

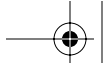
What Is Just Enough Test Automation?

1

This is not going to be a discourse on how to select and implement an automated testing tools suite. There are a number of articles and books available today that offer practical advice on tool selection. It is also not an introductory book on software testing automation. If you are reading this book, we'll assume you have some level of previous experience with test automation. We will also assume that you have some serious questions about the practical aspects of test automation. You may, or may not, have a successful implementation under your belt. In any case, you most probably have experienced the operational, political, and cultural pitfalls of test automation. What you need is a how-to book that has some practical tips, tricks, and suggestions, along with a proven approach. If this is what you want, read on. Our perspective on test automation is what you will be getting in the remaining chapters of this book.

No New Models, Please! _____

"Read My Lips: No New Models!" echoes a sentiment with which we wholeheartedly agree (14). As mentioned in the Preface, there has been a plethora of



models of the software testing process (6,10,11) and models of the automated software testing process (4,7,8,9,12,15), including a software test automation life cycle model (2). While these ideas are all right and in some aspects useful when discussing software testing and test automation, they are of little use to real-world practitioners.

The Software Engineering Institute at Carnegie Mellon University has established a Software Testing Management Key Process Area (KPA) that is necessary to achieve Level 2: Repeatable in the Software Process Capability Maturity Model (CMM) (11). Such a model is useful for general guidance, but it does not define a process that is useful to the test engineer proper. It does give test managers a warm and fuzzy feeling when they pay lip service to it but in reality the testing process activities do not reflect the model at all. The same things hold true for the software test automation life cycle model. We do not believe in life cycles. Instead, we believe in processes that direct workflows. Every testing group has a process. In some instances it is a chaotic process, and in other instances it is more organized.

Krause developed a four-level maturity model for automated software testing (3) that he ties to the software testing maturity model (1) and the SEI Software Process Maturity Model (4) that evolved into the CMM. The levels he specified are Accidental Automation, Beginning Automation, Intentional Automation, and Advanced Automation. While this model may describe what happens from a conceptual standpoint, it offers no practical advice that will facilitate test automation implementation. It merely describes what the author has noted happening in typical organizations.

Dustin, Rashka, and Paul published an Automated Test Lifecycle Methodology (ATLM)—a “structured methodology which is geared toward ensuring successful implementation of automated testing.”(2) It identifies a four-phased methodology: Decision to Automate Test; Introduction of Automated Testing; Test Planning, Design, and Development; Execution and Management of Automated Test.

While this model is useful from a management and control perspective, it is not practical from the test automation engineer’s point of view. Powers offers practical advice that can be very helpful for software testing engineers who are responsible for building and implementing a test automation framework. It includes common-sense discussions of programming style, naming standards, and other conventions that should be applied when writing automated test scripts (9).

There is a comprehensive discussion of the principle of data abstraction, which is the basis of the data-driven approach to automated software test-



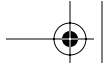
ing. He discusses alternatives for coding how data are defined and used by the test script. According to Powers, "The principle is one of depending less on the literal value of a variable or constant, and more on its meaning, or role, or usage in the test." He speaks of "constraint for product data." He defines this concept as "...the simplest form of this data abstraction is to use named program constants instead of literal values." He also speaks of "variables for product data" and says, "...instead of the literal name 'John Q. Private,' the principle of data abstraction calls for the programmer to use a program variable such as **sFullName** here, with the value set once in the program. This one occurrence of the literal means there's only one place to edit in order to change it." (9)

The immediate impact of the statement Powers makes is that you begin to see the possible benefits derived from data abstraction when it comes to the maintenance of automated test scripts. He further suggests that these values be stored in a repository that will be accessible from the test script code: "All that's required is a repository from which to fetch the values, and a program mechanism to do the retrieval." (9)

This is the underlying principle of Strang's Data Driven Automated Testing approach. His approach uses a scripting framework to read the values from the test data repository. It uses a data file that contains both the input and its expected behavior. His method has taken data abstraction from storing just the literal values to also storing the expected result values. This approach can accommodate both manual and automated data generation. The test script must be coded in such a way that it can determine *right* results from the *wrong* results (12).

Powers's and Strang's work is reflected in the data-driven approaches discussed in Chapters 7 and 8 of this book. Archer Group's Control Synchronized Data Driven Testing (CSDDT) is an example of one such approach that employs the concepts discussed here.

Rational Software Corporation has authored the Rational Unified Process (RUP), which contains specific test phases that are designed to support its automated testing tool suite (10). Even if you are not a Rational user, the testing process information provides a solid base for doing even manual testing. RUP itself comprises process documentation that addresses all of software development, not just testing. It is relatively inexpensive—the RUP CD-ROM sells for under \$1,000. The most important aspect of RUP's testing approach is that it can be used to support a data-driven automated testing framework. That is why we have used it in the past and why it is mentioned in this book.



A Life Cycle Is Not a Process

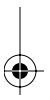
The problem with the approaches taken by the authors cited thus far and other industry gurus is the same problem we have with all life-cycle models—they are management oriented, not practitioner oriented. Again, this approach offers very little in the way of an operational process that we can term an automated testing process. Other approaches, e.g., data-driven automated testing, which these authors have criticized, offer much more in the way of methods and techniques that can actually be applied in day-to-day test automation activities. What this line of thinking really offers is a model to give testing managers the same warm and fuzzy feeling mentioned above with respect to the testing maturity model.

Although purported to be an experiential model, this representation of automated testing has not been developed on a deductive basis. It is a theory based on inductive reasoning, much of which is founded on anecdotal evidence, as are many of the models advocated in information systems (IS) literature. On the other hand, nonmanagement techniques, which are operational, not managerial, and which are applied to specific tasks in the automation process, are based on deductive reasoning. Data-driven testing is an example of a nonmanagement technique. These techniques have evolved through practitioner trial and error—how many of the traditional engineering methods have come to be that are used today.

A Tool Is Not a Process

The most recent results for the minisurvey on the CSST Technologies, Inc., Web site indicate that 40% (102 out of 258 respondents) see software testing methods/process implementation as doing the most to facilitate their testing work. Twenty-four percent (63 respondents) see improved software requirements documentation as the most important facilitation factor. Nineteen percent (50 respondents) believe that software test standards implementation is the most important aspect. Ten percent (25 respondents) cite improved test planning as the most important consideration. Only 7% (18 respondents) think that more time to test would facilitate their work.

Purchasing a software testing tool suite does not constitute implementing a software testing process. Processes are steps that are followed that result in a goal being achieved or a product being produced. The process steps implement testing activities that result in test execution and the creation of test artifacts. Automated software tools support existing processes and, when the process is chaotic, impose some much-needed structure on the activities. One of the primary reasons software testing tool implementa-





tions fail is because there is little or no testing process in place before the tools are purchased.

When we are designing and building automated tests, we do not even see a process. What we see are tasks, a schedule, and personal assignments to complete. For us, just enough software test automation is the amount we need to do our jobs effectively. If we do not have commercial software testing tools we can use, we build our own or use desktop tools that our clients routinely load on workstations.

Figure 1.1 illustrates a testing process that was defined for one of our clients that is loosely based on the RUP testing approach (10). Our process approach differs from Rational's in that we view test script design as part of test implementation whereas in RUP it is still considered to be a test design activity. The reason we differ is we believe that the skill set required for test design does not include previous programming experience, but the one for test implementation does.

How Much Automation Is Enough? _____

This is the question that has been asked since the inception of automated test tools. The tools vendors have presented us with one point of view, and industry experts have given us varied viewpoints. Vendors began with basic capture/playback tools that have evolved into some very sophisticated and highly integrated tool suites. They seem to have left it to the practitioner to determine what lies beyond the basic capture/playback model. The experts in test automation have written many articles and books. They have cited case studies of both successful and failed automation attempts. In the end there has been a modicum of agreement about what we must do, but not how we have to do it. In this text we will give you our point of view on how to do test automation. We believe the industry has debated what to do long enough. Until the tool suites reach a new plateau and until they possess even more sophistication, we have a working archetype for an automation framework that we can use.

To find out how much test automation is enough, we have to look at the areas of the software testing process that *can* be automated followed by the areas that *should* be automated. There is a difference between a tool and a process. Tools are used to facilitate processes. A tool can be used to implement a process and to enforce process standards. In many instances, the built-in procedures that tools bring with them can be construed as processes.



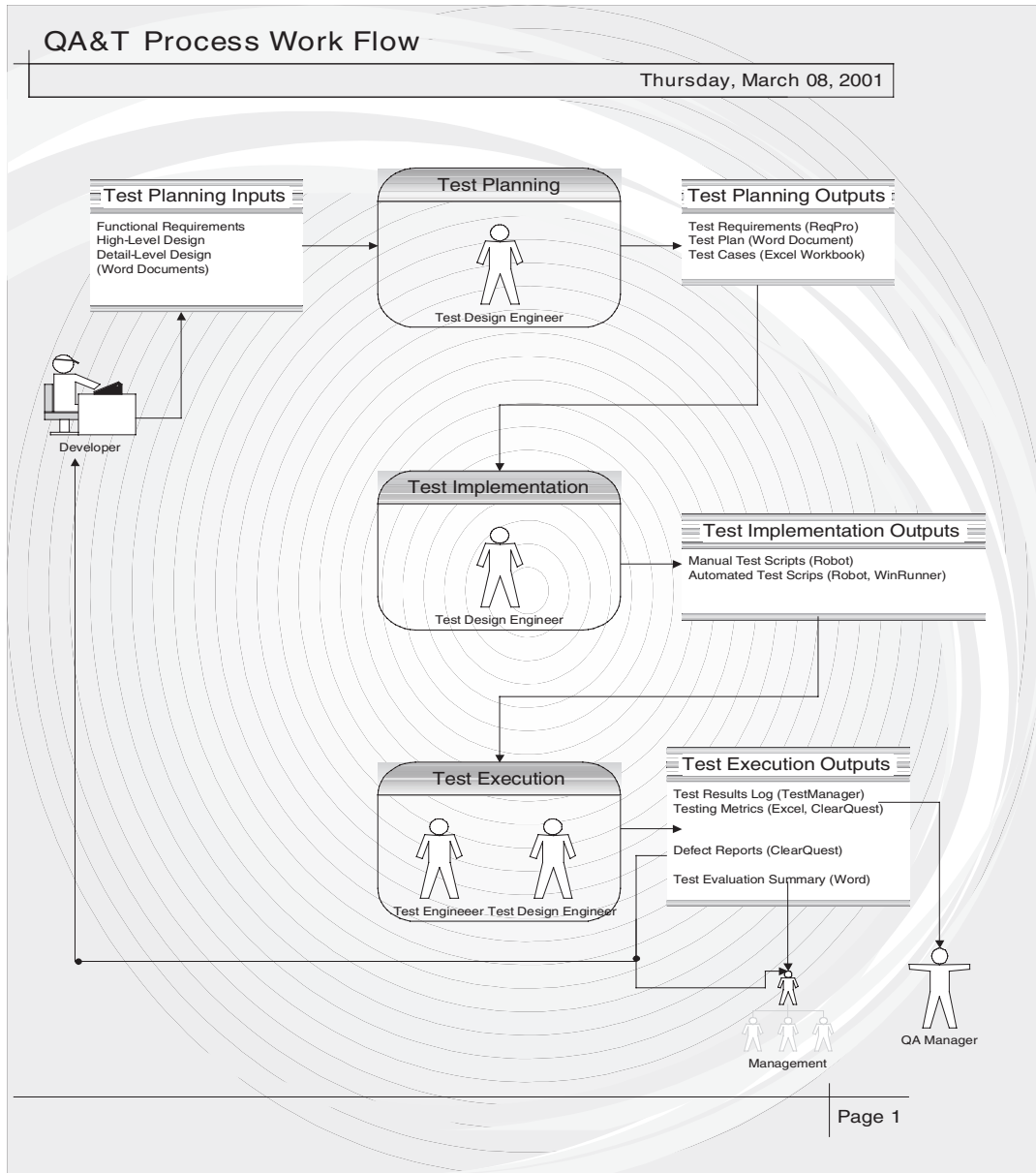


FIGURE 1.1
Quality Assurance and Testing (QA&T) Process



They are, however, frequently incomplete and ill-thought-out processes. The best software testing tools are the ones you can mold to your testing needs. They are the ones that offer a high degree of customizability with respect to workflow and to their tracking and reporting capabilities.

What are the types of tests that can be automated? They include unit, integration, and system testing. Subcategories of automated system tests include security, configuration, and load testing. Automated regression testing should be implemented at the unit, integration, and system levels during development and for use against major and minor system production releases.

What are the regions of the testing process that we should consider? They include the general spheres of test planning, test design, test construction, test execution, test results capture and analysis, test results verification, and test reporting. There are other activities that are closely associated with the testing activities proper. They include problem (defect) tracking and resolution, software configuration management, and software test metrics. Overall, the activities of the testing process are glued together, as is the software development process, by good project management techniques.

All of these areas should be automated to the degree that it makes sense for your organization in terms of time and costs. The more automation you can implement, the better and more effective your process will be. Realize that this statement holds true only if the tools are the appropriate ones and if they have been properly implemented. By *implemented* we mean that an integrated test automation framework has been constructed and is being used.

Testing Process Spheres

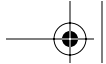
Let's look at each component of the testing process individually.

Test Planning

We'll begin our discussion of the testing process with test planning, the most important activity in the testing process. It involves assessing risk, identifying and prioritizing test requirements, estimating testing resource requirements, developing a testing project plan, and assigning testing responsibilities to test team members. These ingredients can be structured as a formal test plan or they can be developed individually and used at the appropriate times.

The traditional idea of a test plan is the *who, what, when, where, how, and how long* of the testing process. Since using Rational RequisitePro's capabili-





ties, we have adjusted our thoughts on what a test plan is and how it is used. We can import the software requirements document into tools such as Rational RequisitePro and then develop test scenarios directly from such tools into a test plan that is constructed using the RUP test plan template, which we have modified extensively (see Figure 1.2). From the test plan scenarios, we can create test requirements. They can be constructed directly in the scenario tables in the test plan document, or they can be created in a separate test requirements document using the RUP template. From either of these documents, we can create a test requirements view that we can export to a Comma Separated Values (CSV) file. Then we can open the test requirements during testing in Microsoft (MS) Excel and update it online with the test results and import it back into RequisitePro. This is how we defined our manual testing process. We do not yet have this fully implemented, but have piloted it and it works very well.

The point is that we now see the test plan as a working document derived from software requirements and linked to test requirements and test results. It is a dynamic document used during testing. The old idea of test plan is that it is a planning document. It is a document that forces the tester to think about what he is going to do during testing. From this perspective, it becomes a document that, once the planning stage is finished, sits on the shelf. In our experience, very few people refer back to the test plan during test execution. In fact, we have worked in several companies where test plans were actually created after the fact. With our approach, the test plan is created up front, and it is updated when the software requirements are updated; subsequently, the updates are reflected in the test requirements, which can actually be used for testing. The test plan standard is based on a modified version of the template contained in RUP, which accompanies the Rational Suite TestStudio.

The following is how we defined the manual system testing process at one client organization. We have also piloted this and it works very well. From the test plan scenarios, we create test case requirements. They are constructed directly in scenario tables created in the test plan document, as well as in separate test requirements documents (requirements grids). From either of these sources, we can create test requirements views that can be exported to CSV files. For manual testing, we open the CSV files in MS Excel. We use the information to guide manual test execution activities, and we update it online with the test results. We import the updated files back into our automated tool for results analysis and reporting.

Figure 1.2 contains a modified version of the RUP test plan template document table of contents (TOC). The TOC has been simplified in that the number



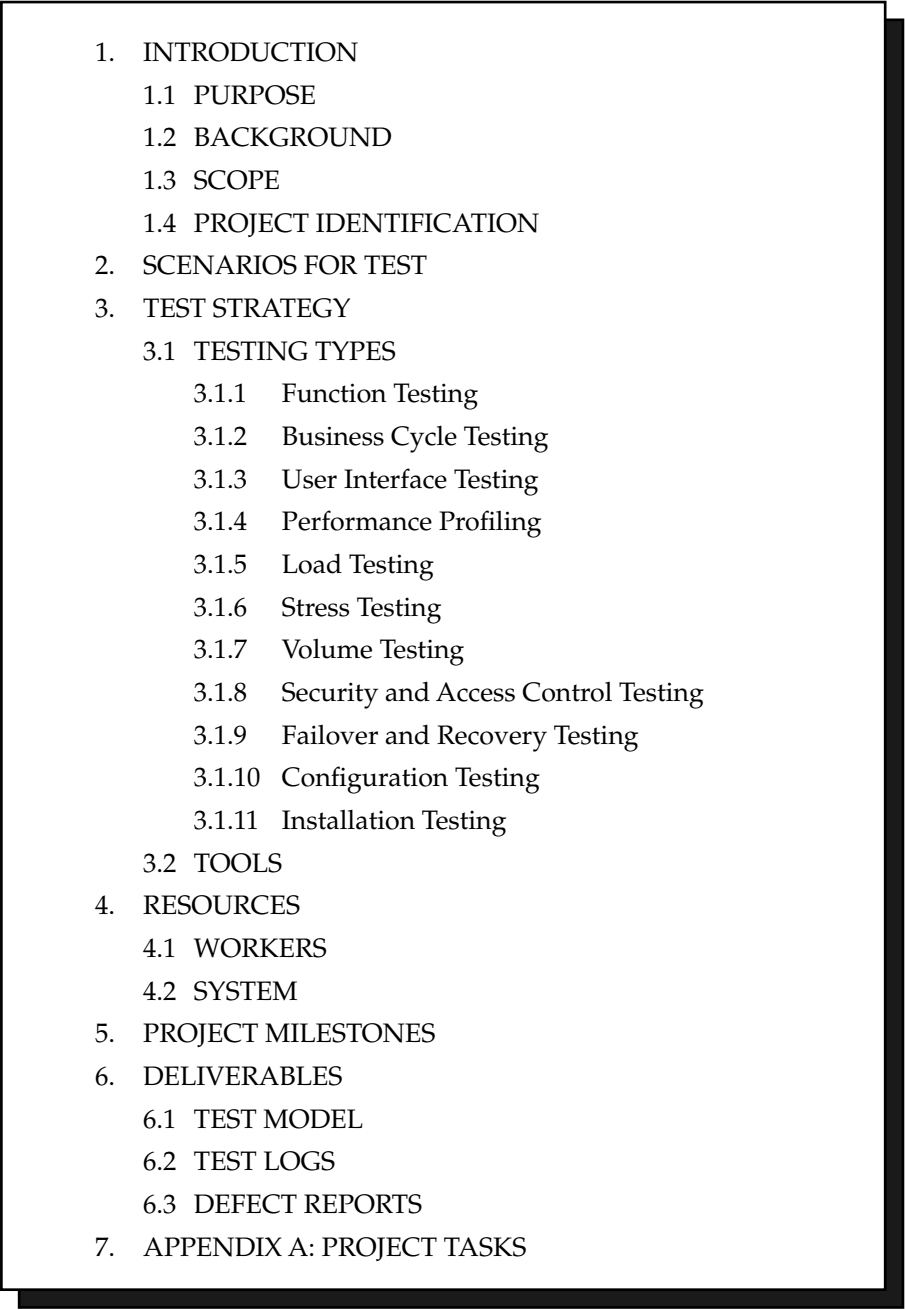
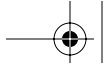
- 
1. INTRODUCTION
 - 1.1 PURPOSE
 - 1.2 BACKGROUND
 - 1.3 SCOPE
 - 1.4 PROJECT IDENTIFICATION
 2. SCENARIOS FOR TEST
 3. TEST STRATEGY
 - 3.1 TESTING TYPES
 - 3.1.1 Function Testing
 - 3.1.2 Business Cycle Testing
 - 3.1.3 User Interface Testing
 - 3.1.4 Performance Profiling
 - 3.1.5 Load Testing
 - 3.1.6 Stress Testing
 - 3.1.7 Volume Testing
 - 3.1.8 Security and Access Control Testing
 - 3.1.9 Failover and Recovery Testing
 - 3.1.10 Configuration Testing
 - 3.1.11 Installation Testing
 - 3.2 TOOLS
 4. RESOURCES
 - 4.1 WORKERS
 - 4.2 SYSTEM
 5. PROJECT MILESTONES
 6. DELIVERABLES
 - 6.1 TEST MODEL
 - 6.2 TEST LOGS
 - 6.3 DEFECT REPORTS
 7. APPENDIX A: PROJECT TASKS

FIGURE 1.2
RUP Test Plan TOC with Modifications



of testing types has been reduced to include only Function Testing, Business Cycle Testing, Setup and Configuration Testing, and User Interface Testing.

The purpose of the test plan is to assemble the information extracted from the requirements/design documents into test requirements that can be implemented as test scenarios. The test scenarios are the portion of the test plan that will directly feed the development of test conditions, test cases, and test data.

Desktop tools such as MS Office and MS Project can be used to automate test planning and project management. For example, checklists created in MS Excel spreadsheets can be used to assess and analyze risk and to document test requirements; MS Project can be used to produce the project plan; MS Word can be used to create a formal test plan that ties it altogether. These documents are test-planning *artifacts*. Just as software development artifacts need configuration management, so do test objects.

Test Design

Test design includes identifying test conditions that are based on the previously specified test requirements, developing all possible functional variants of those conditions, divining the expected behavior(s) of each variant when executed against the application under test (AUT), and executing manual tests during the design process prior to test automation. The manual testing allows the test design engineer to verify that the test data are appropriate and correct for the automated test in which they will be used. It also allows the test designer to become a “human” tester who can identify errors automation might miss. Test design also embraces the layout of the test data that will be input to the AUT. When designing from a data-driven perspective, these same data also control the navigation of the automated test scripts. Last, the test scripts themselves have to be designed.

Designing the tests and the test data is the most time-consuming portion of the testing process. It is also the most important set of activities. If the tests do not test what the requirements have indicated, then the tests are invalid. If the test data do not reflect the intent of the tests, once again the tests are invalid. Test case design is so important that we have included a section in the appendices devoted to test design techniques and their use.

Test designers can use MS Excel to design and build the tests. If you use this approach, it is best to keep everything in a single workbook and to include one test conditions spreadsheet, one test data spreadsheet, and as many detailed test spreadsheets as are needed to adequately describe the environmental, pretest, and posttest activities that are related to each test. For manual testing, a test log should be constructed that will be used during





test execution (used online, not as a printed test log). There are integrated tool suites available that support test data design and creation with mechanisms known as *data pools*. In all cases the data are stored as CSV files that can be read and interpreted by automated test scripts. Those data can be used for either manual or automated testing.

Test Implementation

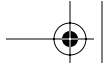
Test implementation can be subdivided into Test Construction, Test Execution, Test Results Capture and Analysis, and Test Result Verification. We'll discuss each activity separately.

Test Construction Test construction is facilitated using the same set of tools that test design employs. The test data are constructed in a spreadsheet in the same workbook as the test conditions. Those data can then be exported to CSV files that can be used at test execution. When the tests are executed via an automated framework, test construction also includes writing the test scripts. Automated test scripts are software programs. As such, they have their own programming languages and/or language extensions that are required to accommodate software testing events. The scripting language is usually embedded in a capture/playback tool that has an accompanying source code editor. The flavors of the languages vary by vendor and, as the associated syntax/semantics vary, so does the difficulty of using a specific product. In addition, some vendors' scripting languages and their recording tools are more robust than others.

The more specialized the commands available as part of the language, the more control the test engineer has over the test environment and the AUT. Specialized tests are built into the languages as commands that, when executed, test specific items—for example, graphical user interface (GUI) object properties and data and window existence—and do file comparisons. Some of the built-in test cases are useful and most are very powerful for GUI testing, but they are not all useful for functional testing. We implement many of the tests through executing external test data (data that reflect the test requirements) and verification of the results. Those data control how the test script behaves against the application; the data contain values the test script uses to populate the target application input data fields.

The design and implementation of the test scripts is left to the scriptwriter. If there are no guidelines for developing test scripts, the scripts that are created will most likely be badly structured and each will be the product of the personality of the individual who coded it. We have seen this happen when several people on the test team were given specific sections of an





application and asked to write automated test scripts for their portion. We even gave them basic templates as starting points and, still, no two were alike. We have written and implemented a set of automated test script writing guidelines that is described in Chapter 8.

If possible, test script coding should be completed in parallel with test data development. Using an approach such as Archer Group's CSDDT allows the test scriptwriters to work independently of the test data designers. This is doable because the data drive the test scripts. The scripts are designed and built to specifics that create a general test-processing engine that does not care about test data content.

It is very important to have test script coding conventions in use if the workload is spread across a number of scriptwriters. It is also important that the right personnel are assigned to this set of activities. We have found that test designers do not like coding test scripts and that test implementers do not enjoy designing and building test conditions and test data. In fact, they both do crappy work when their roles are reversed.

Script writing, as it is used here, is the *coding* of test scripts in a test tool's proprietary test scripting language, in one of the extant application programming languages such as Java or Visual Basic; in one of the standard scripting languages such as Perl, CGI, or VB Script; or in the operating system's command procedure language (for example, coding Unix shell scripts that execute test procedures). As far as test script writing is concerned, you really need people who enjoy programming. Bruce was an engineer who moved into the software industry as a programmer before becoming a test scriptwriter. That type of experience makes a person a natural test scriptwriter. Test script coding requires either prior programming experience or current training in programming concepts such as logic, syntax, and semantics. It also requires an eye for developing complex logic structures. Because of these requirements, don't expect nontechnical testers to write test scripts and, furthermore, don't expect them to be able to create effective test scripts using the capture/playback facilities most tool suites provide. Test scripts developed by that method are a maintenance nightmare.

Test Execution Test execution can be manual, automated, or automated-manual. It is common wisdom in the testing industry that manual tests and automated tests each find different classes of errors. Thus, the experts say that we should do both. We agree to a great extent—that to do full-blown manual system tests and then follow up with automated regression tests is a neat idea, but most testing efforts have the resources to do only one or the other. What we suggest is a combination of both where manual testing occurs in parallel with test case design and construction.





The test designers should have the application open as they design and construct the test data, and they should execute it against the application feature(s) it is designed to test. This accomplishes two things. First, it performs a validation of the data that will eventually be used in the automated regression tests and, second, it implements manual functional (system-level) tests of each application feature prior to the automated regression tests. Many errors can and will be found during test case design and construction when this process is followed. We have successfully applied this approach.

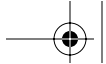
Test Results Capture and Analysis For manual testing, a test log should be developed and implemented online. At the very least it should be a spreadsheet in the same workbook where the test conditions and test data are stored. Ideally, a test results database is used as a central repository for permanent storage. This is the case when the tests are executed using capture/playback tools such as Rational Robot or Mercury Interactive's WinRunner. The test results are trapped as they occur and are written to the repository where later they can be viewed and reports can be printed.

Bruce has developed an automated manual test script (using Rational Robot and the SQABasic language) that does this very thing for manual tests (this tool is fully described in Chapter 9). It displays the manual test scripts much as a teleprompter does, and each step in the script can be selected during test execution. When execution is completed, the test can be logged as pass, fail, or skipped. The manual test script is based on one that was published several years ago by the Phoenix Arizona SQA Users Group (13). It has been completely rewritten and enhanced to add custom comment entries in the test log. This test script is included with the utilities that are on the FTP site that supports this book. It can easily be adapted to scripting languages other than SQABasic.

Test Results Verification Test results verification can be accomplished either manually or through test automation. Manually eyeballing the results and making subjective assessments of correctness is one way of verifying the test results. Test results can further be compared to sets of expected output data values, compared to values that should have been created in database rows based on the transactions tested, and compared against stored files and reports.

Test results verification is a task ripe for automation. Test results should be captured in a test log that is stored in a test repository. The test log should store results from previous test runs that can serve as baseline behaviors with which to compare the current test results. For manual tests, the test log can be an MS Excel workbook where new spreadsheets are created for each





new test iteration. It does not matter how the test results are stored. What is important is that comparisons can be made. If a baseline is not established, then the assessment of a pass or fail status is based on a guess at what the results should be. Being able to define and store baseline behaviors is an important advantage and a strong argument for automated testing.

Commercially available automated testing tool suites offer a variety of automated verification methods. For example, Rational Robot uses SQA-Basic, which has special test cases called *verification points* that can be used to trap AUT characteristics; these can then be used as baselines for future regression tests.

Test Reporting Test reporting is an essential part of the testing process because it documents the test results and their analysis. Two levels of reporting are required—a summary report should be generated for middle- and upper-level technical managers and for customers, and a detailed report should be compiled and presented to the development team members as feedback.

These reports should be presented in standard formats that can be edited and tweaked for each individual test project report. We have employed the reporting template that is in RUP's documentation, but you can create your own. Two versions of this report can be created—a summary report and a detailed report. You can also find templates and examples of test reporting on the World Wide Web.

An important reporting item is also defect tracking information. Defect tracking reports can be generated separately using tools such as Rational ClearQuest and/or MS Excel. Defect information should also be summarized and included in both the detailed and summary test evaluation reports. It is imperative to include a list of known defects that have not been addressed and that will be in the software upon its release. The information in the list should be grouped according to severity. Information such as this can be used to make intelligent release decisions, and help desk personnel can use it after the software goes into production.

Support Activities

Testing Is a Team Effort

Because software testing is done at the team level, it requires tools that support and enhance team member communication and that present an integrated interface allowing team members to share common views of testing





process activities and artifacts. One of the predominant problems at all stages of the testing process is artifact control and storage. One of the areas that provide the most payback to the testing process is automated configuration management of testing deliverables. There are many documents and executables that are created that must be available to all of the test team members. Team members frequently work these entities in parallel. They must be protected from concurrent updates that may overwrite each other when more than one team member is working on the same deliverable. Furthermore, there must be a central repository where the artifacts are stored for public use.

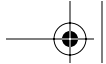
We have worked on many testing projects where there was no central storage and where everyone on the team created and updated the documents on their local desktops. We created rules defining directories for specific deliverables and stating that everyone was supposed to place their work in these shared public directories. This solution was better than no solution, but it still did not provide versioning with check-out and check-in control of the documents.

First and foremost are testing management and requirements management. Testing management can be implemented using a tool such as MS Project. It allows tasking identification, resource management, and progress assessment through the setting of testing milestones. A requirements management tool is also essential because software requirements must be documented and updated during the development process and test requirements must be documented and updated in parallel with development activities and during testing activities.

Our tool of choice for requirements management has been RequisitePro because it integrates software requirements gathering with test requirements specification. Furthermore, its test requirements grids can be exported to MS Project and then used to guide and monitor the test process. There are other requirements management tools available, some of which are integrated with testing tool suites. While this book is not about tool evaluation, there are two essential considerations when assessing these products. First, is the product already integrated with a testing tool suite? Second, if it is not, does it have an open application programming interface (API) that can be used to create your own integration code?

Software configuration management is next. There are products available that can be used to implement configuration management of testing artifacts. They include MS Visual SourceSafe, Rational ClearCase, and Merant's PVCS, just to name a few. It is imperative that all testing artifacts be stored in an automated configuration management database. It is just as important that the particular configuration management tool you have chosen commu-





nicate with the other tools you are using to support test process activities at all levels of testing. If the tool does not, then it must offer and open an API to build the software bridges you need.

Software testing metrics are a necessary component in test evaluation reporting. The metrics should include defect metrics, coverage metrics, and quality metrics. There are many useful defect tracking measures. Defect metrics can be generally categorized as defect density metrics, defect aging metrics, and defect density metrics: the number of daily/weekly opens/closes, the number of defects associated with specific software/test requirements, the number of defects listed over application objects/classes, the number of defects associated with specific types of tests, and so on. Defect reporting should be automated using at a minimum an Excel workbook because Excel has the capability to summarize spreadsheet data in charts and graphs. Defect reporting can also be automated through tools such as Rational ClearQuest, among others.

Testing quality metrics are special types of defect metrics. They include (8):

- Current state of the defect (open, being fixed, closed, etc.)
- Priority of the defect (importance to its being resolved)
- Severity of the defect (impact to the end-user, an organization, third parties, etc.)
- Defect source (the originating fault that results in this defect—the *what* component that needs to be fixed)

Coverage metrics represent an indication of the *completeness* of the testing that has been implemented. They should include both requirements-based coverage measures and code-based coverage measures. For examples of these metrics, see Chapter 9 in reference (6) and the Concepts section under “Key Measures of Testing” in reference (10).

A Test Automation Group’s Scope and Objectives_____

The Scope

A test automation group’s purpose should be to develop automated support for testing efforts. This group should be responsible for the design and implementation of a data-driven automated testing framework. They should design and construct suites of automated tests for regression testing purposes. Figure 1.3 illustrates an automated testing infrastructure that was designed for a well-known company by CSST Technologies, Inc.



The test automation framework should be deployed specifically to support automated test script development and the maintenance related to all levels of testing. The framework should support unit and integration testing and system/regression testing endeavors. This does not imply that other

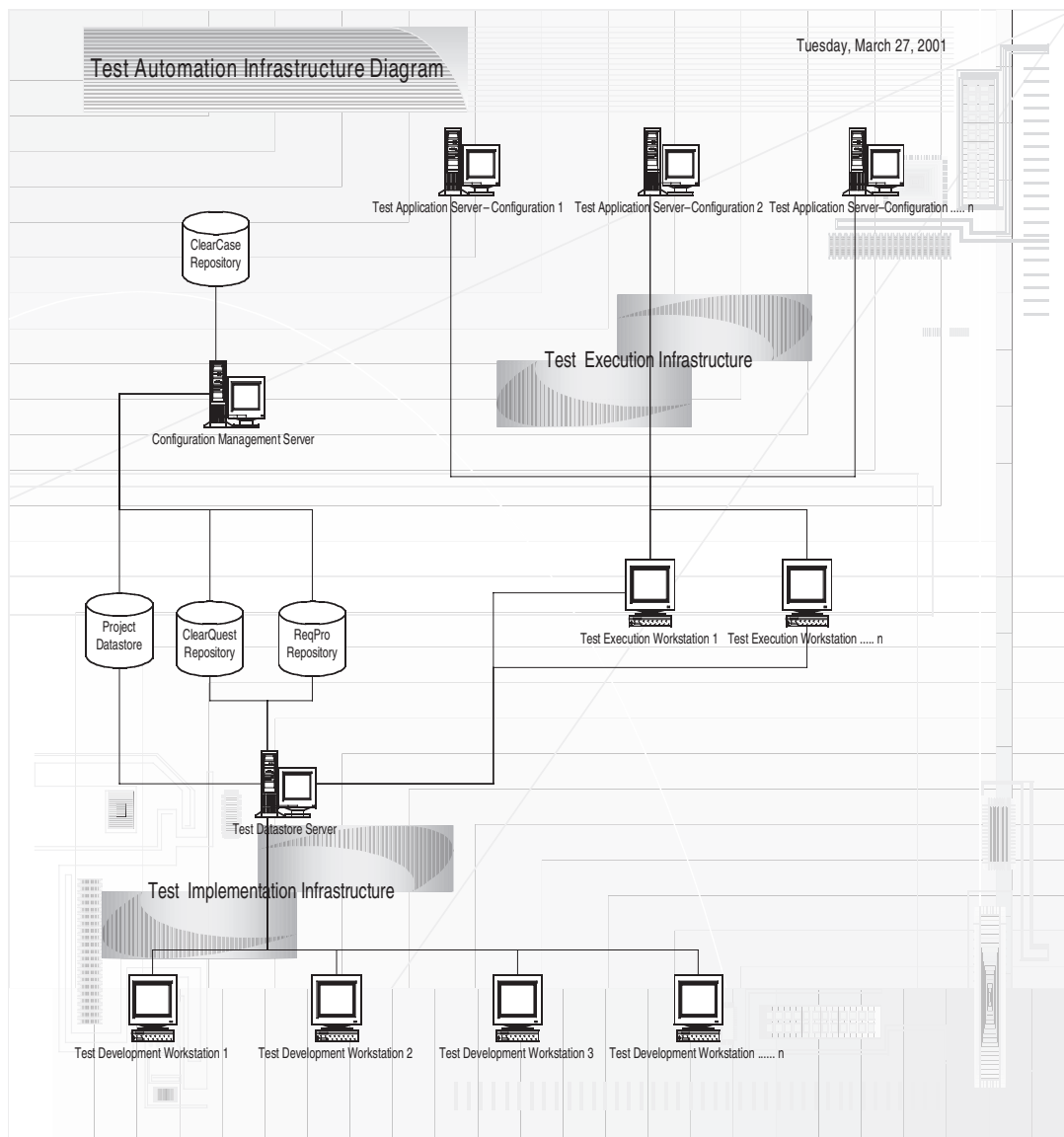
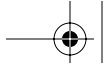


FIGURE 1.3
A Sample Automated Test Infrastructure



areas not included in this scope cannot take advantage of the test automation framework and tool suites. Other departments that may be interested in using the test automation scaffolding and the automation tool suite should fund and coordinate deployments with the automation team. An automation effort should focus on the identified areas of deployment.

The chosen approach should cover the test automation activities that will be performed by an automated tools group. Manual testing activities can serve as precursors to test automation. The goal for manual test efforts should be to manually test all application features and, while in the process, to develop test conditions and data that can be implemented using the automation framework for regression testing.

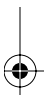
As an example, the data-driven approach could be implemented through structured test scripts that make use of functions and procedures stored in library files, the primary goal being to separate the test data from the test scripts and the secondary goal being to develop reusable test script component architecture. Meeting these goals substantially reduces the maintenance burden for automated test scripts.

Assumptions, Constraints, and Critical Success Factors for an Automated Testing Framework

The following assumptions should be applied.

Assumptions The following assumptions form the basis of the test automation strategy.

- An integrated tool suite must be the primary test management, planning, development, and implementation vehicle.
- The tool suite must be used to direct and control test execution, to store and retrieve test artifacts, and to capture/analyze/report test results.
- The tool suite must include a tool of choice for defect tracking and resolution.
- The tool suite must include a component for test requirements management.
- The tool suite must include a configuration management tool of choice.
- The configuration management tool of choice must be used to put manual and automated test artifacts under configuration management.
- All of the tools described above must be integrated with desktop tools such as MS Office.





- The proper automated testing workspaces must be created on test servers that are separate from development servers.
- The required test engineer desktop-script-development configuration must be defined and implemented.
- Testing standards must be documented and followed.

Constraints These constraints limit the success of the automation effort if they are not heeded.

- The automated tools group resources must remain independent of any manual testing group.
- There may not be a large enough number of available staff on the automation team.
- The level of cooperation of the software development group and their management with respect to automated tool use may be too low.
- There may be a lack of cooperation and information exchange with developers in creating testable applications.
- The release schedules for major versions of the AUT and for customer-specific releases of the AUT can be too tight.
- There is uncertainty associated with the GUI updates in AUT.
- There may be corporate mandates on what tools must be used.

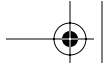
Critical Success Factors We based the following critical success factors on a set of test automation guidelines developed by Nagle (7).

- Test automation must be implemented as a full-time effort, not a side-line.
- The test design process and the test automation framework must be developed as separate entities.
- The test framework must be application independent.
- The test framework must be easy to expand, maintain, and enhance.
- The test strategy/design vocabulary must be framework independent.
- The test strategy/design must hide the complexities of the test framework from testers.

Strategic Objectives These objectives are based on the critical success factors listed above.

- Implement a strategy that will allow tests to be developed and executed both manually (initial test cycle) and via an automation framework (regression test cycles).





- Separate test design and test implementation to allow test designers to concentrate on developing test requirements, test planning, and test case design while test implementers build and execute test scripts.
- Implement a testing framework that both technical and nontechnical testers can use.
- Employ a test strategy that assures that test cases include the navigation and execution steps to perform, the input data to use, and the expected results all in one row or record of the input data source.
- Realize an integrated approach that applies the best features of keyword-driven testing, data-driven testing, and functional decomposition testing.
- Implement an application-independent test automation framework.
- Document and publish the framework.
- Develop automated build validation (smoke) tests for each release of the application.
- Develop automated environmental setup utility scripts for each release of the application.
- Develop automated regression tests for
 - ✕ GUI objects and events
 - ✕ Application functions
 - ✕ Application special features
 - ✕ Application performance and scalability
 - ✕ Application reliability
 - ✕ Application compatibility
 - ✕ Application performance
 - ✕ Database verification

Test Automation Framework Deliverables _____

The following represents a minimal set of test automation framework artifacts that must be created in order to assure success.

- An integrated suite of automated tools that can be used by both technical and nontechnical individuals to test application software
- A strategy for training and periodic retraining of framework users
- A set of reusable test scripts and test script utilities
 - ✕ Automated environmental setup utility scripts





- ✗ Automated smoke test scripts
- ✗ Automated GUI test scripts
 - ✓ Events and objects
 - ✓ Object properties
- ✗ Data-driven automated functional test scripts
 - ✓ GUI-level data validation
 - ✓ Server-level data validation
- ✗ Automated reliability test scripts
- ✗ Automated compatibility test scripts
- ✗ Application performance test scripts
- ✗ Automated test utility libraries (files that contain reusable called procedures and functions) to implement activities such as pretest database loading and posttest database verification

An Automation Plan

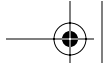
Some do not believe that a plan for automating software testing activities is necessary. In fact, it has been said that such a plan is a waste of time and money and that it can impede an automation effort. Our experience has been that it is important to go through the exercise of writing a plan because it directs your thinking, and, if you follow a plan template, it reduces the chances of omitting important details. Appendix C illustrates an automation plan that was developed for one major company. You could say that it was a waste of time from a management perspective because the plan was submitted to executive-level management in the IS group and was never heard from again. It was not a waste of time for those of us who had to get the effort moving; it gave us guidance and perspective concerning what we thought we could accomplish given the time we had.

Appendix D is a template for a test automation project work breakdown plan. Even if you do not write a formal work breakdown plan, you should at least ponder what you are going to do for each of the areas that are listed in the template.

Categories of Testing Tools _____

A number of different types of automated and manual testing tools are required to support an automated testing framework. Marick has catego-





rized them in a manner that makes sense because it is based on when and how they are used during testing (5).

Test Design Tools. Tools that are used to plan software testing activities. These tools are used to create test artifacts that drive later testing activities.

Static Analysis Tools. Tools that analyze programs without machines executing them. Inspections and walkthroughs are examples of static testing tools.

Dynamic Analysis Tools. Tools that involve executing the software in order to test it.

GUI Test Drivers and Capture/Replay Tools. Tools that use macrorecording capabilities to automate testing of applications employing GUIs.

Load and Performance Tools. Tools that simulate different user load conditions for automated stress and volume testing.

Non-GUI Test Drivers and Test Managers. Tools that automate test execution of applications that do not allow tester interaction via a GUI.

Other Test Implementation Tools. Miscellaneous tools that assist test implementation. We include the MS Office Suite of tools here.

Test Evaluation Tools. Tools that are used to evaluate the quality of a testing effort.

Appendix B is a list of automated testing terms and definitions included for your convenience.

Conclusion

Your job is to look at each testing sphere and, given the scope, goals, and objectives of your organization's automation effort, to decide what category(ies) of test tools (manual or automated) should be implemented to support that sphere. You will find that you have to make compromises and concessions and that an ideal test automation framework is just that: an idea. What you will finally implement is a *mélange* of tools and techniques that are appropriate for your needs.



References

1. Bender, Richard. *SEI/CMM Proposed Software Evaluation and Test KPA*. Rev. 4, Bender and Associates, P.O. Box 849, Larkspur, CA 94977, April 1996.
2. Dustin, Elfriede, Jeff Rashka, and John Paul. *Automated Software Testing*. Addison-Wesley, Reading, MA, 1999.
3. Humphrey, W. S. *Managing the Software Process*. Addison-Wesley, Reading, MA, 1989.
4. Krause, Michael H. "A Maturity Model for Automated Software Testing." *Medical Device and Diagnostic Industry Magazine*, December 1994.
5. Marick, Brian. "Testing Tools Supplier List." www.testingfaqs.org/tools.htm
6. Mosley, Daniel J. *Client-Server Software Testing on the Desktop and the Web*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.
7. Nagle, Carl. "Test Automation Frameworks." Published at members.aol.com/sascanagl/FRAMESDataDrivenTestAutomationFrameworks.htm
8. Pettichord, Bret. "Seven Steps to Test Automation Success." Rev. July 16, 2000, from a paper presented at STAR West, San Jose, November 1999. Available at www.pettichord.com
9. Powers, Mike. "Styles for Making Test Automation Work." January 1997, Testers' Network, www.veritest.com/testers'network
10. Rational Software Corporation. *Rational Unified Process 5.1, Build 43*. Cupertino, CA, 2001.
11. The Software Engineering Institute, Carnegie Mellon University. "Software Test Management: A Key Process Area for Level 2: Repeatable." Available in the "Management Practices" section of www.sei.cmu.edu/cmm/cmm-v2/test-mgt-kpa.html
12. Strang, Richard. "Data Driven Testing for Client/Server Applications." Fifth International Conference on Software Testing, Analysis and Reliability (STAR '96), pp. 395-400.
13. Weimer, Jack. *Manual Test User Interface Program*. Touch Technology International, Inc., www.touchtechnology.com. Available from Phoenix, Arizona, SQA Users Group. This code is free and can be passed on to anyone who wishes to use it, provided the copyright, credits, and instructions stay with the code. www.quanonline.com/phoenix_sqa/tips.html
14. Wiegers, Karl E. "Read My Lips: No New Models!" Whitepaper, Process Impact, (716)377-5110, www.processimpact.com
15. Zambelich, Keith. "Totally Data-Driven Automated Testing." Whitepaper, Automated Testing Specialists (ATS), www.auto-sqa.com/articles.html