STeP-IN SUMMIT 2007

International Conference On Software Testing

Framework for GUI Test Automation

by
Aravind Lakshminarayanan
Texas Instruments (I) Private Ltd, Bangalore

E-mail: aravind.l@ti.com

Copyright: STeP-IN Forum and Quality Solutions for Information Technology Pvt. Ltd.

Published with permission for restricted use in STeP-IN SUMMIT 2007 in agreement with full copyrights from owner(s) / author(s) of material. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior consent of the owner(s) / author(s). This edition is manufactured in India and is authorized for distribution only during STeP-IN SUMMIT 2007 as per the applicable conditions.

Practices Experience Knowledge Automation

Produced By

STeP-IN Forum

Hosted By

www.stepinforum.org

www.qsitglobal.com

Abstract

This paper describes the methodology adopted to develop an automation framework to carryout GUI testing of a window based application at Texas Instruments. The approach reduced the test execution time convincingly without compromising on the quality of testing. The maintenance efforts for the sub-sequent test cycles were very minimal and it warranted complete reuse of test cases.

1. Introduction

Over the past several years, tools that help programmers quickly create Graphical User Interfaces (GUI) based applications have dramatically improved programmer productivity. This has increased the pressure on testers, who are often perceived as bottlenecks to the delivery of software products. Testers are being asked to test more and more code in less time. They need to dramatically improve their own productivity. Test automation is one way to do this.

There are innumerous solutions that exist in the market. But they do not always cater to the needs of testing an application which changes at rapid pace. The maintainability becomes difficult in such cases. The following paper describes development of a tool that was used to test GUI of an application effectively with minimal efforts towards maintaining the tool.

Section 2 analyzes tools available in the market and their pros and cons, and Section 3 describes our implementation. Section 4 describes how we evaluated our system and presents the results. Section 5 presents our conclusions and describes future work.

2. Tools for GUI Test Automation and their suitability for our application

There are several tools available for automating GUI tests. They fall under the following broad categories –

- Capture and Playback tools with/without scripting support
    - They can do effective GUI verification (even up to pixel level)
    - Mostly support recording or scripting test cases (in a proprietary language)
    - **Cons**
        - The test cases are tied to the framework/tool
        - Go based on pixel rather than field (child window) captions in window controls. The effort to maintain (recorded/scripted) test cases is huge for application whose GUI changes frequently
        - The tester is required to learn a proprietary scripting language most of the times
        - They do not reduce test execution time (if tests are manually recorded, they take the same time to execute the test too. Manually changing the idle time in at various points in the test cases might not be feasible if test cases are large in number)
- Script based tools
    - Scripts (mostly) written in proprietary scripting languages
    - Give access to all standard window controls and mostly based on field captions
    - **Cons**
        - Efforts required for learning the scripting language
        - Test cases are tied to the tool

STeP-IN Forum
www.stepinforum.org

QSIT The Knowledge Corporation
www.qsitglobal.com

- Maintainability efforts of the test cases is high since bunch of test scripts need to be changed even for a position or caption change

3. Tool Expectations from application under test and initial trials

In our scenario, the expectations from the automation tool were

- Should exercise GUI controls extensively to carry out various actions (the inputs could be supplied using configuration files too, but GUI had to be tested exclusively since there were tools already present to test when input from configuration files)
- Manages frequent changes in GUI (in terms of addition/removal of fields, frequent screens additions etc)
- Kept isolated from the test cases so that addition of test cases as well as maintaining the framework is easier
- No special efforts should be required to develop test cases (reduce additional overhead on developers/testers)
- The cycle time for test execution should reduce considerably

There were a lot of tools that met the expectations partly but not one that could satisfy all of them. So the need arose to develop a tool that met all the expectations and at the same time did not require a huge effort too.

The expectations from the tool were evaluated and the following requirements were identified for the tool

- Test GUI controls completely
  - Should be able to access all controls including dialogs opened from the application
  - Should provide means to verify all fields including popup dialogs
- Test cases should be separated from the framework to reduce efforts in managing change in GUI controls
- Test case definition should be easy
- The framework to be as generic and easily maintainable as possible
  - Should manage changes in field names and positional changes with minimum or no effort
  - Extending support to new fields should be minimum
  - Should provide both key board as well as mouse support
- Error handling capabilities should be present (like application crash, hang etc)
- Should have logging capacity to help debug failures. The extent of logging should be programmable.

Different architectures were evaluated to satisfy all the requirements mentioned above. Some of the initial trials were -

1. Developing a tool as a part of the application to access various classes and modify and verify values. But this was not exercising the GUI to a great extent and this was tied closely to the application.
2. Third party tools that could be customized to meet the requirements. But the licenses were either very costly or it was not possible to obtain the sources to those tools that could be customized.

But since none of these approaches seemed to satisfy the expectations from the tool completely, a novel architecture was designed and the same is described in the following section

4. GUI Test Automation Tool Design

This section describes the design aspects of the automation tool. It covers the final architecture design and design of the various sub-components of tool.

4.1. Tool Architecture

The following architecture was chosen as it catered to all the requirements. It contained the following sub-components –

1. The Central Framework
2. Input files to specify field details
3. Test Repository (single or multiple files) and list file to specify the path(s)
4. Logging Mechanism
5. Configuration file to specify application specific information

A high level architecture is shown in Figure 1 below

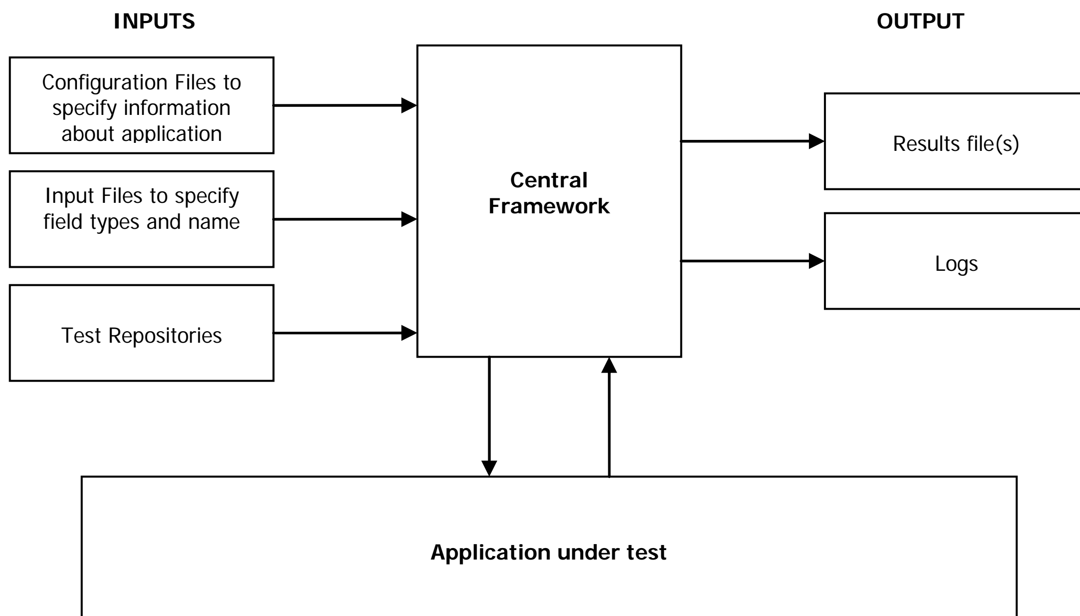

Figure 1 Tool Architecture

4.2. Sub components design

The following sub sections describe the design phase for different sub components in the tool, mentioned in the previous section. It also explains the challenges faced at various phases and how they were overcome.

Central Framework

The central framework was designed in Perl. It had to have the intelligence about the methodology to be employed for different kinds of controls. Various Perl packages were appraised and win32-guitest was chosen as it provided interface to all kinds of window controls. Framework was developed to handle various kinds of controls. Generic procedures were identified to access a control using key board as well as mouse. These procedures were a sequence of win32 API calls that would help achieve desired actions on various types of controls. Verification mechanisms were also identified for different types of controls.

Now that tool knew how to access different types of controls, the information about various controls of our application had to be included in the framework. The access could have been individually programmed (for each and every fields) in the framework but any modification in the future would be tedious. This could become an immense effort (probably equivalent to the complete tool development effort itself later) when multiple fields change drastically and the framework needed to be updated for any future test. Thus a need arose to design a mechanism where in the frame work be made generic to handle different kinds of fields in different pages of the application

Input Files to specify field details

To achieve a generic framework, it was designed to read information about fields from an input file that specified the control names and the control type in various pages of the application (Our application had different pages, which had different kinds and combinations of controls in them). Based on this input and the intelligence to handle different types of controls (that was already defined), the framework was now able to navigate through various fields in different pages to achieve the actions required to be carried out in the application GUI. A screenshot of the input file is shown in Figure 2

Page	Control	Type
General Information	Name	Edit Box
General Information	Sex	Radio Button
General Information	Department	Combo Box Control

Figure 2 Field Type Input File

Test Repository

Now that the framework could handle all fields in the application, the next daunting task that we faced was to identify the manner in which test cases and procedures could be defined, with least effort. Since the test focus for the tools was the application's GUI, test cases would typically be a sequence of user events. The test case could hence be broken down to populating a set of fields in various pages and then carrying out verification based on the expected outcome of the actions.

There were innumerous options available for the mode of input. The framework could run a set of test functions (defined in perl, framework's native language) or read the steps from a file that could be simple text file of any kind of documents like MS Word™, MS Excel™ etc.

After extensive assessment, MS Excel™ was chosen because of its ease of use, capability to hold data in structured manner and possibility to have drop down lists to choose from (so that spelling mistakes in the key words could be avoided). Simple (English) keywords were identified and generalized for various actions.

The tests thus contained a series of actions carried out using keyboard or mouse or a combination of both and then verified for expected results. A snapshot of a sample test repository is shown below in Figure 3

Test Case No	Action	Page	Field	Value	Field	Value
Scenario_1	Populate	General Information	Name	Ram	Department	Sales
	Verify	Summary	Name	Ram	Department	Sales

Figure 3 Test Repository

Logging Mechanism

One of the requirements from the tool was that it aids in narrowing down the possibilities of a failure so that debugging efforts are reduced. A logging mechanism was programmed alongside the framework that would expose a function that can be called with anywhere in the framework to log the output. To allow more flexibility to the user, the level of logging was designed to be programmable. The reason for this being when huge numbers of tests were run the amount of logs generated would be huge that it may disrupt the performance of the framework. When failures occurred, the set of failing tests can be run separately for logs with huge information. The levels that were identified for the logs were –

- 1 – Debug, would typically log all actions done by framework and their responses
- 2- Detailed, would have logs of all actions done by framework and responses for failed ones only
- 3- Minimal, would have responses to framework for failed actions only
- 4 – None

Configuration Files

Now that we were able to define and run test cases, other aspects like environment independence (able to run anywhere, of course limited to window based machines), application specific configurations (like the path to the exe, configuration files required by the application etc) had to be specified to the framework. A configuration file was designed for this purpose that specified the following –

- Path to the directory containing application executable
- Paths to various input files to the application
- Library paths
- Output Log options

The control flow through various sub components is shown in Figure 4 for better clarity.

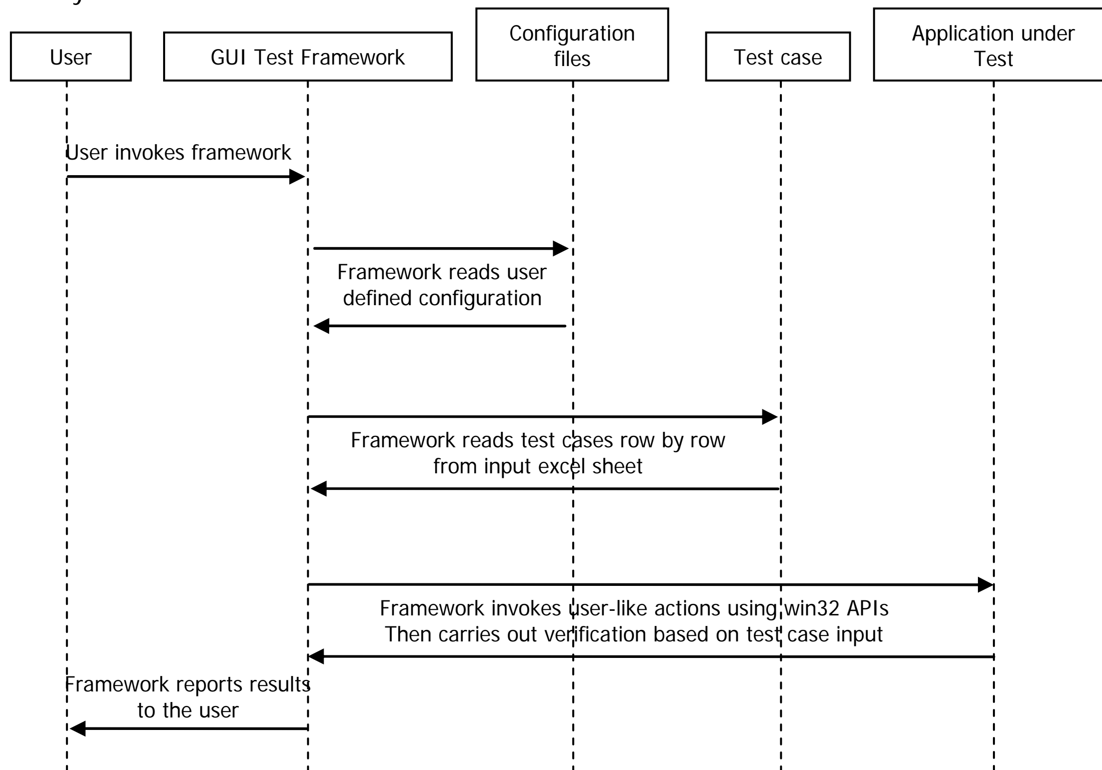

Figure 4 Control flow in the tool

Supplementary features like ability to read test procedures from multiple test repository Excel™ files (so that the files from various test engineers could be run) and handling application crashes and hang (the tests had to continue even if these happen, handy while running a huge number of tests and one of the initial one fails) were then added to the tool.

5. Evaluation

Now that the tool had been developed, it had to be tested. Test plan was charted out to test various features of the tool. Testing was carried out and all identified problems were fixed. The tool was also optimized for performance (typically time lag between steps, this was made programmable by the user so that this could be varied based on the system configuration like RAM and processor speeds).

Test cases were identified for the first trial run and the test execution was carried out in the framework designed. The results were quite promising. The same is indicated in the table below[1] (manual test numbers are also specified for comparison).

S No	Parameter	Manual	GUI Test Automation Framework
1	Test case	NA	8 hours

[1] Results based on a sample of 100 test cases

	specification effort		
2	Execution time	40 hours	8 hours
3	Correctness	100%	100%
4	Defects caught	10	10

Table 1 Metrics on efforts and defects

After further optimization (by fine tuning the idle time between various operations in the framework) we were able to get down the execution time by 3 hours and the test case specification came down on subsequent cycles as the familiarity increased. Thus the total effort required further came down to 10 hrs (1 and ¼ man days)

For sub sequent test cycles where the application GUI changed, the framework could be updated with minimal effort. With minutes of effort to update the input file, the framework could execute the old test cases and was ready to support test cases for the newly added fields as well. This during trial with a capture and playback tool had invited huge amount of efforts (comparable to the previous run) to capture the test cases in the tool again (proportional to the extent of change).

6. Conclusion

Thus we successfully developed a tool that helped us verify the application GUI and reduced the execution time to nearly a quarter without compromising on the quality of testing. It also ensured that we could carryout testing on a more regular basis without the need for extra resources (daily regression tests were designed and included in the build framework for instance).

The tool was also designed to take care of reusability of tests with minimal maintenance effort. All this came at the cost of couple of months of development effort and a few hours of effort required by each tester to familiarize with the tool (for specification of tests in the repositories)
The tool thus proved better suited to test an application whose GUI changes frequently (as compared to a capture and playback tool)

Future Plans

The following enhancements are planned for the tool going forward –

- Reading field type and names from resource file(s) instead of the user having to specify it. This would ensure that this tool could be used to test any window based application with minimal customizations (pertaining to architectural differences in applications)
- Introduce sub routine support in test repositories so that common steps need not be repeatedly specified in the test repository. This would help drive down effort to specify test cases in the framework
- Random testing capability (randomly traverse through fields). This could potentially catch application crashes
- Recursive testing (to test all permissible values in a field, typically only boundary values and a few values in between are checked if the range is vast)

7. References

- Piotr Kaluski, "Using Win32–guitest", v1.8, 27th January 2006
- John McNamara, "Spreadsheet::WriteExcel - Write to a cross-platform MS Excel™ binary file", v2.16, 6th January 2006

o Kawai Takanori, "Spreadsheet::ParseExcel - Get information from MS Excel™ file", v0.15, 2nd February 2006

About the author

Aravind is associated with Texas Instruments (I) Private Limited since January 2006. Prior to joining Texas Instruments, he was engaged with Sasken Communication Technologies Limited, Bangalore where he was involved in various stages of Mobile software testing and has worked on products from different OEMs. He was associated with I2 Technologies, Bangalore before joining Sasken. He holds a Bachelor's degree in Electrical and Electronics Engineering from BITS, Pilani.