Agent SMITH:

Evolution of a Test Tool in Tcl/Tk

By:
John J. Seal
Principal Software Engineer
Platform Support Development Section 21520
john_j_seal@raytheon.com
317.306.4838

Abstract:
This paper describes the evolution of the Story Maker Interface Test Harness (SMITH), a test tool developed by Raytheon Technical Services Company (RTSC) for internal use on one of its projects. Development started with a sketch made by a project engineer, and proceeded in four incremental stages: First, implement the sketch as proof of concept; second, automate the tool using test case files; third, develop a C extension to provide hooks into the target system; and fourth, analyze the target system's response. These stages tracked the changing needs of the project, from development of the interface code before real hardware was available, through unit test, and finally into full-scale system integration and qualification testing.

Prepared for:
15th Annual Tcl/Tk Conference
October 20-24, 2008
Manassas, VA

Prepared By:
Raytheon Technical Services Company LLC
Customized Engineering and Depot Support (CEDS)
6125 East 21st Street
Indianapolis, IN 46219-2058

The Raytheon Technical Service Company (RTSC) Customized Engineering and Depot Support (CEDS) site in Indianapolis, IN, is responsible for developing and maintaining a system known as Story Teller. Our customer asked us to integrate a new system, called Story Maker, to be supplied by a 3rd party. We had the Interface Control Document (ICD), but actual hardware would not be available for quite some time. Instead of just coding blindly to the ICD and hoping for a successful "big bang" integration, we decided to develop a tool to exercise the interface until real hardware was available.

The lead engineer for Story Teller used Microsoft Office tools to sketch out a notional Graphical User Interface (GUI) for that purpose, and marked it up with notes about what the various elements should do [Figure 1]. For comparison, the final tool is shown beside it [Figure 2]. You can see that the final tool matches the initial sketch very closely, but there are significant differences, too. Some of the differences arose early, once the fundamental requirements of the tool were better understood, while others evolved later to meet the needs of different user communities. The evolution of the tool in response to evolving requirements is a good case study in incremental development.

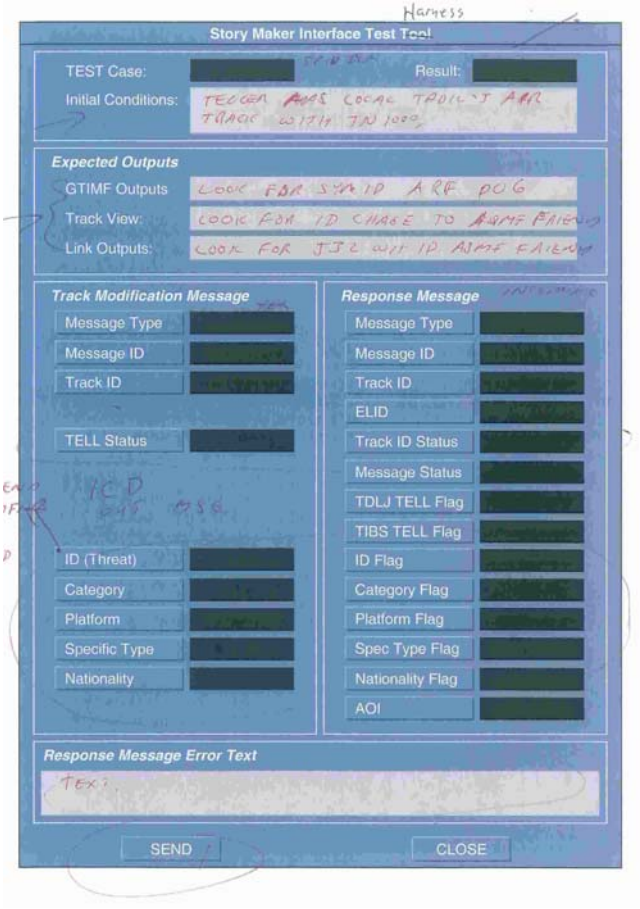

Figure 1 – Initial Sketch

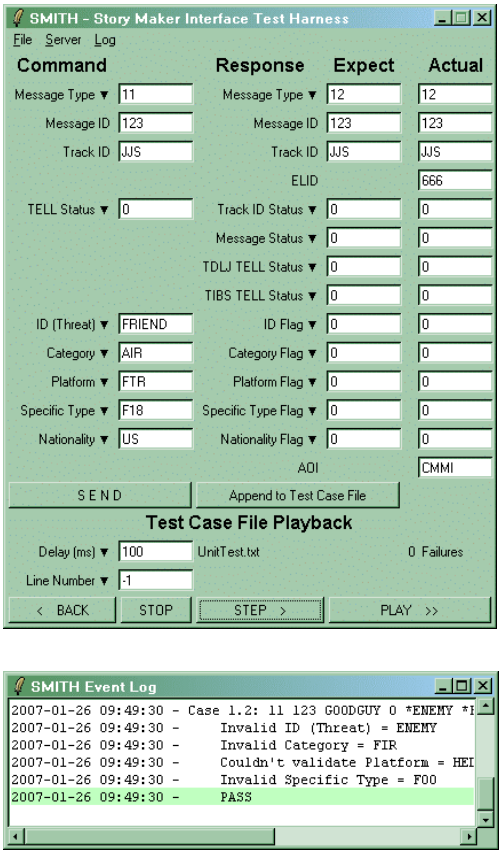

Figure 2 – Final Tool

One of the first orders of business for any project is to pick a name. An earlier project of mine was nicknamed "matrix" because it was being developed when the first Matrix movie was released, and featured a prominent matrix of data in the form of a TkTable.

Another project was a code-counting tool called Count SLOCula. We always used the phrase "Story Maker Interface" when discussing this project, so the acronym SMITH suggested itself rather naturally, but my coworkers tended to refer to it as Agent SMITH, and the name stuck.

The first phase of development was to simply implement the notional sketch almost literally, as a proof of concept. The ICD defined a simple TCP/IP network protocol that was easily implemented in Tcl. The initial Tk GUI allowed the user to construct a message, enter a destination host and port, send the message, and see the response. The initial concept included fields to describe the expected response in narrative form, but we quickly realized that by adding fields to describe the expected response in detail, the tool could automatically check the actual response for correctness. Thus, the tool design quickly settled on three main data entry columns instead of two.

Once the proof of concept was demonstrated, the next step was to add data-entry aids. Story Teller has data files that define the valid values for certain fields, and it uses those files to validate the messages it receives. We use those same files to create pull-down menus (indicated by downward-pointing triangles) for the entry fields. When the user selects a value for one field, it changes the values available for the others; this makes it easy to specify valid and consistent test cases. The user can type directly into the fields to create invalid or inconsistent test cases.

The initial proof-of-concept GUI used tk_optionMenu widgets because they're quick and easy, but they weren't used in the final tool for several reasons. First, they display the current selection; we wanted the menus to double as labels for the entry fields. Second, their default appearance is rather large and visually complex; in a dense grid they made the GUI cluttered and hard to read.

In retrospect, perhaps we should have used labels for the labels, and comboboxes for the entry fields. That also would have resulted in a certain amount of visual noise, with all the combobox arrows lined up vertically. The design we chose, with discreet down arrow glyphs made with characters in the menubutton name, is a visually clean compromise.

A related issue that became apparent during proof-of-concept was that the GUI window tended to be fairly big, because of the large number of entry fields. A standard solution would be to use a tabbed window or multiple windows, but in this case the command and its actual and expected response really is the minimum essential information that has to be presented all together in one place. We did find three ways to save some space:

- Make the Event Log a separate resizable window
- Move the Exit button into a File menu
- Move the destination specification into a Server menu

The elimination of the destination host and port entries was a welcome change to users, who no longer had to remember such details. Instead, the Server menu offers three canned destinations: the actual target system, the development system, and localhost.

The second phase was to automate the tool by allowing it to read test cases from a file. We added entries to the File menu to Open a test case file (and also to show a debug Console). The ability to automate the tool was important because, at that time, the chief role envisioned for the tool was for Unit Test of the interface code. Just verifying that the code followed the ICD required several hundred test cases, and we didn't want the user to have to enter them manually! We wanted a simple format that was easy to parse, and what could be easier than to let Tcl do the parsing? In fact, test case files are just Tcl scripts using a domain-specific language (DSL). Here is an example:

```
# Case  Msg Msg     Track  TELL ID                              Spec
#  Num  Typ ID      ID     Stat Threat      Cat     Plat    Type    Nat
case 1  11  123     JJS     0   ASMDFRND    AIR     FTR     F18     US
get msg "Enter the desired Message ID:" ;# get value of msg from operator
case 2  11  $msg    JJS     0   FRIEND      AIR     FTR    *F-18    US
    Illegal specific type
note "Special setup required for next test case!" ;# stops automatically
case 3  11  123     JJS     0   FRIEND      AIR    *BMR    *F18     US
    Inconsistent platform & specific type
```

The actual test case file template contains a header describing all the commands and formatting rules that have to be followed, for non-programmers. The case procedure, for example, takes 10 mandatory arguments, with any additional args taken as a description; leading asterisks indicate fields where an error is expected. In fact, most of our test cases don't have descriptions and fit all on one 80-column line. The reason you don't see line continuation slashes is a happy example of serendipity: Most editors will wrap long lines, and by using tabs to delimit arguments we can get the nice "indented description of the previous line" effect. There are several other commands in our DSL besides those shown in the example.

We added "transport controls" to the GUI so the test case file could be stepped through, played automatically, rewound, etc. Progress and results are shown in the Event Log window, with discrepancies between actual and expected results highlighted. The controls could have been put in a separate Playback window, but the conceptual coupling between the transport controls and the command/response data (i.e., the test cases being replayed) was strong enough that we felt it desirable keep them in the main window.

At first we required the user to prepare the test case file in a text editor, but the testers (who are not programmers) didn't like it. They pointed out the blindingly obvious: There's already a GUI to specify an individual test case, so just add a button to append it to the test case file! In that way the testers could use the GUI to build the test case files, experimenting with each single test case until they got it just the way they wanted it, then saving it. Bringing the tool to this point took about two weeks of real time. The tool in this intermediate form was used extensively by both developers and testers.

Once the new interface was working well, the Teller engineer started using the tool as if it were a Maker simulator, that is, to exercise the whole Teller system instead of just the new Maker interface. He pointed out that valid commands would cause Teller to send certain messages to other systems, and other systems to send messages back to both

Teller and Maker. Would it be possible for SMITH to verify Teller's external response to Maker messages, rather than just verifying that they were correctly accepted or rejected?

The third phase of development was adding hooks into Teller so SMITH could verify the external message traffic. This involved having SMITH open a server socket so it could receive messages intended for Maker, and registering to receive messages sent and received by Teller. Teller uses a peculiar homegrown Inter-Process Communication (IPC) scheme, so it was necessary to write a simple C extension to wrap the key functions and export them to Tcl. Could I have used CriTcl or FFIDL instead? I don't know, but compiling a C extension to a shared library is simple.

At this point the tool could not only verify that valid messages were accepted, but that Teller responded correctly, too. We added a Log menu so the user could choose what to log (commands, responses, outgoing messages, and incoming messages) and how to log it (simple events or hex/ASCII dumps). The hex/ASCII dump procedure was fun to write, as it was the first time I used the [subst [regsub …]] idiom; for details see <http://wiki.tcl.tk/1599>. The external messages are in Generic Tactical Information Message Format (GTIMF). Teller contains another tool, written in C, that can decode them to "human readable" form, but it's very difficult to use that tool and Agent SMITH together and correlate the results.

The fourth phase was to parse the external GTIMF messages into true human readable form. (The Teller tool parsed the fields and presented them numerically, with no interpretation.) The GTIMF specification documents the meaning of each field, so we wrote a parser to display fully-interpreted messages. The specification of the GTIMF protocol in Tcl is very flexible and could be easily extended to other formats. It took less than a week of real time to design, code, test, and document the GTIMF parser!

The final SMITH tool has been very well received and widely used. System engineers use it to explore various "What if?" scenarios. Developers use it to debug other Teller changes being proposed or implemented. Testers use it to provide repeatable stimulus during system integration and qualification tests.