txt and Reclcnse.txt, respectively).

Fig. 8.13 RECADD Test Script Listing

```
'$include *global.sbl*
Declare Sub AddLicense
'Declare global variables
Global fld(256) as string
Global lfld(256) as string
Global FirstRecord
Sub Main
'Dimension Local Variables
    Dim Result As Integer
    Dim I as Integer
    Dim Msgtext
    Dim Goodrec as Integer
    Dim Badrec as Integer
    Dim LGoodrec as Integer
    Dim LBadrec as Integer
    Dim Testtype
    Dim reccode
```
Fig. 8.13 RECAD Test Script Listing (Continued)

'Initially Recorded: 06/09/98 08:18:15
'Test Procedure Name: Data Entry Template
'Generic add customer location records

'Initialize variables
Goodrec=0
Badrec=0

'Open input files containing test data
Open "H:\SQA\data\Reccustl.txt" for Input as #1
Open "H:\SQA\data\Recclnse.txt" for Input as #2

'Begin processing loop
Do while Not EOF(1)

'Read data
'Must be edited to match number of fields on window
Input #1, reccode,fld(1),fld(2),fld(3),fld(4),fld(5),fld(6),fld(7),
fld(8),fld(9),fld(10)

'Check error code for skip record condition
If (Left$(reccode,1) = "H") then
   GoTo NoProcess
End If

'Check error code for good and bad record count
If (Left$(reccode,1) = "0") then
   Goodrec=goodrec + 1
Else
   Badrec=Badrec + 1
End If

'Put data in GUI fields
Window SetContext, "Name=w_wce_frame", ""
MenuSelect "File→New"
Delayfor 2000
'Puts data values in GUI fields
Window SetContext, "Name=w_wce_frame", ""
Window SetContext, "Name=w_customer_maintenance;ChildWindow", ""
InputKeys fld(1)
DataWindow Click, "Name=dw_customer_record;\;Name=customername",
   "Coords=23,9"
InputKeys fld(2)
DataWindow Click, "Name=dw_customer_record;\;Name=address1",
   "Coords=19,9"
InputKeys fld(3)
DataWindow Click, "Name=dw_customer_record;\;Name=address2",
   "Coords=19,10"
InputKeys fld(4)
DataWindow Click, "Name=dw_customer_record;\;Name=cityname",
   "Coords=13,12"
InputKeys fld(5)
DataWindow Click, "Name=dw_customer_record;\;Name=stateprovince",
   "Coords=14,8"
InputKeys fld(6)
DataWindow Click, "Name=dw_customer_record;\;Name=zipcode",
   "Coords=22,13"
InputKeys fld(7)
DataWindow Click, "Name=dw_customer_record;\;Name=description",
   "Coords=24,11"
Fig. 8.13  RECADD Test Script Listing (Continued)

InputKeys fld(8)  
DataWindow Click, "Name=dw_customer_record;\Name=contact",
"Coords=21,9"  
InputKeys fld(9)  
DataWindow Click, "Name=dw_customer_record;\Name=phoneno",
"Coords=11,9"  
InputKeys fld(10)  
'Save the data record
Window SetContext, "Name=w_wce_frame", **  
MenuSelect "File→Save"  
Delay for 2000  
'Add a license if there is one
If (Left$(reccode,1) = "0") then  
Call AddLicense  
End If  
Window SetContext, "Class=Shell-TrayWnd", **  
TabControl Click, "ObjectIndex=1", "Coords=378,12"  
'Goto label for skipping through loop without processing anything
NoProcess:  
'End of processing loop
Loop  
'Close all data files
Close #1  
Close #2  
'Write the number of good and bad records process to test log
Msgtext= "Customer Location Records Added:  Good= " & Goodrec & "  Bad= " & Badrec  
SQLLogMessage sqaNone, Msgtext, ""  
Msgtext= "Customer License Records Added:  Good= " & lGoodrec & "  Bad= " & lBadrec  
SQLLogMessage sqaNone, Msgtext, ""  
End Sub  
Sub AddLicense  
'Dimension Local variables
Dim Result As Integer  
Dim lreccode  
'Initialize local variables
lGoodrec=0  
lBadrec=0  
'Initially Recorded: 06/09/98 13:29:54  
'Read license data
Input #2,  
lreccode,lfld(1),lfld(2),lfld(3),lfld(4),lfld(5),lfld(6),lfld(7),lfld(8),lfld(9),lfld(10),lfld(11),lfld(12),lfld(13)  
'Begin processing loop
Do while Not EOF(2)  
'Check error code for skip record condition
If (Left$(lreccode,1) = "H") then  
GoTo NoLicense  
End If
Data-Driven Testing and Framework-Based Test Script Examples 221

8.13 DATA-DRIVEN TESTING AND FRAMEWORK-BASED TEST SCRIPT EXAMPLES

Fig. 8.13 RECADD Test Script Listing (Continued)

'Put data in GUI fields
Window SetContext, "Name=w_wce_frame", ""
Window SetContext, "Name=w_customer_maintenance;ChildWindow", ""
PushButton Click, "Name=cb_new_license"
Window SetContext, "Name=w_license_maintenance", ""
InputKeys ifld(1)
DataWindow Click, "Name=dw_1;\Name=federaltype", "Coords=20,10"
InputKeys ifld(2)
DataWindow Click, "Name=dw_1;\Name=transactiontypeid", ""
Window SetContext, "DropDownLB", "Activate=0"
Window Click, "", "Coords=21,3"
Window SetContext, "Name=w_license_maintenance", ""
DataWindow Click, "Name=dw_1;\Name=productproductgroupid", "Coords=23,15"
InputKeys ifld(4)
DataWindow Click, "Name=dw_1;\Name=exemptpercentage", "Coords=22,9"
InputKeys ifld(5)
DataWindow Click, "Name=dw_1;\Name=effectivefrom", "Coords=21,10"
InputKeys ifld(6)
DataWindow Click, "Name=dw_1;\Name=effectiveto", "Coords=22,5"
InputKeys ifld(7)
DataWindow Click, "Name=dw_1;\Name=tax_type_id", "Coords=6,12"
InputKeys ifld(8)
DataWindow Click, "Name=dw_1;\Name=countryid", "Coords=10,10"
InputKeys ifld(9)
DataWindow Click, "Name=dw_1;\Name=stateprovince", "Coords=38,10"
InputKeys ifld(10)
DataWindow Click, "Name=dw_1;\Name=countyid", "Coords=27,18"
InputKeys ifld(11)
DataWindow Click, "Name=dw_1;\Name=cityid", "Coords=24,14"
InputKeys ifld(12)
DataWindow Click, "Name=dw_1;\Name=description", "Coords=35,11"
InputKeys ifld(13)

DataWindow Click, "Name=dw_1;\Name=countryid", "Coords=10,10"
Window SetContext, "Name=w_license_maintenance", ""
InputKeys ifld(10)
DataWindow Click, "Name=dw_1;\Name=stateprovince", "Coords=38,10"
Window SetContext, "Name=w_license_maintenance", ""
InputKeys ifld(11)
DataWindow Click, "Name=dw_1;\Name=countyid", "Coords=27,18"
Window SetContext, "Name=w_license_maintenance", ""
InputKeys ifld(12)
DataWindow Click, "Name=dw_1;\Name=cityid", "Coords=24,14"
InputKeys ifld(13)

'Save License Record
Window SetContext, "Name=w_license_maintenance", ""
PushButton Click, "Name=cb_save"

'Check License error code to determine which command button to click
' AN ATTEMPT TO CLOSE THE CHILD WINDOW
' PushButton Click, "Name=cb_close"
' Window SetContext, "Caption=Close", ""
' InputKeys "y"
' PushButton Click, "Name=cb_close"
If Lreccode=0 then
    PushButton Click, "Name=cb_new_license"
    PushButton Click, "Name=cb_new"
else
Fig. 8.13 RECADD Test Script Listing (Continued)

```plaintext
PushButton Click, "Name=cb_close"
End If
' Read Next License record
Input #1,
  lreccode,lfld(1),lfld(2),lfld(3),lfld(4),lfld(5),lfld(6),lfld(7),lfld(8),lfld(9),lfld(10),lfld(11),lfld(12),lfld(13)
' Goto label used when no record is to be processed
NoLicense:
' End of processing loop
loop
End Sub
```

Fig. 8.14 Reccust1.txt Input File Listing

```plaintext
"E", "00-123", "Acme Company", "125 North South St.", "Suite 103", "Clayton", "MO", "63107", "Too few FEIN #", "John Smith", "3145559876"
"E", "00-1234567", "123456789", "123 North South St.", "Suite 101", "Clayton", "MO", "63105", "Numeric Customer Name", "John Smith", "3145559876"
"E", "00-1234567", "00-000000", "123 North South St.", "Suite 101", "Clayton", "MO", "63105", "Zeroes for CN", "John Smith", "3145559876"
"E", "00-1234567", ",", "123 North South St.", "Suite 101", "Clayton", "MO", "63105", "Spaces for CN", "John Smith", "3145559876"
"E", "00-1234567", ",", "123 North South St.", "Suite 101", "Clayton", "MO", "63105", "Blank CN", "John Smith", "3145559876"
"E", "00-1234567", "!@#$%^&*()!:<>", "123 North South St.", "Suite 101", "Clayton", "MO", "63105", "Special Characters", "John Smith", "3145559876"
```
8.13 DATA-DRIVEN TESTING AND FRAMEWORK-BASED TEST SCRIPT EXAMPLES

Fig. 8.14 R eccentric.txt Input File Listing (Continued)

"E", "00-1234567", "Acme Company", "128 North South St.", ",", "Clayton", "MO", "63105", "Blank Address2", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "129 North South St.", ",", "!@#$%^&*()", "Clayton", "MO", "63105", "Special Characters", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "130 North South St.", "Suite 108", ",", "MO", "63105", "Spaces for City", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "130 North South St.", "Suite 109", ",", "MO", "63105", "Blank city", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "131 North South St.", "Suite 110", ",", "MO", "63105", "Zeroes for City", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "133 North South St.", "Suite 111", "Clayton", "00", "MO", "63105", "Special Characters", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "133 North South St.", "Suite 112", "Clayton", "00", "MO", "63105", "Special Characters", "John Smith", "3145559876"
"E", "00-1234567", "Acme Company", "133 North South St.", "Suite 112", "Clayton", "00", "MO", "63105", "Special Characters", "John Smith", "3145559876"
"E", "00-1234567`, "Acme Company", "133 North South St.", "Suite 112", "Clayton", "00", "MO", "63105", "Special Characters", "John Smith", "3145559876"
**Fig. 8.15** Reclnse.txt Input File Listing

```plaintext
"0", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", "SLT", "1", "MO", "250", "1125", "Good Test Record"
"0", "AA-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", "SLT", "1", "MO", "250", "1125", "Good Test Record"
"0", "ABCDEFG", "123", "ST", "9876", "12345", "01011997", "12311997", "SLT", "1", "MO", "250", "1125", "Good Test Record"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for ELI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Blank Fed Type"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Special Characters for ELI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for Fed Type"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Alpha Characters for Fed Type"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for TTI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Blank TTI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Special TTI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Numeric TTI"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for Product Code"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Blank PC"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Alpha Characters for PC"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for Exempt %"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Blank Exempt %"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Special Characters for Exempt %"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Spaces for Cert Effective"
"E", "123", "ST", "9876", "12345", "01011997", "12311997", "Blank Cert Effective"
```

Fig. 8.15  reclense.txt Input File Listing (Continued)

   "E", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", ",", "1", "MO", "250", "1125", "Spaces for Tax Type Id"
   "E", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", ",", "1", "MO", "250", "1125", "Blank Tax Type Id"
   "E", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", ",", "1", "MO", "250", "1125", "Numeric Characters for Tax Type Id"
   "E", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", ",", "1", "MO", "250", "1125", "Special Characters"
   "E", "00-1234569", "123", "ST", "9876", "12345", "01011997", "12311997", ",", "1", "MO", "250", "1125", "Special Characters"

Fig. 8.16 is an example of a script that retrieves the records from a database table, runs a script that adds new records, and retrieves the records from the table a second time. It is used to verify that the records were added to the table. The verification here is that n. new records were added.

Fig. 8.16 Add Record Verification Procedure

' $include "GLOBAL.SBH"
'SCString$:
' Initially Recorded: 10/01/97 12:40:55
' Test Procedure Name: Get Txxxxx table

Sub Main
Dim Result As Integer
Dim Fileparms As String*128
Dim Length_1 As Long
Dim Length_2 As Long
Dim wrstr As String
'tnum is a global variable in GLOBAL.SBH that must be set to the table number in
'test
' if tnum = '00' error
    If tnum = "00" then
        'Error
        SQALogMessage sqaFail, "Calling procedure Error", _
        "The calling procedure did not set the Table number in 'tnum'"
        Exit Sub
    End If
'Build the isql input file
    Open "w:\sqa97\inhouse\sqlscrpt\sqlin.txt" for Binary Access Write as #1
    Len=2
    wrstr= "select * from T" + tnum + "\r\n"
    Put #1, 1, wrstr
    wrstr= "go\r\n"
    Put #1,, wrstr
    Close #1
    DelayFor (2000)
'Build the parm string for SQAShellExecute
    Kill "w:\sqa97\inhouse\outdata\t" + tnum + ".txt"
    Fileparms = "w:\sqa97\inhouse\sqlscrpt\",Fileparms
    SQAShellExecute "gett.bat","w:\sqa97\inhouse\sqlscrpt\",Fileparms
    DelayFor (5000)
    Result = FileT (Exists, "Name=w:\sqa97\inhouse\outdata\t" + tnum + ".txt",
    "CaseID=GETTCF1")
    length_b = 0
    length_a = 0
'Get the file length before the add record procedure is called
    length_b = FileLen("w:\sqa97\inhouse\outdata\t" + tnum + ".txt")
    Kill "w:\sqa97\inhouse\outdata\t" + tnum + ".txt" 'get rid of the first version
    of the file
    DelayFor (1000)
'Call the add record procedure
    CallProcedure RECADD
    DelayFor (5000)
'Re-acquire the txxxx table
    SQAShellExecute "gett.bat","w:\sqa97\inhouse\sqlscrpt\",Fileparms
    DelayFor (5000)
    Result = FileT (Exists, "Name=w:\sqa97\inhouse\outdata\tc" + tnum + ".txt",
    "CaseID=GETTCF1")
'Get the file length after the add record procedure is called
    length_a = FileLen("w:\sqa97\inhouse\outdata\t" + tnum + ".txt")
'Test it
    If length_b = length_a Then
        'Error
        SQALogMessage sqaFail, RECADDadd record failure", _
        "The T" + tnum + " table does not indicate new (additional) records."
        Else
            SQALogMessage sqaNone, "The Txxxx table shows new records", ""
        End If
'set tnum to '*00' to insure calling procedure sets in a legal table number
    tnum = "00"
End Sub

---

**Fig. 8.16 Add Record Verification Procedure (Continued)**

- Dim Length_2 As Long
- Dim wrstr As String
- 'tnum is a global variable in GLOBAL.SBH that must be set to the table number in
test
- if tnum = '00' error
  - If tnum = "00" then
    - 'Error
    - SQALogMessage sqaFail, "Calling procedure Error", _
    - "The calling procedure did not set the Table number in 'tnum'"
    - Exit Sub
  - End If
- 'Build the isql input file
  - Open "w:\sqa97\inhouse\sqlscrpt\sqlin.txt" for Binary Access Write as #1
  - Len=2
  - wrstr= "select * from T" + tnum + "\r\n"
  - Put #1, 1, wrstr
  - wrstr= "go\r\n"
  - Put #1,, wrstr
  - Close #1
  - DelayFor (2000)
- 'Build the parm string for SQAShellExecute
  - Kill "w:\sqa97\inhouse\outdata\t" + tnum + ".txt"
  - Fileparms = "w:\sqa97\inhouse\sqlscrpt\",Fileparms
  - SQAShellExecute "gett.bat","w:\sqa97\inhouse\sqlscrpt\",Fileparms
  - DelayFor (5000)
  - Result = FileT (Exists, "Name=w:\sqa97\inhouse\outdata\t" + tnum + ".txt",
  - "CaseID=GETTCF1")
  - length_b = 0
  - length_a = 0
- 'Get the file length before the add record procedure is called
  - length_b = FileLen("w:\sqa97\inhouse\outdata\t" + tnum + ".txt")
  - Kill "w:\sqa97\inhouse\outdata\t" + tnum + ".txt" 'get rid of the first version
  of the file
  - DelayFor (1000)
- 'Call the add record procedure
  - CallProcedure RECADD
  - DelayFor (5000)
- 'Re-acquire the txxxx table
  - SQAShellExecute "gett.bat","w:\sqa97\inhouse\sqlscrpt\",Fileparms
  - DelayFor (5000)
  - Result = FileT (Exists, "Name=w:\sqa97\inhouse\outdata\tc" + tnum + ".txt",
  - "CaseID=GETTCF1")
- 'Get the file length after the add record procedure is called
  - length_a = FileLen("w:\sqa97\inhouse\outdata\t" + tnum + ".txt")
- 'Test it
  - If length_b = length_a Then
    - 'Error
    - SQALogMessage sqaFail, RECADDadd record failure", _
    - "The T" + tnum + " table does not indicate new (additional) records."
    - Else
      - SQALogMessage sqaNone, "The Txxxx table shows new records", ""
    - End If
- 'set tnum to '*00' to insure calling procedure sets in a legal table number
  - tnum = "00"
End Sub
If further, perhaps visual, inspection of the stored data is required to determine if the values in each record were written correctly, utilities such as WISQL, MS Query, and VISDATA can be used. With testing tools that support ODBC protocols directly, you do not have to use an external utility program to access the database. SQA TeamTest is ODBC-compliant and this utility script was rewritten to directly access any database we need to verify.

The test script illustrated in Figure 8.13 is data driven but it was not designed from the framework-based perspective. It could be retrofitted in or made more structured. For example, the file open could be broken out into a separate procedure. In addition, the DOS path names could be entered as variables that could be specified at runtime. Figure 8.17 depicts a procedure for the open input file function. As another example, the read input record function could be implemented as a separate procedure, as in Figure 8.18. The portion of the test script that places the data in the GUI fields could also be put into its own procedure (see Figure 8.19).

---

**Fig. 8.17 Open Input File Procedure**

```
'$include "Global.sbh"
Sub Main
  Dim Result As Integer
  'Initially Recorded: 06/09/98  08:40:10
  'Test Procedure Name: Reads Input for #1
  Open "H:\SQA\data\Recwce01.txt" for Input as #1
End Sub
```

**Fig. 8.18 Read Input Record Procedure**

```
'$include "Global.sbh"
Sub Main
  Dim Result As Integer
  'Initially Recorded: 06/09/98  08:40:10
  'Must be edited to match number of fields on window
  Input #1,
    reccode,fld(1),fld(2),fld(3),fld(4),fld(5),fld(6),fld(7),fld(8),fld(9),fld(10)
End Sub
```

**Fig. 8.19 Populate GUI Fields Procedure**

```
'$include "Global.sbl"
Sub Main
  'Initially Recorded: 06/09/98  08:44:53
  'Test Procedure Name: Puts data values in GUI fields
  Window SetContext, "Name=w_wce_frame", ""
  Window SetContext, "Name=w_customer_maintenance;ChildWindow", ""
  InputKeys fld(1)
  DataWindow Click, "Name=dw_customer_record;\;Name=customername", "Coords=23,9"
  InputKeys fld(2)
  DataWindow Click, "Name=dw_customer_record;\;Name=address1", "Coords=19,9"
```
All of the procedures would be called then from the main test script. The main test script would be invoked in turn from a shell test procedure that starts the application, runs the main test procedure, and closes the application when the test is complete.

8.14 Adding Controls to the Test Data

The examples above demonstrate the basic ideas and implementation of the data-driven approach. To make this approach more rigorous, my associate, and scripting guru, Bruce Posey of the Archer Group has evolved the test scripts by placing control data in each data record. The control values determine how the script interacts with the application under test. As discussed above, this is an alternative approach to the one advocated by Kit [20].

In order to determine where to go and what to do in the application, the test script uses several values. For example, the test data and test scripts for an application in which several of the screens have multiple tabs could be easily developed. As illustrated in the example test record below (test data record is in boldface), the first six positions in each test data record are used as control values and the remainder as test data. The example contains only one actual data field, but the number of data fields can be very large because they are read into an array by the test script.

“G”, ”MT1J”, ”DELETE”, ””, ”1”, ”Delete a record”, ”FEDERAL”

- Error Code (Field 1): The “G” indicates this should be a good record with no errors expected.
- Control Code 1 (Field 2): The ”MT1J” code represents a particular menu item selection, a tab number, and a radio button selection.
- Control Code 2 (Field 3): ”DELETE” indicates the action is to delete a record that has been previously selected with the data of ”FEDERAL.”
Field 4 is not used in this example, but this field is secondary to Field 3 and is used as an additional control, if needed.

Data Length Control (Field 5): The "1" indicates the number of data fields after the comment field.

Comment Field (Field 6): "Delete a record" is the comment.

Data Field 1: "FEDERAL" is value in the single data field.

8.15 CONCLUSION

Cross-level functional testing involves a mixed bag of methods and techniques. There are the ones that were developed for testing mainframe system and can now be used to design and construct test cases for execution on distributed client-server systems. There are also methods and techniques developed for testing object-oriented systems that are applicable to testing client-server systems. There are new methods and techniques that are geared to the unique aspects of testing client-server systems.

Traditional Black and White Box techniques are our legacy from the mainframe world. Techniques such as Binder’s FREE methodology evolved from the object-oriented arena [4]. Techniques for testing GUIs, DLLs, and APIs are a function of the unique aspects of programming languages such as VB and Visual C++ that have come to dominate in the client-server development community.

8.16 REFERENCES


