PhUSE 2011

Paper TS08

Data-driven Software Testing for Agile Development

Priyadarshan Shinde, Cytel, Pune, India
Ajay Sathe, Cytel, Pune, India

ABSTRACT

Automated comprehensive testing of software functionality, especially GUI, conventionally requires extensive scripting for the Testing Tool. Agile development being dynamic with multiple changes in specifications, poses major challenges for the Testing to keep pace, making it arduous and time consuming. For software with complex GUI, comprehensive test coverage is very difficult. This paper presents a Workbench that is Testing-Tool-neutral, and that neatly addresses the demands of re-testing with changing software specs; creating extensive test-cases with mere mock-ups; accommodating new or modified GUI features; and re-using test-cases across similar software products. The Workbench integrates TestComplete™ with Microsoft® Excel® for storing test scenarios – collections of sophisticated test cases – in pre-defined structure. This approach accommodates testing of growing or changing feature-combinations with a relatively small effort of defining the test parameters. The workbench, in active use over several release cycles is now being extended to QC numeric results of algorithmic engines.

1. AUTOMATED SOFTWARE TESTING

Today's software development environment is characterized by short and rapid test cycles and frequent change requests. In this scenario, test automation plays a crucial role in software testing.

Automation of testing has evolved from being a desirable to an essential part of software development. It offers four major advantages:

- Coping with disproportionate increase in testing needs as number of features increase and cause interdependent effects – the cumulative coverage of features
- Generating and executing numerous test cases rapidly by leveraging a common logic of testing
- Accelerating the software to release-worthy completion
- Minimizing the probability of potentially costly defects or bugs

Automation also offers the advantages of higher ***speed*** and ***accuracy***; easy ***repeatability*** of a battery of test cases and ***access*** to data, objects, protocols and other components of the software, that a manual tester may not have.

2. MANUAL vs. AUTOMATED TESTING

To take an illustration from the authors' work context, let us consider development of statistical software to be run on clients' desktops. The software deals with complex algorithms to implement the statistical tests and analyses. It needs to run on Windows Vista, Windows 7 and Windows XP, and needs to support Operating System versions in English, French, German, and Japanese. In the example case, 5000 scenarios were identified to cover all possible combinations of one particular algorithm.

For this situation, in a manual testing context, the time was estimated to be 116 person-days. In contrast, by using automated testing the actual time spent was 37.5 person-days, including 12 person-days spent in developing the automation. That considerable saving is slightly offset by the fact that this calls for computer hardware available 24x7 to actually execute the tests, and multiple machines to run different environments in parallel.

Clearly there is a case for substantial usage of automation where feasible, and not surprisingly there is widespread adoption of automation in testing.

3. EVOLUTION OF TEST AUTOMATION[1]

As the Figure 1 depicts below, the approach to automation of testing has evolved. Starting from simple record-and-playback techniques, it has moved to using a structured storage of data to drive a large number of automated tests, and then beyond, to utilizing keywords from a defined vocabulary to describe the problem to be tested.

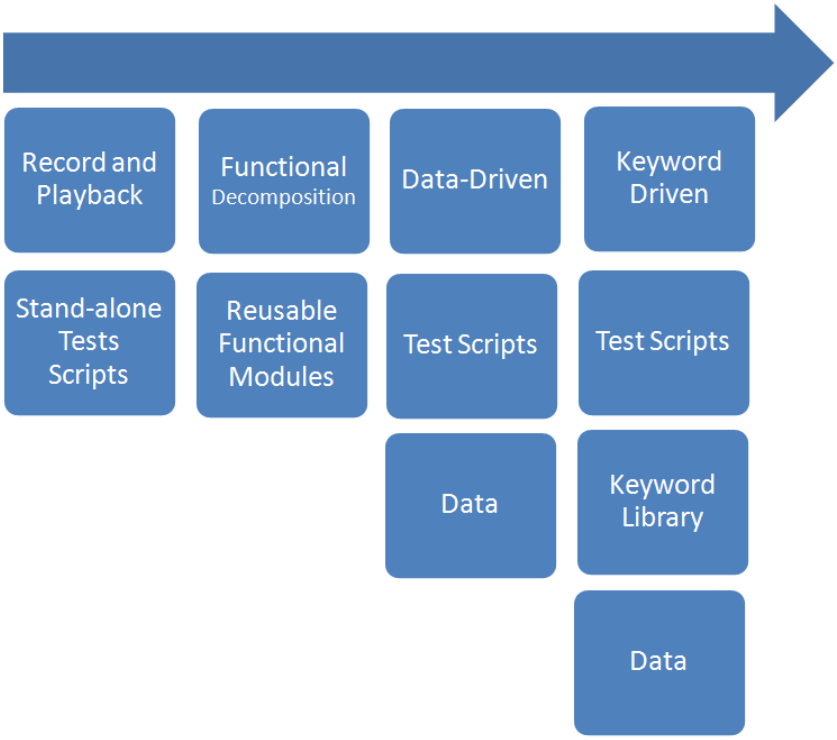


Figure 1

The relative advantages and challenges of the evolved approaches are compiled in Table 1 below.

Comparison of Automated Testing Approaches			
Approach	**How it works**	**Advantages**	**Disadvantages**
Record and Playback	User's actions captured and then play back	Not much technical expertise required	Difficult to maintain Limited reusability Lots of modification Not Extendable
Functional Decomposition	Re-usable & repeatable functions created and used across	Modular Approach Reusable Functions Hierarchical Structure	Test data is within the scripts, so limited reusability Dependency on Technical expertise Test Script is dependent on software
Data-Driven	Data is in external file	Improved maintainability	Dependency on Technical expertise Test Script is dependent on software
Keyword-Driven	Robust, software independent reusable keyword libraries are built	Ease of maintenance High scalable Reduced dependency on software	Requires great deal of efforts and time consuming Dependency on Technical expertise

Table 1

This paper focuses its attention on the combination of Data-Driven and Keyword-Driven approaches.

4. KEY ISSUES IN CONVENTIONAL AUTOMATED TESTING

The defining characteristic of the conventional method of testing is that test scripts are manually written by testers for each individual test-case, and are therefore “hard-coded”. These methods therefore lead to some limitations. Figure 2 describes the conventional approach:

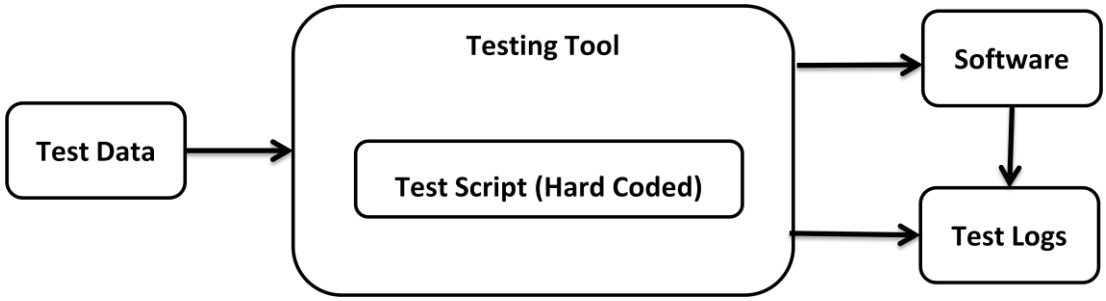


Figure 2

This method has the following demands and limitations:

- Software being stable is a mandatory requirement. Conventional test script assumes stable software. Even minor software changes will hinder script execution unless corresponding changes are made in the script. Bugs if any must be fixed before recording a scenario.
- It is not easy to co-ordinate efforts of multiple testers, opening up possibilities of redundant repetitions and sometimes omissions.
- Scripts tend to be hard-coded, making maintenance a non-trivial and sometimes complex task. An example could be the changed sequence or number of controls in a dialog box. A single feature change spawns multiple changes in the script.
- Script life tends to be short, because new testers find it a challenge to locate and modify existing scripts, and often prefer to discard existing scripts and write new ones.

The conventional automation by using hard-coded scripts gives rise to the following practical challenges:

- Object names tend to be long and unreadable. For instance, it may be difficult to see which control has the following name:

```
Sys.Process('notepad').Window('#32770', 'Font', 1).Window('ComboBox', '',
3).Window('Edit', '', 1)).
```

- Object name depends on object properties that may not be persistent. For instance, window and control recognition attributes (class name and caption) can change from one software build to another. To run tests successfully on the new build, old object names often need to be changed to the new ones.
- Changes need to be known in advance, so as to modify the test library accordingly. Test scripts cannot be used to find software changes in the form of errors.
- Operating System (OS) compatibility can pose a challenge. For instance, script written for German language OS may not work for English or French OS. It is a tedious job to modify the scripts.
- Differences in coding styles of different testers can cause problems of future maintenance.
- Tester needs to be fairly good at programming with the test script, as virtually every test case needs to be converted into a test script.

  Figure 3 depicts an example of a conventional automated testing script. This script is developed in TestComplete. As we can see, this script is large and needs expert knowledge to interpret.

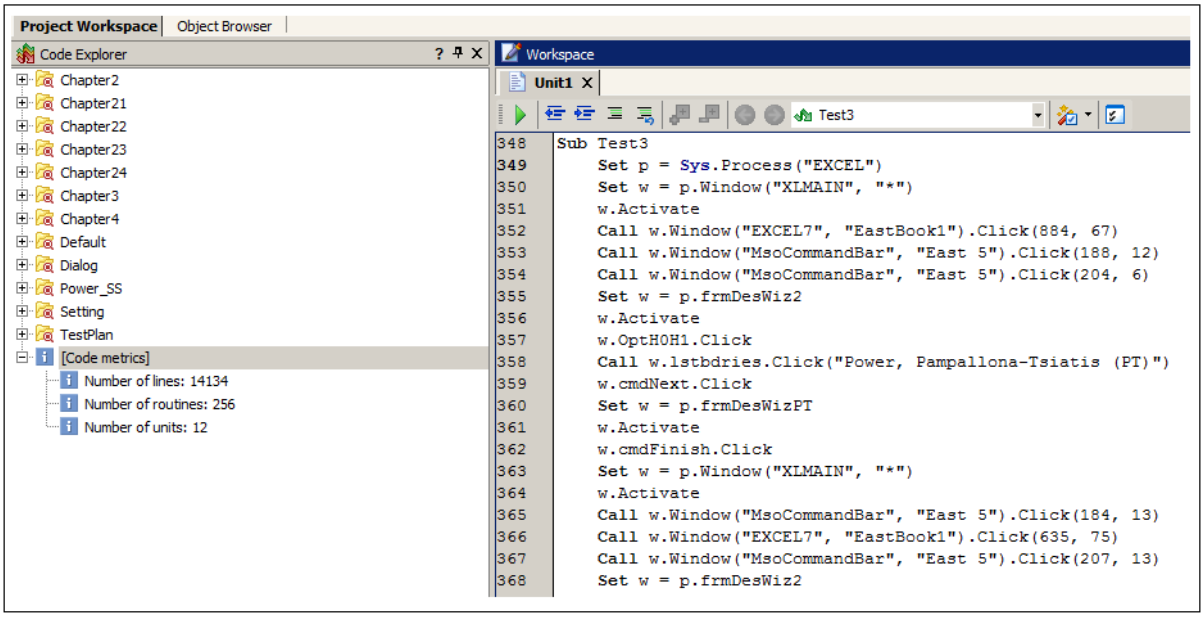


Figure 3

5. A NOVEL APPROACH TO DATA-DRIVEN SCRIPTING

Today most software packages use common GUI controls and sometimes custom GUI controls. To test and verify the software for its functional and business requirements, the idea of writing test-case specific scripts is generally not the ideal approach. This paper presents an alternative way of developing the test scripts for handling the GUI controls. This approach was developed by the authors and their colleagues as a real part of their work, and is in active use in several product development projects. We will call this approach as a ***Workbench approach***. This approach helps reduce the scripting time and efforts required for different software packages or versions since the common controls are handled identically by the Workbench across different pieces of software.

Workbench integrates ***Microsoft® Excel®*** with a testing tool called ***TestComplete***™. Microsoft Excel provides an easy-to-use interface and hence is used as the test repository. The repository stores Test Scenarios – the collection of sophisticated test cases in pre-defined structured. The pre-requisites for executing the test cases are sequence of actions, control names, values and expected results. All these are stored in an Excel workbook under a predefined structure.

TestComplete is used for handling the GUI controls available in the software. Once a piece of code is written for handling a particular GUI control then there is no need of rewriting the code for same control used in another software.

Workbench provides a way to create automated test scripts that can be run without human intervention. It can cover a lot of software areas in a repeatable manner and therefore it significantly cuts down time required for test scenario creation and test execution. It can run continuously and unattended, round the clock if necessary. Test scripts can be executed on the same GUI that the end-user or Customer would see. Workbench retains all the good concepts of test automation like data-driven, keyword-driven and modular approaches. It integrates all these in an easy to use manner without taking away any of the power that a scripting-based tool provides.

WORKBENCH

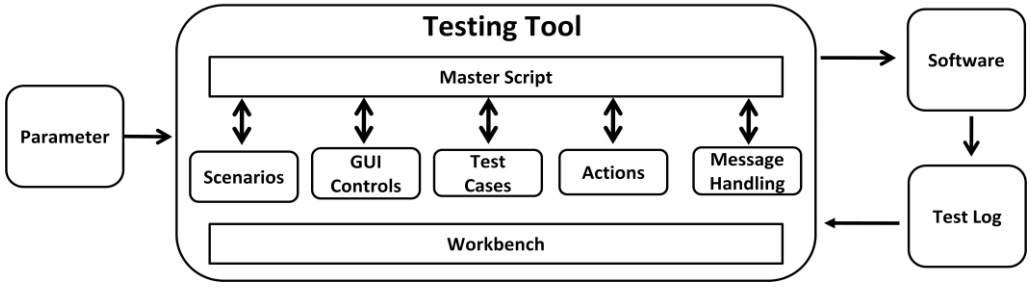


Figure 4

GUI CONTROL SEARCH TECHNIQUE

Most of the popular testing tools provide supporting technologies with which we can have access to internal properties of GUI controls like Microsoft Active Accessibility (MSAA). With the help of MSAA technology, the application provides information about its graphical user interface to the testing tools like TestComplete. It provides information about the type of the underlying control, its name, location, current state, and also the methods for handling that control. TestComplete has a very good mechanism through which we can get easy access to software specific control & its methods. TestComplete treats each process, window, control, etc. as an independent object. It uses values of object's properties. Each object has a number of properties such as "class name", "text", "enabled" and so on.

In TestComplete typically object is searched by the method described below. For example, following code is meant for searching a button called "***Compute***" button in the ***Architect*** software. TestComplete supports two methods `Find` & `FindChild`. `FindChild` is used in this example because it searches within the specified parent.

```
Set btn = Sys.Process("Architect").FindChild("WndCaption", "Compute")
  If btn["Exists"] Then
    btn.Click
  Else
    Log.Error "The button ""Compute"" was not found."
  End If
```

Here, `FindChild` method searches for a child object with the specified values "Compute" of the specified properties "`WndCaption`". Once the object is found then it can be used for further actions.

SETTING A PROCESS

TestComplete gets access to the elements of the software under test though the Windows process. Actual process is obtained through the `Sys.Process()` function. TestComplete searches for among all processes that currently exist in the operating system. Using data driven approach, Workbench uses `FindChild` method dynamically. It will first set the process of software under test with the following command:

```
Set P = Sys.Process(AUTProcess)
```

Here, `AUTProcess` is a variable name, which accepts a specified value from the test repository stored in Excel. The created process object, P will be used in the test script for further use and TestComplete will ignore all other processes which are running simultaneously in operating system.

SETTING A WINDOW OBJECT

Once the process is set, next step is to find & set window Control within the above process. This can be done by using the following code:

```
Set WndObject = Sys.Process(P).FindChild("Name", ControlName, 1000)

If WndObject.Exists = True then
       WndObject.Activate
       Log.Message("Specified Window is exists and activated")
                 :
                 :
                 :
Else
       Log.Error("Specified Window not found")
End If
```

Here, `ControlName` is a variable name which accepts the specified value from the test repository and finds the specified window using the process object. If Window control exists then the created window object is used for further actions.

Object creation of a particular window is an important step since this reduces the time required for searching other controls within that window.

Another purpose of setting a unique window object is to distinguish between the two controls with same name if present on different windows.

FINDING AND SETTING CONTROLS WITHIN A WINDOW

Once we are sure that the specified window exists and also the corresponding object is created, the next step is to search specific Child control within that Window. This can be done by the following code:

```
WndObject.Refresh
WndObject.Activate

Set Control = WndObject.FindChild("Name", ControlName, 100)
If Control.Exists = True Then
       Control.Refresh
       Control.Activate
          :
          :
```

Here `FindChild` method searches the "Name" property for specified `ControlName`. It will only search within the created window object. This will reduce the test execution time.

This step is then repeatedly used for searching all specified Graphical User Interface (GUI) controls in a test case within a window.

GUI CONTROL HANDLING

By the above methods, the process, the window object and a Child control within that window are set.

Different types of actions can be performed on this Control. According to the control type, the set of possible actions can be classified. The efficient use of Workbench can be done by classifying all the controls based on their types. Some of the control types are: a button, a text box, and a combo box. This kind of classification helps in future while inserting a new type of control or handling the existing controls of similar type.

This can be done with regular **`Select Case`** method as shown in following code:

```
Select Case ControlType
       Case "Button"
              Call ButtonAction
              Call ButtonProperties
       Case "Textbox"
              Call TextBoxAction
              Call TextboxProperties
       Case "ComboBox"
              Call ComboBoxAction
              Call ComboboxProperties
       Case "CustomGrid"
              Call CustomGridAction
              Call CustomGridProperties
                                  :
                                  :
End Select
```

The desired set of actions to be performed on these controls is specified in the test repository. Workbench reads these actions and interacts with the software.

For example, a Click action can be performed on a button control. An action called 'Select Item' can be performed on a Combo box. Textbox is to be used for entering values. Workbench handles these actions using the following code:

```
If Control.Exists = True Then
       Select Case Action
              Case "Click"
                     Control.Click
              Case "SelectItem"
                     Control.ClickItem(ControlValue)
                     :
                     :
       End Select
Else
       Log.Error("Controldoesnotexists:" & ControlName)
End If
```

VERIFICATION OF OBJECT PROPERTIES

Another important part of the automated test execution is of verifying the observed responses given by the system after performing a particular action on a control. One action can generate one or multiple system responses.

For instance, by selecting an item in a particular combo box it is expected that the system may have three responses like setting a selected value in combo box, hiding one of the existing text boxes and displaying a new textbox with a default value. In traditional hardcoded scripting, this can be done by inserting checkpoints which are always hardcoded and software specific. In an agile environment, this activity can be tedious and time consuming. Workbench uses the same test repository for the list of checkpoints and handles their properties dynamically.

Workbench reads the values of a particular object property by the following list of commands:

```
Control.Enabled
Control.Exists
Control.Visible
Control.WndCaption
Control.AutoSize
Control.Font.Name
Control.wText
Control.VisibleonScreen
```

For example, to check the visibility of a control on screen, it checks the runtime value of `control.VisibleonScreen` and verifies it with expected values in Test repository

Workbench can also identify and verify all the visible and invisible properties of an object. A set of property checkpoints can be used throughout the software to verify the consistency. The checkpoints like Font, Font Type & Font Size and Alignment can be cross-checked throughout the software. Reusability of property checkpoints can be done without any change in the Workbench.

HANDLING UNEXPECTED EVENTS

In any automated scripting, one of the required tasks is to handle unexpected events. Due to some major bugs in the software, some unexpected events like crashes occur which lead to interruption in the test execution. Manual intervention is required to execute the further script. Workbench handles these unexpected events efficiently and proceeds with the further script. After every action Workbench checks for any unexpected event that may have occurred. If that is indeed the case, the details are printed in the test log. This can be done by using the `WaitWindow` method and the code for the same is:

```
Set MsgWind = Sys.Process(P).WaitWindow("#32770", "*", -1, 1000)
```

CUSTOM CODE INTEGRATION

While working with Workbench, some situation may occur where common methodologies used may not handle some controls. Here, Workbench provides a way of writing custom code. It easily integrates the custom code without affecting Workbench code. This generally applies when TestComplete is not able to recognize a particular control or identify some properties of a control.

THE TEST REPOSITORY

The Test Repository has a predefined standard structure. This Excel workbook can contain sheets like Main Sheet, Message List and Windows Specific Details.

MAIN SHEET

The sheet named “Main Sheet” contains the list of Test Scenarios and the required details. It contains Process Name, Build date, Scenario Id, Scenario Description and some other quantities as shown in the Figure 5.

Process Name	Architect
Executed on Build	01-Apr-11

Test Scenarios List					
Scenario ID	Description	Detail Scenario Flow		Run	Pass/ Fail
		From Row	To row		
1	Single field validation on sample size tab	17	18	Run	
2	Single field validation on Dose Toxicity Curve tab	20	22	no	
3	Single field validation on Design Method tab(Model type)	24	27	Run	
4	Application flow Testing	29	43	no	
5	Dialog navigation Testing	45	50	no	
6	Design method tab validation (Prior distribution)	52	58	no	

Figure 5

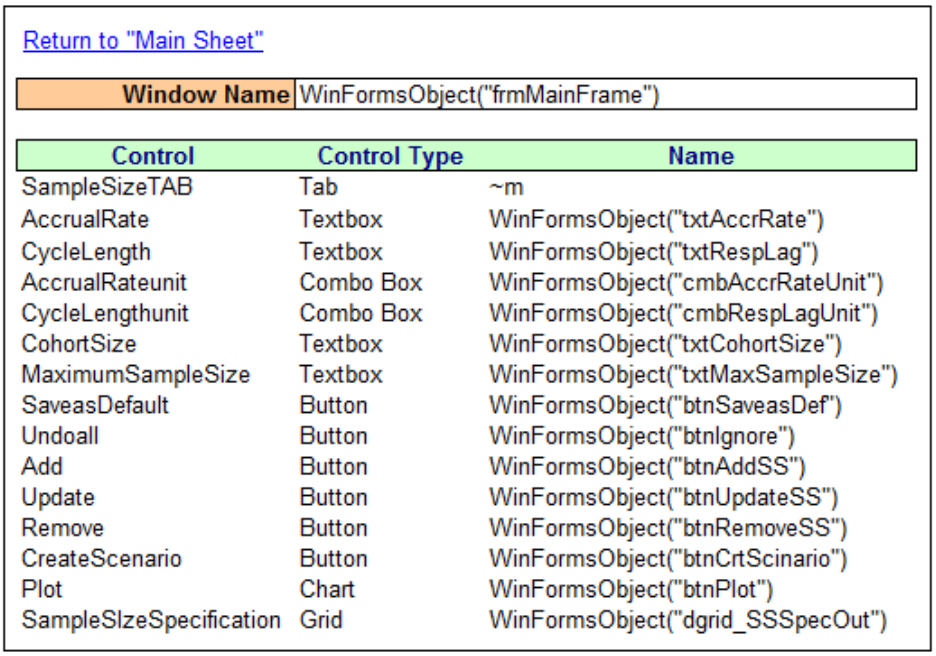
The section ‘Detail Scenario Flow’ is based on a modular approach. Test scenarios can be split into different modules as per the requirements. Because of the modular approach, test cases created for a particular module can be reused across different test scenarios. With this approach, efforts required for writing the test cases are reduced the test cases can be easily maintained. Refer to Figure 6 for more details.

Detail Scenario Flow					
Scenario ID	Module Sequence	Control Actions & Test Data			Comment
		On Sheet Name	Execute From row	Upto row	
1	Insert CRM	Menu Bar	10	10	
	Add Sample Size Scenario	Sample Size	34	37	
2	Insert CRM	Menu Bar	10	10	
	Add Dose Toxicity Curve	Dose Toxicity Curve	65	65	
	Add Dose Toxicity Curve	Dose Toxicity Curve	38	63	
3	Insert CRM	Menu Bar	10	10	
	Add Design Method	Design Method	58	58	
	Add Design Method	Design Method	62	63	
	Add Design Method	Design Method	37	39	
4	Insert CRM	Menu Bar	10	10	
	Add Sample Size Scenario	Sample Size	11	11	
	Add Dose Toxicity Curve	Dose Toxicity Curve	65	65	
	Add Dose Toxicity Curve	Dose Toxicity Curve	27	27	
	Add Dose Toxicity Curve	Dose Toxicity Curve	48	48	
	Add Dose Toxicity Curve	Dose Toxicity Curve	53	53	
	Add Dose Toxicity Curve	Dose Toxicity Curve	11	11	
	Add Design Method	Design Method	58	58	
	Add Design Method	Design Method	14	14	
	Add Design Method	Design Method	47	47	
	Add Design Method	Design Method	16	16	
	Add Design Method	Design Method	11	11	
	Add Design Method	Design Method	52	52	
	Add Design Method	Design Method	57	57	
	Add Simulation Settings	SimulationSettings	14	14	

Figure 6

WINDOW-SPECIFIC DETAILS

This sheet should contain the window name, list of controls and test cases. This list can look like the one shown in Figure 7. It consists of columns like control names used in test cases, control type, and real time control name which will be used during test execution. Using this type of structure, test scenarios and test cases can be written well in advance *even when the actual software build is not ready*. At this stage the two columns, Control and Control Type are filled. This can be done with the help of mock-ups or protocols. Once the software build is received then the last column of Actual Control Name can be updated. In traditional method of scripting, this kind of test case creation is not possible and it also requires stable build for preparing the automated test cases.



Return to "Main Sheet"

Window Name	WinFormsObject("frmMainFrame")

Control	Control Type	Name
SampleSizeTAB	Tab	~m
AccrualRate	Textbox	WinFormsObject("txtAccrRate")
CycleLength	Textbox	WinFormsObject("txtRespLag")
AccrualRateunit	Combo Box	WinFormsObject("cmbAccrRateUnit")
CycleLengthunit	Combo Box	WinFormsObject("cmbRespLagUnit")
CohortSize	Textbox	WinFormsObject("txtCohortSize")
MaximumSampleSize	Textbox	WinFormsObject("txtMaxSampleSize")
SaveasDefault	Button	WinFormsObject("btnSaveasDef")
Undoall	Button	WinFormsObject("btnIgnore")
Add	Button	WinFormsObject("btnAddSS")
Update	Button	WinFormsObject("btnUpdateSS")
Remove	Button	WinFormsObject("btnRemoveSS")
CreateScenario	Button	WinFormsObject("btnCrtScinario")
Plot	Chart	WinFormsObject("btnPlot")
SampleSlzeSpecification	Grid	WinFormsObject("dgrid_SSSpecOut")

Figure 7

CONTROL-SPECIFIC DETAILS

Automated test cases are the set of actions that are to be performed in a sequence on different controls. Re-execution of this set will help us in validating the software against the expected output.

For example, let us consider ***Architect*** software. Following actions are to be performed on Architect:

1. Click on Sample Size tab
2. From the Combo Box of Accrual Rate Unit, select “Day”
3. Specify Accrual rate=10
4. Click on Compute button
5. Verify results

Here, every step is a combination of a control, an action to be performed on it, the value to be entered for that control and the response verification. For instance, in step 2, “Accrual Rate Unit” is a control, “Select” is an action, “Day” is a value to be entered and the system response is the desired value, “Day”.

The above events can be stored in a structure as shown in Figure 8.

Test case Step 1				Test case step2			
Control	**Action**	**Value**	**Property Validation**	**Control**	**Action**	**Value**	**Property Validation**
Sample Size Tab	Click	NA	NA	Acc. Rate Unit	Select Item	Day	Day

Figure 8

WORKBENCH WORKFLOW

Figure 9 describes the workflow in the Workbench.

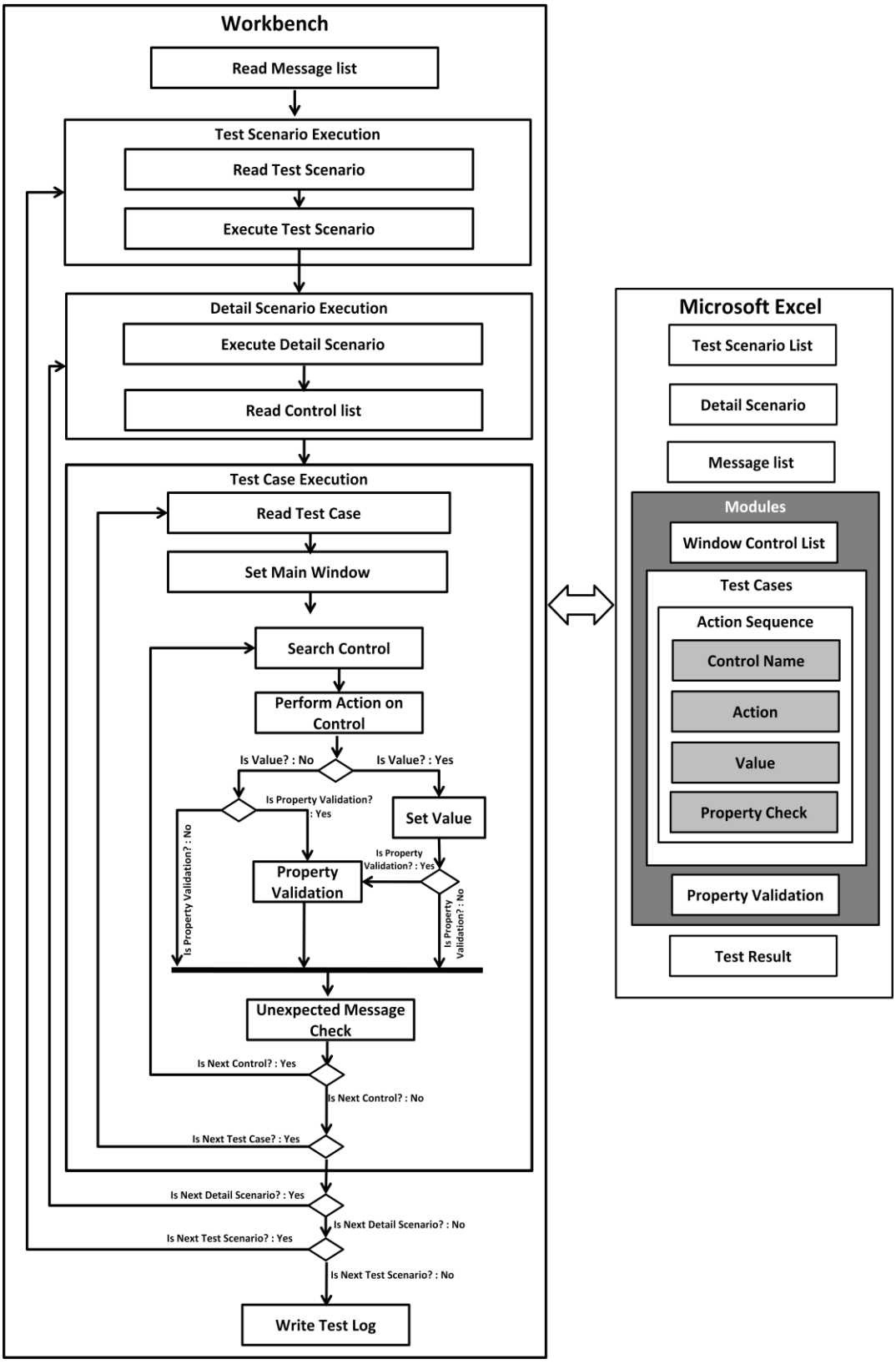


Figure 9

IMPLEMENTATION

Initially, Workbench was developed in ***TestComplete*** version 7 and was used for testing the well-known Cytel products, namely, ***SiZ***® version 2.0, ***Compass***® version 1.2 and ***Architect***. Workbench could successfully run hundreds of test scenarios without any code change in TestComplete. Here, the TestComplete code was common for all the three products. This resulted into drastic reduction in testing time and effort.

As compared to the conventional method (Figure 3), Workbench reduces the size of the code and also makes it much simpler to comprehend. Refer to Figure 10 which is the part of the code from actual Workbench.

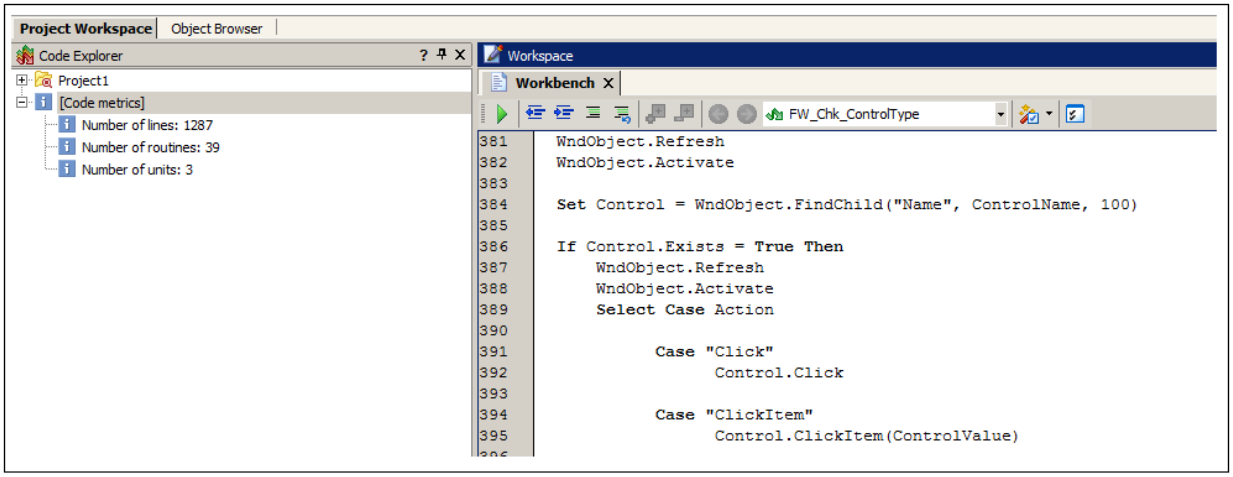


Figure 10

Workbench is now successfully ported onto ***Visual Studio 2010*** with no change in the Repository structure. The Workbench is now being used for testing software ***East***® version 6 of Cytel.

6. WORKBENCH ADVANTAGES

The main advantage of this approach is that it provides maximum maintainability and flexibility.

DEVELOP TEST CASES, SCRIPTS EARLIER

Test cases can be constructed much earlier in the development cycle, and can be developed as data files through a predefined structure in Microsoft Excel. The elements of the test cases can be easily modified and extended as the software itself becomes more and more defined. The scripts that process the data can be created as soon as the mockups and methods have been defined.

MINIMIZED MAINTENANCE

By constructing the test script out of modular, reusable routines, maintenance is minimized. The script routines need not be modified unless a new type of control is added that supports different actions; adding new control types simply requires additions to the handle script for new control.

PORTABLE ARCHITECTURE BETWEEN TOOLS

This approach is also portable between different test tools. As long as the underlying script language has the equivalent set of commands, test cases in this format could be executed by a script library created in any tool. This means you are free to use different tools for different platforms if necessary, or to migrate to another tool.

Figure 11 explains the portability of test cases from TestComplete to Visual Studio 2010 coded UI test.

Between TestComplete and Visual Studio the difference is only in the control names. By replacing the control names in the Repository and creating Workbench in Visual Studio, the test cases were ready to reuse.

Control	Control Type	TestComplete	Visual Studio 2010
SampleSizeTAB	Tab	~m	~m
AccrualRate	Textbox	WinFormsObject("txtAccrRate")	txtAccrRate
CycleLength	Textbox	WinFormsObject("txtRespLag")	txtRespLag
AccrualRateunit	Combo Box	WinFormsObject("cmbAccrRateUnit")	cmbAccrRateUnit
CycleLengthunit	Combo Box	WinFormsObject("cmbRespLagUnit")	cmbRespLagUnit
CohortSize	Textbox	WinFormsObject("txtCohortSize")	txtCohortSize
MaximumSampleSize	Textbox	WinFormsObject("txtMaxSampleSize")	txtMaxSampleSize
SaveasDefault	Button	WinFormsObject("btnSaveasDef")	btnSaveasDef
Undoall	Button	WinFormsObject("btnIgnore")	btnIgnore
Add	Button	WinFormsObject("btnAddSS")	btnAddSS
Update	Button	WinFormsObject("btnUpdateSS")	btnUpdateSS
Remove	Button	WinFormsObject("btnRemoveSS")	btnRemoveSS
CreateScenario	Button	WinFormsObject("btnCrtScinario")	btnCrtScinario
Plot	Chart	WinFormsObject("btnPlot")	btnPlot
SampleSIzeSpecification	Grid	WinFormsObject("dgrid_SSSpecOut")	dgrid_SSSpecOut

Figure 11

NO TOOL EXPERTISE NEEDED TO CREATE TEST CASES

Once the whole logic is defined and stored, the person using the Workbench does not need expertise in TestComplete or Visual Studio. All that is needed is an understanding of the software components, their names, valid methods and the values to be entered. This knowledge is essential for writing the test scenarios in any case

7. CONCLUSION

Data-driven automation provides significant advantages to the testing process, especially in the context of frequent feature changes and delayed availability of working software builds. It also lets the tester skill be focused on developing test cases rather than dealing with test script programming complexities. The Workbench described in this paper implements these ideas by enhancing test automation's productivity and avoiding a rigid distinction between the manual tester and the automation engineer.

CAUTION

Automation of custom UI components may need coding expertise.

The test automation tool used may not support all the technologies. In this case, one has to check with the tool vendor for any specific technology requirements.

8. REFERENCES

1. Bharath Anand R, Harish Krishnankutty, Kaushik Ramakrishnan, Venkatesh V. C., "*Business Rules-Based Test Automation: A novel approach for accelerated testing*" – an Infosys white paper
2. Linda G. Hayes, "*The Automated Testing Handbook*"

9. ACKNOWLEDGEMENTS

The authors would like to thank their colleagues at Cytel – VP Chandran and Vidyagouri Prayag for their ideas and directions, and also other colleagues in Cytel's QA team.

10. CONTACT INFORMATION

Priyadarshan Shinde priyadarshan.s@cytel.com +91 20 6604 0106
Ajay Sathe ajay.sathe@cytel.com +91 20 6604 0119

Cytel Statistical Software & Services Pvt. Ltd., 8th Floor, Siddharth Towers, Off Karve Road, Kothrud, Pune 411029, India