Automated Regression Testing Framework using Keyword Driven Approach

Rajeev Randhawa (rarandhawa@sapient.com)
Lokesh Pandey (lpandey@sapient.com)

Sapient Corporation Pvt. Ltd
Sapient Tower (D & E)
DLF Cyber Greens
DLF Phase III, Sector-25A
Gurgaon 122002
Haryana, INDIA

1 Abstract

A framework approach to create an Automated Regression Test Bed has many benefits over the traditional simple capture/playback or custom scripts methodology. This paper shares our experiences evolving a keyword driven framework approach for automating the regression test bed of scripts. The paper shares a case study on the implementation of a Keyword Driven Framework within Sapient and some data on the benefits reaped by using the approach over a period of a year in two of our projects. The case study lists our organizational requirements for an automated test suite and how the keyword driven framework implementation caters to all of them.
2 Why Automation?

- **Reuse of Tests**: When a certain set of test scripts require frequent execution it makes more sense to automate them and reap the benefits of unattended automated execution of test scripts. Another example will be in cases where a single script needs to be executed with multiple data sets. In such cases the effort to automate a single script and running it with multiple data sets is far less than the manual execution effort for all those data sets.

- **Time Save**: Running unattended automated test scripts saves human time as well as machine time than executing scripts manually.

- **Better use of people**: While automated scripts are running unattended on machines, testers can do more useful tasks.

- **Cost Saving**: On test engagements requiring a lot of regression testing, usage of automated testing reduces the people count and time requirement to complete the engagement and helps reduce the costs.

- **Machines are more reliable than humans**: Great confidence will be gained when a system is released, when we use automated testing.
3 Case study on the usage of Keyword Driven Framework approach in Sapient

When Sapient started in the direction of using Automated Testing, we came up with a set of requirements that we wanted to meet as a result of our effort. The requirements were divided into two buckets: Business & Technical Requirements.

3.1 Requirements for the Automated Suite

3.1.1 Business requirements

3.1.1.1 Maintainability

It should be easy to maintain and update the scripts with newer releases or changes done to the application.

3.1.1.2 Selective testing

- There should be an easy way to only test the breadth of the application without doing a lot of depth testing. This is to be used for stability testing of an application
- There should be an easy way to select test scenarios at a more granular level in order set up more targeted regression

3.1.1.3 Recovery

If test script execution is stopped at any moment for any reason, the test suite should be able to resume testing from that point at any time later. We should not need to re-test functionality that had passed earlier.

3.1.1.4 Scalability

It should be possible to execute tests from different machines at the same time in order to gain time efficiencies

3.1.1.5 Inter-environment portability

The scripts should run on any given test environment without any code change (technical intervention)

3.1.1.6 Usability

- With minimal amount of training, anyone on the team must be able to execute the automated suite
- The script must run without any manual intervention
3.1.1.7 Reporting

- Test results must be reported in a way that lets a user know the status of each test script easily
- Failed scripts must be accompanied by appropriate error messages and screenshot(s) that helps the user understand the exact problem
- As far as possible, the information generated by logs of failed scripts must be necessary and sufficient for the user to log a defect

3.1.2 Technical requirements

3.1.2.1 Data abstraction

The test data needed to execute scripts must be completely isolated from the actual script code. All data needed by scripts must be stored in data sheets, and the process for updating the data sheet must be documented.

3.1.2.2 Maintainability

The automated suite should easily be maintainable for updates to application’s functionality. The following items detail out the technical requirements that will help achieve maintainability.

3.1.2.2.1 Separation of concerns

Distinct layers in architecture must address concerns at each layer. In particular:
- Logging, reporting, exception handling etc must be handled in a separate layer
- Core functionality for testing must be handled in a separate layer

This will help the script author to focus on the core task of writing a script that addresses the business functionality; while delegating exception handling and other non-core tasks to other layers.

3.1.2.2.2 Modular code

Code written should be modular, so that:
- **Reuse is possible** – if a functionality has already been coded, the functions written should be generic enough to be reusable in a different context
- **Centralized changes** – if a functionality changes, the script author must be able to make change to one central function that should suffice for the whole suite

3.1.2.2.3 Portability

The suite must be portable to:
3.1.3 Feasibility Study on the various Automation approaches

The various approaches to Test Automation are listed below:

**Generation 1: Record and Replay**
- Interacts with the application in a similar fashion as an end user
- Usually involve writing an automated script per test script
- Usually involve unrepeatable and undefined processes leading to high maintenance costs

**Generation 2: Custom Scripts**

---

![Diagram of Generation 1: Record and Replay]

**Figure 1 : Generation 1 – Record and Replay**
• Enhancing record and replay scripts to work more than once
• Recording tools usually generate scripts based on the recorded user actions and allow the test developer to enhance the script for expanded functionality
• Libraries of common functions get developed
• Reuse of code starts happening

**Generation 3: Data Driven Testing**

• Application input data are extracted from the test script for maintainability and expansion of scope
• Validation data are maintained separately and are compared with the input data
• Data is stored in flat files, spreadsheets, or databases depending upon the data complexity and the extent to which it needs to be accessed by test scripts

---

**Generation 4: Keyword Driven Framework**

• Overall approach and strategy is defined for all test processes
• Enables writing tests in a more abstract manner using Keyword approach
• Enables a non-technical subject matter expert / business user to design tests without ever touching a script

---

**Figure 2 : Generation 2 and 3 – Custom Scripts & Data Driven**
White Paper on Best Practices

- Allows automation to be started earlier in the SDLC even before a stable application is delivered for testing thus providing more time for testing
- Allows a high degree of reuse among all artifacts: data, code, input files, comparison data, error handling, and reporting mechanisms

![Diagram: Generation 4 – Keyword Driven](image.png)

**Figure 3: Generation 4 – Keyword Driven**

### 3.1.4 The Solution: Keyword Driven Framework Approach

Based on the feasibility study done on the various approaches mentioned in the previous section, Sapient zeroed down on developing a Keyword Driven Framework solution for Test Automation.

This solution approach is motivated by efficient usage of people by using the right level of people skills to perform the various testing tasks. Test design is treated as a separate but integrated activity from test automation, so we can assign differently skilled people to each activity. Test design requires knowledge of the application domain as well as test design techniques, but does not require familiarity with programming. Test automation requires familiarity with programming and optimization techniques.

From the SME / business user perspective, test cases should be designed at a higher level of abstraction with little or no knowledge required on the tool specific programming language. To cater to this need, we used the spreadsheet for test case design in which the test case designer would write the test case in a high level language. A test case processor implemented within the framework will then interpret and process the spreadsheet.

This high-level test design language contains keywords and optional data needed for each keyword. The keywords are the test steps within the test case and generally encapsulate a business event or activity. The keywords are atomic in structure and can be reused in multiple test cases. The optional data provided with a keyword may be a specific test case input or expected results. Keywords may be specific to software under test and may vary from one to another.

The test case processor within the framework comprises of an interpreter containing general functions to read the test cases specified in the spreadsheet. On
encountering a keyword, the function implementing that specific keyword is called and execution results will be written into a test results report.

3.1.4.1 Framework Overview

In the implementation of the Keyword Framework Approach within Sapient, we have used WinRunner as the Record/Replay tool.

A typical script for automation involves a number of actions on a sequence of pages. For example, a script that leads from home page to reaching the accessories page is described by the following steps:

1. Open http://www.yahoo.com
2. Click “Mail” on the header
3. Enter “user id” and “password”
4. Click on “Inbox”

To automate such a flow in a Record/Replay tool, the following steps have to be performed:

1. All GUI objects have to be recognized into a GUI Map file (e.g. windows like “Homepage” and “Inbox”, links like “Mail” and “Inbox”, buttons like “Delete” and so forth)
2. An automated script must now be written, which moves screen by screen, taking relevant actions on each screen (e.g. clicking on a link, checking some text, entering some field values and submitting a form, and so on)
3. Data sheets must be prepared for any inputs required by the script (e.g. what link to click, what text to input). The script must refer to this data sheet for relevant inputs

In writing such a script, a lot of functionality is routine, for example:

1. Opening a new browser window
2. Setting focus on a particular window
3. Clicking on a link that has certain properties (e.g. a “text” property called “Add to cart”)
4. Reading items from a data sheet, and using them as inputs for fields on the page

In each of these operations, while some part is context dependent (e.g. the text of the link to click, or the title of the window to set the focus on), the other part is constant (code for clicking on a link, or setting focus on a window). Besides, a lot of other code (such as – logging the event of click on a link, or reporting an exception if a window of the particular name is not found) remains more or less the same, except for context dependent parameters.

The framework separates out the context dependent part (metadata about actions being performed specifically for a given script) from the more generic framework (which holds common code for all scripts). Users of the framework simply specify metadata about what has to be tested for a given script, while the framework interprets the metadata and calls appropriate Record/Replay tool code for executing the actions.
The following diagram summarizes the key components involved:

- **Core Framework**
  - Generic services: Logging, Exception handling, Reporting...
  - WinRunner wrappers: Clicking on a link/button, Checking a checkbox/radio, Opening a browser window, Checking object/text...
  - Execution Engine: Utility functions, INIT script, MASTER script, EXECUTE_APP script

- **Metadata**
  - Screen metadata
  - Screen actions
  - Flow metadata

Figure 4: High Level Design of the Framework

The design can be broken down into two main parts:

- **Core framework** – which holds common code for functions, the controller script and generic services like logging, exception handling
- **Metadata** – spreadsheets which hold metadata about screens to test, what actions to test on those screens (e.g., "Click" action on "Add" link on "Shopping cart" page), and what data to test them with (e.g., what phone to buy in an order script)

The script designer is responsible for either automating new test scripts, or updating existing automated scripts with changes pertaining to the engagement. Based on this framework, the script designer does the following:

1. Updates the metadata sheets with new screen definitions, actions on those screens, new flows, updates to existing screens etc
2. Requests the automation engineer to enhance the framework with new custom functions or actions which were not provided by default and are needed for implementing the new functionality
3.1.4.2 Team Composition and Role Definitions

- **Test Designer** – creates the actual Keyword driven test cases in a spreadsheet, including pre- and post-conditions and Keyword parameters. This role will be played by a non-technical subject matter expert / business user

- **Automation Engineer** – maintains the core framework and enhances the same for any new customizations or enhancements. This role is a technical person and is well versed with coding and the scripting language of the Record/Replay tool

- **Test Executioner** – Executes the automated test scripts as and when required. This role need not be a technical one but requires application data requirement understanding to provide the input data for the automated script to run

3.2 Meeting requirements

This section maps back the organizational requirements noted earlier for the automation test suite, and describes how the keyword driven framework approach addresses each design concern.

3.2.1 Business requirements

3.2.1.1 Maintainability

With each new release, the suite needs to be updated based on new/modified functionality. To update the automated suite, a few key activities need to happen. Subsequent sections outline what tasks constitute maintenance activities, and how the architecture helps reduce effort involved in these tasks.

3.2.1.2 Understanding existing functionality

Before a new engagement team begins to incorporate changes due to new functionality, it must understand existing code. This could mean traversing through several lines of code in a typical architecture.

This effort is largely reduced in the new architecture for these reasons:

- The script author only has to look at the metadata for each script to understand its existing functionality. With documents that map every automated script to a corresponding manual script, coupled with solid naming conventions for script actions and screen names, this is relatively very easy compared to browsing hundreds of lines of code

- It’s easy to identify impacts to existing scripts with the metadata approach. Thus for example, if a new screen has been added between 2 existing screens, all we need to do is to do a Find in excel for that particular screen

3.2.1.3 Making code changes

Existing code needs to be changed to reflect updated functionality, while new code needs to be added for any new functionality. A metadata driven approach significantly cuts down development effort due to several reasons:

- Script authors focus on updating metadata, which represents core functionality (as opposed to dealing with WinRunner code in which they must deal with WinRunner
API for core functionality, aside from exception handling, logging/reporting and other coding constructs)

- Centralized changes can be made for functionality. If a new button has been added to an existing form, all we need to do is to modify the screen metadata for that screen to include the extra button

- Even for changes that are not centralized in nature, Find/Replace operation on excel will suffice. For example, consider a new screen added in a flow. We can find in the metadata sheet all flows that include the previous screen in the sequence, and then modify to add the new screen. In a traditional architecture, we would have added a function, and then browse through code to determine all dependent functions, and modified them

3.2.1.4 Testing

After code changes have been made, code must be unit/integration tested. We can choose, using metadata sheets, exactly what data combinations to run during a particular integration run. That saves us the time involved in running the tests, as well as analyzing results – and helps focus on exactly functionality that has been impacted by code changes.

Besides, if issues are found during testing, framework’s error messages can be very intuitive and immediately indicate if the issue is with the application or with the automated script, hence reduce the discover->fix cycle.

3.2.1.5 Design/Code Reviews

Design documents and code written needs to be reviewed. Here is how the architecture helps this effort get reduced and focused:

- In design reviews, the focus is on understanding how functionality is broken down into flows, and what actions those flows contain. There is minimal technical effort if new custom functions have to be written based on the new functionality

- In code reviews, only artifacts to review are metadata sheets, GUI Map objects and any custom functions written. This is in contrast with typical architectures, where a majority of effort in reviews is spent in ensuring adherence to coding best practices (like variable names or comments in headers). Also, reviewing metadata helps focus on what functionality is being tested in the automated suite, and whether or not it is necessary and sufficient for testing the functionality given – which leads to a robust, comprehensive regression test suite

3.2.1.6 Selective testing

To selectively test certain flows of the application, a “regression_level” column is used in the data sheet. Script authors can set this column to “N” for any data combination that does not need to be tested.

Further, the controller script provides the user a GUI to choose what regression level to run (Level 1, Level 2 or Sniff), which can help user test depth (Level 2) or simply breadth (Level 1) of the functionality.
3.2.1.7 Recovery

In case of catastrophic failure (network outage, machine going down temporarily etc) or if the scripts are intentionally stopped for some time (e.g. when environment is down), scripts must recover from where they last left – to save execution time.

To achieve this, the controller script automatically sets the “execution_required” column to “N” every time a data combination is executed. Next time when the same application is run again, combinations marked “N” are not run.

3.2.1.8 Scalability

Parallel execution of scripts can be achieved across two levels:

- **Across all applications** – As the framework provides flexibility to execute a specific application using the GUI interface, we can execute different applications on different machines in parallel, thus achieving scalability in execution

- **Within an application** – To scale execution within an application, the datasheet can be modified to include only specific data combinations (by setting “execution_required” column to “Y” for those rows). By executing different combinations on different machines, parallel execution can be achieved within an application

3.2.1.9 Inter-environment portability

To meet this goal, framework code must be independent of an environment. Typically, the only dependency on environments is based on specific data used for environments. For example, a test environment might have different username/passwords and URL used from the production environment. Given that data used for executing scripts is located in datasheet, the design must allow localization of these data sheets based on environments.

3.2.1.10 Usability

Following sections describe how both usability requirements are met.

3.2.1.10.1 Low ramp up curve

In a traditional architecture, script authors are expected to write custom code for every script in the system. A typical WinRunner ramp up course, including exercises to be completed with the tutorial, takes about 5 days. Further, productivity of script authors is low for the initial period as they go through a ramping curve while coding.

In the new architecture, script authors focus on entering screen metadata. Thus for a majority of the team members, a simple framework usage ramp up will suffice. A small team of automation engineers which maintains the framework in steady state can be responsible for writing custom functions.
3.2.1.10.2 Minimal manual intervention

Framework reports each action, and failure at each step of execution. In case of an application bug or an issue with metadata etc, the framework logs the particular issue and moves on with further execution. Thus scripts can be executed unattended, and results can be later analyzed for fixing.

3.2.1.10.3 Reporting

Following sections describe how the three reporting requirements are met:

- **Status per test script**
  
  For every data combination, the test result sheet contains the test script ID and description for each flow executed. After the success or failure of each script, the reporting mechanism of the framework indicates the status of the test script.

- **Screenshot or any other error message**
  
  At any point of failure, a screenshot of the screen on which the failure happened is captured on the local hard drive of the testing machine. Further, the path of the screenshot, as well as any actions that caused the error with the return code is also logged by the framework.

- **Information for defect logging**
  
  With each error that the automated script logs, a defect must be logged with the exact steps to reproduce the defect. To achieve this, in case a script fails, the actions are logged as a statement in the result sheet, which can directly be copy-pasted by tester into a new defect after validating that the defect exists.

3.2.1.11 Technical requirements

3.2.1.12 Data abstraction

This is achieved by ensuring that all data used by all applications is moved to application-specific data sheets.

3.2.1.13 Maintainability

Section below cover different aspects of maintainability:

3.2.1.13.1 Separation of concerns

The various layers are explained in Figure 1. Generic services (logging, exception handling, reporting, generic functions etc) are all coded in the core framework layer, while the end user is focused on entering metadata.
3.2.1.13.2 Modular code

3.2.1.13.3

- **Reuse is possible** – the whole concept of reusable keywords allows for code reuse
- **Centralized changes** – for any kind of changes required, the usage of wrapper functions allows the flexibility for centralized changes without the need to make changes at the script level

3.2.1.13.4 Portability

**Different Operating system & Tool**– The framework has been designed in a fashion which hides the tool specific code within the wrapper functions and thus separates the test case design from operating system or tool dependency. For a change in operating system or tool the changes can be made directly in the wrapper functions and the test case design need not be touched
3.3 ROI using the Keyword Driven Framework based Automation

The Keyword driven framework approach has been successfully used within two projects in Sapient and the comparative data in effort save is shared in the below section.

<table>
<thead>
<tr>
<th>Project</th>
<th>ROI %</th>
<th>LOE saved (person days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project 1</td>
<td>38.34%</td>
<td>175</td>
</tr>
<tr>
<td>Project 2</td>
<td>40.95%</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 1: ROI percentage gain against investment in automated suite creation

The table lists the ROI (Return on Investment) percentage achieved and the LOE (Level of Effort) saved in person days over a period of a year on the two projects. The comparative figures show consistency in results.

<table>
<thead>
<tr>
<th>Activity</th>
<th>LOE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Execution LOE (person days)</td>
<td>522</td>
</tr>
<tr>
<td>Automation Execution LOE (person days)</td>
<td>255</td>
</tr>
<tr>
<td>Automation Update LOE (person days)</td>
<td>91.52</td>
</tr>
<tr>
<td>LOE saved per annum (person days)</td>
<td>175.48</td>
</tr>
<tr>
<td>Automation Creation LOE (person days)</td>
<td>457.62</td>
</tr>
</tbody>
</table>

Table 2: LOE comparisons

The two tables show LOE values for the manual vs. automated execution on the two projects. It also lists the LOE saved by automated testing using the framework approach over manual testing.

Details of the ROI calculation are listed below:

\[
\text{ROI} \% = \left( \frac{\text{Savings per annum due to automation}}{\text{One time cost of automating the scripts}} \right) \times 100
\]

Positive ROI signifies that at some point of time we will achieve the break even point (a point in time where effort saved is equal to the effort invested in automating the scripts).

For example, if the ROI is 50 % then we will achieve the breakeven point in two years. After two years we will become profitable on our investment in the automated test suite.
Figure 5: Effort Saved in the two projects in terms of person days

Figure 6: ROI percentage in the two projects as a percentage of LOE saved over LOE invested on creating the automation test bed
4 Way ahead for the Framework Approach

The framework has been developed and used successfully for the web based testing. Following enhancements are being worked upon to derive greater efficiency and benefits of the automation to testing:

- Expand the capability to cater to multiple testing platforms other than web based testing – power builder, VB, people soft and java based applications
- Building supporting tools to take care of the data preparation and tying it back to the automation framework to make the automation even more painless and comprehensive
- Intelligent front end to automate the scripts so as to reduce the effort of creating and maintaining the automated scripts
5 Author Profile

Rajeev Randhawa has more than four years of experience in Software Testing, Automated SQA tools, and is currently playing the role of QA Manager and managing team of 8 people.

Prior to joining Sapient Rajeev worked as a Software Tester for Xavient Technologies and NIIT ESRI.

Rajeev holds a MCA degree from GGSIPU Delhi and has done his graduation in computer science from University of Delhi. He is a QAI certified CSTE.

Lokesh Pandey has more than three years of experience in Software Testing, Automated SQA tools. Over the past one year Lokesh has worked on QA automation efforts on a client engagement and managed multiple QA tracks within the project.

Prior to joining Sapient, Lokesh worked as a Technical Support Professional for an Industry leading Automation Testing Tools company where he handled technical issues on the project and managed a team of 8 support professionals for a period of six months.

Lokesh has a BS from University of Delhi in Computer Science & Information Technology.

Company and Contact Address:

Sapient Corporation Pvt. Ltd
Sapient Tower (D & E)
DLF Cyber Greens
DLF Phase III, Sector-25A
Gurgaon 122002
Haryana, INDIA
www.sapient.com