Automated Testing approaches

- Thirumalai V.M

By Thirumalai V.M.

For additional information or assistance please contact Thirumalai at 91-44-2254 0555. www.cognizant.com • No. 4 Canal Bank Road, Taramani, Chennai 600 113. India.

Table of Contents

Automated Testing

Abstract

Automated testing is recognized as a cost-efficient way to increase application reliability, while reducing the time and cost of software quality programs . Some of the common reasons for automating are Reducing Testing Time, Reducing Testing Costs, Replicating Testing Across Different Platforms, Repeatability and Control, Application Coverage and Results Reporting.

There are various tools available in the market, Here we are going to present about the Winrunner. The architecture we adopted is the Data Driven. Automation is done for the client server application. It's an Insurance domain application. The application has to be tested for different States in U.S.

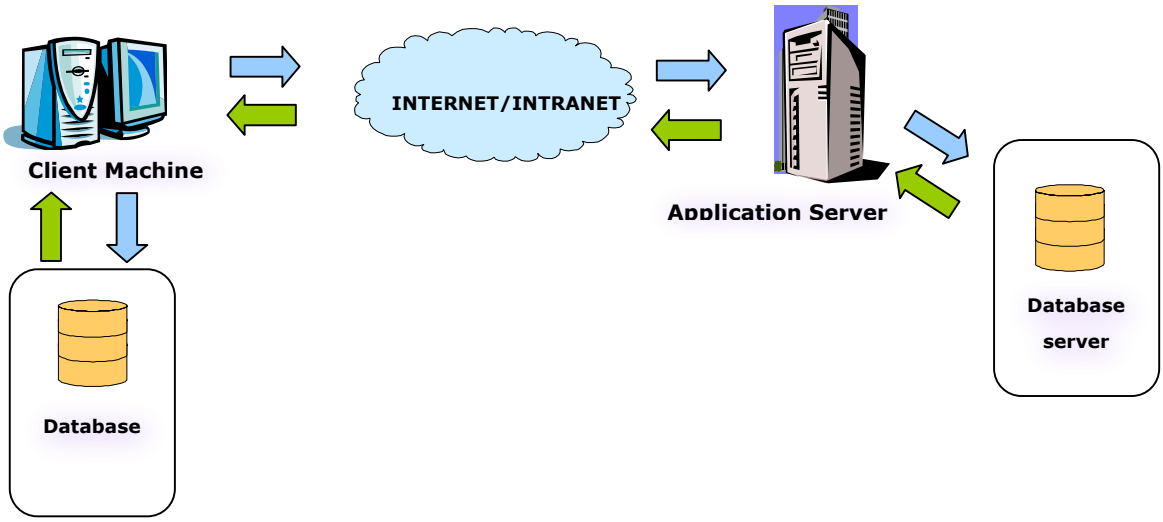
About Application

Introduction

It's an Insurance domain based application. Its an client server application. Its the leading independent agency writer of automobile insurance in California and has been one of the fastest growing automobile insurers in the nation. Automobile insurance is also written in Florida, Georgia, Texas, Illinois, Oklahoma, New York, and Virginia. In addition to automobile insurance, It writes other lines of insurance in various states, including mechanical breakdown and homeowners insurance.

Brief Description of Application?

Application takes the input from the Insurer who really wants to covered by a typical policies like auto, homeowners and so on. The application is automated for the Homeowners module which covers the HO-3, HO-4 and HO-6 types of policies. Depends on the Policy types the GUI changes at the run time. Homeowners module is divided into Basic Info, Dwelling Info, Additional Questions, Coverage, Point of Sale and Binding modules. Each module is designed in an hierarchy and separated by Tab's and Sub Tab's. Once the valid data's entered in each module, the premium has to be calculated for the coverage he has chosen. The application is used to generate the policy for the Insurer.



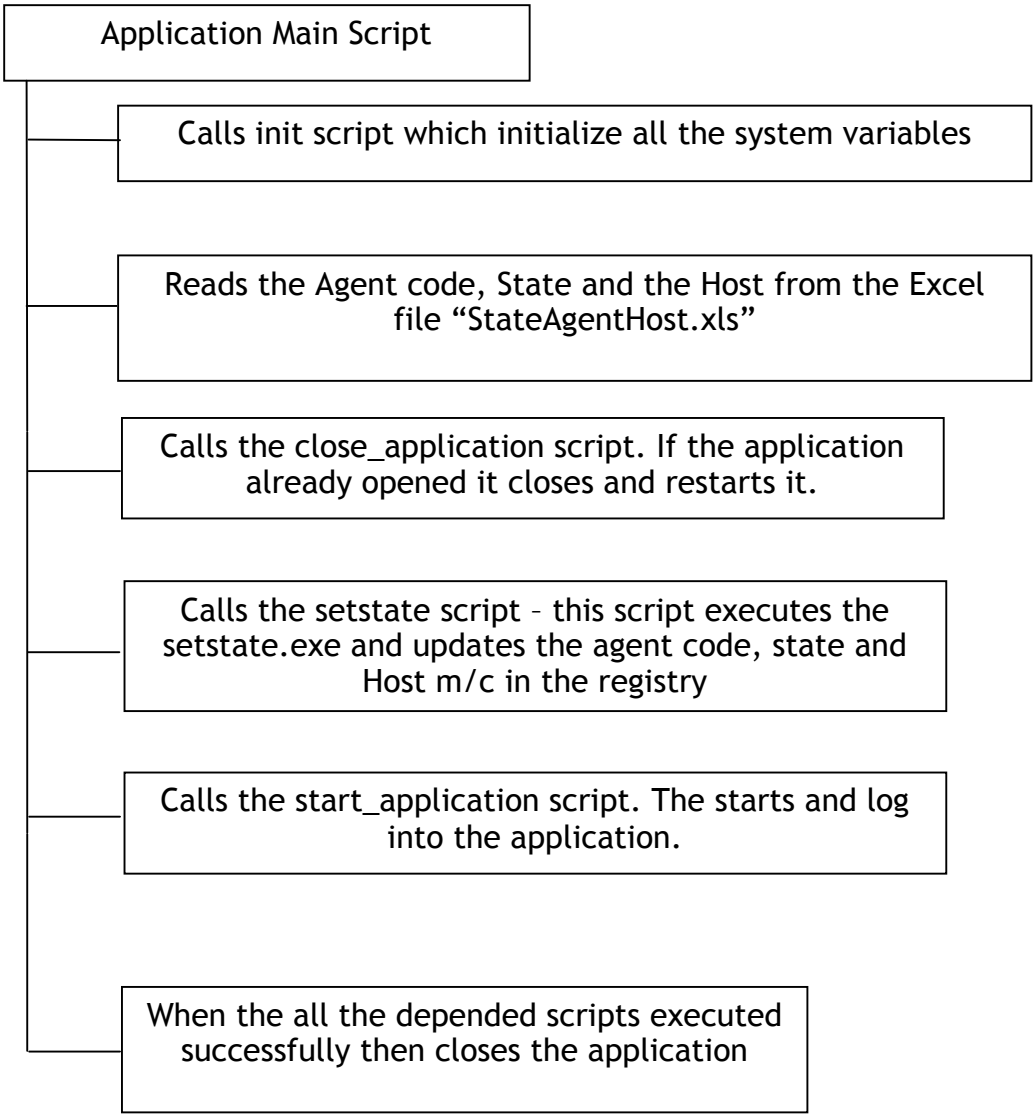
The above figure can give a high level view on Application architecture

Architecture Incorporated

The Approach

This section speaks about the process we adopted to automate the testing process. Data Driven architecture has been adopted we have prepared data sheets for each locales because the functionality may differ based upon the different locales. Some locale has a specific coverage for Earth quake, Hail Strom and so on. It is difficult to handle these in the Test script. The Data Driven excel sheet is prepared based on the Locale and the script is made generic. The script is driven based on the Excel sheet.

The Script Layout



Introduction

The Main script made use of existing supporting WR (WinRunner) Function Libraries and a new library based on application depended functionality. At the time of writing, most function libraries used by this script are location: <drive>\<State>\<Module Name>\New Scripts\Libs\ folder. Some libraries are located in the WR installation path but these are libraries installed with the product.

This script can be executed from either TD (Test Director) or WR. When executed directly from WR, the State to execute is set into the variable state_to_run (which is at the beginning of the script).Note that it is necessary to "set" the State value so the name of the DT (data table) can be determined. The path to find the data table(s) is defined in the variable DT_path at the beginning of the function library Application_lib. When executed from TD the full path of the DT must be passed to the script via the parameter scriptparm_Datatable. When passing this path, the double backslashes must be retained between the folder and file names (and four backslashes at the beginning of the path when using UNC notation). The State and agent values are obtained from the DT and must be present in the first row on the DT in the columns State and Agent respectively.

The state value is also used to build the name of the WR GUI file to load to allow processing of different states. The GUI file(s) are names must end with the two-character abbreviation of the state. For example: Homeowners_CA, Homeowners_FL. The GUI files are stored in TD.

This script also uses the QS_NB GUI file. This file contains all the windows not related directly to Homeowners Assignment.

This script was designed to handle positive data. It can handle the error window when it appears (which indicates edit/cross-edit errors) and will stop processing the current test case and move to the next test case.

This script is data-driven. By using the information found in the data table, the script navigates through the app and populates objects found in the major tabs (and sub-tabs) found in the application.

Script Flow

The data table, which drives the script, has eight columns that allow application navigation and entry of data. The columns are as follows:

Test_Case: The value in this column is used to identify a test case. This can contain any value (numeric or alpha numeric). When the value changes this will trigger a new insurance application to be started (pressing the New button). Note: when the test case name changes the name of the test case is written to the Test Results and Log file.

Tab: The value in this column represents the navigation through the main tabs of the application or an action to perform against the application. The values in this column must be Basic Info, Dwelling Info, Additional Questions, Coverage, Servant's Info, Additional Question Details, Loss History, Coverage, CA Earthquake Authority (the preceding is not a tab but a popup window which appears in the Homeowners Assignment), Coverage, Additional Info, Point of Sale, Binding, Validate Rate, Exp Results. Validate Rate and Exp Results are not tabs but actions to take upon the application. Validate Rate will cause the Validate Rate button to be pressed and Exp Results are the expected results found in the Discount/Rate Factors (sub) tab. Order is very important as this represents the order to navigate thru the tabs.

Sub_Tab: The value in this column represents sub tabs found in some tabs. Note that not all tabs have sub tabs. The valid sub tabs are Warranty, Add Warranty, Description, Additional Info, Mortgagee Info and Payment Info

Next_Tab: The value in the column "Yes" is used to refer the Data is in the Next main tab. So that script can identify and process to the Next main Tab.

Occurrence: Denotes the occurrence that data represents. Drivers and vehicles have occurrences. Occurrence is also used to associate a driver to a vehicle.

Field: The value in this column represents the name of the field to populate or the name of the column to validate.

Data: The value in this column represents the data to place in the field. If the data value is empty or null the field will not be populated (see PopulateField function for exceptions to the above rule). Remember to correctly format fields. For example: date and time fields.

Actual Results: The Actual value found in the application will be read and write besides the excepted column row. This feature is providing so that user can easily check whether the excepted and actual result matches.

As mentioned earlier, the State and Agent columns need only have the first row populated. The state and agent are set once at the beginning of script execution.

The relationship between the data table values and the method used to identify objects to populate is critical in this script. The GUI map for this script consists of the objects found in the application using the following naming convention: Tab, Sub_Tab and Field. There are occasions when Occurrence is used to identify fields (and appended to the end of the Tab, Sub_Tab, Field combination). The combination of the column values is used to build the name of the object to look for in the GUI map. The names used in the DT match those found in the GUI map. All the field names in the GUI map are upper case and the Tab, Sub_Tab and Field column data are converted to upper case to make the match with the GUI map.

This script, which may seemingly be complex, is really simple. The main script itself drives the navigation through the application and a User Defined Function named PopulateObject does the population of the data into the application.

The script processes all records found in the data table. For the most part, each row of the data table represents the population of an object on the application (the exceptions being Validate Rate and Expected Results). Each data table contains one to many test cases. Each State should be its own data table. The script could be modified to process multiple states. Based on the change of the entry found in the Tab or Sub_Tab column the script will perform some navigation action (usually pressing F8, F5, selecting a tab or pressing a button).

This approach allowed the population of the apps objects to be a very simple routine. The logical name of the object is constructed (as outlined above) and then the object's class is extracted from the GUI map using the GUI_buf_get_desc function.

There are only six types of objects in the app (edit, list and check_button, radio button, Tab, button) and two of those are used 99% of the time. Once the class is determined then the appropriate "set" or "select" function is used to perform the action upon the object. The User Defined function, which does all this, is called PopulateObject. There are some exceptions to the way certain fields are handled and this is all coded in the PopulateObject function. Some fields behaved better

when handled other than the “normal” way. This mainly applies to edit objects where a obj_type works better than the edit_set (for example, date and time fields). Also, some fields select a default value and ignore any entry in the data table.

This library contains many of the supporting functions for the Application Main script. All the functions have a short description in the library itself. Some function will have some further details below.

HomeownersInit – This function is executed as soon as the HomwOwners_lib is loaded. Remember that the variable declarations at the beginning of HomwOwners_lib are automatically done when the library is loaded.

PopulateField – This function handles population of all objects in the app. Keep in mind that some fields require special handling. This is all done in this function

Major Functions

TabSetWindow – Performs the set_window on the appropriate window based on the name of the tab found in the DT. Most tabs belong to the Application window but there are three exceptions that are handled in this function. To press any button on the left side of the app, the current window must be Application. This is the only window that has these button defined.

ValidatePaymentFactors– This function will perform the validation of the expected results found in the DT (“Expected Results” in Tab column). The DT “Field” column has the name of the table column to check. An associative array at the beginning of the function indicated which physical column of the table relates to these columns. When all expected matches actual data, one message is written to the Test Results indicating all matched. When a mismatch occurs, only the mismatch is written to the Test Results (the matches are not).

CheckForErrors – This function attempts to allow the script to recover from the error dialog that can display upon navigation from a tab. When an error is found the script should log the error messages to the Test Results and then skip processing the remainder of the test case.

Other Information

For debugging purposes, a column named Debug can be added to the DT. When a non-empty value if found in this column then the script will break just inside of the main DT loop.

Conclusion

Thus we have practically automated the complete work flow for this project. The time consumed for scripting the work flow is considerably took less time and the maintainability is of code is very easy compared with the play back technique. The main advantage of this architecture is if we want to test any new functionality we don't want to touch any scripts. Instead update the Data Driven Excel sheet is more than enough. Thus we can automate for all the functional areas in the application we want to test in a short span of time.