Better Testcases Through Improved Testability

Kanwarpreet Singh Grewal
Lead PV Engineer
Cadence Design Systems

Introduction:

Software today is getting more and more complex. This complexity brings fresh challenges in testing the software UI and its underlying functionality. Test engineers have to worry about a lot of different kinds of software controls, interfaces between software modules and etc. Test Engineers find some software modules easier to test than others. These test modules have "testability" built into them which help the test engineers make better testcases faster, tests that are easier to automate and hence have better chances of catching bugs in the code.

This paper defines testability and discusses its importance for software applications. We take a look at why it gets left out of most software components and why it needs to be there right at the design stage of the software development life cycle.

This paper also examines common software controls from testability point of view and suggests improvements that should be made to ensure better testability. We also believe that testability and usability in software go together and ensuring one generally ensures another.

Defining Testability

Testability is defined as the degree to which a system facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (from IEEE 90)

In other words Testability refers to the ease of making good tests and automating them. The Hardware world has always known the importance of testability and DFT (Design For Test) has always been a important criteria for the acceptance of any design. The importance of testability in the software world is just as important so as to ensure better testcases and thus a higher quality of shipped software.

The vital questions that need to be asked to see if a particular software piece has testability or not are:

1) Does the software have any features that can help Test / Automate it :

   This refers to the presence of features in the software that help in making good testcases and automating them. For example lets take a Internet Browser. A more testable Browser may be the one, which uses standard controls, logs messages in text files, keeps user informed about errors and their causes etc.

2) Does the Software give enough Information about its present state :

   The more the information given by a software execution at run time the better it is. For example if our file browser outputs all actions performed on it and all sites visited during a particular session into a log file then it would be easy to test it by comparing this text file with a golden (expected) file and checking the pass/fail status.

3) Does the software have controls that are recognised by third party UI automation tools :

Most third party UI automation tools can operate only on a set of commonly used controls. These controls are referred to as standard controls. UI automation tools can effectively automate only software, which uses these standard controls.

Causes For The Lack of It

If testability is so important then why does it get left out? Testability gets left out because developers are more concerned with the functionality the customers want and don't really spend much time thinking about what testers want.

But the problem is not with the developers alone. Testers also were never aware that they could (and should) demand functionality / hooks that can help them test more effectively.

Another reason is that sometimes the Management does not support the cause of testability enough. If the Management gives lower priority to enhancement requests for testability versus customer enhancements then testability gets left out. This later hurts the software quality and increases the time needed to test the software.

More Testability Means Better Testcases and Increased Automation:

Testability generally means that testing engineers have more ways of approaching a feature than one and thus have many paths to a feature. This increases the number of test critical paths that can be effectively explored with a aim of testing them. Testability also results in more information at each stage of the run time and thus increases the tester's visibility into the internals of the software helping him make better testcases. Thus increased testability means a easier coverage of the software features through testcases.

Testability and Usability go together and ensuring one goes a long way ensuring the other. This is especially true in case of UI intensive software where more usability means Keyboard shortcuts, consistent tab controls and standard UI controls. As we will see later these features also increase the testability of the UI. Standard UI controls which are recognized by third party controls also are more usable. A more usable software gives out more informative messages and is open to interfaces with other software applications. These very features make the software more testable too. These information/error messages can be trapped to know the state of the system under test and the interfaces can be used to make test utilities that can operate on the software.

Testable software is built with the test engineer and his tools in mind and it realizes the limitations of the test engineer and his tools. Thus making testcases for such software is easier and the testcases are more effective.

Testability Guidelines for UI intensive tools:

The greatest need for testability is felt in UI intensive applications. UI applications tend to hide the insides of the software and usually give only limited interfaces to the user. Also the methods used to create UIs are very diverse and dependent on the operating system. All these limit the amount of visibility a tester has into the system and makes testing and automation a great challenge. Thus testability is really important for UI applications. We discuss some general UI tools testability guidelines below.

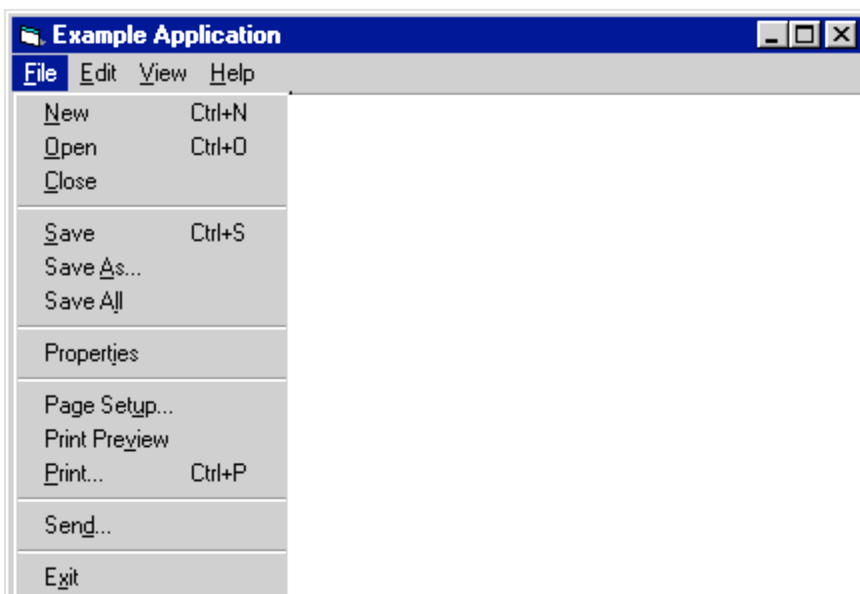
The way to test UI applications is to use specialized UI automation tools. However these tools assume that the UI is built in a particular way. If this is actually the case then testing the application becomes easy. The most commonly used UI controls are Editboxes, Comboboxes, Buttons, Menuitems, toolbars, trees, TabControls , taskbars , statusbars or controls derived from these. As far as possible these controls should be used to design the UI. This would ensure a good level of coverage for automation from UI automation tools. We describe here some guidelines that can help design testable UI modules. But before that the most important guideline for UI controls:
"Any Changes in UI should be carefully done. Many times a Test engineer's tests may break because of some seemingly harmless changes in UI by the development team "

Standard and Non-Standard UI Controls

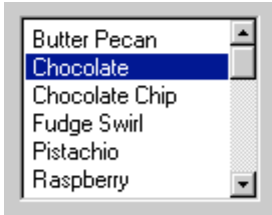
Software should use as far as possible supported UI controls. They are the easiest to test. The supported/standard UI controls are the ones that one or the other third party UI automation tools can understand. If you cannot find any supported control that can do that job for you then think about controls that tools like Visual Test can be made to understand using class definition commands which map a UI control to its corresponding standard UI control. However non-standard controls have to be used then it is important to make sure that there should be a way to access the control by using the keyboard. The reason for this is that UI automation tools may not be able to operate on the controls and the automation may have to be done be giving keyboard actions in scripts. Using the mouse may result in coordinate actions, which are not likely to be robust enough.

KeyBoard Access

It is very important that ALL UI controls should have a keyboard access/shortcut. This not only gives the tester an alternate method to access the UI but also is a safety net incase non standard controls are being used . The keyboard actions are always recognized by UI automation tools .It should be possible to do most things (if not all things) using the keyboard. This decreases the test's dependence on the mouse clicks and on coordinates (if the controls are non-standard).



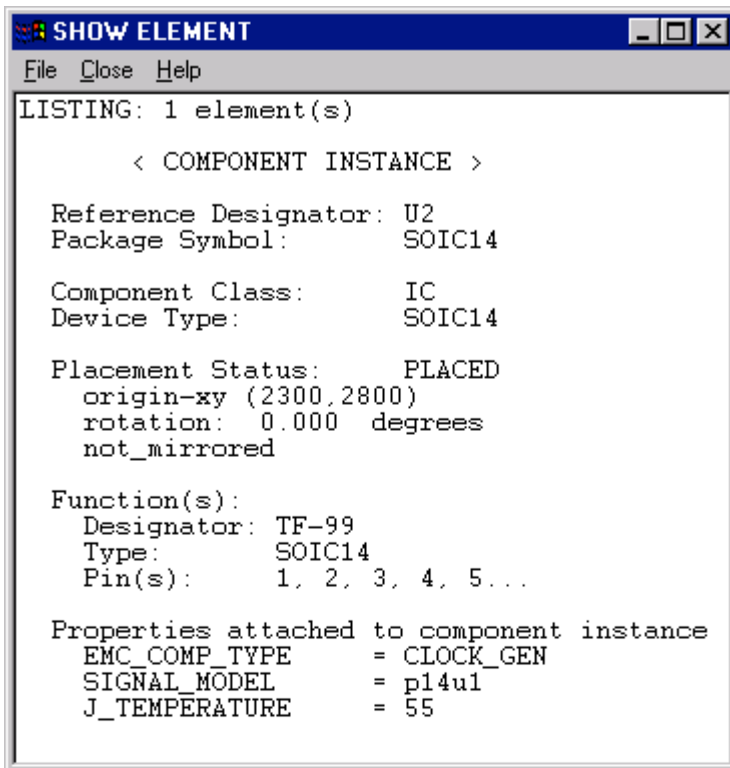
For example in the above menu item it should be possible for the user to access the Exit menu item by pressing the Alt+x key. Also it should be possible to move from one control in the form to another using the TAB button.



For example in the above list box it should be possible to just type in the name of whatever we want instead of using the mouse. And we should be able to use the ARROW keys to do the selection and the TAB key to move to the next control.

Information Display on the UI:

Whenever an information is displayed on the UI there should be a mechanism to save that information to a file. This makes the information useful/usable for the test and its automation script.



For Example we should be able to save the above form to a file by using the File->saveas menu item. Then it can be compared with a golden/expected file using OS specific utilities.

If however the tool does not support displaying information to forms then one should consider displaying such information on a standard UI control/form. Third party UI automation tools support any information extraction from

their supported controls. So if the information is a part of a standard control checking functions / counting functions can be done on it enabling the test to evaluate if the output/information is as expected.

Information must NEVER by dumped in a non-standard UI control. It is visible but you can never get it out or check it. Automating and verifying such a thing is nearly impossible.

Also the Application Under Test (AUT) should have lots of information displays. The more the better. The information displays tell the test about the status of the execution.

Checking the Form's UI:

Verifying that the UI is exactly what it should be is generally a very difficult task. For example a simple form like the one shown below would require a lot of code to verify it. One has to look at each UI control and check if it exists and if it has the default value in it.

Drawing Parameters
Project: C:\
Drawing: sig_tutor.brd
Type: Drawing
User Units: Mils
Size: A
Accuracy: 0 (decimal places)
DRAWING EXTENTS
Left X: 0 Lower Y: 0
Width: 11000 Height: 8500
MOVE ORIGIN
X: 0 Y: 0
OK Cancel Reset Help
No changes made until 'OK'.

One testability feature that makes testing this really easy is one, which is similar to the printform command in some Cadence software. This command prints the values of any form in a text file, which can be compared with a golden file. This lets the test engineer check the UI contents easily and very fast.

And the following info goes into the text file for the above form As one can see the info is complete and can be used to compare with the expected output.

FORM: <<< Drawing Parameters >>>
Project: C:\
Drawing: sig_tutor.brd
Type: Drawing

User Units: Mils
Size: A
Accuracy: 0 (decimal places)

DRAWING EXTENTS
Left X: 0 Lower Y: 0
Width: 11000 Height: 8500

MOVE ORIGIN

X: 0 Y: 0

OK Cancel Reset Help

Testability in Forms:

All Forms should be resizable and one should be able to maximize or minimize them. This enables the test engineer to do a setting of the position and sizes before operating on the form. This makes the coordinate functions more reliable.

When a form launches another form the focus should shift to the new form and it should come back to the original from when the new form is closed.

The positions and names of the UI controls in a particular form should stay the same for new releases of the software. Since the UI automation tools rely on the positions and the Class names of the Controls for their operations it is important that they don't change too much. The tests created for one release should keep running for the next release too.

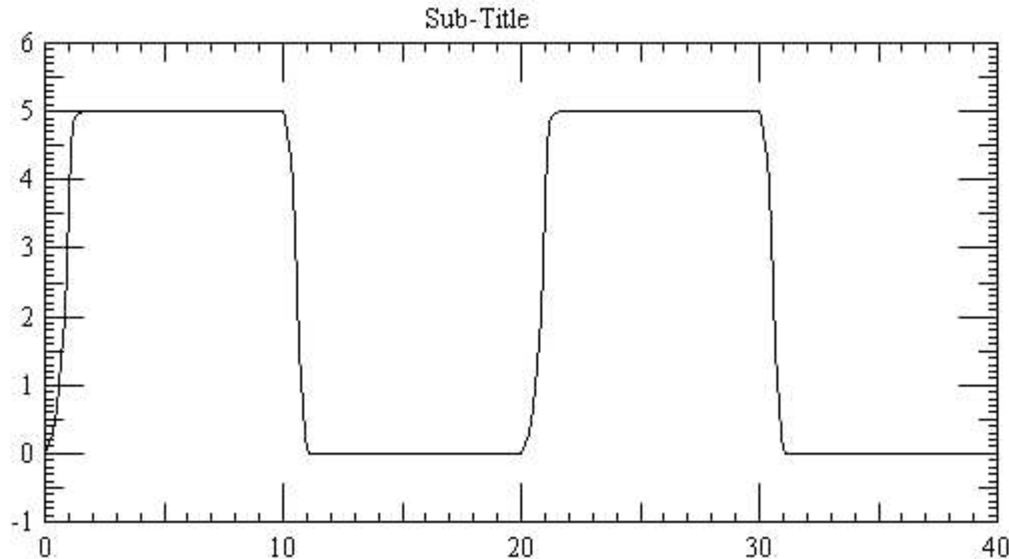
Record Replay Capability:

One of the most useful testability features that could go into a tool is a record replay/script capability. Once recorded the script can be replayed and the script reproduces the recording. The scripting language need not be too general. It can just be a set of commands that are sent to the AUT and the AUT performs those functions. A Record Replay feature is always a little time consuming to built but once done it can add a lot of value to the testcases.

The difference between an internal record replay facility and a third party record replay facility is that the internal utility can be made to operate on the UI as well as the underlying functionality. Also the problems coming from non-standard controls can be bypassed by issuing commands directly to the internal functions which are called by the controls. So the customized record replay facilities are better than more general UI testing tools.

Dealing With Waveform Display Tools:

Tools that display waveforms are particularly difficult to test/automate. Testing whether the waveform that is displayed is right requires a careful visual comparison, which can be very time consuming and prone to human errors.



So there should be a feature in the tool to dump the waveform in a tabular text format. This would enable the Test Engineer to compare this tabular version of the waveform with his golden waveform. Also such a table should be able to be imported into the tool to display the corresponding waveform.

How to go about it

Implementing testability or taking it up requires a joint effort of the development team, the testing team and the management. The starting point is that everybody realize the importance of testability and how short term efforts in it can pay long term dividends. Here is how in our view testability can be approached:

1) The first step is to identify the areas in the software that are difficult to test and to list down the reasons that make these areas difficult to test and what kind of testability features would be needed to make these easier to test.
2) The management team and the developers have to be convinced about the importance of testability.
3) Get a buy in from the development team that the testability features would be implemented.
4) The testcases should take advantage of these testability features by using them as pivots to access “difficult to test areas” of the software.

The right attitude to testability is when the developers are thinking about adding testability for better testcases and the testers are thinking about the features that can help them test a particular feature. So this is in a way a reversal of roles with developers thinking about tests and the testers thinking about features.

The Right time to influence testability

The right time to influence testability is at the design stage of the software development process. This is because at this stage the testability features can be build in the framework of the software design and can be really effective in testing the insides of the software. Testability features that get missed out in the design cycle are very difficult to put in later and can only be patched into the software.

Our Experiences

We learnt the importance of testability the hard way. While testing Cadence's Schematic Capture Software and its related interfaces we faced ever increasing failures in our automated regression test suite. The automated regression tests used to fail every new release and the time to fix the scripts became enormous. We realized that the problem was in testability and this was causing Visual Test testcases to fail release after release. We decided to separate the UI testcases and batch testcases and have a scripting utility to test the main functionality of the software and to reduce the UI testcases to just test the interface with the user. We used an internal APIs to create utilities to do frequently used actions. We made a set of testability guidelines and convinced the development to incorporate the features suggested by the guidelines in their software.

By improving the testability in the software we got huge improvements in our testcases and the robustness of their automation. To take a particular example: In a cross referencer application which was a utility application to the design capture software we had to test the dumping of cross reference symbols on various pages and hierarchies of the schematic sheets. All these symbols were graphic data and testing it meant either bitmap comparisons or visual checks. None of the approaches was very good. Bitmap comparisons are not robust enough and manual testing was too time consuming.

We convinced the development team to write a utility to dump out these symbols and the relevant data in a test file, which we could compare. This utility was written in an API called Skill. The results were very encouraging. We achieved 100% automation and the testcases were very robust.