Testing X/Motif Applications

Achieving Quality Goals Through Automated GUI Testing

Integrated Computer Solutions Incorporated

The User Interface Company™

Phone: 617.621.0060
Email: info@ics.com

www.ics.com

Automated Testing of X-Window Applications
Choosing the Ideal Tool for Automated GUI Testing

Table of Contents

Automated Testing of X-Window Applications
Choosing the Ideal Tool for Automated GUI Testing

Overview

GUI testing is different from other types of testing and requires the support of special tools. In this whitepaper, we look at features that are critical for effective automated GUI testing. Throughout the discussion, we use a simple example to illustrate how one commercial product, Replay Xcessory (more information as PDF), compares against the required features and we will assess its ability to help the achievement of quality goals through an effective testing process. Instructional webinars are held on this and other development topics. A schedule is available by clicking here.

The importance of GUI testing

Development teams are spending more time today than ever before crafting user interfaces that make application use easy to use, intuitive, and foolproof. Some applications have over 60% of their code devoted to the user interface. [MEM02] Many safety-critical systems, such as flight control software and medical systems rely on the correctness of the complete application, including the user interface. Errors in any part of these systems can result in loss of life.

GUI testing is more than just pushing buttons or selecting menu items. It is a complete set of methods and practices that supports testing. It guides the quality professional or the developer to test code that implements the user interface as rigorously and completely as they test other code in the system. GUI testing tools must support all types of testing, including exploratory testing, regression testing, boundary testing, and stress testing for the system through the user interface. Before we continue with a discussion of GUI testing tools, let's look a little at what we mean by each type of testing.

Exploratory testing

Exploratory testing is a method for designing tests while executing tests. When performing exploratory testing, the tester literally explores the product's features in a systematic way. During the exploration, we find defects, report bugs, and generate ideas for future tests. The tester keeps notes of what she did during the exploration and identifies future areas for exploration. Exploratory testing is an extremely effective use of testers' time when a new product, or new release of a product, is initially tested.[1]

Regression testing

Software products are built and tested many times during their construction. Organizations that follow modern software development methods use iterative, incremental development. This lets the development team add functionality in small

[1] Further information about exploratory testing can be found at: http://www.satisfice.com/articles/what_is_et.htm.

increments, ensuring that the new features work and that previous work is not broken by the new additions. Regression tests are required to support iterative, incremental development.

Regression testing is defined as: *The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software.*[2]

One important characteristic of regression tests is that they are usually automated. Lack of automated tests results in expensive, error-prone human resource expenditures to rerun the tests for every build or release.

Boundary testing

Boundary tests are tests that exercise the software by using data that is at the boundaries of valid values – usually upper and lower limits. When there is a GUI, we identify those elements that allow data input or display output, identify valid and invalid values and group these into equivalence sets. We then execute test cases that exercise the boundary value and values just inside and outside of the boundary. For example, if there is a rule that says the login name must be at least eight characters containing at least one non-alphabetic character, we might try names that are seven, eight, and nine characters long. We might also try values that have no alphabetic characters in them, and so on.

Stress testing

Sometimes software has built into it areas that are particularly susceptible to different forms of stress. Depending upon the type of software and the potential loss from a software failure, part of the testing resources should be spent on stress testing. For example, if you develop a system for stock trading, you need to be able to handle not just the usual amount of trades, but be sure that you are able to handle trades during the most hectic peak periods the end user might experience. To test these conditions, you need to find a way to force extremely high traffic loads on the system.

GUI Testing Tools

All of the testing techniques described above apply to software that has a GUI component, and they all can benefit from automation. Even exploratory testing can be improved by having a tool that captures the gestures of your exploration and can play them back, or allow you to modify them in interesting ways.

There are many types of GUI testing tools, each with its own unique capabilities. When considering adopting a testing tool, you will likely want one that is as broad as possible, yet gives you as much control and power for each area that it addresses. Any testing tool should minimally support the above four testing techniques. However, there is more to

[2] See http://www.pcwebopedia.com/TERM/R/regression_testing.html.

look at in a testing tool, especially a GUI testing tool. The following list describes many attributes you should look for in your testing toolset.

- *Intuitive interface.* Any tool should be intuitive to the user. One expects that a sophisticated tool will contain some features that are specific to the discipline, such as testing. However, you do not want a tool that takes a long time to learn and from which it takes a long time to begin to get results.
- *Adapt to changes easily.* Change is the one constant we have in software development. As a software product evolves, one of the most changeable parts of the system is the user interface. As beta customers use the product and provide feedback the user interface will change to reflect preferred colors, locations, text, and so forth. When this inevitable event occurs, you don't want to have to rework all of your tests.
- *Scriptable.* This is perhaps the most important attribute to look for. The real power of any testing tool lies in the capabilities provided by its scripting language and extensions. The ability to create and modify test scripts translates directly to reduced manual effort.
- *Easy to implement test cases.* Testers of all levels want to be able to design and implement test cases easily. A testing tool must make it as natural as possible for the tester to go from concept, to implementation, to execution, and to analysis of test cases quickly and efficiently.
- *Easy to debug and analyze results.* Developing automated tests is like developing software. You need to be able to incrementally develop your test and quickly identify and fix errors. Then when the tests are running, the tool must provide a rich analytic capability to let you determine product readiness quickly and easily.

Developing initial GUI tests

Through the rest of this paper, we will look at different ways to test GUI applications effectively using Replay Xcessory. We will use the example `xcalc` program that is delivered with the Replay Xcessory distribution. `xcalc` is a simple scientific calculator application with an interface shown in Figure 1. The Replay Xcessory documentation contains a tutorial on how to set up the testing environment for `xcalc`.

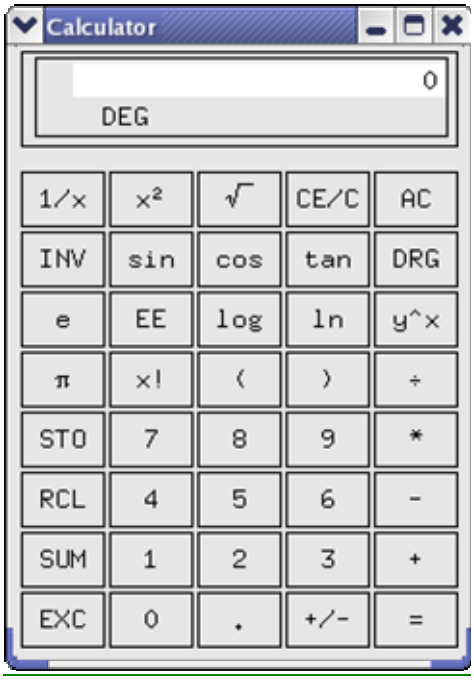

Figure 1. The xcalc application.

When faced with a new application, one of the most effective ways to begin testing is through exploratory testing. In exploratory testing, you explore the application in a systematic, but not formal manner. You keep notes on what you find, and record any defects. You use the notes to help you decide what areas seem most promising for future testing – that is, which areas you think might contain defects that your testing can uncover.

Looking at the interface of `xcalc` in Figure 1, one can begin to see different types of operations that might be grouped into test sets. During exploratory testing, we should try at least one test case from each set, just to verify normal operation of the product. Such tests would include:

- basic arithmetic
- mathematical functions
- overflow and underflow

Using a GUI test tool during exploration

When you perform exploratory testing, you tend to wander through the application. Sometimes you will want to back up to a state that you previously visited and take a different path. You may also make a mistake in your interaction with the product under test and need to back up. If you could have a step-by-step record of all your actions, you can easily ensure that you can recreate the state you need. A GUI test tool that records all of your gestures provides you with such a record.

Replay Xcessory writes a test script as you use the application. The script is your record. When you finish an exploration, simply use your favorite editor to annotate the script, remove those parts you don't want, and use the script as the basis for future testing.

Listing 1 shows a part of the listing from an exploration of xcalc. This section of the script describes the gestures taken to test some of the trigonometric functions. The scripting language is Tcl.[3]

```
subtest    5
click      {/CE/C}
click      {/0}
click      {/tan}
click      {/=}
snapshot    {vr0005.snp} -object {LCD}
subtest    6
click      {/CE/C}
click      {/9}
click      {/0}
click      {/tan}
snapshot    {vr0006.snp} -object {LCD}
```

Listing 1. Section of script that tests trigonometric functions.

The code in listing one contains two subtests. The first subtest (5) gets the tangent of 0 degrees. The second (6) gets the tangent of 90 degrees. Each subtest ends with a snapshot of the LCD object. The LCD object is the calculator window that displays the numbers. There are many different types of snapshots. Object-level snapshots are usually fine for most tests. The last snapshot is captured in the file vr0006.snp. This file is shown in Listing 2.

```
! rx:snapshot
*LCD.sensitive:          True
*LCD.label:    6.189871e+14
```

Listing 2. Snapshot of the result of tan(90).

Notice that the value shown in the label is 6.189871e+14. This is the number that is displayed by the calculator. It is also wrong. The tangent of 90 degrees is either

[3] Tcl is the *Tool Command Language* developed by John Ousterhout. Tcl was designed to be a scripting language that could be easily integrated with other programs to drive them and interact with them. More information about Tcl can be found in the references or at http://www.tcl.tk/software/tcltk/.

undefined, or infinity. This is clearly a problem with the `xcalc`. It also indicates an area that should be explored further.

Testing values such as 0 and 90 degrees for trigonometric functions are good examples of boundary tests. Whenever you test you want to design tests that exercise boundary conditions. In fact, the rule-of-thumb is to test each boundary condition, a small increment below the boundary condition, and a small increment above the boundary condition.

There is one feature of the snapshot file that is worth noting. Since the snapshot granularity is at the object level, there is no capture of pixels – just the simple values of the object's properties are recorded in textual form. This means that if the size, color, or some other non-important (for our testing purposes) property changes, the test is immune to such changes. This is one of the characteristics we want in our GUI testing tools.

We can now annotate the script to provide a record of what we did, what we expected, and what we plan to do. This is shown in Listing 3.

```
subtest    "tan(0) = 0"
click      {/CE/C}
click      {/0}
click      {/tan}
click      {/=}
#snapshot    {vr0005.snp} -object {LCD}
# Test passes
#
subtest    "tan(90) = error"
click      {/CE/C}
click      {/9}
click      {/0}
click      {/tan}
#snapshot    {vr0006.snp} -object {LCD}
# Test fails. Result is 6.189871e+14
# Note: Test other possible error conditions and make
```

Listing 3. Annotated script.

In just a few minutes, we have a record of our actions and a plan of attack. We identify those areas that have the most promise for finding errors. Now we go onto a systematic, in-depth investigation of the application for the areas we flag as problematic.

Basic scripting

There is another advantage for recording exploratory sessions. The resulting scripts can be used to see how to write additional scripts for further testing. By studying the generated scripts and looking at a few pages of the testing tool's user manual, you can usually begin writing your own scripts within a couple of hours from when you start using the tool. With Replay Xcessory, you only need to know a little bit about Tcl and then look at the documentation to find some of the extended commands that Replay Xcessory adds to the language, such as the `click` and `subtest` commands.

Listing 4 shows a revised script that was developed from the script in Listing 3. This script does the following:

- start the xcalc application and move it to a central point on the screen
- create a subtest with an appropriate name
- calculate tan(90)
- validate the results and indicate whether the test passes or fails

```
# replay:xcessory
#
# Script to test the tan(90) calculation in xcalc
startup    /mnt/share/Replay/examples/replay/bin/xcalc
activate   {xcalc}

currentwin {/Calculator}
move       {.}  @731,346
subtest    "Expect tangent error"
click      {/9}
click      {/0}
click      {/tan}
click      {/=}
set        result [getvalue {*LCD} {label}]
if {$result == "error"} then {
   pass "Got error as expected on tan(90)"
} else {
   fail "Expected error on tan(90) and got $result"
}
click      {/CE/C}
closewin   {.}
```

Listing 4. Script to test and record results of tan(90).

This subtest contains a few additional Tcl commands to save the value of the LCD label in the result variable. It then tests to see if the `result` was the string "error" or some other value. It then passes or fails the test based upon the comparison. When the test is

executed, the results are written to the test log that we can easily scan to see if the tests pass or fail. This is shown in Listing 5.

You usually do not want to run every test separately. You want to group them into a complete regression test run. However, you need to have the granularity of tests such that you are able to run jus the tests you want. Since all of the test scripts for Replay Xcessory can be run in batch mode, it is a simple task to create a makefile that runs just the subset of tests that you need. Each run can produce a single report that lets you immediately see the results of your tests. This is a very important feature to have as the release date approaches and you need to have a quick turn around on running and evaluating your tests.

```
                    Replay Xcessory Test Report

                   Generated on Sun Feb 13 22:13:16 2005

INPUTS
------
Test Package:  /mnt/share/replay-work/papersuite/exploratory
Script:        tangent.tcl
Baseline:
Result:
Run Command:       /mnt/share/Replay/bin/replay -display :0.0 -p -l tangent.tcl


OUTPUT LOG
----------


22:13:17  ** Subtest Expect tangent error **
22:13:22  Expected error on tan(90) and got 6.189871e+14          FAILED


TESTS SUMMARY
-----------------

Subtest Expect tangent error                FAILED

Percent of Subtests Passed:    0.0% (0/1)

Session Elapsed Time:                       0:00:08.00
Application CPU Time:                       0:00:00.00
```

Listing 5. Example test report.

Automated regression testing

The phrase “automated regression testing” is somewhat redundant. The only practical way to perform regression testing is through automation. The purpose of regression testing is to ensure that new code does not break old code. This can be especially important with GUI development.

User interfaces are complex. Even the simplest user interface will involve several graphical objects, events, and multiple threads of execution. Determining all of the

possible effects of a seemingly simple change is difficult, if not impossible. The following example will help clarify this.

Most applications with GUIs have several menus. Each menu selection may have a keyboard accelerator, such as CTL-S to save the current file. What happens when you add a new menu selection and inadvertently assign the same keyboard accelerator? Some user interfaces allow this if it is in a different menu and others will not recognize the second one. Will you remember to test all of the possible accelerators every time you make a change? Will you make sure that all events are going to the right places in your code?

Certainly, even if you remember to test all of the combinations, this can be a significant use of time if it is done manually. Therefore, you need to apply automation to the task. Every time you make a change to your code base, there are two things you should do before committing the new code:

1. Write new tests to exercise the new code.
2. Run *all* of the tests and do not check in the new code unless they all pass.

Your testing tools must be able to be executed in an efficient manner for this type of testing to be practical. With Replay Xcessory, you simply add a *tests* target to your makefile and make that target. Since Replay Xcessory can be executed from the command line as well as interactively, it satisfies the ability to work with a single test from the graphical user interface, and it can run all of the tests from the command line through the makefile.

Stress testing

A program must be more than simply correct in terms of its logic, it also has to be robust. The term “robust” has different meanings to different people. When you think about user interfaces, one attribute of robustness is whether the events are handled quickly enough and, if not, are they able to be queued in a reasonable manner. Possibly more important is whether the program will hang if there is too much GUI activity too quickly. You also need to determine if the application works properly when the number of objects it has to display or manage becomes large.

Having a testing tool that can be easily scripted allows you to test for these stressful conditions easily. Consider the following situation: you’d like to test the `xcalc` program to ensure that nothing bad happens when the user simply keeps pressing a button. The code snippet shown in Listing 6 does the trick.

```
$i = 10000
while {$i > 0} {
     click     {/2}
     $i = $i - 1
}
```

Listing 6. 10,000 key presses.

With just a few lines of Tcl code, we are able to insert stress tests as part of our normal testing. After pressing the "2" key 10,000 times we can check to make sure that the calculator screen has the maximum number of 2s displayed. If the program does not behave that way, or hangs, it becomes obvious and you can run this test whenever you need to without pressing the "2" button thousands of times each test run.

Seeing how the application scales with a different number of objects is also an easy task with tools that have good scripting. We discuss an effective way to do this in the section of this paper called "Raising the bar on your testing."

Developing GUI tests before the GUI is available

A popular testing technique today is called Test First Programming (TFP). It is also called Test Driven Design (TDD). The premise is a simple one. You write your tests first, and then develop the code to make the tests work. There are several benefits attributed to this approach. See the references for information about TFP.

If your organization has adopted TFP, it seems natural that you should develop all unit tests for your project, including the tests of the GUI components. While this might seem like an impossible task, having a scripting language that lets you use tags, such as LCD in the previous examples, makes TFP possible even for your GUI components. Let's look at how this works.

Let's assume that the `xcalc` application has not been developed and that you are one of the people responsible for testing the application; specifically you are responsible for testing the user interface. You want to perform testing like the tests above, but you have no code to work with. In some projects, you might go off, do other things, and hope that when the code is ready you have the time to get the GUI testing done.

Maybe you can take a different approach. It is a realistic to expect that the developers will have sketched out the user interface, even if they have not agreed upon all of the possible functions or placement of the widgets and other aesthetic properties. You decide to work with a developer to find out the names (or tags) of each of the visible widgets. This list of the widget names and the type of widgets is all you need to begin writing tests.

For xcalc, you might create a table like the following, incomplete, one:

Tag	Type	Description
CE/C	Button	Clear entry, clear the accumulator
0	Button	0 key
1	Button	1 key
2	Button	2 key
…		
9	Button	9 key
+	Button	Add to accumulator
…		
LCD	Display	Label contains the value in the accumulator
Calculator	Window	Main application window
…		

With this table, a little Tcl knowledge, and the Replay Xcessory user's guide, you can begin writing tests. If you wanted to write a test that performed a "smoke test" of each of the basic arithmetic functions, you would write a Tcl script as shown in Listing 7. This is only a partial listing, but the approach should be evident. While the developers create the code that implements the application, you can write tests that are ready to run as soon as the code is ready. All you have to do is insert the line that starts the application. If you have already set up the testing environment, you can put the code in now and remove the second line of the script – the comment about what needs to be done. This approach reduces the time between code and feedback and, ultimately, shortens the development time for the total application.

```
# Test of basic arithmetic in xcalc.
# TBD: insert actual application startup here.

activate   {xcalc}
currentwin {/Calculator}

subtest          "Addition smoke test"
click            {/CE/C}
click            {/2}
click            {/+}
click            {/2}
click            {/=}
set              result [getvalue {*LCD} {label}]
if {$result == "4"} then {
   pass "2+2 = 4"
} else {
   fail "2+2 = $result"
}
…
```

Listing 7. Test-first approach.

Adding a little more Tcl

At this point, you might be thinking that it will become tedious to check the results of each application in the way shown in Listing 7. You are correct. Now would be a good time to learn a little more Tcl. The language allows you to have functions and procedures. You can write a simple result check procedure called `checkResults` that takes a description of the test and the expected value. The procedure retrieves the result from the GUI component and compares it to the expected result. It then posts the pass or fail message. This is shown in Listing 8.

```
# Test of basic arithmetic in xcalc.
# TBD: insert actual application startup here.

#
# Test the result in the accumulator against an expected value
#
proc checkResults { testDescr expected } {
        set             result [getvalue {*LCD} {label}]
        if {$result == $expected} then {
           pass "$testDescr = $expected"
        } else {
           fail "$testDescr = $result"
        }
}
activate   {xcalc}
currentwin {/Calculator}

subtest         "Addition smoke test"
click           {/CE/C}
click           {/2}
click           {/+}
click           {/2}
click           {/=}
checkResults {"2+2" 4}
```

Listing 8. Adding a procedure to the previous script.

If you follow this approach, you gain several benefits:

- You develop tests from the beginning of the project.
- You learn the scripting language incrementally. You don't have to know everything about the language in order to write tests.
- You begin to develop reusable test assets that you can use as a library of test accelerators, such as the results checking procedure above. These can be Tcl modules or simple Tcl files that you "source" into your Tcl script.[4]

Raising the bar on your testing

Once you become comfortable with your testing tool's scripting language you advance to being a power user of the tool. At some point, you will find that you only use the record capability of the tool to get started with a new application, as we've done in this paper. Once you understand the roadmap of your application, you will spend your time in the scripting language. You can become a real power user when you end up developing your

[4] See any Tcl tutorial or manual for information on how to do this.

own language to express the tests for your particular application, or application domain succinctly.

There are different ways to develop such scripting capabilities. One is through a data-driven approach. The other is to develop a little language that reads in a "program" written in the new, domain-specific language, and translates it into the appropriate commands that drive your testing tool. Each of these approaches is, in effect, a little language for your application.

Replay Xcessory comes with an example of the data-driven approach to the `xcalc` program testing. You can find this example in the xcalc.tcl program in the examples. It reads a file named test.data. The data file is shown in Listing 9. This simple input format contains one calculation on each line. The line is read by xcalc.tcl and separated into a list of token. Depending upon the token, the appropriate action is taken. Either a button is pressed, or the result in the LCD label is checked against a pre-determined value.

```
0 + 2 = 5
1 + 1 = 2
5 - 2 = 3
3 * 7 = 21
10 - 2 = 5
1 * 9 = 9
1 + 1 + 1 = 2
5 * ( 1 + 4 ) = 25
1.1 + 2.3 = 3.4
4.4 + 2 = 2.2
```

Listing 9. test.data file for input to xcalc.tcl.

You can see how the data-driven approach makes testing applications like `xcalc` easy and removes the burden of knowing the scripting language from the person who writes the data files.

Sometimes, however, it may take more than just data input. You may need to extend the capabilities of the input data by adding specific commands. This concept will become clearer with an example.

Let's say that you are working on a project team that is writing software to monitor network performance. The application has an interface that shows the nodes in the network, their status, and other statistics. There are many requirements placed upon the user interface. You need to perform menu selections, enter data, and so on. One test you need to perform is to make sure that the system can "scale up" to handle hundreds, or even thousands of nodes on the network. The system is designed so that you can add a node to the network for monitoring purposes, or remove a node from the network. You certainly do not want to record a test session where you add 1,000 nodes to the network.

Not only would you have to make the appropriate gestures 1,000 times, but also you would have to create node names (since this is also part of the add node operation).

The little language approach lets us refine our testing in two parts. First, we will build a script that will accept data for adding and removing nodes. This is easy to do in a language like Tcl. We write two procedures, one called `addnode` and one called `removenode`. Each of these causes the appropriate actions and gestures in the application under test. It might involve filling in text fields in dialogs, making menu selections, and pressing several buttons. We also write an assert command that will test a condition and determine if it is true. If not, we write a failure message to the test report. If the condition is true, we write a pass message to the test report. At the end of the subtest, if there were any failure messages, the subtest fails.

When we read a line from the data file, we assume the whole line is nothing but a call to one of these methods, or is a predefined command, and use the Tcl `eval` command to evaluate the line. That is, it calls the procedure named by the first word on the line.[5]

After this first step, we would now be able to write a test by creating a data file like Listing 10.

```
subtest "add nodes"
addnode "node1"
addnode "node2"
addnode "node3"
assert nodecount == 3
removenode "node2"
assert nodecount == 2
```

Listing 10. Data for the first little language.

This is a useful extension to our scripting language. It will certainly make it easy for someone who knows the domain, but not the underlying test tool or scripting language, to write tests. But we can do better.

Since we want to ensure that our application scales up, we need to write a test that adds a thousand nodes. It would be good to remove the nodes from the network too. Writing each addition as a line in a script can become tedious quickly, and error prone. We can fix this by adding a simple looping construct to our language. We just add a `repeat` command that takes an integer indicating the number of repetitions of the loop. This

[5] Readers who have taken some computer science courses will recognize this as the standard read-eval-execute loop that is taught in several courses, especially those involving functional languages such as Lisp or Scheme.

command gathers all of the lines that follow it in the script up to an `end` command. Then it executes the gathered lines the specified number of times.

We can further improve the language by not requiring a name for each node. We alter the `addnode` command to generate a name if one isn't specified. We also modify the `removenode` command to remove a random node from the network if a name is not supplied. Now our final input to the little scripting language we have created looks like the file in Listing 11, which adds 1,000 nodes to the network and then removes them.

```
subtest "add nodes"
repeat 1000
addnode
end
assert nodecount == 1000
repeat 1000
removenode
end
assert nodecount == 0
```

Listing 11. Input to enhanced little language.

This is a powerful tool! If you were only going to write a single test, you would not develop a little language, but you want to write many tests that share similar characteristics. Our little language provides a vehicle for significant reuse.

Meaningfulness and portability

Designing and testing are not mutually exclusive activities in the software development process. In order for each to be effective, they must be connected. You want to design your code so that it can be easily tested. You want to design your tests so they are resilient to change in the design. For GUI testing this can be quite a problem. The user interface is one part of an application that will change many times during a project. It is subject to more iterations and increments than other parts of the application due to the nature of GUI development. You create something for the user to try and then adjust it based upon the feedback they give you. This cycle is repeated throughout the product development life cycle.

Not only will the visual elements of your interface change position, but they will also be renamed, resized, and have their attributes changed. In order to ensure minimum disruption due to these changes, there are a few guidelines of which you should be aware:

Establish a naming convention for your widgets and stick to it. There are many conventions people adopt when naming the elements of their user interface. You should adopt one and review your code and your tests to ensure that they follow the conventions

properly. Having a simple, yet consistent, naming convention makes it easy to determine the name from the widget type and its visible attributes.

Use a standard way to refer to the graphical elements. Depending upon the tool you are using, you will have the option of referring to widgets using a fully qualified name, or a minimized name, using wildcard characters. Neither of these methods is ideal, but mixing them only leads to confusion. Just as you should have a consistent naming convention to follow when you create widgets, you need a consistent naming convention to follow to refer to the widgets in the tests.

Use the most portable name possible. Depending upon the platform upon which your program is running, there are some techniques that can be leveraged to make your test scripts more portable. For example, when running a Motif program, there are symbolic names for the different keys on the keyboard. By using the names that will be recognized in the most contexts you insulate your tests from many types of changes.

Use meaningful tag names when possible. Some test tools, such as Replay Xcessory provide a facility for you to provide your own names for the different widgets. You should take advantage of this. It allows you to use names that are more meaningful to your testing and to the final application than the names that were used by the developer (sometimes developers use names that are meaningful for the graphics platform, or the program libraries, but have little or no meaning to people who need to use the software). Using widget tags enable you to make your tests more readable and meaningful.

Just as important as making your tests readable with widget tags, you can use them to insulate against changes to the widget names made by the developers. If you use the widget names in your tests, and the underlying name is changed, you only have to change the test code in one place- the map of the actual widgets to the tags. The rest of your test code remains unchanged. As your test base grows, following this guideline can save you hundreds of hours of work.

Summary

In order to focus on testing applications with graphical user interfaces, you need tools that support the testing discipline. Such tools must be easy to use, extensible, adaptable to change, and scriptable. The best way to adopt such tools is to start out by using the record / playback capabilities and then advance to writing scripts. After the scripting language becomes familiar, you develop your own little languages for each application or domain type and become a power tool user and expert tester.

We have shown how a tool such as Replay Xcessory (see the PDF) exhibits the desirable characteristics of a good testing tool. We also showed how to learn and use, incrementally, powerful features of the tool and its scripting language.

References

[AST03] David Astels. *Test-Driven Development A Practical Guide*, Prentice Hall PTR, 2003. ISBN 0131016490.

[BEN86] Jon Bentley, "Little Languages," *Communications of the ACM*, 29(8):711-21, August 1986.

[KAN01] Cem Kaner, James Bach and Bret Pettichord. *Lessons Learned in Software Testing*, Wiley, 2001. ISBN 0471081124.

[MEM02] Atif M. Memon. *GUI Testing: Pitfalls and Practices*, IEEE Computer, August 2002, pp.87-88.

[WEL03] Brent Welch, Ken Jones and Jeffrey Hobbs. *Practical Programming in Tcl and Tk 4th ed.*, Prentice Hall PTR, 2003. ISBN 0130385603.

About ICS

Driven by the belief that the success of any software application ultimately depends on the quality of the user interface, Integrated Computer Solutions, Inc., The User Interface Company™, of Cambridge, MA, is the world's leading provider of advanced user-interface development tools and services for professional software engineers in the aerospace, petrochemical, transportation, military, communications, entertainment, scientific, and financial industries. Long recognized as the platform of choice for visually developing mission-critical, high-performance Motif applications, ICS' BX series of GUI builders has recently been expanded to provide a complete line of tools that accelerate development of Java™. ICS is the largest independent supplier of add-on products to the Qt multi-platform framework developed by Trolltech. Supporting the software development community since 1987, ICS also provides custom UI development services, custom UI component development, training, consulting, and project porting and implementation services. More information about ICS can be found at http://www.ics.com

Instructional Webinars:

ICS sponsors a series of instructional web-based seminars on a variety of software development topics. The webinars are offered at no cost, but pre-registration is required. A complete schedule is posted at: www.ics.com/webinars.

Integrated Computer Solutions Incorporated

The User Interface Company™

Phone: 617.621.0060
Email: info@ics.com

www.ics.com