Automated Testing Tool

Damon Courtney, Gerald Lester, Lauren Vaughn and Tim Thompson

October 2, 2006

Abstract

This paper presents details of the design and implementation of a Automated Testing Tool for a Tcl/Tk based database application suite. The tool does record and playback of the of user interaction events, much like TkReplay. In addition, the tool also records and compares database interactions, display of error messages, and widget tree state.

Discussed are:

- The advantages of building a custom tool vs use of generic tool in testing the application.
- The "tricks" required for accurate record and playback under Microsoft Windows.

While the initial implementation is Microsoft Windows based, care has been taken to ensure that with minor modifications the Tool will run on Unix under X11.

1 Overview

1.1 Computer Aided Process Planning & Shop Floor Manufacturing System

Visiprise Computer Aided Process Planning (CAPP), part of the Visiprise Process Planning solution, allows process planners to easily define the manufacturing and assembly process using a multimedia delivery system that quickly captures, retrieves, authors, manages and distributes manufacturing information and work instructions.

Visiprise Shop Floor Manufacturing (SFM), part of the Visiprise Manufacturing Operations Solution, is a comprehensive manufacturing shop floor management solution that delivers complete visibility of the entire shop floor to decision-makers across your shop floor including manufacturing management, shop floor supervisors and shop floor operators/mechanics. Visiprise SFM executes manufacturing and maintenance processes and manages all of the information required for fabrication, tooling, sub-assembly and final assembly.

The current system consist of a Tcl/Tk based Client-Server application running on the end user's desktop and zero or more daemon processes running on servers. The daemon processes may be anyone of:

1. Transaction Monitors that monitor for incoming IBM MQ messages
2. Print Servers that process background printing for the end users
3. Batch Servers that process scheduled background task, e.g. nightly data exports
4. Transaction Daemon that process background task for the end users

In short, our application presents a Graphical User Interface to an Oracle Db and enforces certain Business Rules.

1.2 Goals

The section of the system that the Automated Testing Tool is to address is the Graphical User Interface part of the Client-Server application running on the end user's desktop. The API which the Graphical User Interface and daemon process are built on top of can be tested using the existing Tcl Test package. Thus the task is simplified to ensuring the correctness of:

1. Verifying the starting and ending state of the GUI.
2. Verifying that any additional dialogs are posted (e.g. "Busy Dialog").
3. Verifying that all database transactions occurred and occurred correctly.

1.3 Gray box testing

Black box testing[1], also known as functional testing. A software testing technique whereby the internal workings of the item being tested are not known by the tester. For example, in a black box test on a software design the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs.

The advantages of this type of testing include:

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need knowledge of any specific programming languages.

- The test is done from the point of view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.

The disadvantages of this type of testing include:

- The test can be redundant if the software designer has already run a test case.
- The test cases are difficult to design.
- Testing every possible input stream is unrealistic because it would take a inordinate amount of time; therefore, many program paths will go untested.

White box testing[2], also known as glass box, structural, clear box and open box testing. A software testing technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data. Unlike black box testing, white box testing uses specific knowledge of programming code to examine outputs. The test is accurate only if the tester knows what the program is supposed to do. The tester can then see if the program diverges from its intended goal. White box testing does not account for errors caused by omission, and all visible code must also be readable.

Gray box testing[3] involves inputs and outputs, but test design is educated by information about the code or the program operation of a kind that would normally be out of view of the tester. Gray box testing can be seen as the blending of structural and functional testing methods throughout the entire testing procedure.

Gray box testing examines the activity of back-end components during test case execution. There are two types of problems that can be encountered during gray-box testing. The first is when a component encounters a failure of some kind, causing the operation to be aborted. For example, an edit check to allow dollars does not accept dollar amounts, i.e. "AAA". The second is when the test executes in full, but the content of the results is incorrect. Example: calculations - produces a number but it is incorrect.

The Automated Testing Tool that we have implement is a Gray box testing application, it:

- Gathers important state information at the start of the test
  - Execution environmental information
  - Database information
  - State of the GUI

- Monitor Database transactions
- Monitor the appearance and disappearance of toplevel widgets
- Gathers important state information at the end of the test
  - State of the GUI

1.4 Tester Friendly - Record/Playback Model

Our testers, and those of our end users, tend to be non-programmers. Thus our software had to be aimed, primarily at non-programmers. To this end we decided early on to use the Record/Playback model with a "VCR Control" like GUI. We felt that most of our testers would not use a tool that required them to write scripts but would use tools that allowed them to use a simple GUI to record, modify and play back their test.

One goal was for the recorded test to be editable. This was done at two levels. The first was to provide a GUI to:

1. Delete recorded event
2. Change recorded timing on an event
3. Change the description of an event

The second level this was done at was to store the recorded events as a human readable plain text file. This allows for the developers and advance testers to quickly modify a test.

The architecture of the testing tool also had to be extendable in the the future as the product evolved. This was done by mapping the recorded test events to commands and by using a plugin architecture for displaying and editing events.

1.5 Non-Goals

We did not set out to build a generic testing tool. The testing group at the company had attempted, with varying small degrees of success, to use several of the generic testing tools on the market. Since these tools work either by doing screen scrapes or looking at an MFC widget tree, neither worked well with our Tcl/Tk based application.

Those tools that did screen scrapes were very sensitive to a tester (or a release of Microsoft Windows) changing system colors. They also do not allow for the movement or resizing of a widget without rerecording the test. Resizing in this case also include the slider in the through of a scrollbar.

Those tools that look at the MFC widget tree have particular problems with Tk since, by and large it does not use MFC widgets. Thus most of the application is "invisible" to those testing tools.

2 Design

Part of the design for testing of our application included the ability to capture and compare database queries while the application ran. This isn't something that every application would have a need for, but because of the sheer amount of database work in our application, and because we consistently have problems with database results, it was important to our testing.

Because all of our database functions are abstracted through a Tcl layer, the implementation of capturing the database calls was easy. We handle this by "hijacking" the database commands when the user enters record mode and then replacing them once the recording has stopped. This mostly allows us to capture the database queries as they occur, but it has the added benefit of letting us stop commits and rollbacks to the database.

With the Automated Testing Tool's ability to create savepoints and then restore the database to a known state, it was necessary for us to stop the code from doing its normal routine of commits and rollbacks and let us handle the brains behind it. All of this is done by renaming our database procs out of the way and replacing them with our own. We then rename the original procs back once the record mode has ended.

Most of the database interaction could, of course, be done using Tcl's excellent command tracing capabilities just easily as renaming the procs. With command traces, we could just as easily have captured the queries before the actual commands are called and then captured the result on the way out of the proc. Using command traces, one could easily handle the capture of database queries without the necessity of a database abstraction layer. You could simply trace the actual library calls for your specific database and record the queries that way.

With all this in place, the Automated Testing Tool can record all of the database interaction from the application in a way that can be verified during playback. The tool does not play back the database results, it simply stores them in its script and upon playback, it compares the queries going out with the ones stored to first verify that the queries are the same. And, secondly, it compares the results coming back to make sure we got the same results as when the test script was first created.

This allows us to not only tell when a query has been modified without our knowledge, but it also tells us whether that query is returning the exact same results as we expect. If not, the new query may be wrong, and this would be a red flag in our testing that something was not correct.

3 Tk State Capture

At start and end of test the visible widget tree is walked and a subset of attributes captured, these include the contents of associated variables. Only the visible widget tree is walked instead of the entire widget tree since our application caches "forms" instead of destroying them. By only walking the visible widget tree, this allows tests to be reused regardless of the previous test that have been run.

4 Application State Capture

Besides the state of the Tk widget tree, we also capture some application information that effects the execution of our application and hence of our test:

- Current user of our application specific user database that the test is being run under. This may be different from both the operating system user and the Oracle DB user.
- Current user enabled role. This consist of the application specific Profile and Group that are currently selected.
- Contents of ::tcl_platform

5 Oracle Data Capture

This information is used to assist with performance issues and error resolution. Gathering this information up front saves tracking down a DBA to gather it later. It allows for proactive problem resolution. The following information is captured at the beginning of a test capture or replay of a test:

- Database Version
- List of Parameters
- Data Dictionary Information (Tables, Columns, Indexes)
- Server Statistics
- Connected Session Information
- Execution Plans

6 Tk Event Capture

The capturing of events in record mode is all handled at the Tcl level using Tk's events. While this could have been done at a lower level, it does not allow us the ability to record data that is specific to Tk, like: widget name, widget class, widget attributes, etc... This works out to our advantage most times, but it does lead to some problem.

Some things are just not possible to record when recording our test scripts using Tk's events. Chief among them being that events on menus on the Windows platform are sorely lacking, so we can only record our best an estimation of what the user is doing and just hope and pray that we're right. Menus, in particular, are handled by overriding the commands of each menu when record mode begins and then replacing them when it ends. This will cause big problems if the code somehow wants to modify the commands of a menu, but it's what we've got to work with.

One thing we were able to do in Tk that would have been very difficult at a lower level was to record the events as the occur on the widgets themselves rather than based on coordinates. Most of your common testing tools available record actions based on the X,Y coordinates of where the action occurred. This works fine when the GUI is not going to change, but if you, say, move a button 100 pixels to the left, your script is worthless.

By recording events like buttons clicks and entry focus based on the widget names, we are able to play back a script based on the current locations of the widgets instead of where they were when the script was recorded. This was crucial to our testing strategy because of the volume of changes that our GUIs often go through. With customer demands constantly coming in, our GUIs can change drastically from release-to-release.

This method is not without its problems though. If the name of the widget in the GUI changes, your script will need to be recorded again. However, we find that this type of change occurs far less frequently than just moving the button over 100 pixels to make room for another button.

The record mode takes events from Tk and appends them to a long list as the script builds. When recording stops, the scripts is then analyzed for events that need to be stripped out. Because we end up with a lot of events from Tk that are garbage to our playback, we need to walk through and try to determine what can stay and what can go.

This usually includes stripping out Window configure events down to the last event, meaning that we don't care how the window was resized, we only care about the last event that tells us the final geometry. We didn't really feel it was necessary to record every tiny movement in between.

It is also necessary to rework some of the timing of the events to make things a little smoother. Movement through a popup menu, for example, needs to be

spaced out a little to give the OS time to draw the menus during cascading. The timing can always be adjusted by the user after the script has been record if it's necessary.

Once the user is finished recording and the analyzer has cleaned up the script, the user is presented with the finished script including some basic comments about the events as they occurred. The user can go through and modify the script as they choose without ever having to actually learn how the scripts are saved.

This was another big win for us because our testers are often unskilled in any kind of programming, and while it would be possible to train them, the effort could be better spent actually testing our application instead of programming our testing tool. With the script editor, the basic functions of a script can all be changed from within the tool itself.

7 Future Directions

The Testing Tool was proposed to in the spring of 2005 to the management team at HMS Software. Work, both design and coding, proceeding on a part time basis through the first quarter of 2006. During this time, at the end of 2005, HMS Software was acquired by Visiprise. This effected the future direction of this work.

Items that had been planned to be added in later releases were:

1. Parameter driven substitution of keyboard entered data
2. Parameter driven looping of test

However, Visiprise's vision is to convert the entire code base of the application to Java within five years. Such plans, along with restructuring of the testing department, have caused a large decrease in support for this product – at least at this time.

8 Summary

We believe that our tool allows for successful testing of our application. We also believe that for a large number of applications that are essentially GUI front ends to a data base that a Gray Box testing tool like the one described in this paper is the correct approach to testing.

References

[1] http://www.webopedia.com/TERM/B/Black_Box_Testing.html

[2] http://www.webopedia.com/TERM/W/White_Box_Testing.html

[3] http://www.geocities.com/xtremetesting/WorldofGrayBoxTesting.html