Paper on Software Test Automation- Architecture and Implementation
Submitted at

Testing Conference 2006

By
Sreenivasa Pisupati
Senior Vice President
W3Softech India Private Limited
sreenivasa.pisupati@w3softech.com

Table of Contents

1. Software Test Automation- Architecture and Implementation

1.1 Abstract

Test automation becomes an increasingly critical and strategic necessity to keep up with the new explosive pace of web-enabled deployment while retaining satisfactory test coverage and reducing risk.
Automated testing is best used when tests must be run many times. Regression testing often requires that tests be performed repeatedly with each successive software build. Therefore, the use of an automated testing tool is often worth the effort required to automate because of the significant reduction in the amount of manual effort to rerun tests. Furthermore, because automated testing is more accurate and reliable than manual testing, better test results can be achieved that can be consistently reproduced.

This paper explains the automation architecture, objectives, strengths, guidelines, frameworks and tools. Automated testing is, however, not free. Extra work will be needed to prepare test scripts in a form that they can be executed automatically using the automated testing tools available to the project. The extra effort required to automate should be estimated and compared with the anticipated benefits of automation to justify the added overhead. If the effort to automate the test scripts and maintain those test scripts cannot be justified by an overall reduction in work effort and an increase in testing throughput, then automated testing may not be justified. Test automation must be approached as a full-blown software development effort in its own right. Without this, it is most likely destined to failure in the long term.
The test automation strategy to be used should be thoroughly considered during the test planning and test design phases and documented in the test plan. Sufficient numbers of resources skilled in the use of the automated testing tools and framework should be allocated to the testing initiative to provide adequate capacity to complete the testing and to support the automation effort. Insufficient allocation of resources to support automation may result in successfully automating some tests but failing to fully test the application. A more reasonable approach may include a combination of manual as well as automated testing. The payback from automation is realized during regression testing only after an initial investment in script development and maintenance.

1.2 Automation Objectives

The objectives for employing automated testing techniques to facilitate testing are listed below:

- Test cycles are usually compressed because IT projects are market-driven, not effort-based. Automation addresses this reality by improving the regression testing cycle times to shorten the overall Systems Development Life Cycle (SDLC)
- Testing teams are usually under-staffed. Automation allows greater breadth and depth of coverage with fewer dedicated resources.
- Automation increases coverage and saves execution time, thus allowing dedicated resources the time for more analysis, design, and development of new tests for additional functionality.
- Increased coverage reduces cost. Test teams can find defects earlier and avoid costly production problems.

- Improve traceability of the testing process, spanning requirements, test planning, test execution, and defect tracking.
- Improve the quality of the applications by allowing more robust scripts to be executed under shorter time frames.
- Improve the overall testing cycle by providing a central repository for requirements, test plans, test cases, and collateral documentation.

- Provide seamless integration between manual and automated test processes.

1.3 Automation Strengths

- Ability to execute repetitive tasks accurately
- Ability to exactly repeat test conditions and steps
- Ability to deal with vast amounts of data and verification steps
- Ability to create load and stress on a system

Automation may not be a good solution for:

- Ad hoc or unstructured tests
- Exploratory tests
- Tests with constantly changing steps or dependent applications

1.4 Good Test cases for Automation

- **A good manual test case**
- **Fully stepped**

Tests that are to be automated should be tests that have detailed steps. The idea is that a reasonably experienced tester, who has little or no prior experience with the product being tested, could run the test with limited assistance. Many teams rely on the expertise and experience of their testers to “fill in the gaps” in a test case. While this practice can be necessary in emergency situations, it can actually introduce risk into the testing process if experienced testers are not available for a particular test cycle. Also, not maintaining full test cases limits the opportunity for cross-training and will accelerate tester burnout. It is always best practice to have a fully stepped test case.

- **Contain any data needed to complete it the test**

Additionally, a test case is not complete unless the data sets necessary to complete the test are included in a data or parameter table and included with the test.

- **Easily maintained and designed in a modular fashion that encourages re-use**

A modular test case can be created any time a test has a series of common steps with other similar test cases.

- **Maintained using version control to trace changes and reviewed for accuracy and completeness, by either peers or test customers**

Version control should be enabled so that only one tester can edit a specific test case at any given time. Also, all changes in a test case are captured and a version record is created so that changes in the test case can be reviewed and restored if necessary. The functionality also allows for a potential policy change that might benefit teams. Some teams already choose to perform test case reviews before allowing a new check in on a shared test case or module. These reviews should be conducted in a systematic fashion by either peer on the testing team, by the project management, or development staff attached to the project. It is "best practice" to perform

peer reviews on all manual tests. It is even more important to do these reviews when tests are modular, as more than one tester might be using the test as a part of his or her testing effort. Also, it is a good idea to have another person test keyword automated tests so that they can be assured of running on more than one machine. While version control for both manual and automated scripts is mandatory, peer review of the test cases is not.
When the complexity of the test environment increases, and the number of testers involved with the effort likewise increases, it is wise to manage change in the scripts themselves. However, peer review is standard for anyone making changes to the code layer of the automation framework through a code review and check in process.

- **Often repeated functionality**

1.4.1 Criteria

- Test script requires the verification and validation of multiple attributes, objects and components
- Test script needs to be executed with external data that resides in a database system or spreadsheet
- Business process to be automated has a finite number of input values and fields and was constructed with an orthogonal array to provide coverage for all its permutations.
- Test script will be used for security testing
- Test script will be used for either one or all of these: stress, volume, load, soak, performance testing (add a point for each type of test that it meets)
- Test script will be used for regression, functional testing
- Test script will be used to validate values, calculations, etc, displayed on a customized report, or online report
- Test script will be repeated many times or is highly repetitive (i.e. Test script will need to be executed multiple times on different software releases or builds)

1.5 Guidelines for Automation

- Full time dedicated resources should be allocated for test framework design and coding
- The test framework must be well thought out and documented
- The test design and the test framework are totally separate entities
- The test design details how the particular functions of our application will be tested. It will tell us what to do, how and when to do it, what data to use as input, and what results we expect to find. All of this is specific to the particular application being tested. The test automation framework is an execution environment for the automated tests. The skill set for each one of these is completely different.
- Application independent Test Framework
  Remove all application-specific contexts and reuse virtually everything we develop for every application that comes through the automated test process. This framework should handle all the details of ensuring we have the correct window, verifying that the element of interest is in the proper state, performing the action on the element, and logging the success or failure of the entire activity. This does not mean we will never have to develop application-specific test scripts, but does limit it to a very small percentage.
- Test Framework to deal with the development of common components for different applications
- Development of Test Framework with module independence

Each module should be independent and separate from all other modules. The functionality available within each module can be readily expanded without affecting any other part of the system

- The test strategy/design vocabulary should be framework-independent
- Test Framework should distinguish between test designers and Automation team.

1.6 Automation Frameworks

Modern automation test applications have “record-playback” functionality that allow for script code to be generated as a manual tester tests, but this technology has limitations. we cannot and *must not* expect pure capture and replay of test scripts to be successful for the life of a product. For example, many testers record unnecessary steps, alternatively if the application changes only slightly then the recorded test script fails to run. The generated scripts can be difficult to maintain and require that a tester understand the program code to modify them.

1.6.1 Data Driven Automation Frameworks

Data driven scripts are those application-specific scripts captured or manually coded in the automation tool’s proprietary language and then modified to accommodate variable data. Variables will be used for key application input fields and program selections allowing the script to drive the application with external data supplied by the calling routine or the shell that invoked the test script.
A data-driven test is automated test case that executes identical steps each time, but with different data values (or parameters) for each run. The test will run once for each set of parameters in the data table that is associated with it. Data driven tests rely on parameters. Once the expected outcome of the script is identified and automated, the data values are replaced by parameters. These parameters are represented by column headings in an Excel file, and the data values can be listed under those headings. The automation framework will then re-run the test for each row of data values entered in that spreadsheet. This enables many combinations of data values to be tested in a short period of time, and without the need for repetitive data entry. Data driven, modular and keyword test case design strategies can be combined. Modular data-driven keyword tests, while difficult to design, allow for one tester to execute many combinations and thus, exercise many code paths within an application under test. It is an advanced technique, but has been demonstrated to be not only possible, but maintainable by a tester with sufficient training in the technology and test case design.

Variable Data, Hard Coded Component Identification:
These data driven scripts often still contain the hard coded and sometimes very fragile recognition strings for the window components they navigate. When this is the case, the scripts are easily broken when an application change or revision occurs. And when these scripts start breaking, we are not necessarily talking about just a few. We are sometimes talking about a great many, if not all the scripts, for the entire application.

The following is an example of activating a server-side image map link in a web application with an automation tool scripting language:

Image Click "DocumentTitle=Welcome;\;ImageIndex=1" "Coords=25,20"

This particular scenario of clicking on the image map might exist thousands of times throughout all the scripts that test this application. The preceding example identifies the image by the title

given to the document and the index of the image on the page. The hard coded image identification might work successfully all the way through the production release of that version of the application. Consequently, testers responsible for the automated test scripts may gain a false sense of security and satisfaction with these results.
However, the next release cycle may find some or all of these scripts broken because either the title of the document or the index of the image has changed. Sometimes, with the right tools, this might not be too hard to fix. Sometimes, no matter what tools, it will be frustratingly difficult. Remember, we are potentially talking about thousands of broken lines of test script code. And this is just one particular change. Where there is one, there will likely be others.

Highly Technical or Duplicate Test Designs:
Another common feature of data driven scripts is that virtually all of the test design effort for the application is developed in the scripting language of the automation tool. Either that or it is duplicated in both manual and automated script versions. This means that everyone involved with automated test development or automated test execution for the application must likely become proficient in the environment and programming language of the automation tool. A test automation framework relying on data driven scripts is definitely the easiest and quickest to implement if you have and keep the technical staff to maintain it. But it is the hardest of the data driven approaches to maintain and perpetuate and very often leads to long-term failure.

1.6.2 Keyword or Table Driven Test Automation

Keyword Automation is an attempt to put the technology and control of a custom scripted automation into the hands of the testers who know the application the best. Once testers understand keyword automation, they can write and maintain their own automation suites without writing program code. The program code can be maintained by specially trained automation engineers or even the software engineers who are writing the application being tested. Also the keyword cases can be run as a manual test case, eliminating the need for separate test cases representing a manual or automated process.

Process Champions

It will be necessary to identify the people who will be responsible for automation in the teams. Training will still be necessary for keyword automation, and there is still a significant learning curve for the manual testers who may or may not be experts on the environment in which the testing is being accomplished. The people selected should at very least be testers who have a full understanding of the product that is being tested and can write and maintain test cases. Any previous programming or scripting experience will also prove helpful, but is not necessary for correct use of the Keyword Automation.

Action Steps:

1. Analyze current test coverage
2. Identify areas that would benefit from automation
3. Develop an automation plan and select cases for automation
4. Identify resources who can be made available to implement automation (Process Champions)
5. Schedule training
6. Begin automation

Standards and Procedures

Manual to automated script conversion

As a part of a comprehensive automation plan, it may be possible to convert existing manual test cases to keyword automated cases. Most groups will quickly discover that the level of detail necessary for an automated keyword test is much higher than the level of detail present in typical manual tests. There are many steps that the computer needs to be instructed to take that our brains automatically fill in for us as we are reading. So, there will not be a 1 to 1 relationship between steps in a manual test and a keyword automated test.

Modular Test Case Design

The goal of modular test case design is to reduce the duplication of effort that goes into maintaining test procedures. Many test cases have groups of shared steps that can be combined into a module. These modules can then be called from main tests and function as if those steps are simply part of the test that calls them. So, if there is a change in the application that the module is testing the change only needs to be made in the module, and not in the tests that use it.
It is typically an application-independent automation framework designed to process our tests. These tests are developed as data tables using a keyword vocabulary that is independent of the test automation tool used to execute them. This keyword vocabulary should also be suitable for manual testing.

Action, Input Data, and Expected Result ALL in One Record:
The data table records contain the keywords that describe the actions we want to perform. They also provide any additional data needed as input to the application, and where appropriate, the benchmark information we use to verify the state of our components and the application in general.
For example, to verify the value of a user ID textbox on a login page, we might have a data table record as seen in Table 1:

WINDOW	COMPONENT	ACTION	EXPECTED VALUE
LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Table 1

Reusable Code, Error Correction and Synchronization:
Application-independent component functions are developed that accept application-specific variable data. Once these component functions exist, they can be used on each and every application we choose to test with the framework.
Test Design for Man and Machine, With or Without the Application:
Table 2 reiterates the actual data table record run by the automation framework above:

WINDOW	COMPONENT	ACTION	EXPECTED VALUE

LoginPage	UserIDTextbox	VerifyValue	"MyUserID"

Table 2

Note how the record uses a vocabulary that can be processed by both man and machine. With minimal training, a human tester can be made to understand the record instruction as deciphered below:

On the LoginPage, in the UserIDTextbox,
Verify the Value is "MyUserID".

Once they learn or can reference this simple vocabulary, testers can start designing tests without knowing anything about the automation tool used to execute them.
Another advantage of the keyword driven approach is that testers can develop tests without a functioning application as long as preliminary requirements or designs can be determined. All the tester needs is a fairly reliable definition of what the interface and functional flow is expected to be like. From this they can write most, if not all, of the data table test records.

Keyword Driven Automation Framework Model

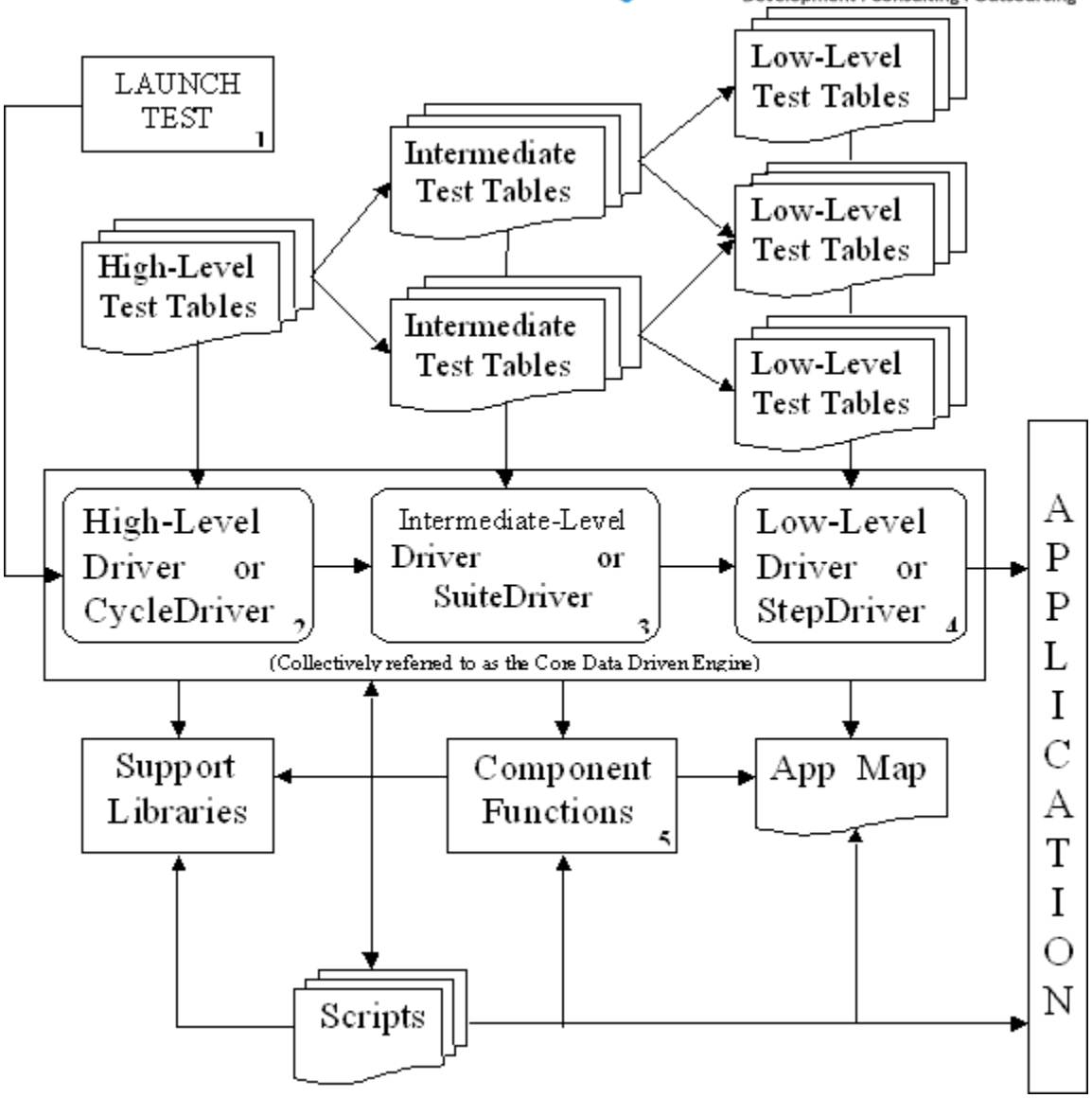


Figure 1

In brief, the framework itself is really defined by the Core Data Driven Engine, the Component Functions, and the Support Libraries. While the Support Libraries provide generic routines useful even outside the context of a keyword driven framework, the core engine and Component Functions are highly dependent on the existence of all three elements.

The test execution starts with the LAUNCH TEST(1) script. This script invokes the Core Data Driven Engine by providing one or more High-Level Test Tables to CycleDriver(2). CycleDriver processes these test tables invoking the SuiteDriver(3) for each Intermediate-Level Test Table it encounters. SuiteDriver processes these intermediate-level tables invoking StepDriver(4) for each Low-Level Test Table it encounters. As StepDriver processes these low-level tables it attempts to

keep the application in synch with the test. When StepDriver encounters a low-level command for a specific component, it determines what Type of component is involved and invokes the corresponding Component Function(5) module to handle the task.
All of these elements rely on the information provided in the App Map to interface or bridge the automation framework with the application being tested.
Findings:
The keyword driven automation framework is initially the hardest and most time-consuming data driven approach to implement. After all, we are trying to fully insulate our tests from both the many failings of the automation tools, as well as changes to the application itself.
To accomplish this, we are essentially writing enhancements to many of the component functions already provided by the automation tool: such as error correction, prevention, and enhanced synchronization.
Fortunately, this heavy, initial investment is mostly a one-shot deal. Once in place, keyword driven automation is arguably the easiest of the data driven frameworks to maintain and perpetuate providing the greatest potential for long-term success.

1.6.3 Hybrid Test Automation

The most successful automation frameworks generally accommodate both keyword driven testing as well as data driven scripts. This allows data driven scripts to take advantage of the powerful libraries and utilities that usually accompany a keyword driven architecture.
The framework utilities can make the data driven scripts more compact and less prone to failure than they otherwise would have been. The utilities can also facilitate the gradual and manageable conversion of existing scripts to keyword driven equivalents when and where that appears desirable.
On the other hand, the framework can use scripts to perform some tasks that might be too difficult to re-implement in a pure keyword driven approach, or where the keyword driven capabilities are not yet in place.

1.7 Designing Automated Test Environment

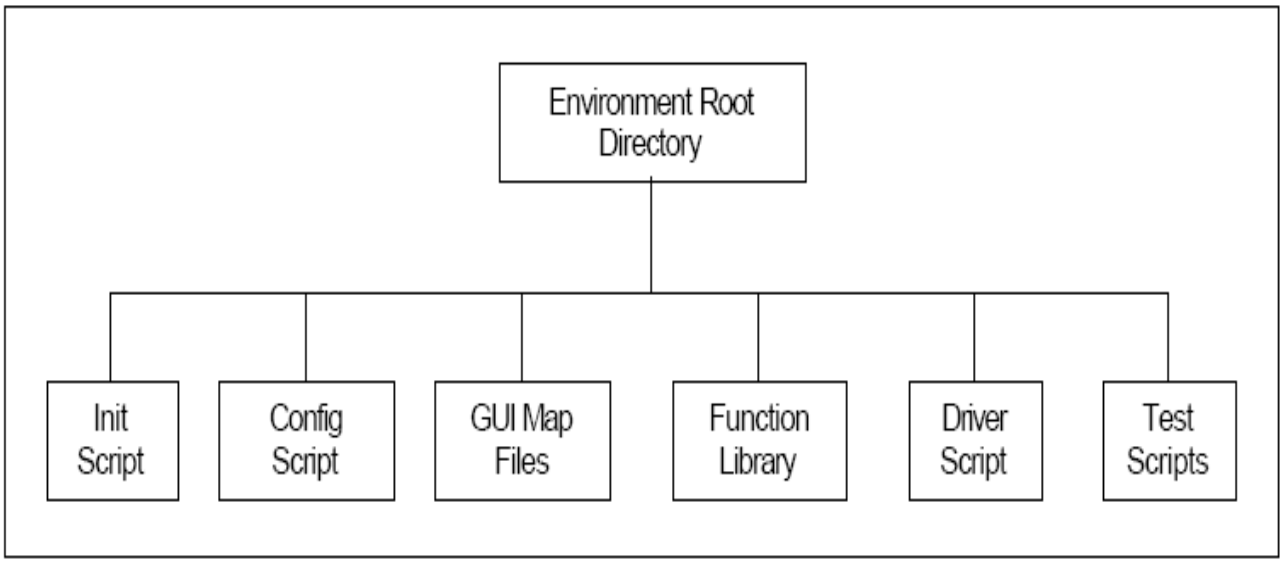
1.7.1 Considerations in Designing Automated Test Environment

Automation should be treated like mini software development projects with due consideration to the following points.
Script maintainability and flexibility
The automation environment and scripts need to be designed in a way that minimizes the amount of work involved in maintenance.
Script performance
If the performance of automated tests is ignored or taken for granted, the time saved by running automated tests can be drastically reduced.
The different levels of testing
Depending upon the nature of a build, the time given to test it may vary from a few minutes to a few days (may be more). Full regression may not always be an option. Only certain types of testing actions might be allowed, and these different types of actions comprise different levels of testing.
The number of environments testing takes place in
A software project may have several different environments that testing occurs in. Most have at least the following three:
1. Functional test environment
2. Development environment

3. Production environment
Each environment may be on a different machine, connect to a different database, and limit the types of testing that can be executed. The user shouldn't have to go to several different places to set up several different parameters every time the automation test suite is ready to be run.

1.7.2 Design Steps

Create the appropriate directory/test suite structure



Component Definitions:
Init Script: brings the environment to a controlled stable point.
Configuration Script: sets parameters needed for the test run.
GUI Map Files: an interpreter that enables the automation tool to communicate with the AUT.
Function Lib: set of functions used in the test suite.
Driver Script: coordinates test activities by calling other scripts.
Test Scripts: actually perform tests, and verify that the AUT works as desired

1.8 Automation Tools

1.8.1 Tools Evaluation

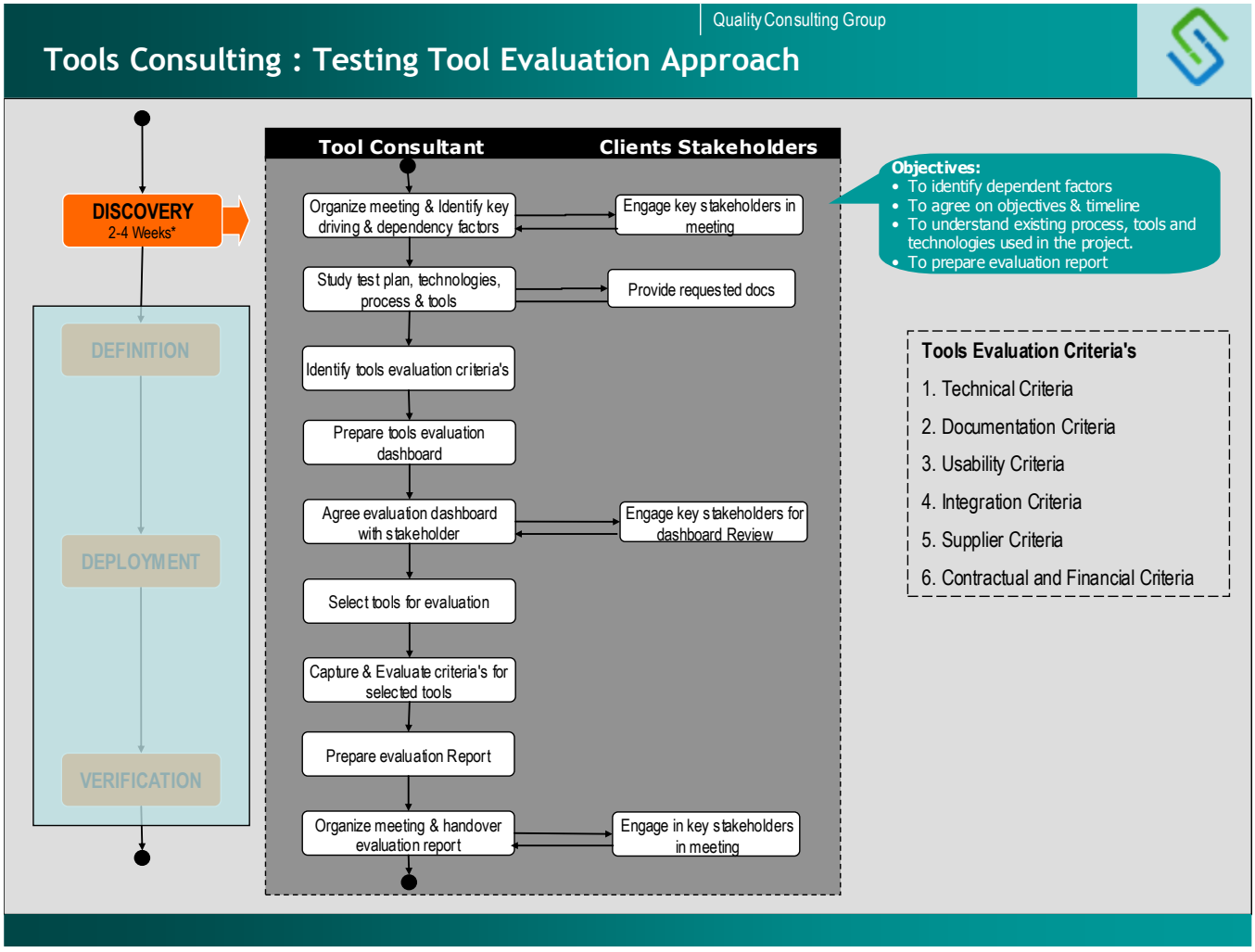


Figure 2

1.8.1.2 Evaluation Matrix

Functional Test Tool Matrix

The Tool Matrix is provided for quick and easy reference to the capabilities of the test tools. Each category in the matrix is given a rating of 1 – 5. 1 = Excellent support for this functionality, 2 = Good support but lacking or another tool provides more effective support, 3 = Basic/ support only. 4 = This is only supported by use of an API call or third party add-in but not included in the general test tool/below average, 5 = No support. In general a set of criteria can be built up by using this matrix and an indicative score obtained to help in the evaluation process. Usually the

lower the score the better but this is subjective and is based on the experience of the author and the test professionals opinions used to create this document.

	Record & Playback	Web Testing	Database tests	Data functions	Object Mapping	Image testing	Test/Error recovery	Object Name Map	Object Identity Tool	Extensible Language	Environment support	Integration	Cost	Ease of use	Support	Object Tests
WinRunner																
QA Run																
Silk Test																
Visual Test																
Robot																

Table 3

2.0 Case Studies

Using Rational Robot

System Testing

Case Study: Rational Robot

- Project : E&PE Domain
- Test Cases : 1399
- No of Regressions : 3
- Manual Effort : 18 PW
- Tool Effort : 4.2 PW
- 76% Effort Saving Using Rational Robot for Functionality testing for 3 runs of regression

Benefits:-

- Good user-friendly tool to perform functional testing when coupled with a well-structured automation framework.
- Easy identifications for all common GUI objects such as menus, lists etc

Challenges:-

- Verification points based on Window Images are prone to failures
- Window Image based VPs need to be avoided as much as possible

18

Figure 3

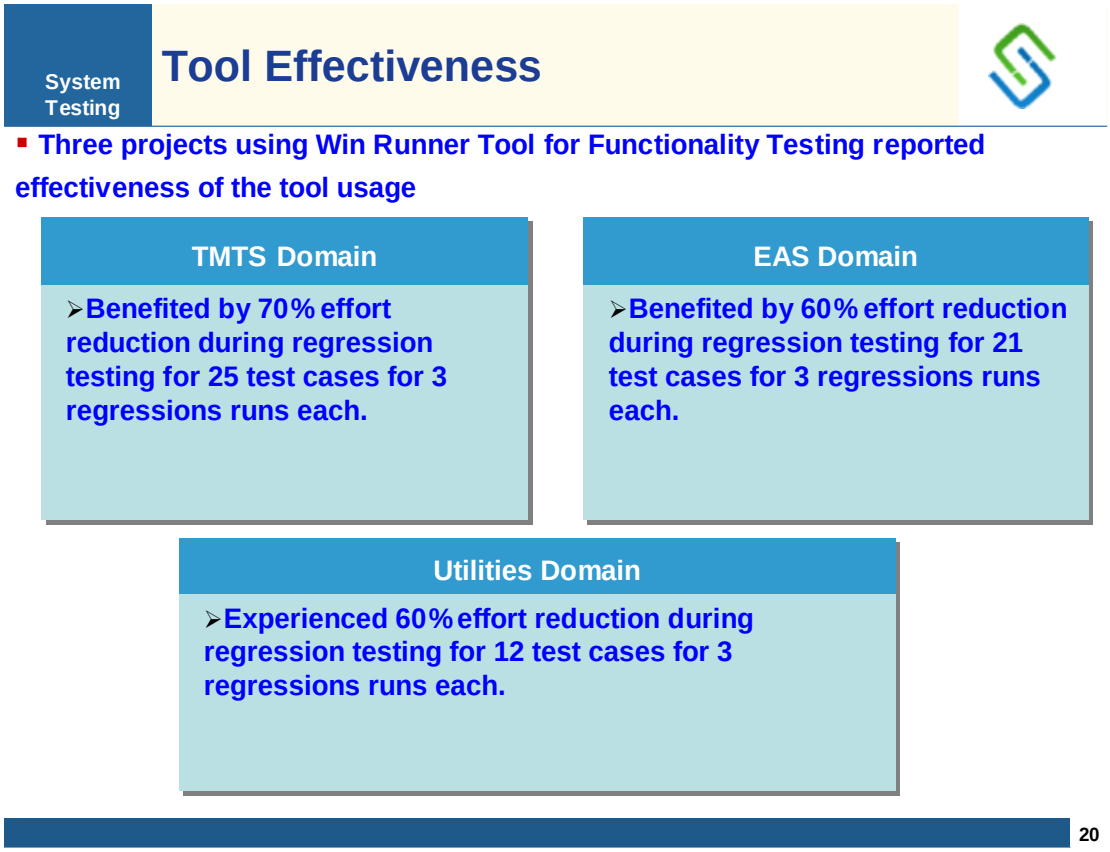

System Testing
Tool Effectiveness
Three projects using Win Runner Tool for Functionality Testing reported effectiveness of the tool usage
TMTS Domain
Benefited by 70% effort reduction during regression testing for 25 test cases for 3 regressions runs each.
EAS Domain
Benefited by 60% effort reduction during regression testing for 21 test cases for 3 regressions runs each.
Utilities Domain
Experienced 60% effort reduction during regression testing for 12 test cases for 3 regressions runs each.
20

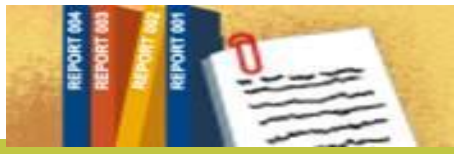
Figure 4

Using WinRunner, QuickTestPro and LoadRunner

Case Study – CRM Testing using Automation

Client profile

Client is a leading Financial Services Corporation in USA with over 29000 Employees, Headquarters at California and has multiple Enterprise CRM Projects spanning various divisions and a Central QA/ Emerging Technologies QA team responsible for all CRM QA activities



Situation

Challenges facing the client

- Few Resources to handle QA Effort for 6+ CFC CRM Projects (Bank, CLD, OOP, Claim source, PRM, Marketing etc..)
- Multiple CRM project releases in a short period of time
- Need to reduce QA cycle time and costs
- Remove QA person dependencies
- Reduce Time to Market
- Measure/Improve CRM system performance

What we did

- **Onsite/Offshore Model**
- 1 Test Consultant at Onsite (25 – 50 % Time), 1 Dedicated Offshore Consultant
- Develop Automated Scripts (Offshore), Test & Execute Scripts (Onsite)
- **Streamlined Test Management Process**
- Testdirector usage across all CRM projects. Served as central repository for creating, running, collecting, analyzing, maintaining test cases. Also used as Defect tracking system
- **Automated Functional Test cases**
- QuickTestPro used as Functional Test Automation tool. Projects to be automated decided on case by case basis
- **Removed Performance bottlenecks**
- Load Runner used as Load Testing tool. Performance bottlenecks identified and removed
- Topaz monitoring scripts were used to identify system availability

Results

Business benefits

- Increased productivity and reduced risk by automating manual tasks in system life cycle
- Reduced time to market by Automating functional testing
- Improved quality through repeatable sustainable processes

Reduced QA Effort & Time

Office of President CRM Application

- 3 Successful run of all automated test cases
- 76% of Test scripts automated (120/158)
- Execution time down from 8 days/2 Consultants to 4 days/1 Consultant)

CLD CRM application

- 2 Successful run of all automated test cases
- 85% of Test scripts automated
- Test execution time down from 3 days to ½ day

Bank CRM application

- 1 Successful run of all automated test cases
- 75% of Test scripts automated

Figure 5

3.0 References

1. Software Test Automation: Effective Use of test Execution tools, Fewster & Graham Addison-Wesley, 1999

2. Effective Methods for Software Testing by William E. Perry, John Wiley & Sons, © 2000
3. The Art of Software Testing by Glenford J. Myers, John Wiley & Sons , © 1979
4. Automated Software Testing: Introduction, Management and Performance by Elfrieda Dustin, Jeff Rashka and John Paul
5. Automated Testing Handbook by Linda Hayes
6. Effective Software Test Automation by Kanglin Li, Mengqi Wu
7. Test Automation Snake Oil by James Bach

Web site: Automated Testing
This site is dedicated to presenting information and facilitating dialogue about Automated Testing.
http://www.automated-testing.com/

Author's biography
Name: Sreenivasa Pisupati
Current Position: Senior Vice President, W3Softech India Pvt Limited.
Previous Positions:
Group Manager-QA, Oracle Corporation, India
Head-Testing services- LG Soft India
Test Consultant in Vancouver, BC. Canada
Test Manager, Ensim Corporation, Sunnyvale, California. – QA
Test Lead, Hyperion Solutions, Sunnyvale, California.

Educational Qualifications:
M.Tech in Industrial engineering & management with 8.09 CGPA on 10 point scale from Indian Institute of Technology (IIT) Kharagpur, India. (Specialization in Information systems)
B.Tech in Chemical Engineering with 1st class from Andhra University, India

Certifications:
Certified quality analyst (CQA) from Quality Assurance Institute (QAI)

Certified Course in WinRunner Web from Mercury Interactive, Sunnyvale, California

Certified Course in Test Director from Mercury Interactive, Sunnyvale, California

Certified ITSM (ITIL Foundation course) Professional from ISEB

Total years of experience:
More than 18.0 years IT experience with more than 14 years in Software Testing and Process consulting. More than 5 years testing in product development companies in US, Canada and Singapore

4.0 Appendix

Specific terms, abbreviations, and acronyms used within this document are defined here to facilitate understanding of the information presented herein.

Automated Testing	This is a large part of the validation process. To determine what has been developed works as anticipated, automated testing tools are advantageous. Although vendors may promise utopia when using these tools, enterprises must realize that the use must be technical in nature .Gartner advocates parity between the developer and the test engineer. Along with the professional parity comes the parity of the tools.

Defect	A discrepancy between system operation and system specification. Synonymous with Bug. Another definition: Deviation of the implementation with respect to the system requirements.
Functional Testing	Application of test data derived from the specified functional requirements without regard to the final program structure. Functional testing is a subset of black-box validation that compares actual behavior to the functional specification.
Integration Testing	For simplicity, Integration Testing is the testing of two or more integrated software components distributed across multiple platforms (e.g., client, web server, application server and database server) within the same system or across external interfacing systems.
Load Runner	LoadRunner is used to simulate multi-user concurrent sessions to verify an application is ready for a production environment. It is the primary load testing tool and can be used to simulate up to 4000 virtual users at the enterprise level allowing us to load, stress, baseline or volume test web, client/server, terminal emulator, mail services, and COM type applications.
Pass/Fail Criteria	Decision rules used to determine whether a software item or feature passes or fails a test.
Quality Assurance	The planned and systematic activities implemented in a quality system so that quality requirements for a product or service will be fulfilled.
Record/Playback	Record/playback tools are part of automated and functional testing. These tools reside locally on the desktop, capture every input and then may be played back to identify behavior. Record/playback sounds simple, but it requires scripting; therefore, it may only be cost-effective if the application's user interface is stable. Record/playback ensures that the application does what is recorded, which may or may not be intended.
Recovery testing	Recovery testing validates that the system recovers from major exception conditions without loss of data.
Regression Testing	Testing with the intent of determining if bug fixes have been successful and have not created any new problems. Also, this type of testing is done to ensure that no degradation of baseline functionality has occurred.
Requirements	A set of specific criteria that a system must meet in order to be acceptable to the clients. Source: TestDirector User Guide and Defect Tracking Guide
Stress / Load / Volume Testing	Determine the limits of the application by simulating or replicating actual performance in the intended production environment. Re-create a true

	production environment during system testing to catch errors that might otherwise go undetected and impact existing production applications. This is testing with the intent of determining how well the product handles when a load is placed on the system resources that nears and then exceeds capacity. To test and measure the network traffic, CPU utilization or memory allocation, etc.
System Testing	Testing two or more modules or functions together with the intent of finding interface defects between the modules or functions. Testing that attempts to discover defects that are properties of the entire system rather than of its individual components.
Test Case	A specific set of test data and associated procedures developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. Each test case consists of at least 4 components: Description/Objective, Inputs, Procedure and Expected Result. Source: TestDirector User Guide and Defect Tracking Guide.
Test Coverage	Percentage of the system tested in project selected elements. Elements could include requirements, unique paths, functions, decision points, lines of code, units, routines, hardware components, etc.
Test Cycle	A logical grouping of test cases/scripts. This grouping is made according to the convention that makes sense for the project. Source: TestDirector User Guide and Defect Tracking Guide.
Test Plan	A document based upon the testing strategy, prescribing the steps to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning.
Test Script	Commonly used to refer to the instructions for a particular test that will be carried out by an automated test tool.
Test Set	A series of related test cases combined to test specific system requirements, or to perform a specific category of testing. A test set can be created within Test Director, and results reported for the execution of the test set. Source: TestDirector User Guide and Defect Tracking Guide.
Test Specification	A list of all tests and a brief description of each. Source: TestDirector User Guide and TestDirector Defect Tracking Guide.
Test Step	The step-by-step instructions for executing a test case.

Test Strategy	A section of the Master Test Plan document that describes the high level approach to achieving the testing goals and objectives.
Testing	The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.
Unit Testing	Unit testing is the first review of software at the individual component level within the development lifecycle.
User Acceptance Testing	Formal testing conducted to determine whether or not a system satisfies its defined acceptance criteria and to enable the customer or its representative to determine whether or not to accept the system.
WinRunner	WinRunner is an automated test tool for GUI/WUI/CHUI applications.