The Automated Testing Handbook

The Automated Testing Handbook 1

About the Author 3

Introduction 3

Why automate? 4
When not to automate 8
How not to automate 9
Setting realistic expectations 11
Getting and keeping management commitment 15
Terminology 17

Fundamentals of Test Automation 19

Maintainability 20
Optimization 22
Independence 23
Modularity 25
Context 26
Synchronization 29
Documentation 30

The Test Framework 32

Common functions 32
Standard tests 37
Test templates 39
Application Map 41

Test Library Management 44

Change Control 44
Version Control 45
Configuration Management 46

Selecting a Test Automation Approach **48**

Capture/Playback *50*
Structure 51
Advantages 52
Disadvantages 52
Comparison Considerations 55
Data Considerations 57

Data-Driven *58*
Structure 60
Advantages 61
Disadvantages 62
Data Considerations 63

Table-Driven *64*
Structure 65
Advantages 66
Disadvantages 69

The Test Automation Process **70**

The Test Team *70*

Test Automation Plan *73*

Planning the Test Cycle *76*

Test Suite Design *77*

Test Cycle Design *79*

Test Execution **81**

Test log *81*

Error log *84*

Analyzing Results *85*
Inaccurate results 85
Defect tracking 87

Test Metrics **88**

Management Reporting **95**

Historical trends *97*

About the Author

Linda G. Hayes is an accountant and tax lawyer who has founded three software companies, including AutoTester - developer of the first PC-based test automation tool, and Worksoft – developer of the next generation of test automation solutions.

She is an award-winning author and popular speaker on software quality. She has been a columnist continuously since 1996 in publications including Computerworld , Datamation and StickyMinds and her work has been reprinted for universities and the Auerbach Systems Handbook. She co-edited Dare to be Excellent with Alka Jarvis on best practices and has published numerous articles on software development and testing.

But most importantly she brings two decades of personal experience with thousands of people and hundreds of companies that is distilled into practical advice.

Introduction

The risk of software failure has never been greater. The estimated annual economic impact ranges from $60 billion for poor testing to $100 billion in lost revenues and increased costs. Unfortunately, market pressure for the delivery of new functionality and applications has also never been stronger. This combination creates increasing pressure on software test organizations to improve test coverage while meeting ever-shorter deadlines with static or even declining resources. The only practical means to achieving quality goals within the constraints of schedules and budgets is to automate.

Since software testing is a labor-intensive task, especially if done thoroughly, automation sounds instantly appealing. But, as with anything, there is a cost associated with getting the benefits. Automation isn't always a good idea, and sometimes manual testing is out of the question. The key is to know what the benefits and costs really are, then to make an informed decision about what is best for your circumstances.

The unfortunate fact is that many test automation projects fail, even after significant expenditures of time, money and resources. The goal of this book is to improve your chances of being among the successful.

Why automate?

The need for speed is practically the mantra of the information age. Because technology is now being used as a competitive weapon on the front lines of customer interaction, delivery schedules are subject to market pressures. Late products can lose revenue, customers, and market share. But economic pressures also demand resource and cost reductions as well, leading many companies to adopt automation to reduce time to market as well as cut testing budgets.

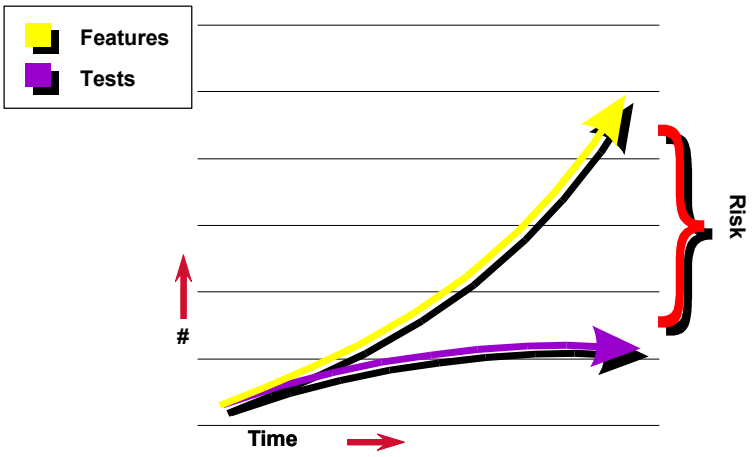
While it might be costly to be late to the market, it can be catastrophic to deliver a defective product. Software failures can cost millions or even billions, and in some cases entire companies have been lost. So if you don't have enough people or time to perform adequate testing to begin with, adding automation will not reduce software instability and errors. Since it is well-documented that software errors – even a single one – can cost millions more than your entire testing budget, the first priority should be first to deliver reliable software. Once that is achieved, then focus on optimizing the time and costs. In other words, if your software doesn't work, it doesn't matter how fast or cheap you deliver it.

Automated delivers software tests provide three key benefits: cumulative coverage to detect errors and reduce the cost of failure, repeatabililty to save time and reduce the cost to market, and leverage to improve resource productivity.

But realize that the test cycle will be tight to begin with, so don't count on automation to shorten it - count on it to help you meet the deadline with a reliable product. By increasing your coverage and thus reducing the probability of failure, automation can help to avoid the costs of support and rework, as well as potentially devastating costs.

Cumulative coverage

It is a fact that applications change and gain complexity over their useful life. As depicted in the figure below, the feature set of an application grows steadily over time. Therefore, the number of tests that are needed for adequate coverage is also constantly increasing.



Just a 10% code change still requires that 100% of the features be tested. That is why manual testing can't keep up – unless you constantly increase test resources and cycle time, your test coverage will constantly decline. Automation can help this by allowing you to accumulate your test cases over the life of the application so that both existing and new features can always be tested.

Ironically, when test time is short, testers will often sacrifice regression testing in favor of testing new features. The irony is that

the greatest risk to the user is in the existing features, not the new ones! If something the customer is already doing stops working – or worse, starts doing the wrong thing – then you could halt operations. The loss of a new feature may be inconvenient or even embarrassing, but it is unlikely to be devastating.

But this benefit will be lost if the automated tests are not designed to be maintainable as the application changes. If they either have to be rewritten or require significant modifications to be reused, you will keep starting over instead of building on prior efforts. Therefore, it is essential to adopt an approach to test library design that supports maintainability over the life of the application.

Leverage

True leverage from automated tests comes not only from repeating a test that was captured while performed manually, but from executing tests that were never performed manually at all. For example, by generating test cases programmatically, you could yield thousands or more - when only hundreds might be possible with manual resources. Enjoying this benefit requires the proper test case and script design to allow you to take advantage of external data files and other constructs.

Faster time to market

Because software has become a competitive weapon, time to market may be one of the key drivers for a project. In some cases, time is worth more than money, especially if it means releasing a new product or service that generates revenue.

Automation can help reduce time to market by allowing test execution to happen 24X7. Once the test library is automated, execution is faster and run longer than manual testing. Of course, this benefit is only available once your tests are automated.

Reduced cost

Software is used for high risk, mission critical applications that

of failure represent revenue and productivity. A single failure could cost more than the entire testing budget for the next century! In one case a single bug resulted in costs of almost $2 billion. The national department of standards and technology estimates the cost of correcting defects at $59.5 billion a year, and USA Today claims a $100 billion annual cost to the US economy.

Automation can reduce the cost of failure by allowing increased coverage so that errors are uncovered before they have a chance to do real damage in production.

Notice what was NOT listed as a benefit: reduced testing resources. The sad fact is that most test teams are understaffed already, and it makes no sense to try to reduce an already slim team. Instead, focus on getting a good job done with the time and resources you have. In this Handbook we will present practical advice on how to realize these benefits while keeping your expectations realistic and your management committed.

When not to automate

The cornerstone of test automation is the premise that the expected application behavior is known. When this is not the case, it is usually better not to automate.

Unstable design

There are certain applications that are inherently unstable by design. For example, a weather-mapping system or one that relies on real-time data will not demonstrate sufficiently predictable results for automation. Unless you have a simulator that can control the inputs, automation will be difficult because the expected results are not known

Also, if you can't control the application test environment and data, then automation will be almost impossible. The investment required to develop and maintain the automated tests will not be offset by the benefits, since repeatability will be doubtful.

If your application is highly configurable, for example, or has other attributes that make its design variable, then either forget automation or focus on implementing only selected configuration profiles. Whatever you do, don't try to reproduce all of the configurability of the application into the test library, otherwise you will end up with excessive complexity, high probability of test failure, and increased maintenance costs.

Inexperienced testers

If the person(s) automating the test are not sufficiently experienced with the application to know the expected behavior, automating their tests is also of doubtful value. Their tests may not accurately reflect the correct behavior, causing later confusion and wasted effort. Remember, an automated test is only as good as the person who created it.

If you have inexperienced testers who are new to the team, they make the best manual testers because they will likely make the same mistakes that users will. Save automation for the experts.

Temporary testers

In other cases, the test team may be comprised primarily of personnel from other areas, such as users or consultants, who will not be involved over the long term. It is not at all uncommon to have a "testfest" where other departments contribute to the test effort. But because of the initial investment in training people to use the test tools and follow your library design, and the short payback period of their brief tenure, it is probably not time or cost effective to automate with a temporary team. Again, let them provide manual test support while permanent staff handles automation.

Insufficient time, resources

If you don't have enough time or resources to get your testing done manually in the short term, don't expect a tool to help you. The initial investment for planning, training and implementation will take more time in the short term than the tool can save you. Get through the current crisis, then look at automation for the longer term.

Keep in mind that automation is a strategic solution, not a short term fix.

How not to automate

Whatever you do, do not simply distribute a testing tool among your testers and expect them to automate the test process. Just as you would never automate accounting by giving a program compiler to the accounting department, neither should you attempt to automate testing by just turning a testing tool over to the test group.

It is important to realize that test automation tools are really just specialized programming languages, and developing an automated test library is a development project requiring commensurate skills.

Automation is more than capture/replay

If you acquired a test tool with the idea that all you have to do is record and playback the tests, you are due for disappointment. Although it is the most commonly recognized technique, capture/replay is not the most successful approach. As discussed in a later chapter, Selecting an Automation Approach, capture and replay does not result in a test library that is robust, maintainable or transferable as changes occur.

Don't write a program to test a program!

The other extreme from capture/replay is pure programming. But if you automate your tests by trying to write scripts that anticipate the behavior of the underlying program and provide for each potential response, you will essentially end up developing a mirror version of the application under test! Where will it end? Who tests the tests? Although appealing to some, this strategy is doomed - no one has the time or resources to develop two complete systems.

Ironically, developing an automated test library that provides comprehensive coverage would require more code than exists in the application itself! This is because tests must account for positive, negative, and otherwise invalid cases for each feature or function.

Automation is more than test execution

So if it isn't capture/replay and it isn't pure programming, what is it? Think of it this way. You are going to build an application that automates your testing, which is actually more than just running the tests. You need a complete process and environment for creating and documenting tests, managing and maintaining them, executing them and reporting the results, as well as managing the test environment. Just developing scores of individual tests does not comprise a strategic test automation system.

Duplication of effort

The problem is, if you just hand an automation tool out to individual testers and command that they automate their tests, each one of them will address all of these issues - in their own unique and

personal way, of course. This leads to tremendous duplication of effort and can cause conflict when the tests are combined, as they must be.

Automation is more than test execution

So if it isn't capture/replay and it isn't pure programming, what is it? Think of it this way. You are going to build an application that automates your testing, which is actually more than just running the tests. You need a complete process and environment for creating and documenting tests, managing and maintaining them, executing them and reporting the results, as well as managing the test environment. Just developing scores of individual tests does not comprise a strategic test automation system.

Need for a framework

Instead, approach the automation of testing just as you would the automation of any application - with an overall framework, and an orderly division of the responsibilities. This framework should make the test environment efficient to develop, manage and maintain. How to develop a framework and select the best automation approach are the focus of this handbook.

Remember, test tools aren't magic - but, properly implemented, they can work wonders!

Setting realistic expectations

All too often, automated testing tools are expected to save the day by making up for too little time, resources, or expertise. Unfortunately, when these expectations are inevitably disappointed, automation or the tool itself gets a bad name. Before any effort can be deemed a success, realistic expectations must be set up front.

There are three important things to remember when setting expectations about test automation: one, an initial as well as ongoing investment in planning, training and development must be made before any benefits are possible; two, the time savings come only when automated tests can be executed more than once, by more than one person, and without undue maintenance requirements; three, no tool can compensate for the lack of expertise in the test process.

Test automation is strategic

If your test process is in crisis and management wants to throw money at a tool to fix it, don't fall for it. Test automation is a long term, strategic solution, not a short term band-aid. Buying a test tool is like joining a health club: the only weight you have lost is in your wallet! You must use the club, sweat it out and invest the time and effort before you can get the benefits.

Use consultants wisely

Along the same lines, be wary about expecting outside consultants to solve your problems. Although consultants can save you time by bringing experience to bear, they are not in and of themselves a solution. Think of consultants as you would a personal trainer: they are there to guide you through your exercises, not to do them for you! Paying someone else to do your situps for you will not flatten your stomach.

Here's a good rule of thumb to follow when setting expectations for a test tool. Calculate what your existing manual test iteration requires, then multiply by (5) five for a text-based user interface and (10) ten for a GUI, then add on the time scheduled for training and planning. GUI interfaces have inherently more complexity than text interfaces.

This will approximate the time it will take to properly automate your manual tests. So, if it takes you two weeks to execute one iteration of tests manually, plan for ten to twenty weeks after training and planning are complete to get through your first automated iteration. From there on out, though, you can cut each iteration in half or more. Naturally, these are only approximations and your results may be

different. For intensive manual test processes of stable applications, you may see an even faster payback.

Not everything can be automated

But remember, you must still allow time for tasks that can't be automated - you will still need to gather test requirements, define test cases, maintain your test library, administer the test environment, and review and analyze the test results. On an ongoing basis you will also need time to add new test cases based on enhancements or defects, so that your coverage can constantly be improving.

Accept gradual progress

If you can't afford the time in the short term, then do your automation gradually. Target those areas where you will get the biggest payback first, then reinvest the time savings in additional areas until you get it all automated. Some progress is better than none!

Plan to keep staff

As pointed out earlier, don't plan to jettison the majority of your testing staff just because you have a tool. In most cases, you don't have enough testers to begin with: automation can help the staff you have be more productive, but it can't work miracles. Granted, you may be able to reduce your dependence on temporary assistance from other departments or from contractors, but justifying testing tools based on reducing staffing requirements is risky, and it misses the point.

The primary goal of automation should be to increase test coverage, not to cut testing costs. A single failure in some systems can cost more than the entire testing budget for the next millennia. The goal is not to trim an already slim testing staff, it is to reduce the risk and cost of software failure by expanding coverage.

Reinvest time savings

As your test automation starts to reap returns in the form of time savings, don't automatically start shaving the schedule. The odds are that there are other types of tests that you never had time for before, such as configuration and stress testing. If you can free up room in the schedule, look for ways to test at high volumes of users and transactions, or consider testing different platform configurations. Testing is never over!

When setting expectations, ask yourself this question: Am I satisfied with everything about our existing test process, except for the amount of time it takes to perform manually? If the answer is yes, then automation will probably deliver like a dream. But if the answer is no, then realize that while automation can offer great improvements, it is not a panacea for all quality and testing problems.

The most important thing to remember about setting expectations is that you will be measured by them. If you promise management that a testing tool will cut your testing costs in half, yet you only succeed in saving a fourth, you will have failed! So take a more conservative approach: be up front about the initial investment that is required, and offer cautious estimates about future savings. In many cases, management can be satisfied with far less than you might be.

For example, even if you only break even between the cost to automate and the related savings in direct costs, if you can show increased test coverage then there will be a savings in indirect costs as a result of improved quality. In many companies, better quality is more important than lower testing costs, because of the savings in other areas: failures can impact revenues, drive up support and development costs, and reduce customer confidence.

Getting and keeping management commitment

There are three types of management commitment needed for successful test automation: money, time and resources. And it is just as important to keep commitment as it is to get it in the first place! Keep in mind that test automation is a project that will continue for the life of the application under test.

Commit money

Acquiring a test automation tool involves spending money for software, training and perhaps consulting. It is easier to get money allocated all at once instead of piece meal, so be careful not to buy the software first then decide later you need training or additional services. Although the tool itself may be advertised as "easy to use", this is different from "easy to implement". A hammer is easy to swing, but carpentry takes skill.

Do a pilot

Just because the money is allocated all at once, don't spend it that way! If this is your first time to automate, do a small pilot project to test your assumptions and prove the concept. Ideally, a pilot should involve a representative subset of your application and have a narrow enough scope that it can be completed in 2-4 weeks.

Take the time to carefully document the resource investment during the pilot as well as the benefits, as these results can be used to estimate a larger implementation. Since you can be sure you don't know what you don't know, it is better to learn your lessons on a small scale. You don't learn to drive on a freeway!

Commit time

All too often tools are purchased with the expectation that the acquisition itself achieves automation, so disappointment sets in when results aren't promptly forthcoming. It is essential to educate management about the amount of time it takes to realize the benefits, but be careful about estimating the required time based on marketing

literature: every organization and application is different.
A pilot project can establish a sound basis for projecting a full scale rollout.

When you ask for time, be clear about what will be accomplished and how it will be measured.

Commit resources

Remember that even though test automation saves resources in the long run, in the short term it will require more than a manual process. Make sure management understands this, or you may find yourself with a tool and no one to implement it.

Also be sure to commit the right type of resources. As further described in the Test Team section of this Handbook, you will need a mix of skills that may or may not be part of your existing test group. Don't imagine that having a tool means you can get by with less skill or experience: the truth is exactly the opposite.

Track progress

Even though benefits most likely won't be realized for several months, it is important to show incremental progress on a regular basis - monthly at the least. Progress can be measured in a number of ways: team members trained on the tool, development of the test plan, test requirements identified, test cases created, test cases executed, defects uncovered, and so forth.

Identify the activities associated with your test plan, track them and report them to management regularly. Nothing is more disconcerting than to wait for weeks or months with no word at all. Also, if you run up against obstacles, it is critical to let management know right away. Get bad news out as early as possible and good news out as soon as you can back it up.

Adjust as you go If one of your assumptions changes, adjust the schedule and expectations accordingly and let management know right away. For example, if the application is not ready when expected, or if you lose resources, recast your original estimates and inform everyone concerned. Don't wait until you are going to be late to start explaining why. No one likes surprises!

Plan for the long term Be sure to keep focus on the fact that the test automation project will last as long as the application under test is being maintained. Achieving automation is not a sprint, it is a long distance run. Just as you are never through developing an application that is being actively used, the same applies to the test library.

In order for management to manage, they must know where things stand and what to expect. By letting them know up front what is needed, then keeping them informed every step of the way, you can get their commitment and keep it.

Terminology

Throughout this Handbook we will be investing certain terms with specific meanings.

Requirement A required feature or function of the application under test. A business requirement is a statement of function that is necessary for the application to meet its intended use by the customer: the "what" of the system. A design feature is an attribute of the way in which the functions are actually implemented: the "how" of the system. A performance requirement spells out the volume and speed of the application, such as the maximum acceptable response or processing time and the highest number of simultaneous users.

Test This term will be used to describe the combination of a test case and a test script, as defined below.

Test Case A test case is a set of inputs and expected application response that will confirm that a requirement has been met. Depending on the automation approach adopted, a test case may be stored as one or more data records, or may be stored within a test script.

Test Script A test script is a series of commands or events stored in a script language file that execute a test case and report the results. Like a program, a test script may contain logical decisions that affect the execution of the script, creating multiple possible pathways. Also, depending on the automation approach adopted, it may contain constant values or variables whose values change during playback. The automation approach will also dictate the degree of technical proficiency required to develop the test script.

Test Cycle A test cycle is a set of individual tests that are executed as a package, in a particular sequence. Cycles are usually related to application operating cycles, or by the area of the application they exercise, or by their priority or content. For example, you may have a build verification cycle that is used to establish acceptance of a new software build, as well as a regression cycle to assure that previous functionality has not been disrupted be changes or new features.

Test Schedule A test schedule consists of a series of test cycles and comprises a complete execution set, from the initial setup of the test environment through reporting and cleanup.

Fundamentals of Test Automation

It is a mistake to assume that test automation is simply the capture and replay of a manual test process. In fact, automation is fundamentally different from manual testing: there are completely different issues and opportunities. And, even the best automation will never completely replace manual testing, because automation is about predictability and users are inherently unpredictable. So, use automation to verify what you expect, and use manual testing for what you don't.

So, your chances of success with automation will improve if you understand the fundamentals of test automation.

Test process must be well-defined

A key consideration is that you cannot automate a process that is not already well-defined. A fully manual process may not have the formality or documentation necessary to support a well-designed automation library. However, defining a complete test process is outside the scope of this handbook; entire books have been written about software testing. For our purposes, we will assume that you know what needs to be tested.

Testware is software

But even when the test process is reasonably well-defined, automation is still a challenge. The purpose of this handbook is to bridge the gap between what should be tested and how it should be automated. This begins by laying out certain fundamental principles that apply which must be understood before success is possible. All of these principles can be summarized in one basic premise: ***testware is software!***

Test automation is two different

As odd as it sounds, test automation is really two different things. There is testing, which is one discipline, and automation, which is another. Automating software testing is no different than automating

disciplines accounting or any other business function: in each case, a computer is being instructed to perform a task previously performed manually. Whether these instructions are stored in something called a script or a program, they both have all of the characteristics of source code.

Test	Automation
Application expertise	Development expertise
What to test	How to automate
Test Cases	Test scripts

The fact that testware is software is the single most important concept to grasp! Once this premise is understood, others follow.

Maintainability

Just as application software must be designed in order to be maintainable over its useful life, so must your automated tests.

Applications are maintained continuously One reason maintainability is so important is that without it you cannot accumulate tests. On average, 25% of an application is rewritten each year; if the tests associated with the modified portions cannot be changed with a reasonable amount of effort, then they will be obsolete. Therefore, instead of gradually improving your test coverage over time by accumulating more and more test cases, you will be discarding and recreating tests instead. Since each new version of the application most likely has increasing functionality, you will be lucky to stay even!

Changes must be known in advance It is also important to know where and how to make changes to the test library in advance. Watching tests execute in hopes of finding application changes in the form of errors is not only extremely inefficient, it brings the validity of test results and metrics into question. A failed test may in fact be a correct result! If a person must watch the test to determine the results, then the test is not truly

automated.

In most cases, the application source code will be managed by a source control or configuration management system. These systems maintain detailed change logs that document areas of change to the source code. If you can't get information directly from development about changes to the application, ask to be copied on the change log. This will at least give you an early warning that changes are coming your way and which modules are affected.

Cross-reference tests to the application

Identifying needed changes is accomplished by cross-referencing testware components to the application under test, using consistent naming standards and conventions. For example, by using a consistent name for the same window throughout the test library, when it changes each test case and test script which refers to it can be easily located and evaluated for potential modifications. These names and their usage is is described more fully in the section on the Application Map.

Design to avoid regression

Maintainability is achieved not only by assuring that changes can be easily identified and made, but also that they do not have an unexpected impact on other areas. Unexpected impact can occur as a consequence of poor test design or implementation. For example, a test script that selects an item from a list box based on its relative position in the list is subject to failing if the order or number of items in the list changes. In this case, a maintainable test script would be designed to enter the selected item or select it based on its text value. This type of capability may be limited by your test tool; if you are evaluating tools, look for commands that use object-based commands ("select list box item XYZ") instead of coordinate-based events (click window @ 451,687).

Maintainability can be designed into your test cases and scripts by adopting and adhering to an overall test framework, discussed in the next section.

Optimization

When designing your tests, remember that more is not always better. The more tests you have, the more time it will take to develop, execute and maintain them. Optimization is important to be sure you have enough tests to do the job without having too many to manage.

One test, one requirement

Well-designed tests should not roam across the entire application, accessing a wide variety of areas and functions. Ideally, each test should be designed to map to a specific business or design requirement, or to a previously reported defect. This allows tests to be executed selectively when needed; for example, to confirm whether a defect has been corrected.

Having no requirements is no excuse

If you don't have formally defined requirements, derive them from the tests you are going to perform instead of the other way around, but don't just ignore them altogether. Examine your tests and decide what feature or function this test verifies, then state this as a requirement. This is important because you must know what test cases are affected if an application requirement changes; it is simply not practical to review every test case to see whether it remains valid.

Understanding test results

Another reason to specify as precisely as possible what each test case covers is that, if the test case fails, it reduces the level of diagnostics required to understand the error. A lengthy, involved test case that covers multiple features or functions may fail for any number of reasons; the time it takes to analyze a failure is directly related to the complexity of the test case itself. A crisp tie-in between requirements and test cases will quickly indicate the type and severity of the failure.

Requirements measure readiness

Once you have them, requirements can be assigned priorities and used to measure readiness for release. Having requirements tied to tests also reduces confusion about which requirements have been satisfied or failed based on the results of the test, thus simplifying the test and error log reports. Unless you know what requirements have been proven, you don't really know whether the application is suitable for release.

A requirements matrix is a handy way of keeping track of which requirements have an associated test. A requirement that has too many tests may be too broadly defined, and should be broken down into separate instances, or it may simply have more tests than are needed to get the job done. Conversely, a test that is associated with too many requirements may be too complex and should be broken down into smaller, separate tests that are more targeted to specific requirements.

There are tools available that will generate test cases based on your requirements. There are two primary approaches: one that is based on addressing all possible combinations, and one that is based on addressing the minimum possible combinations. Using the former method, requirements are easier to define because interdependencies are not as critical, but the number of tests generated is greater. The latter method produces fewer tests, but requires a more sophisticated means of defining requirements so that relationships among them are stated with the mathematical precision needed to optimize the number of tests.

Independence

Independence refers to the degree to which each test case stands alone. That is, does the success or failure of one test case depend on another, and if so what is the impact of the sequence of execution? This is an issue because it may be necessary or desirable to execute less than all of the test cases within a given execution cycle; if dependencies exist, then planning the order of execution becomes more complex.

Independent data

Independence is most easily accomplished if each test case verifies at least one feature or function by itself and without reference to other tests. This can be a problem where the state of the data is key to the test. For example, a test case that exercises the delete capability of a record in a file should not depend on a previous test case that creates the record; otherwise, if the previous test is not executed, or fails to execute properly, then the later test will also fail because the record will not be available for deletion. In this case, either the beginning state of the database should contain the necessary record, or the test that deletes the record should first add it.

Independent context

Independence is also needed where application context is concerned. For example, one test is expected to commence at a particular location, but it relies on a previous test to navigate through the application to that point.

Again, if the first test is not successfully executed, the second test could fail for the wrong reason. Your test framework should give consideration to selecting common entry and exit points to areas of the application. and assuring that related tests begin and end at one of them.

Result independence

It is also risky for one test case to depend on the successful result of another. For example, a test case that does not expect an error message should provide assurance that, in fact, no message was issued. If one is found, steps should be added to clear the message. Otherwise, the next test case may expect the application to be ready for input when in fact it is in an error status.

If proper attention is paid to independence, the test execution cycle will be greatly simplified. In those cases where total independence is not possible or desirable, then be certain that the dependencies are well documented; the sequence, for example, might be incorporated into the naming conventions for test cases (ADD RECORD 01,ADD RECORD 02, etc.).

Modularity

Modularity in this context refers to test scripts, whereas independence refers to test cases. Given that your test library will include a number of scripts that together make up an automated test environment, modularity means scripts that can be efficiently assembled to produce a unified system without redundancy or omission.

Tie script design to application design

Ideally, the test scripts should be comprised of modules that correspond to the structure of the application itself, so that when a change is made to the application, script changes are as localized as possible. Depending on the automation approach selected, this may require separate scripts for each window, for example, or for each type of method of interacting with a control.

But modularity should not be taken to an extreme: scripts should not be broken down so minutely that they lose all individual meaning. This will raise the same issues that lengthy, convoluted scripts do: where should changes be made?

Identify common scripts

Modularity also means that common functions needed by all tests should not be duplicated within each individual script; instead, they should be shared as part of the overall test environment. Suggested common routines are described further in the Test Framework chapter.

Context

As described earlier, context refers to the state of the application during test playback. Because an automated test is executing at the same time the application is, it is critical that they remain synchronized. Synchronization takes two forms: one, assuring that the application is in fact located at the point where the test expects to be, and two, assuring the test does not run ahead of the application while it is waiting or processing. We will cover the second type in the next section, *Synchronization.*

Context controls results

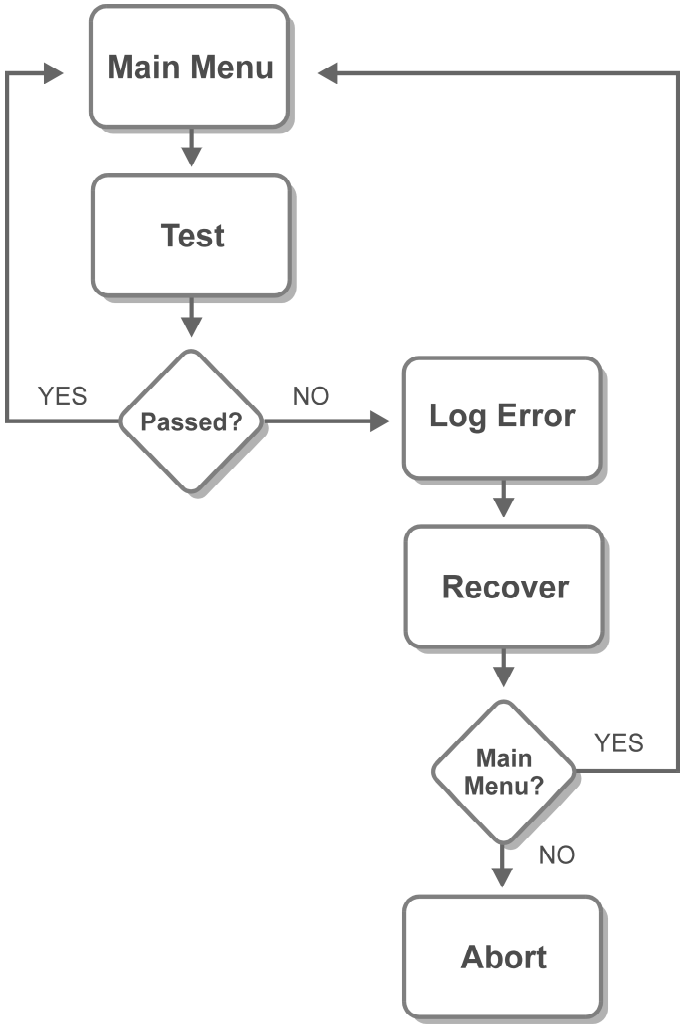
Because tests are performing inputs and verifying outputs, it is imperative that the inputs be applied at the proper location in the application, and that the outputs appear where expected. Otherwise, the test will report an incorrect result. Also, when multiple tests run one after the other, the result from one test can affect the next. If one test begins at the main menu and ends at a sub-menu, the following test must either expect to begin at the sub-menu or risk failure. Similarly, if a test which expects to complete at the main menu instead fails and aborts within a window, the next test will most likely begin out of context.

The Main menu approach

The simplest solution to beginning and ending context is to design all tests to begin and end at the same point in the application. This point must be one from which any area of the application can be accessed. In most cases, this will be the main menu or SIGNON area. By designing every test so that it commences at this point and ends there, tests can be executed in any order without considering context.

Enabling error recovery

Adopting a standard starting and ending context also simplifies recovery from unexpected results. A test which fails can, after logging its error, call a common recovery function to return context to the proper location so that the next test can be executed. Granted, some applications are so complex that a single point of context may make each individual test too long; in these cases, you may adopt several, such as sub-menus or other intermediate points. But be aware that your recovery function will become more complex, as it must have sufficient logic to know which context is appropriate. Designing test suites, or combinations of tests, will also be more complex as consideration must be given to grouping tests which share common contexts.



The key to context is to remember that your automated tests do not have the advantage that you have as a manual tester: they cannot make judgment calls about what to do next. Without consistency or logic to guide them, automated tests are susceptible to the slightest aberration. By proper test design, you can minimize the impact of one failed test on others, and simplify the considerations when combining tests into suites and cycles for execution.

Synchronization

Synchronization between the test and the application requires that they execute at the same rate. Because different conditions may exist at the time of playback than existed when the test was created, precise timing coincidence may not be possible. For example, if heavier system traffic increases processing time, the application may respond more slowly than it did previously. If the test does not have a means for compensating for fluctuating application speed, it may fail a test if the result does not appear in the time frame expected, or it may issue input when the application is not ready to receive it.

Synchronization is complicated when there are multiple platforms involved. Methods for synchronizing with a local application are different from those for synchronizing with a remote host or network server. But in any case, synchronization can affect the result of your automated tests and must be accounted for.

Global indicators Some test tools compensate for local synchronization by waiting for the application to cease processing. In Windows applications, for example, this may take the form of waiting while the hourglass cursor is being displayed. In other cases, this may require that the tool check to see that all application activity has ceased. Unfortunately, neither method is infallible. Not all applications use the hourglass cursor consistently, and some conduct constant polling activities which never indicate a steady state. Verify your tool's synchronization ability against a subset of your application under varying circumstances before developing large volumes of tests that may later require rework.

Local indicators Other tools automatically insert wait states between windows or even controls, causing the test script to suspend playback until the proper window or control is displayed. This method is more reliable, as it does not rely on global behavior that may not be consistent.

However, this approach also requires that some form of timeout processing be available; otherwise, a failed response may cause playback to suspend indefinitely.

Remote indicators When a remote host or network server is involved, there is yet another dimension of synchronization. For example, the local application may send a data request to the host; while it is waiting, the application is not "busy", thus risking the indication that it has completed its response or is ready for input. In this case, the tool may provide for protocol-specific drivers, such as IBM 3270 or 5250 emulation, which monitor the host status directly through HLLAPI (high level language application program interface). If your tool does not provide this, you may have to modify your scripts to detect application readiness through more specific means, such as waiting for data to appear.

Synchronization is one of the issues that is unique to automated testing. A person performing a manual test instinctively waits for the application to respond or become ready before proceeding ahead. With automated tests, you need techniques to make this decision so that they are consistent across a wide variety of situations.

Documentation

Documentation of the testware means that, in a crunch, the test library could be executed manually. This may take the form of extensive comments sprinkled throughout the test cases or scripts, or of narrative descriptions stored either within the tests or in separate documentation files. Based on the automation approach selected, the form and location of the documentation may vary.

Document for transferability

It may not be evident from reading an undocumented capture/playback script, for example, that a new window is expected to appear at a certain point; the script may simply indicate that a mouse click is performed at a certain location. Only the person who created the script will know what was expected; anyone else attempting to execute the script may not understand what went wrong if the window does not appear and subsequent actions are out of context. So, without adequate documentation, transferability from one tester to another is limited.

Mystery tests accumulate

Ironically, mystery tests tend to accumulate: if you don't know what a test script does or why, you will be reticent to delete it! This leads to large volumes of tests that aren't used, but nevertheless require storage, management and maintenance. Always provide enough documentation to tell what the test is expected to do.

More is better

Unlike some test library elements, the more documentation, the better! Assume as little knowledge as possible, and provide as much information as you can think of.

Document in context

The best documentation is inside the test itself, in the form of comments or description, so that it follows the test and explains it in context. Even during capture/playback recording, some test tools allow comments to be inserted. If this option is not available, then add documentation to test data files or even just on paper.

The Test Framework

The test framework is like an application architecture: it outlines the overall structure for the automated test environment, defines common functions, standard tests, provides templates for test structure, and spells out the ground rules for how tests are named, documented and managed, leading to a maintainable and transferable test library.

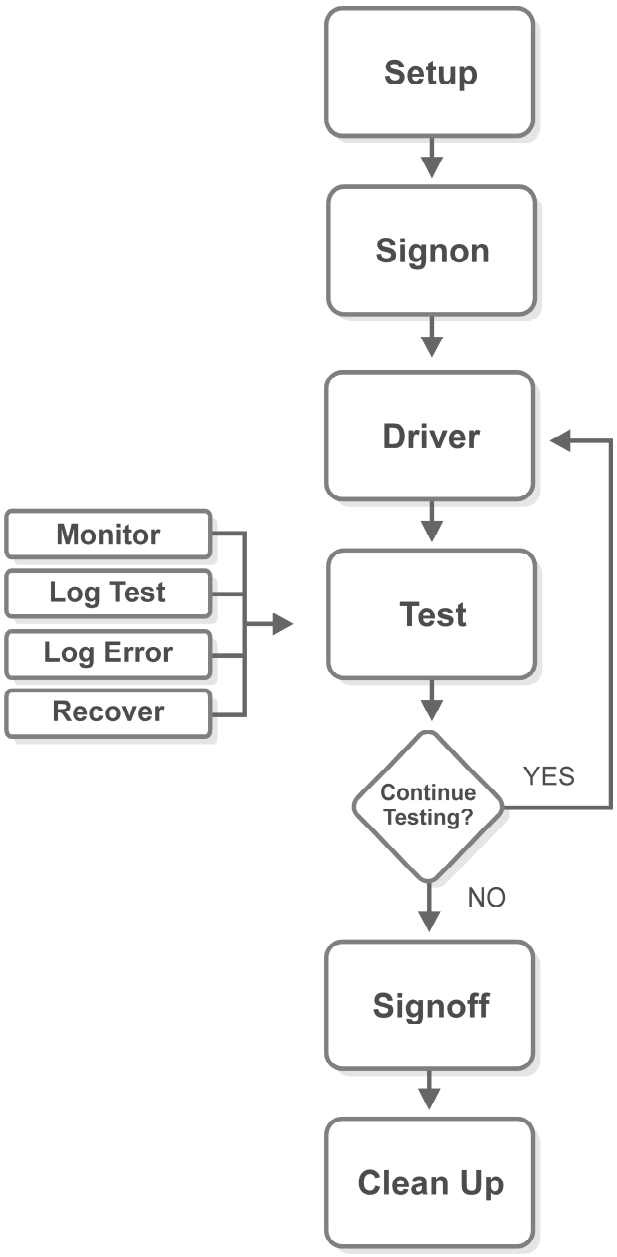
The need for a well-defined and designed test framework is especially great in testing. For an application, you can at least assume that the developer has a basic understanding of software design and development principles, but for automated tests the odds are high that the tester does not have a technical background and is not aware of, much less well-versed in, structured development techniques.

The test framework presented in this chapter can be applied to any of the automation approaches described in this Handbook. The only difference between one approach and another is in how the individual test cases and scripts are structured. By adopting a framework, you can enjoy the efficiency that comes from sharing common functions, and the effectiveness that standard tests and templates provide.

Common functions

Common functions are those routines which automate tasks that are shared throughout the entire test library. Some functions may be shared by all tests, such as routines which recover from unexpected errors, log results and other similar tasks. These functions should usually be structured as subroutines, which means that they can be called from any point and return to the next step after the one which called them.

Other types of common functions are utility scripts: for example, refreshing the database or populating it with a known set of records, deleting temporary work files, or otherwise managing the test environment. Clearly defining and sharing these routines will reduce and simplify testware development and maintenance. These scripts should be structured so that they can be executed stand-alone, or linked together sequentially as part of an integrated test cycle.



Following are suggested common functions:

SETUP The SETUP function prepares the test environment for execution. It is executed at the beginning of each test cycle in order to verify that the proper configuration is present, the correct application version is installed, all necessary files are available, and all temporary or work files are deleted. It may also perform housekeeping tasks, such as making backups of permanent files so that later recovery is possible in the event of a failure that corrupts the environment. If necessary, it may also intialize data values, or even invoke sorts that improve database performance. Basically, SETUP means what it says: it performs the setup of the test environment. It should be designed to start and end at a known point, such as the program manager or the command prompt.

SIGNON The SIGNON function loads the application and assures that it is available for execution. It may provide for the prompting of the user ID and password necessary to access the application from the point at which the SETUP routine ends, then operate the application to another known point, such as the main menu area. It may also be used to start the timer in order to measure the entire duration of the test cycle. SIGNON should be executed after SETUP at the beginning of each test execution cycle, but it may also be called as part of a recovery sequence in the event a test failure requires that the application be terminated and restarted.

DRIVER The DRIVER function is one which calls a series of tests together as a suite or cycle. Some test tools provide this capability, but if yours does not you should plan to develop this function. Ideally, this function relies upon a data file or other means of storing the list of tests to be executed and their sequence; if not, there may be a separately developed and named DRIVER function for each test suite.

Remember if you are using a DRIVER to design each individual test to return to the DRIVER function when it ends, so that the next test can be called.

MONITOR The MONITOR function may be called after each transaction is submitted, or at other regular intervals, in order to check the status of the system. For host-based applications, this may be the status line; for networked applications, this may be the area in which system messages are broadcast. The purpose of this script is to check for asynchronous messages or events - those which are not expected but which may nevertheless occur. Because result comparison is usually based on what is expected, some manner of checking for the unexpected is necessary; otherwise, host or network failures or warnings may go undetected.

RECOVER The RECOVER function is most often called by the LOGERROR function, but in fact may be called by any script that loses context during playback. Instead of simply aborting test execution altogether, or blindly continuing to execute and generating even more errors, a routine like RECOVER can be used to attempt to restore context to a known location so that subsequent tests in the suite can be executed. This may include navigating through the application to reach a predefined point, such as the main menu, or terminating the application and restarting it. In the latter event, the RECOVER routine may also call the SIGNON script to reload the application.

For instances where the steps to recover are not standard throughout the application and human intervention is needed, it may be helpful to insert an audible alarm of some type, or to halt playback and display a message, that alerts the test operator that assistance is needed.

If correctly designed, this intervention can be provided without interfering with continuation of the test cycle. For example, the displayed message might instruct the operator to suspend playback,

return context to a particular window, then resume.

SIGNOFF The SIGNOFF routine is the sibling script to SIGNON. It terminates the application and returns the system to a known point, such as the program manager or command prompt. It should be used at the end of the last test suite, before other shared routines such as CLEANUP are executed. SIGNOFF may also stop the test cycle timer, thus providing a measure of how long the entire cycle required for execution.

LOGTEST The LOGTEST function is called at the end of each test case or script in order to log the results for the component just executed. This routine may report not only pass/fail status, but also elapsed time and other measurements. Results may be written to a text file, to the clipboard, or any other medium that can be later used to derive reports. A test logging function may already be integrated into your test tool; if not, develop a function to provide it.

LOGERROR The LOGERROR function is called by any test that fails. Its primary purpose is to collect as much information as possible about the state of the application at the time of the error, such as the actual context versus expected; more sophisticated versions may invoke stack or memory dumps for later diagnostics. A secondary purpose may be to call the RECOVER function, so that context can be restored for the next test.

CLEANUP The CLEANUP function is the sibling script to SETUP. It begins at a selected point, such as the program manager or command prompt, and it does what its name implies: it cleans up the test environment.

This may include deleting temporary or work files, making backups of result files, and otherwise assuring that the test environment does not accumulate any detritus left behind by the test execution process. A properly designed CLEANUP routine will keep your test environment

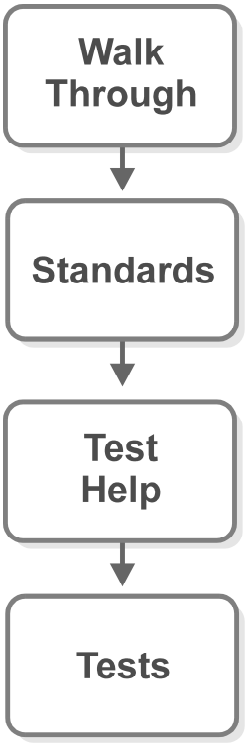
organized and efficient.

By designing your test framework to include common functions, you can prevent the redundancy that arises when each individual tester attempts to address the same issues. You can also promote the consistency and structure that provides maintainability.

Standard tests

The concept of common functions can be extended even further when you consider standard tests. A common function is shared among tests; a standard test is shared among test suites, cycles or even applications. Certain types of standard tests might also be shared with the development or production support groups.

For example, each test library might include a standard test that performs a complete walkthrough of the application, down each menu branch and through each window and control. Although this type of test could be shared by all testers, it need not be developed by all of them. These tests should be structured just as any other test, so that they can be executed stand-alone or as part of a complete test suite.



WALKTHRU As described above, the WALKTHRU standard test navigates through the application, assuring that each menu item, window and control is present and in the expected default state. It is useful to establish that a working copy of the application has been installed and that there are no major obstacles to executing functional tests. Each test execution cycle can take advantage of this standard test in order to assure that fatal operational errors are uncovered before time and effort are expended with more detailed tests. This type of test could be executed by the development group after the system build, before the application is delivered for testing, or by the production support group after the application has been promoted into the production environment.

STANDARDS The STANDARDS test is one which verifies that application design standards are met for a given component. While the WALKTHRU test assure that every menu item, window and control is present and accounted for, the STANDARDS test verifies that previously agreed upon standards have been satisfied.

For example, it may be a design criteria that every window have a maximize and minimize button, vertical and horizontal scroll bars, and both an OK and CANCEL push button. It might also verify standard key behaviors, such as using the ESC key to cancel. This type of test could be executed by developers against each individual window as part of the unit test phase.

TESTHELP The TESTHELP standard test, like the STANDARDS test, is one which might be useful on every screen or window. It assures that the help function is available at all points in the application. Depending on whether the help function is context-sensitive or not, this test may require additional logic to verify the correct response. If your application has similar functionality that is common to multiple areas of the application, you may consider developing standard tests specific to those functions.

It is well worth the time to think through the particulars of testing your application in order to identify those tests that are widely applicable. By developing as many common and standard functions and tests as possible, you can streamline and standardize your test library development.

Test templates

A test template provides the structure for the development of individual tests. It may be used to speed development by allowing a single format to be quickly copied and filled in, saving time for new tests and promoting consistency. Although naming conventions for tests and their contents are important, and are more fully described in the next section on the Application Map, it is also important that each individual test follow a common structure so that it can be easily linked into the test framework.

For example, tests which are expected to be called as subroutines and shared with other tests must be developed in order to permit a return to the calling test; likewise, tests which are to be executed from a driver or other control mechanism must be capable of returning control when they are completed. The precise means of accomplishing this will vary with each automation approach, and is discussed in the related section for each approach.

However, some elements of structure are common to all approaches.

HEADER Just as a document has a Header section that describes important information about its contents, a test case or script should contain an area that stores key data about the test. Depending on the tool and approach selected, this information may be found within the test script, the data file, or on paper. The Header is designed to provide later testers with enough information about the test to execute or modify it.

NEXT The NEXT area is used for those tests that rely on external files, and it indicates the point at which the next record is read from the file. It is used as a branch point within the test after processing is complete for a single record.

END At the end of each test there should be an ending area, which is the last section to be executed before the test terminates. For tests that read external files, this may be the branch point for an end-of-file condition. In most cases, this area would provide for the test to be logged, such as by calling the LOGTEST routine. For subroutine scripts or tests that are shared by other routines, this area would include the command(s) necessary to return control to the calling script, such as RESUME. For scripts that are executed stand-alone, this might simply say STOP.

Test Case Header

Application: General Ledger 5.1.1 Test Case ID: 112-0000
Date Created: 01/01/2X By: Teresa Tester
Last Updated: 01/11/2X By: Lucinda Librarian

Test Description:

This test case deletes an existing chart of accounts record that has a zero balance. The script DELETE_ACCTS is used to apply the test case.

Inputs:

This test case begins at the Account Number edit control; the account number 112 and sub-account number 0000 are entered, then the OK button is clicked.

Outputs:

The above referenced account is retrieved and displayed. Click DELETE button. The message "Account Deleted" appears. All fields are cleared and focus returns to Account Number field.

Special requirements:

The security level for the initial SIGNON to the general ledger system must permit additions and deletions.

Dependencies:

Test Case 112-0000 should be executed by the ADD_ACCTS script first so that the record will exist for deletion. Otherwise, the completed chart of accounts file ALL_ACCTS should be loaded into the database before execution.

Application Map

Similar to a data dictionary for an application, the Application Map names and describes the elements that comprise the application and provides the terminology used to tie the tests to the application. Depending on the type of automation approach you adopt, these elements may include the components that comprise the user interface of the application, such as windows, dialog boxes and data elements or controls.

Test Vocabulary

Think of your Application Map as defining the "vocabulary" of your automated tests. This vocabulary spells out what words can be used in the test library to refer to the application and what they mean. Assuring that everyone who contributes to the test process uses the same terminology will not only simplify test development, it will assure that all of the tests can be combined into a central test library without conflict or confusion.

Naming Conventions

In order to develop a consistent vocabulary, naming conventions are needed. A naming convention simply defines the rules by which names are assigned to elements of the application. The length and format of the names may be constrained by the operating system and/or test automation tool. In some cases, application elements will be identified as variables in the test script; therefore, the means by which variables are named by the tool may affect your naming conventions. Also, test scripts will be stored as individual files whose names must conform to the operating system's conventions for file names.

Cross-reference names to application

Because your tests must ultimately be executed against the application, and the application will inevitably change over time, it is crucial that your tests are cross-referenced to the application elements they impact. By using consistent names for windows, fields and other objects, a change in the application can be quickly cross-referenced to the potentially affected test cases through a search for the name(s) of the modified elements.

Following is an excerpt from the Application Map for the sample general ledger system; the Data-Driven approach is assumed.

Object Names

Conventions:

Sub-menus are named within the higher level menu; windows are named within their parent menus. Controls are named within their parent window. Data files are named by the script file that applies them; script files are named by the parent window.

Name	Description	Object Type	Parent
CHT_ACCTS	Chart of accounts	Window	CHT_MENU
CHT_ACCTS	Text file	.TXT	CHT_ACCTS
CHT_ACCTS	Script file	.SLF	CHT_ACCTS
ACCTNO	Account number	Edit control	CHT_ACCTS
SUBACCT	Sub account number	Edit control	CHT_ACCTS
ACCTDESC	Account description	Edit control	CHT_ACCTS
STMTTYPE	Statement type	Radio button	CHT_ACCTS
ACCTTYPE	Account type	List box	CHT_ACCTS
HEADER	Header	Check box	CHT_ACCTS
MESSAGE	Message	Information box	CHT_ACCTS
OK	Accept record	Push button	CHT_ACCTS
CANCEL	Cancel record	Push button	CHT_ACCTS

Test Library Management

Just as an application source library will get out of control if changes and different versions are not managed, so will the test library eventually become useless if it is not managed properly. Regardless of how many testers are involved, there must be a central repository of all test scripts, data and related information that can be effectively managed over time and turnover. Individual, uncoordinated test libraries have no long term value to the organization; they are only as good - and around only as long - as the person who created them.

Test library management includes change control, to assure that changes are made only by authorized persons, are documented, and are not made concurrently to different copies so that overwriting occurs; version control, to assure that tests for different versions of the same application are kept segregated; and configuration management to account for any changes to the test environment.

Change Control

Change control refers to the orderly process of introducing change to the test library.

Documentation is key

Changes may take the form of new tests being added or existing tests being modified or deleted. It is important to not only know that a change was made, but who made it, when and why. Documentation of the nature of the change to an existing module should ideally include a delta file, which contains the differences between the old module and the new one. At a minimum, an explanation should be provided of what was changed and where.

Change log The test librarian should manage the change control process, keeping either a written or electronic log of all changes to the test library. This change log should list each module affected by the change, the nature of the change, the person responsible, the date and time. Regular backups of the test library are critical, so that unintended or erroneous changes can be backed out if needed.

Test your tests The librarian should also take steps to assure that the test being added to the library has been itself tested; that is, it should have been executed successfully at least once before being introduced into the permanent library.

Synchronize with source control There should also be some level of correspondence between the change log for the application source and the test library. Since changes to the application will often require changes to the affected tests, the test librarian may take advantage of the application change log to monitor the integrity of the test library. In fact, it is ideal to use the same source control system whenever possible. If the change to a test reflects a new capability in a different application version, then the new test should be checked into a different version of the test library instead of overwriting the test for the prior version. See Version Control, following, for more information.

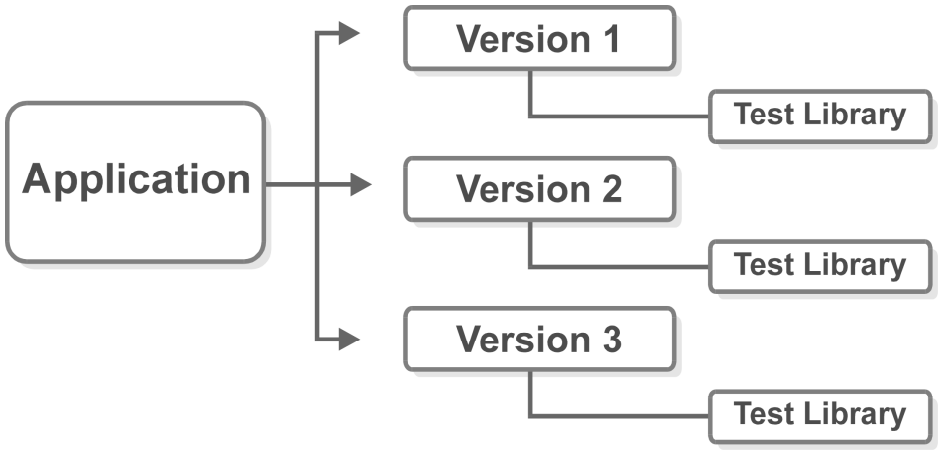
Version Control

Just as the application source code must be kept aligned by versions, so must the test library.

Multiple At any given time, more than one version of the application may

application versions require testing; for example, fixes may be added to the version in the field, while enhancements are being added to the next version planned for release.

Multiple test library versions Proper version control of the test library allows a test execution cycle to be performed against the corresponding version of the application without confusing changes made to tests for application modifications in subsequent versions. This requires that more than one version of the test library be maintained at a time.



Configuration Management

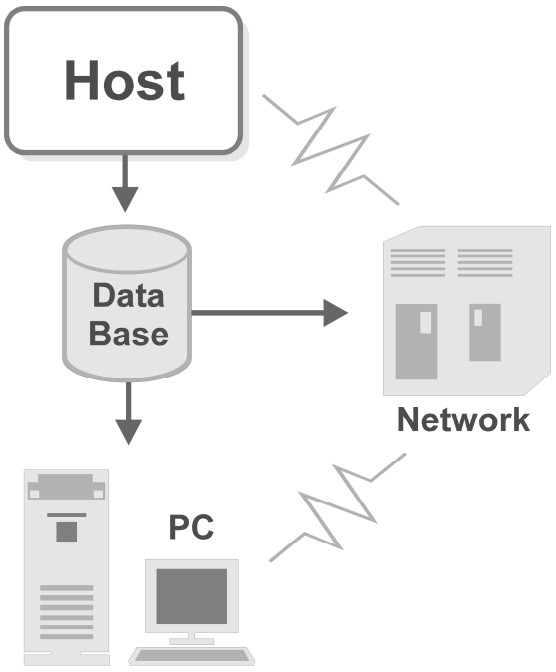
A thorough test exercises more than just the application itself: it ultimately tests the entire environment, including all of the supporting hardware and surrounding software.

Multiple layers affect the test In today's complex, layered environments, there may be eight or more different variables in the environment: the workstation operating system and hardware configuration, the network protocol and hardware connection, the host or server communications protocol, the server's hardware configuration and operating system, and the state of the database. It is risky to test an application in one

environment and deliver it in another, since all of these variables will impact the functionality of the system.

Test integrity requires configuration management

This means that configuration management for the test environment is crucial to test integrity. It is not enough to know what version of the software was tested: you must know what version and/or configuration of every other variable was tested as well. Granted, you may not always be able to duplicate the production environment in its entirety, but if you at least know what the differences are, you know where to look if a failure occurs.



Selecting a Test Automation Approach

There are as many ways to approach test automation as there are testers, test tools and applications. This is one reason why it is important to develop an overall approach for your automated test environment: otherwise, each tester will adopt his or her own, leading to a fragmented test library and duplication of effort.

For our purposes, we will refer to three major approaches as described below. These approaches are not exclusive of each other, or of other approaches; rather, they are intended to describe options that may be mixed and matched, based on the problem at hand. Indeed, a single test library may contain tests designed according to each approach, depending on the particular type of test being automated.

But before you can get started, you have to know where you are starting from. The following assessment is designed to help you evaluate where you stand in terms of your application, test team and test process. Based on the results, you will be able to select the automation approach that is right for your needs. Start by answering these questions:

What phase of development is the application in?

_____ Planning
_____ Analysis
_____ Design
_____ Code
_____ Test
_____ Maintenance

What is the skill set of the test team?

_____ Primarly technical

_____ Some technical, some non-technical

_____ Primarily non-technical

How well documented is the test process?

_____ Well-documented

_____ Somewhat documented

_____ Not documented

How stable is the application?

_____ Stable

_____ Somewhat stable

_____ Unstable

Based on your answers to these questions, you should select an automation approach that meets your needs. Each of the approaches is described in more detail below.

Approach	Profile
Capture/Playback	Application already in test phase or maintenance Primarily non-technical test team Somewhat or not documented test process Stable application
Data-Driven	Application in code or early test phase Some technical, some non-technical test team Well or somewhat documented test process Stable or somewhat stable application
Table-Driven	Application in planning, analysis or design Some technical, most non- technical test team Well documented test process Unstable or stable application

These profiles are not hard and fast, but they should indicate the type of approach you should consider. Remember that you have to start from where you are now, regardless of where you want to end up. With a little prior planning, it is usually possible to migrate from one method to another as time and expertise permits.

Capture/Playback

The capture/playback approach means that tests are performed manually while the inputs and outputs are captured in the background. During subsequent automated playback, the script repeats the same sequence of actions to apply the inputs and compare the actual responses to the captured results; differences are reported as errors. Capture/playback is available from almost all automated test tools, although it may be implemented differently.

Following is an excerpt from an example capture/playback script:

```
Select menu item "Chart of Accounts>>Enter Accounts"
Type "100000"
Press Tab
Type "Current Assets"
Press Tab
Select Radio button "Balance Sheet"
Check box "Header" on
Select list box item "Asset"
Push button "Accept"
Verify text @ 562,167 "Account Added"
```

Notice that the inputs - selections from menus, radio buttons, list boxes, check boxes, and push buttons, as well as text and keystrokes - are stored in the script. In this particular case, the output - the expected message - is explicit in the script; this may or may not be true with all tools - some simply capture all application responses automatically, instead of allowing or requiring that they be explicitly declared. See Comparison Considerations below for more information.

Structure

In order to allow capture/playback script recording to be distributed among multiple testers, a common structure should be adopted.

One script, one requirement

The ideal structure is to have one script per requirement, although multiple instances of the requirement - i.e. test cases - might be grouped together. This also allows the requirements to be distributed among multiple testers, and the name of the script can be used as a cross-reference to the requirement and clearly indicate the content and purpose of each script.

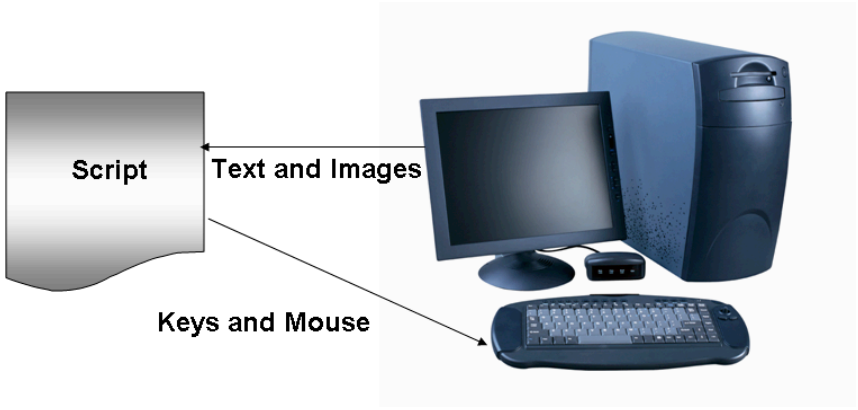
Associate scripts by application areas

These scripts can be packaged together into test suites that are related by common characteristics, such as beginning and ending context and data requirements, and/or by the area of the application they exercise. This makes it easier for a single tester to focus on certain areas of the system, and simplifies later maintenance when changes are needed.

Callable scripts

Depending on the capabilities provided by your tool, you may need to build in the capability of tying scripts together into suites and/or test cycles.

If your tool does not have a built-in mechanism, you should consider making each script callable from another, so that when it completes it returns processing to the next instruction in the calling script. A master or driver script can then be created which contains a series of calls to the individual scripts for execution.



Advantages

Capture/playback is one of the earliest and most common automated test approaches and offers several advantages over other methods.

Little training or setup time The main advantage of this approach is that it requires the least training and setup time. The learning curve is relatively short, even for non technical test operators.

Develop tests on the fly Tests need not be developed in advance, as they can be defined on the fly by the test operator. This allows experienced users to contribute to the test process on an ad hoc basis.

Audit trail This approach also provides an excellent audit trail for ad hoc or usability testing; in the event an error occurs, the precise steps that created it are captured for later diagnosis or reproduction.

Disadvantages

There are, however, several disadvantages of capture/playback, many of which have led to more advanced and sophisticated test tools, such as scripting languages.

Requires manual capture

Except for reproducing errors, this approach offers very little leverage in the short term; since the tests must be performed manually in order to be captured, there is no real leverage or time savings. In the example shown, the entire sequence of steps must repeated for each account to be added, updated or deleted.

Application must be stable

Also, because the application must already exist and be stable enough for manual testing, there is little opportunity for early detection of errors; any test that uncovers an error will most likely have to be recaptured after the fix in order to preserve the correct result.

Redunancy and omission

Unless an overall strategy exists for how the functions to be tested will be distributed across the test team, the probability of redundancy and/or omission is high: each individual tester will decide what to test, resulting in some areas being repeated and others ignored. Assuring efficient coverage means you must plan for traceability of the test scripts to functions of the application so you will know what has been tested and what hasn't.

Tests must be combined

It is also necessary to give overall consideration to what will happen when the tests are combined; this means you must consider naming conventions and script development standards to avoid the risk of overwriting tests or the complications of trying to execute them as a set.

Lack of maintainability

Although subsequent replay of the tests may offer time savings for future releases, this benefit is greatly curtailed by the lack of maintainability of the test scripts. Because the inputs and outputs are hard-coded into the scripts, relatively minor changes to the application may invalidate large groups of test scripts. For example, changing the number or sequence of controls in a window will impact any test script that traverses it, so a window which has one hundred test transactions executed against it would require one hundred or more modifications for a single change.

Short useful script life

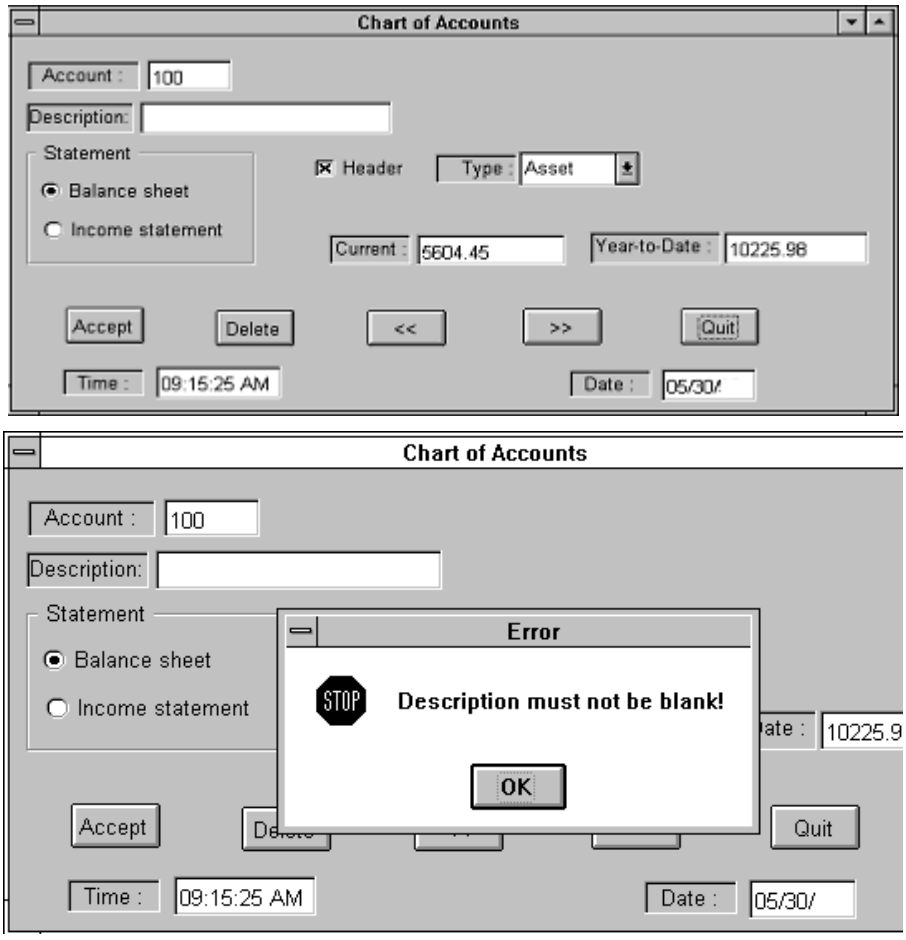
This issue is exacerbated by the fact that the test developer will probably require additional training in the test tool in order to be able to locate and implement necessary modifications. Although it may not be necessary to know the script language to capture a test, it is crucial to understand the language when making changes. As a result, the reality is that it is easier to discard and recapture scripts, which leads to a short useful life and a lack of cumulative test coverage.

No logic means more tests fail

Note also that there is no logic in the script to be sure that the expected window is in fact displayed, or that the cursor or mouse is correctly positioned before input occurs - all the decisions about what to do next are made by the operator at the time of capture and are not explicit in the script. This lack of any decision-making logic in the scripts means that any failure, regardless of its true severity, may abort the test execution session and/or invalidate all subsequent test results. If the application does not behave precisely as it did when the test was captured, the odds are high that all following tests will fail because of an improper context, resulting in many duplicate or false failures which require time and effort to review.

Comparison Considerations

The implied assumption in capture/playback is that the application behavior captured at the time the test is created represents the expected, or correct, result. As simple as this sounds, there are issues that must be addressed before this is effective.



Identify results to verify

For fixed screen format character-based applications, the comparison criteria often includes the entire screen by default, with the opportunity to exclude volatile areas such as time and date. In the case of windowed applications or those without a fixed screen format, it may become necessary to rely only on selected areas. In either event, it is critical to evaluate what areas of the display are pertinent to the verification and which are not.

Use text instead of bitmaps when possible

For graphical applications, full screen or window bitmap comparisons are usually impractical. Simply capturing, storing and comparing the huge amount of information present in a graphical image is a tremendously resource intensive task. Also, merely moving the test from one computer to another may invalidate the comparison altogether, since different monitor resolutions return different values for bitmaps. Further, the very nature of graphical applications is to be fluid instead of fixed, which means that the same inputs may not result in precisely the same outputs. For example, the placement of a window is often determined by the window manager and not by the application. Therefore, it is usually more accurate to use text to define expected results instead of using images.

Verify by inclusion instead of exclusion

If your tool permits it, define the test results by inclusion rather than exclusion. That is, define what you are looking for instead of what you are not looking at - such as everything except what is masked out. Explicit result verification is easier to understand and maintain - there is no guesswork about what the test is attempting to verify. Having said that, however, also be aware that minimally defined results may allow errors to go unnoticed: if, for example, system messages may be broadcast asynchronously, then you might miss an error message if you are not checking the system message area.

Of course your tool will control the types of comparison available to you and how it is defined, to some degree. Familiarize yourself with your options and adopt a consistent technique.

Data Considerations

Because capture/playback expects the same inputs to produce the same outputs, the state of the data is critical.

Static data The beginning state of the application database is essential to predictable results in any automated test method. Assure that your test cycle contains steps to prepare the database to a known state, either by refreshing it with a new copy or populating it with known records.

Dynamic data In some cases, the application will generate a data value dynamically that cannot be known in advance but must be used later during the test. For example, a unique transaction identifier may be assigned to each new record as it is entered, which must be input later in order to access the record. Because capture/playback hard codes the test data in the script at the time of capture, in this situation subsequent playback will not produce the same results.

Using variables In these cases, it may be necessary to implement variable capability for the dynamic field, so that the value can be retrieved during playback and saved for later reference. This will require at least one member of the test team to become familiar with how the test tool defines and manipulates variables in order to substitute them in place of the fixed values which were captured against the dynamic field.

If your test tool can store its scripts in a text format, you can use your favorite word processor to copy the script for a single transaction, then simply search and replace the data values for each iteration. That way, you can create new tests without having to perform them

manually!

Data-Driven

The difference between classic capture/playback and Data-Driven is that in the former case the inputs and outputs are fixed, while in the latter the inputs and outputs are variable. This is accomplished by performing the test manually, then replacing the captured inputs and expected outputs with variables whose corresponding values are stored in data files external to the script. The sequence of actions remain fixed and stored in the test script. Data-Driven is available from most test tools that employ a script language with variable data capability, but may not be possible with pure capture/playback tools.

The following page contains an example of the previous capture/playback script, modified to add an external file and replace the fixed values with variables. Comments have been added for documentation:

```
Select menu item "Chart of Accounts>>Enter Accounts"
Open file "CHTACCTS.TXT"            * Open test data file
Label "NEXT"                        * Branch point for next record
Read file "CHTACCTS.TXT"            * Read next record in file
End of file?                        * Check for end of file
   If yes, goto "END"               * If last record, end test
Type ACCTNO                         * Enter data for account #
Press Tab
Type ACCTDESC                       * Enter data for description
Press Tab
Select Radio button STMTTYPE        * Select radio button for statement
Is HEADER = "H"?                    * Is account a header?
  If yes, Check Box HEADER on * If so, check header box
Select list box item ACCTTYPE * Select list box item for type
Push button "Accept"
Verify text MESSAGE                 * Verify message text
  If no, Call LOGERROR              * If verify fails, log error
   Press Esc                        * Clear any error condition
CALL LOGTEST                        * Log test case results
Goto "NEXT"                         * Read next record
Label "END"                         * End of test
```

Example file contents:

Test Case	ACCT NO	SUB ACCT	ACCT DESC	STMT TYPE	ACCT TYPE	HEADER	MESSAGE
1000000	100	0000	Current Assets	Balance Sheet	Asset	H	Account Added
1001000	100	1000	Cash in Banks	Balance Sheet	Asset		Account Added

Structure

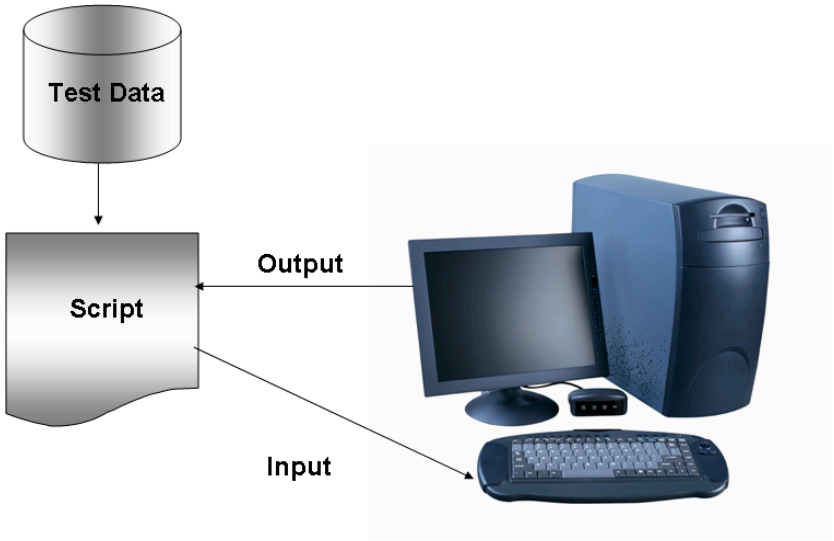
In order to permit test cases to be defined as data records to be processed by as external files to scripts, the application data elements associated with each process must be known. This should be provided by the Application Map. Also, the following structure should be followed:

One script, one process A Data-Driven script is tied to a single processing sequence but will support multiple test cases. A sequence of steps that enters or processes data may include many test cases relating to individual elements of the data or steps in the sequence. Select a sequence of steps that require a consistent set of data for each iteration, and name the script for the process or application window it addresses.

One record, one test case Each record in the test data file should relate to a single test case, and the test case identifier should be stored in the data record. This allows a single script to process multiple test cases while logging results for each. Notice in the example that the test results are logged for each record, instead of at the end of the script.

Data-intensive Implied in this approach is that the application is fairly data intensive; that is, the same steps tend to be repeated over and over with different data. In the general ledger example, the steps to enter one account are identical to those needed to enter hundreds.

Consistent behavior Also assumed is that the same steps are repeated without significant variance, so that different inputs do not have a major impact on the sequence of actions. For example, if the value of one field causes a completely different processing path to take effect, then the amount of logic required to process each test case increases exponentially.



Advantages

There are several advantages to Data-Driven over simple capture/playback.

Create test cases earlier Data-Driven allows test cases - the inputs and expected outputs - to be created in advance of the application. The software does not have to be stable enough to operate before test cases can be prepared as data files; only the actual script has to await the application.

Flexible test case creation Because they are stored as data, the sets of inputs and outputs can be entered through a spreadsheet, word processor, database or other familiar utility, then stored for later use by the test script. Familiarity with, or even use of, the test tool is not required for test cases.

Leverage Data-Driven provides leverage in the sense that a single test script can be used to apply many test cases, and test cases can be added later without modifications to the test script.

Notice that the example script could be used to enter one or one

thousand different accounts, while the capture/playback script enters only one. Cut and paste facilities of the selected utility can be used to rapidly "clone" and modify test cases, providing leverage.

Reduced maintenance This approach reduces required maintenance by not repeating the sequence of actions and logic to apply each test case; therefore, should the steps to enter an account change, they would have to be changed only one time, instead of once for each account.

Disadvantages

The main disadvantage of this approach is that it requires additional expertise in the test tool and in data file management.

Technical tool skills required In order to convert the script to process variable data, at least one of the testers must be proficient in the test tool and understand the concept of variable values, how to implement external data files, and programming logic such as if/then/else expressions and processing loops.

Data file management needed Similarly, the test case data will require someone with expertise in creating and managing the test files; large numbers of data elements in a test case may lead to long, unwieldy test case records and awkward file management. Depending on the utility used, this may require expertise in creating and manipulating spreadsheet macros, database forms or word processor templates, then exporting the data into a file compatible with the test tool.

Data Considerations

Because the test data is stored externally in files, there must be a consistent mechanism for creating and maintaining the test data.

One script, one file Generally, there will be a file for each script that contains records comprising the set of values needed for the entire script. Therefore, the content and layout of the file must be defined and organized for easy creation and maintenance. This may take the form of creating macros, templates or forms for spreadsheets, word processors or databases that lay out and describe the necessary fields; some level of editing may also be provided. Obviously, the more that can be done to expedite and simplify the data collection process, the easier it will be to add and change test cases.

Using multiple files It is of course possible to have more than one test data file per script. For example, there may be standalone data files that contain a list of the equivalence tests for specialized types of fields, such as dates. Instead of repeating these values for every script in which dates appear, a single file may be read from multiple scripts. This approach will require additional script logic to accomodate nested processing loops for more than one file per script iteration.

Dynamic data This approach may also require the same manipulation for dynamic variables that was described under Data Considerations for capture/playback, above.

Table-Driven

Table-Driven differs from Data-Driven in that the sequence of actions to apply and evaluate the inputs and outputs are also stored external to the script. This means that the test does not have to be performed manually at all. The inputs and expected outputs, as well as the sequence of actions, are created as data records; the test scripts are modular, reusable routines that are executed in a sequence based on the test data. The logic to process these records and respond to application results is embedded in these routines.

Another key differentiator is that these script routines are reusable across applications. They are completely generic in that they are based on the type of field or object and the action to be performed. The exact instance of the field or object is defined in the Application Map and is provided to the routine when the test executes.

Below is an excerpt of the test script routines and test data file that would process the same entry of the chart of accounts:

```
Open file TESTDATA                  * Open test data file
Label "NEXT"                        * Branch point for next test case
Read file @TESTDATA                 * Read next record in file
   End of file?                     * Check for end of file
       If yes, goto "END"           * If last record, end test
Does @WINDOW have focus?            * Does the window have focus?
   If no, Call LOGERROR             * If not, log error
Does @CONTROL have focus?           * Does the control have focus?
   If no, set focus to @CONTROL     * Try to set the focus
      Does @CONTROL have focus?     * Was set focus successful?
         If no, Call LOGERROR       * If not, log error
Call @METHOD                        * Call test script for method
Call LOGTEST                        * Log test results
Goto "NEXT"
Label "END"
```

SELECT_MENU script

```
Does menu item VALUE exist?* Does the menu item exist?
```

```
    If no, Call LOGERROR          * If not, log error
Is menu item VALUE enabled?* Is the menu item enabled?
    If no, Call LOGERROR          * If not, log error
Select menu item VALUE            * Select the menu item
Resume                            * Return to main script
```

Example file contents:

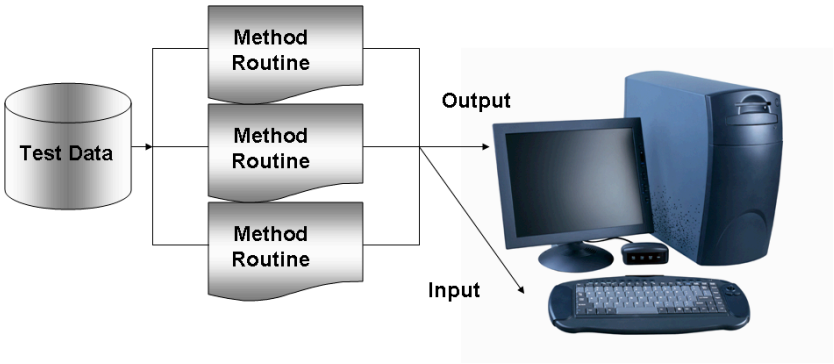
Test Case	Window	Object	Method	Value	On Pass	On Fail
Add Account	MAINMENU	CHART.MENU	Select	Chart of Accounts>>Enter Accounts	Continue	Abort
Add Account	Chart of Accounts	Account Number	Enter	100000	Continue	Continue
Add Account	Chart of Accounts	Account Description	Enter	Current Assets	Continue	Continue
Add Account	Chart of Accounts	Statement Type	Select	Balance Sheet	Continue	Continue
Add Account	Chart of Accounts	Header	Check	On	Continue	Continue
Add Account	Chart of Accounts	Account Type	Select	Assets	Continue	Continue
Add Account	Chart of Accounts	OK	Push		Continue	Continue
Add Account	Chart of Accounts	Message Box	Verify Text	Account Added	Continue	Continue

Structure

Like Data-Driven, this method requires that the names of the application data elements be known; however, it also requires that the type of object and the valid methods it supports also be defined and named, as well as the windows and menus. In the earlier stages of development, the object types and methods may not yet be known, but that should not prevent test cases from being developed. A simple reference to input or output can be later refined as the precise methods are implemented. For example, the input of the statement type can eventually be converted to a check box action. This information will support the following structure:

One file, multiple scripts A single test data file in Table-Driven script is processed by multiple scripts. In addition to common and standard scripts, there will be a master script that reads the test file and calls the related method scripts. Each object and method will have its own script that contains the commands and logic necessary to execute it.

Multiple records, one test case A single test case is comprised of multiple records, each containing a single step. The test case identifier should be stored in each data record to which it relates. This allows a single set of scripts to process multiple test cases. Notice in the example that the test results are logged for each step, instead of at the end of the test case.



Advantages

The main advantage to this approach is that it provides the maximum maintainability and flexibility.

Develop test cases, scripts earlier Test cases can be constructed much earlier in the development cycle, and can be developed as data files through utilities such as spreadsheets, word processors, databases, and so forth. The elements of the test cases, can be easily modified and extended as the application itself becomes more and more defined. The scripts that process the data can be created as soon as the objects

(screens, windows, controls) and methods have been defined.

Minimized maintenance

By constructing the test script library out of modular, reusable routines, maintenance is minimized. The script routines need not be modified unless a new type of object is added that supports different methods; adding new windows or controls simply requires additions to the tool's variable file or GUI map, where the objects are defined.

Portable architecture between applications

Another key advantage of Table-Driven is that the test library can be easily ported from one application to another. Since most applications are composed of the same basic components - screens, fields and keys for character-based applications; windows and controls for graphical applications - all that is needed to move from one to another is to change the names and attributes of the components. Most of the logic and common routines can be left intact.

Portable architecture between tools

This approach is also portable between test tools. As long as the underlying script language has the equivalent set of commands, test cases in this format could be executed by a script library created in any tool. This means you are free to use different tools for different platforms if necessary, or to migrate to another tool.

No tool expertise needed to create test cases

Because logic is defined and stored only once per method, and there is substantial implied logic for verifying the context and state of the application to assure proper playback, individual testers may create test cases without understanding logic or programming. All that is needed is an understanding of the application components, their names, and the valid methods and values which apply to them.

Disadvantages

The central disadvantage of the Table-Driven approach is the amount of training and setup time required to implement it.

Extensive technical and tool skills required

In order to properly design and construct the script library, extensive programming skills and test tool expertise are needed. Programming skills are needed to implement a library of modular scripts and the surrounding logic that ties them together and uses external data to drive them.

Data conventions and management critical

Because all test assets are maintained as data, it is essential to have a means of enforcing conventions and managing the data. While this can be done in spreadsheets, a database is far more powerful.

The Test Automation Process

In an ideal world, testing would parallel the systems development life cycle for the application. This cycle is generally depicted as:

Software

Planning	Requirements	Design	Code	Test	Maintain

Testware

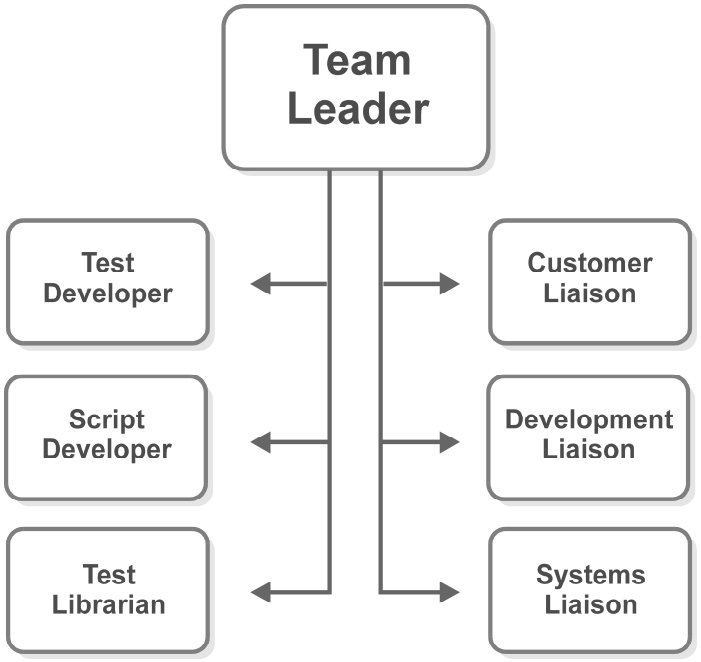
Test Plan	Test Cases	Test Scripts	Test Execution/Maintenance

Unfortunately, not all test efforts commence at the earliest stage of the software development process. Depending on where your application is in the timeline, these activities may be compressed and slide to the right, but in general each of these steps must be completed.

The Test Team

But regardless of the approach you select, to automate your testing you will need to assemble a dedicated test team and obtain the assistance of other areas in the company. It is important to match the skills of the persons on your team with the responsibilities of their role. For example - although the type and level of skills will vary somewhat with the automation approach you adopt - developing test scripts is essentially a form of programming; for this role, a more technical background is needed.

You must also be sure that the person in each role has the requisite authority to carry out their responsibilities; for example, the team leader must have control over the workflow of the team members, and the test librarian must be able to enforce procedures for change and version control.



Following are suggested members of the test team and their respective responsibilities:

Team Leader The Team Leader is responsible for developing the Test Plan and managing the team members according to it, as well as coordinating with other areas to accomplish the test effort. The Team Leader must have the authority to assign duties and control the workflow of those who are dedicated to the test team.

Test Developers Test Developers are experts in the application functionality, responsible for developing the test cases, executing them, analyzing and reporting the results. They should be trained on how to develop tests, whether as data records or as scripts, and use the test framework.

Script Developers Script Developers are experts in the testing tool, ideally with technical programming experience. They are responsible for developing and maintaining the test framework and supporting scripts and publishing the Application Map.

Test Librarian The Test Librarian is responsible for managing the configuration, change and version control for all elements of the test library. This includes defining and enforcing check in and check out procedures for all files and related documentation.

Customer Liaison The Customer Liaison represents the user community of the application under test and is responsible for final approval of the test plan or any changes to it, and for working with the Test Developers to identify test cases and gather sample documents and data. Even though the Customer Liaison may not be a dedicated part of the testing organization, he or she must have dotted line responsibility to the Test Team to assure the acceptance criteria are communicated and met.

Development Liaison The Development Liaison represents the programmers who will provide the application software for test and is responsible for delivering unit test cases and informing the Test Librarian of any changes to the application or its environment. Even though the Development Liaison may not be a dedicated part of the testing organization, he or she must have dotted line responsibility to the Test Team to assure the software is properly unit tested and delivered in a known state to the Test Team.

Systems Liaison The Systems Liaison represents the system or network support group and database administrator, and is responsible for supporting the test environment to assure that the Test Team has access to the proper platform configuration and database for test execution. The Systems Liaison must also inform the Test Librarian of any changes to the test platform, configuration or database.

Test Automation Plan

A Test Automation Plan describes the steps needed to automate testing. It outlines the necessary components of the test library, the resources which will be required, the schedule, and the entry/exit criteria for moving from one step to the next. Note that the Test Automation Plan is a living document that will be maintained throughout the test cycle as progress is made and additional information is obtained. Following is an example Plan:

Document Control

Activity	Initials	Date	Comments
Created	LL	5/1/XX	F:\GENLDGR\TESTPLAN.DOC
Updated	CC	5/21/XX	Added acceptance criteria
Updated	TT	5/31/XX	Added test cases
Revised	PP	6/3/XX	Modified schedule
Published	LL	6/15/XX	Circulated to team members for approval
Approved	CC	6/17/XX	Updates and revisions accepted

This section is used to control additions and changes to the plan. Because the plan will likely be modified over time, keeping track of the changes is important.

Application

```
Version 1.0 of the General Ledger system.
```

Describe the application under test in this section. Be sure to specify the version number. If only a subset is to be automated, describe it as well.

Scope of Test Automation

```
Black box tests for each of the listed requirements will be
automated by the test team. Unit string and integration testing
will be performed by development manually using a code debugger
when necessary.  Performance testing will be done using
Distributed Test Facility to create maximum simultaneous user
load. Ad hoc testing for usability will be captured by the
automated tool for reproducing conditions that lead to failure but
will not be checked into the test library.
```

The statement of scope is as important to describe what will be tested as what will not be, as well as who will be responsible.

Test Team

Name	Role	Initials
Mike Manager	Team Leader	MM
Tina Tester	Test Developer	TT
Steve Scripter	Script Developer	SS
Loretta Librarian	Test Librarian	LL
Carla Customer	Customer Liaison	CC
Dave Developer	Development Liaison	DD
Percy Production	Systems Liaison	PP

List the names and roles of the test team members, and cross-reference each of the steps to the responsible party(ies).

Be sure you have a handle on the test environment and configuration. These factors can affect compatibility and performance as much as the application itself. This includes everything about the environment, including operating systems, databases and any third party software.

Schedule

Phase	Scheduled Date	Entry Criteria	Exit Criteria	Date Complete
Test requirements defined		Planning completed	SIGNOFF by customer	
Configuration of test environment		Hardware and software	SIGNOFF by system support	
Publication of Application Map		Design completed	SIGNOFF by development	
Development of test cases		Requirements defined	SIGNOFF by customer	
Initial installation of application		Coding completed	SIGNOFF by development	
Development of test scripts		Application installed	SIGNOFF by team leader	
Execution of tests and result reporting		Cases, scripts completed	SIGNOFF by team leader	
Result analysis and defect reporting		Execution completed	SIGNOFF by team leader	
Test cases for defects found		Defect reporting	SIGNOFF by customer	

Schedule (continued)

Test script modifications for execution errors		Result analysis	SIGNOFF by team leader	
Second installation of application		Changes completed	SIGNOFF by development	
Execution of tests		Application installed	SIGNOFF by team leader	
Result analysis and defect reporting		Execution completed	SIGNOFF by team leader	
Test cases for defects found		Defect reporting	SIGNOFF by customer	
Test script modifications for execution errors		Result analysis	SIGNOFF by team leader	
Third installation of application		Changes completed	SIGNOFF by development	
Execution of tests		Application installed	SIGNOFF by team leader	
Ad hoc and usability testing		No known or waived defects	SIGNOFF by customer	
Performance testing		No known or waived defects	SIGNOFF by systems	
Result analysis and defect reporting		All tests executed	SIGNOFF by team leader	
Application release		No known or waived defects	SIGNOFF by all test team	

Planning the Test Cycle

In an automated environment, the test cycle must be carefully planned to minimize the amount of supervision or interaction required. Ideally, an execution cycle should be capable of automatically preparing and verifying the test environment, executing test suites or individual tests in sequence, producing test result reports, and performing final cleanup.

Test Suite Design

A test suite is a set of tests which are related, either by their function or by the area of the application they impact, and which are executed as a group. Not only the set of tests but their sequence within the suite should be considered. The execution of a suite may be a feature available from the test tool, or may have to be scripted within the tool itself using a driver.

Related tests A test suite usually contains tests that are related by the area of the application they exercise, but they may also be selected by their priority. For example, each suite of tests may be designated by the priority level of the requirements they verify. This allows the most critical tests to be executed selectively, and less important ones to be segregated. Another means of differentiation is to identify tests by their type; for example, verifying error messaging or other type of design requirements.

Context All tests in a suite should share the same beginning and ending context, as well as the expected state of the database. This allows the suite to be packaged so that the data is prepared at the beginning, and all tests that depend on each other for to be executed in the proper sequence. Any RECOVERY routine that is included should also coincide with the desired context. If the suite is packaged to be executed with a driver script, each individual test within the suite should end with a return to the calling driver.

Documenta- Test suite documentation should include the set and sequence of

tion individual tests, the beginning and ending context, as well as any data or sequence dependencies with other test suites.

Test Cycle Design

In addition to designing test suites, the entire test execution cycle must also be designed. There may be different types of test cycles needed; for example, a regression cycle that exercises the entire test library, or a fix cycle that tests only targeted areas of the application where changes have been made. Although there may be varying cycles, certain aspects of each cycle must always be considered, such as the configuration of the test platform as well as initialization, setup and cleanup of the test environment.

Setup

The cycle should commence with the setup of the test environment, including verifying the configuration and all other variables that affect test execution. Preparing the test environment for execution requires that the platform be properly configured. A test cycle executed against the wrong platform configuration may be worthless. The configuration includes not only assuring that the hardware, operating system(s), and other utilities are present and of the expected model or version, but also that the version or level of the application and test library are properly synchronized.

Certain portions of the configuration may be automated and included in a shared routine, such as SETUP; others may require human intervention, such as loading software. Whatever the required steps, the configuration of the test platform should be carefully documented and verified at the beginning of each test execution cycle.

Context

The beginning and ending context of a cycle should be the same point, usually the program manager or command prompt. Care should be taken to synchronize the suites within the cycle to assure that the context for the first and last suite meets this requirement.

In addition to assuring that the test platform is configured, it may be

important to initialize the state of the database or other data elements. For example, a clean version of the database may be restored, or a subset appended or rewritten, in order to assure that the data is in a known state before testing begins. Data elements, such as error counters, may also require initialization to assure that previous test results have been cleared.

Schedule sequence A test schedule is often comprised of a set of test cycles. The sequence should reflect any dependencies of either context or data, and standard tests, such as a WALKTHU, should be packaged as well. A test schedule template may be useful for assuring that all standard tests and tasks are included for each run.

Cleanup The cycle should end with the cleanup of the test environment, such as deleting work files, making file backups, assembling historical results, and any other housekeeping tasks.

Test Execution

Since an ideally automated test cycle does not depend on human intervention or supervision, the test execution process must thoroughly document results. This documentation must be sufficient to determine which tests passed or failed, what performance was, as well as provide additional information that may be needed to assist with diagnosis of failures.

Test log

The test log reports the results of the test execution for each test case. It is also useful to include the elapsed time for each test, as this may indicate performance problems or other issues. For example, a test which executes too quickly may indicate that it was not run all the way to completion; one that takes too long might raise questions about host response time.

Pass/fail Each individual test case - whether as data record(s) or scripts - should be logged as to whether it executed successfully or not. Ideally, each case should be cross-referenced to a requirement that has a priority rating, so that high priority requirements can be easily tracked.

Performance In addition to reporting the elapsed time for each test, if host or server based testing is involved the test log should track the highest response time for transactions as well as the average. Many service level agreements specify the maximum allowable response time, and/or the expected average, for given areas of the system.

Performance measurements may also include the overall time required to execute certain functions, such as a file update or other batch process. It is of course critical to establish the performance criteria for the application under test, then assure that the necessary tests are executed and measurements taken to confirm whether the requirements are in fact met or not.

Configuration Each test log should clearly indicate the configuration against which it was executed. This may take the form of a header area or comments. If subsequent logs show widely varying results, such as in the area of performance, then any changes to the configuration may provide a clue.

Totals Total test cases executed, passed and failed, as well as the elapsed time overall, should be provided at the end of the execution log to simplify the updating of historical trends.

Test Log

Date: 01/01/2X TIME: HH:MM Page: XXX

Application: GENLEDGR Version: 5.1.1 Test Cycle: ALL

Test Suite	Test Script	Test Case	Begin	End	Status
CHT_ACCTS			08:11:12		
	NEW_ACCTS	100-0000	08:11:15	08:12:21	Passed
	NEW_ACCTS	101-0000	08:12:23	08:13:25	Passed
	NEW_ACCTS	102-0000	08:13:29	08:14:31	Passed
	NEW_ACCTS	102-1000	08:14:34	08:15:42	Passed
	NEW_ACCTS	102-2000	08:15:45	08:16:50	Passed
	NEW_ACCTS	110-0000	08:16:53	08:18:01	Passed
	NEW_ACCTS	111-0000	08:18:05	08:19:17	Passed
	NEW_ACCTS	111-1000	08:19:20	08:19:28	Passed
	NEW_ACCTS	111-2000	08:19:33	08:20:54	Passed
	DEL_ACCTS	112-0000	08:21:02	08:22:19	Failed

Elapsed Time: 00:11:07 Cases Passed: 9 Cases Failed: 1

Test Log Summary

Failed:	**Previous**	**New**	**Resolved**	**Remaining**
Priority 1	9	10	9	10
Priority 2	58	10	22	46
Priority 3	70	25	30	65
Total	137	45	61	121

Passed: 172

Total Executed 217

Ratios: 21% **defects** 55% **recurrence**

Error log

For every test which fails, there should be a corresponding entry in the error log. The error log provides more detailed information about a test failure to support diagnosis of the problem and determination about its cause.

Test case and script The test case which failed, as well as the script being executed, should be documented in the error log to enable later review of the error condition. Errors do not necessarily indicate a defect in the application: the test case or script may contain errors, or the application context or data may be incorrect.

State of application When an error occurs, it is important to document the actual state of the application for comparison against what was expected. This may include a snapshot of the screen at the time of failure, the date and time, test case being executed, the expected result, and whether recovery was attempted.

Diagnostics Whenever possible, the error log should include as much diagnostic information about the state of the application as is available: stack or memory dumps, file listings, and similar documentation may be generated to assist with later diagnosis of the error condition.

Analyzing Results

At the conclusion of each test cycle, the test results - in the form of the execution, performance and error logs - must be analyzed. Automated testing may yield results which are not necessarily accurate or meaningful; for example, the execution log may report hundreds of errors, but a closer examination may reveal that an early, critical test failed which in turn jeopardized the integrity of the database for all subsequent tests.

Inaccurate results

Inaccurate results occur when the test results do not accurately reflect the state of the application. There are generally three types of inaccurate results: false failures, duplicate failures, and false successes.

False failure from test environment

A false failure is a test which fails for a reason other than an error or defect in the application. A test may fail because the state of the database is not as expected due to an earlier test, or because the test environment is not properly configured or setup, or because a different error has caused the test to lose context.
Or, a test which relies on bitmap comparisons may have been captured against one monitor resolution and executed against another.

False failure from application changes

Another type of false failure can occur if a new field or control is added, causing the script to get out of context and report failures for other fields or controls that are actually functional. Any of these situations will waste resources and skew test results, confusing the metrics which are used to manage the test process.

False failure from test errors

It is unfortunately true that the failure may also be the result of an error in the test itself. For example, there may be a missing test case record or an error in the script. Just as programmers may introduce

one problem while fixing another, test cases and scripts are subject to error when modifications are made.

Duplicate failure

A duplicate failure is a failure which is attributable to the same cause as another failure. For example, if a window title is misspelled, this should be reported as only one error; however, depending on what the test is verifying, the name of the window might be compared multiple times. It is not accurate to report the same failure over and over, as this will skew test results.

For example, if a heavily-used transaction window has an error, this error may be reported for every transaction that is entered into it; so, if there are five hundred transactions, there will be five hundred errors reported. Once that error is fixed, the number of errors will drop by five hundred. Using these figures to measure application readiness or project the time for release is risky: it may appear that the application is seriously defective, but the errors are being corrected at an astronomical rate - neither of which is true.

False success from test defect

A false success occurs when a test fails to verify one or more aspects of the behavior, thus reporting that the test was successful when in fact it was not. This may happen for several reasons. One reason might be that the test itself has a defect, such as a logic path that drops processing through the test so that it bypasses certain steps. This type of false success can be identified by measurements such as elapsed time: if the test completes too quickly, for example, this might indicate that it did not execute properly.

False success from missed error

Another false success might occur if the test is looking for only a specific response, thus missing an incorrect response that indicates an error. For example, if the test expects an error to be reported with an error message in a certain area of the screen, and it instead appears elsewhere. Or, if an asynchronous error message appears, such as a broadcast message from the database or network, and the test is not looking for it. This type of false success may be avoided by building in standard tests such as a MONITOR, described in this Handbook.

Defect tracking

Once a test failure is determined to be in fact caused by an error in the application, it becomes a defect that must be reported to development for resolution. Each reported defect should be given a unique identifier and tracked as to the test case that revealed it, the date it was logged as a defect, the developer it was assigned to, and when it was actually fixed.

Test Metrics

Metrics are simply measurements. Test metrics are those measurements from your test process that will help you determine where the application stands and when it will be ready for release. In an ideal world, you would measure your tests at every phase of the development cycle, thus gaining an objective and accurate view of how thorough your tests are and how closely the application complies with its requirements.

In the real world, you may not have the luxury of the time, tools or tests to give you totally thorough metrics. For example, documented test requirements may not exist, or the set of test cases necessary to achieve complete coverage may not be known in advance. In these cases, you must use what you have as effectively as possible.

Measure progress The most important point to make about test metrics is that they are essential to measuring progress. Testing is a never-ending task, and if you don't have some means of establishing forward progress it is easy to get discouraged. Usually, testers don't have any indication of success, only of failure: they don't hear about the errors they catch, only the ones that make it into production. So, use metrics as a motivator. Even if you can't test everything, you can get comfort from the fact that you test more now than before!

Code coverage Code coverage is a measurement of what percentage of the underlying application source code was executed during the test cycle. Notice that it does not tell you how much of the code passed the test - only how much was executed during the test. Thus, 100% code coverage does not tell you whether your application is 100% ready.

A source level tool is required to provide this metric, and often it requires that the code itself be instrumented, or modified, in order to capture the measurement. Because of this, programmers are usually the only ones equipped to capture this metric, and then only during their unit test phase.

Although helpful, code coverage is not an unerring indicator of test coverage. Just because the majority of code was executed during the test, it doesn't means that errors are unlikely. It only takes a single line - or character - of code to cause a problem. Also, code coverage only measures the code that exists: it can't measure the code that is missing.

When it is available, however, code coverage can be used to help you gauge how thorough your test cases are. If your coverage is low, analyze the areas which are not exercised to determine what types of tests need to be added.

Requirements coverage

Requirements coverage measures the percentage of the requirements that were tested. Again, like code coverage, this does not mean the requirements were met, only that they were tested. For this metric to be truly meaningful, you must keep track of the difference between simple coverage and successful coverage.

There are two prerequisites to this metric: one, that the requirements are known and documented, and two, that the tests are cross-referenced to the requirements. In many cases, the application requirements are not documented sufficiently for this metric to be taken or be meaningful. If they are documented, though, this measurement can tell you how much of the expected functionality has been tested.

Requirements

However, if you have taken care to associate requirements with your

satisfied test cases, you may be able to measure the percentage of the requirements that were met - that is, the number that passed the test. Ultimately, this is a more meaningful measurement, since it tells you how close the application is to meeting its intended purpose.

Priority Requirements Because requirements can vary from critical to important to desirable, simple percentage coverage may not tell you enough. It is better to rate requirements by priority, or risk, then measure coverage at each level. For example, priority level 1 requirements might be those that must be met for the system to be operational, priority 2 those that must be met for the system to be acceptable, level 3 those that are necessary but not critical, level 4 those that are desirable, and level 5 those that are cosmetic.

In this scheme, 100% successful coverage of level 1 and 2 requirements would be more important than 90% coverage of all requirements; even missing a single level 1 could render the system unusable. If you are strapped for time and resources (and who isn't), it is well worth the extra time to rate your requirements so you can gauge your progress and the application's readiness in terms of the successful coverage of priority requirements, instead of investing precious resources in low priority testing.

Exit criteria Successful requirements coverage is a useful exit criteria for the test process. The criteria for releasing the application into production, for example, could be successful coverage of all level 1 through 3 priority requirements. By measuring the percentage of requirements tested versus the number of discovered errors, you could extrapolate the number of remaining errors given the remaining number of requirements.
But as with all metrics, don't use them to kid yourself. If you have only defined one requirement, 100% coverage is not meaningful!

Test case Test case coverage measures how many test cases have been

coverage executed. Again, be sure to differentiate between how many passed and how many were simply executed. In order to capture this metric, you have an accurate count of how many test cases have been defined, and you must log out each test case that is executed and whether it passed or failed.

Predicting time to release Test case coverage is useful for tracking progress during a test cycle. By telling you how many of the test cases have been executed in a given amount of time, you can more accurately estimate how much time is needed to test the remainder. Further, by comparing the rate at which errors have been uncovered, you can also make a more educated guess about how many remain to be found.

As a simple example, if you have executed 50% of your test cases in one week, you might predict that you will need another week to finish the cycle. If you have found ten errors so far, you could also estimate that there are that many again waiting to be found. By figuring in the rate at which errors are being corrected (more on this below), you could also extrapolate how long it will take to turn around fixes and complete another test cycle.

Defect Ratio The defect ratio measures how many errors are found as a percentage of tests executed. Since an error in the test may not necessarily be the result of a defect in the application, this measurement may not be derived directly from your error log; instead, it should be taken only after an error is confirmed to be a defect.

If you are finding one defect out of every ten tests, your defect ratio is 10%. Although it does not necessarily indicate the severity of the errors, this metric can help you predict how many errors are left to find based on the number of tests remaining to be executed.

Fix rate Instead of a percentage, the fix rate measures how long it takes for a

reported defect to be fixed. But before you know if a defect is fixed, it must be incorporated into a new build and tested to confirm that the defect is in fact corrected.

For this metric to be meaningful, you have to take into account any delays that are built into the process. For example, it may only take two hours to correct an error, but if a new build is created only weekly and the test cycle performed only once every two weeks, it may appear as though it takes three weeks to fix a defect. Therefore, measure the fix rate from the time the defect is reported until the corresponding fix is introduced into the source library.

Recurrence ratio

If a code change that is purported to fix a defect does not, or introduces yet another defect, you have a recurrence. The recurrence ratio is that percentage of fixes that fail to correct the defect. This is important because although your developers may be able to demonstrate a very fast turnaround on fixes, if the recurrence ratio is high you are spinning your wheels.

This ratio is extremely useful for measuring the quality of your unit and integration test practices. A high recurrence ratio means your developers are not thoroughly testing their work. This inefficiency may be avoided to some degree by providing the programmer with the test case that revealed the defect, so that he or she can verify that the code change in fact fixes the problem before resubmitting it for another round of testing.

So temper your fix rate with the recurrence ratio. It is better to have a slower fix rate than a high recurrence ratio: defects that recur cost everyone time and effort.

Post-release defects

A post-release defect is a defect found after the application has been released. It is the most serious type of defect, since it not only reflects a weakness in the test process, it also may have caused

mayhem in production. For this reason, it is important to know not just how many of these there are, but what their severity is and how they could have been prevented.

As discussed earlier, requirements should be prioritized to determine their criticality. Post-release defects should likewise be rated. A priority 1 defect - one which renders the system unusable - should naturally get more attention than a cosmetic defect. Thus, a simple numerical count is not as meaningful.

Defect prevention

Once a defect is identified and rated, the next question should be when and how it could have been prevented. Note that this question is not about assessing blame, it is about continuous process improvement. If you don't learn from your mistakes, you are bound to repeat them.

Determining when a defect could have been prevented refers to what phase of the development cycle it should have been identified in. For example, a crippling performance problem caused by inadequate hardware resources should probably have been revealed during the planning phase; a missing feature or function should have been raised during the requirements or design phases.

In some cases, the defect may arise from a known requirement but schedule pressures during the test phase may have prevented the appropriate test cases from being developed and executed.

Continuous improvement

Whatever the phase, learn from the problem and institute measures to improve it. For example, when pressure arises during a later cycle to release the product without a thorough test phase, the known impact of doing so in a previous cycle can be weighed against the cost of delay. A known risk is easier to evaluate than an unknown one.

As to how a defect could be prevented, there are a wide range of possibilities. Although the most obvious means of preventing it from being released into production is to test for it, that is really not what this is about. Preventing a defect means keeping it from coming into existence, not finding it afterwards. It is far more expensive to find a defect than to prevent one. Defect prevention is about the entire development cycle: how can you better develop high quality applications in the future?

By keeping track of post-release defects as well as their root causes, you can not only measure the efficacy of your development and test processes, but also improve them.

Management Reporting

Although there are many sophisticated metrics for measuring the test process, management is usually interested in something very simple: when will the application be ready? If you can't answer this question, you run the risk that the application will be released arbitrarily, based on schedules, instead of based on readiness. Few organizations can make open-ended commitments about release dates.

Once management has invested time and money in test automation, they will also want to know what their return was. This return could take three forms: savings in money, time, and/or improved quality. By assuring that you have these measurements at your fingertips, you can increase the odds of keeping management committed to the test automation effort.

Estimated time to release

Although you can never precisely predict when or even if an application will be defect-free, you can make an educated guess based on what you do know. The best predictor of readiness for release is the requirements coverage as affected by the defect ratio, fix rate and recurrence ratio.

For example, if after four weeks you are 80% through with 100 test cases with a 20% defect ratio, a two day fix rate and a 5% recurrence ratio, you can estimate time to release as:

4 weeks = 80% 20% defects = 16

1 week = 20% 5% recurrence = 1

2 day fix rate = 34 days

1 week + (34 days/5 days per week) + 5 weeks to test fixes = 13 weeks to release

Saving money

There are two kinds of savings from automated testing. The first is the productivity that comes from repeating manual tests. Even though you may not actually cut staff, you can get more done in less time. To measure this savings - the amount you would have spent to get the same level of test coverage - measure the time it takes to manually execute an average test, then automate that test and measure the time to execute it.

Divide the automated time into the manual test time. If it takes two hours to perform the test manually but it will playback in thirty minutes, you will get a productivity factor of 4. Next, execute a complete automated test cycle and measure the total elapsed time, then multiply that times the productivity factor. In this example, a twelve hour automated test cycle saves 48 hours of manual test time.

So, if you have four releases per year and three test iterations per release, you are saving (4 times 3 times 48 hours) 576 hours per year. Multiply that by your cost per man hour; if it's $50, then you are saving $28,800 per year.

Saving time

Getting the application into the market or back into production faster also saves the company time. In our above example, you are shaving 3.6 weeks off the release time (3 iterations times 48 hours/40 hours per week). This is almost a month of time savings for each release. If the reason for the release is to correct errors, that extra time could translate into significant productivity.

Higher quality It is hard to measure the impact of higher quality: you can't really measure the amount of money you *aren't* spending. If you do a thorough job of testing and prevent defects from entering into production, you have saved money by not incurring downtime or overhead from the error.

Unfortunately, few companies know the cost to fix an error. The best way to tell if you are making progress is when the post-release defect rate declines.

Better coverage Even if you can't tell exactly what it is saving the company, just measure the increasing number of test cases that are executed for each release. If you assume that more tests mean fewer errors in production, this expanded coverage has value.

Historical trends

In all of these metrics, it is very useful to keep historical records so that you can measure trends. This may be as simple as keeping the numbers in a spreadsheet and plotting them graphically. Remember to also keep the numbers that went into the metric: not just test case coverage, for example, but the total number of test cases defined as well as executed that went into the calculation.

The reason historical trends are important is that they highlight progress - or, perish the thought, regression. For example, the number of requirements and test cases which have been defined for an application should be growing steadily. This indicates that enhancements, as well as problems found in production, are being added as new requirements and test cases, assuring that your test library is keeping pace with the application. A declining recurrence ratio might indicate that programming practices or unit testing has improved.

Another reason to analyze historical trends is that you can analyze the impact of changes in the process. For example, instituting design reviews or code walkthroughs might not show immediate results, but later might be reflected as a reduced defect ratio.