Testing Initiatives at the New York Stock Exchange

Al Lowenstein
Securities Industry Automation Corporation

ABSTRACT

The Securities Industry Automation Corporation (SIAC) was established by the New York Stock Exchange (NYSE) and American Stock Exchange (AMEX) to develop, operate, and maintain their computer software and hardware systems. Last year, the NYSE traded over $10 trillion in securities. With so much at stake, operational reliability and integrity is critical.

Three years ago, the NYSE and SIAC established an initiative to improve the productivity and quality of testing. This paper discusses the key architecture and infrastructure elements of that initiative including:

- The suite of tools needed to test and control 20 major software systems that are written in many different computer languages, operate on heterogeneous hardware platforms and operating systems, and are in a constant state of change.
- The development, operation, and maintenance of robust, reusable business-oriented Regression Scripts.
- Key issues and lessons learned.

BACKGROUND

SIAC computer systems are used to trade NYSE and AMEX stocks and to generate Trade and Quote tickers for exchange-listed stocks. A variety of Tandem, IBM, DEC, UNIX, and Windows computers and operating systems are used to host these applications. The applications have been developed over the past 25 years using many computer languages, as well as internal and third party software. New revisions of individual components are constantly being installed. To ensure reliability, an issue of paramount importance, SIAC initiated efforts to improve testing.

The Test Support Group (TSG) was created to develop SIAC's testware infrastructure. The group's initial efforts were directed at automation of business and system level testing. Although the group built the utilities and tools to facilitate the development, use, and maintenance of test scripts, it did not (and does not) write the actual scripts, a function left to others having detailed knowledge of business and technical issues. While a comprehensive discussion of requirements addressed in this initiative is beyond the scope of this paper, the paper highlights some requirements that have not received attention in the design and test communities.

There was some concern that the introduction of new tools, even effective new tools, may have an undesirable impact. The NYSE Computer Systems are among the most reliable in the world. Consequently, NYSE's existing design and quality processes are successful in balancing business and technical validation concerns. To achieve this balance, experts with diverse knowledge and experience in business and technology must participate in the validation. This balance could easily be altered if new tools are introduced that empowered technical experts to a significantly greater extent than business experts. Hence, the architecture of the testware was designed to address such concerns.

Our initial target user was the business expert performing system validation. This forced us from the outset to address the diversity of computer, operating system, and test tools employed. A test tool was needed to interact with each interface. The individual Application (APPL) interfaces included ASCII command line interfaces, custom and standard UNIX Graphical User Interfaces (GUIs), and Windows GUIs. For standard interfaces, we acquired public domain and commercial off-the-shelf test tools including Tcl/Expect for text interfaces, XRunner for MOTIF, and WinRunner for Windows. We built a custom tool for the non-standard X-based GUI. The test tools use multiple languages. As a result, our architecture was required to support the use of multiple languages in a Test Script.

The APPL development teams, first to use our tools, influenced our initial efforts, helped to validate the APPL testware prior to integration into the testware infrastructure, and were instrumental in identifying deficiencies and setting priorities. The business orientation of the tools was particularly valuable to the development teams and helped the team's software experts understand and test business requirements. Automation minimized user error during test execution, and equally important, increased confidence in test results. Furthermore, automation lowered the cost and effort of running the entire Test Script suite, and avoided the tendency to run only the *"relevant"* or "*most important"* tests.

TEST SCRIPTS

At SIAC, an APPL may be upgraded and released many times a year. Each release needs new tests for the new features and also a Regression Script for the stable features. Test script development and maintenance are important cost and quality drivers of the overall APPL project [BB_95]. Consequently, many testware architectural elements were incorporated to minimize development and maintenance, and ensure the durability and robustness of the test scripts [JH_99].

Test APIs (Application Programming Interfaces) were developed so that scripts may be written using business vocabulary. The business underlying an APPL typically changes more slowly than the user interface. Consequently, a test of business functions is more stable than a test based on the user interface, i.e., mouse clicks and keystrokes. Furthermore, the testware elements that underlie the test script allow many maintenance activities to be performed once to the testware infrastructure. In practice, major changes to an APPL have been tested with minimal maintenance to a test script. As discussed below, changes were confined to other underlying testware elements.

Some believe that the cost benefits are secondary. If a test script isn't stable, confidence in the script, especially a regression script, is compromised and revalidation of the script is necessary. One of the most important successes of this testing initiative has been the development of true regression tests. Our existing regression tests are stable. New tests developed to test new business functions are created, generalized, and appended to the regression test suite for the next release of the APPL. Regression script stability is of fundamental importance.

One reason APPL project team embraced the testware was because of its support for debugging. Also, priority was given to features that improved the productivity of test script developers and users, sometimes by deferring the availability of features to test business functions. A control panel was provided to permit the script developer to monitor what was happening and *look under the hood*. To some, a control panel is considered a *nice-to-have* optional feature. We argued against that view by asking the question, "Would one expect a test pilot to conduct the maiden flight with a non-working instrument panel?" We also had to deal with the *chicken-or-egg* dilemma. Namely, how does one develop infrastructure testware before having test scripts, and how does one develop test scripts without the infrastructure? During this early period, debug features were especially critical to productivity and quality.

The readability of test scripts is a high priority [LW_89]. The purpose of a line-of-code, sequence of steps in a test, and tests in a script must all be easily recognized. If not, the test will be difficult to develop, debug, validate, and maintain. Our approach, as described below, successfully addressed this issue and also supported the use of multiple test tools using different languages. We were able to mix languages, even within a file, and still avoid the risky, error-prone, and expensive process of translation.

TESTWARE ARCHITECTURE

This section of the paper discusses the architecture utilized to meet these requirements. Figure 1 represents one view of the basic architectural elements and their interaction when testing an APPL. The major elements are the Master Controller and Agents. They are independent software processes that communicate using a master-slave protocol. The Master Controller is responsible for reading the script, informing the Agents what to do, and collecting and analyzing results. An Agent consists of both the APPL and the APPL-specific testware needed to respond to the Master Controller's direction. An Agent is a slave that waits in a polling loop until it is activated to carry out a command. The slave Agent does what it is told, and returns the data it is asked for. It also returns error information that may be used by a test script or interactively by the tester to diagnose a problem. The Agent concept mapped well to how existing scripts were written and used, and facilitated communications with business and technical experts.

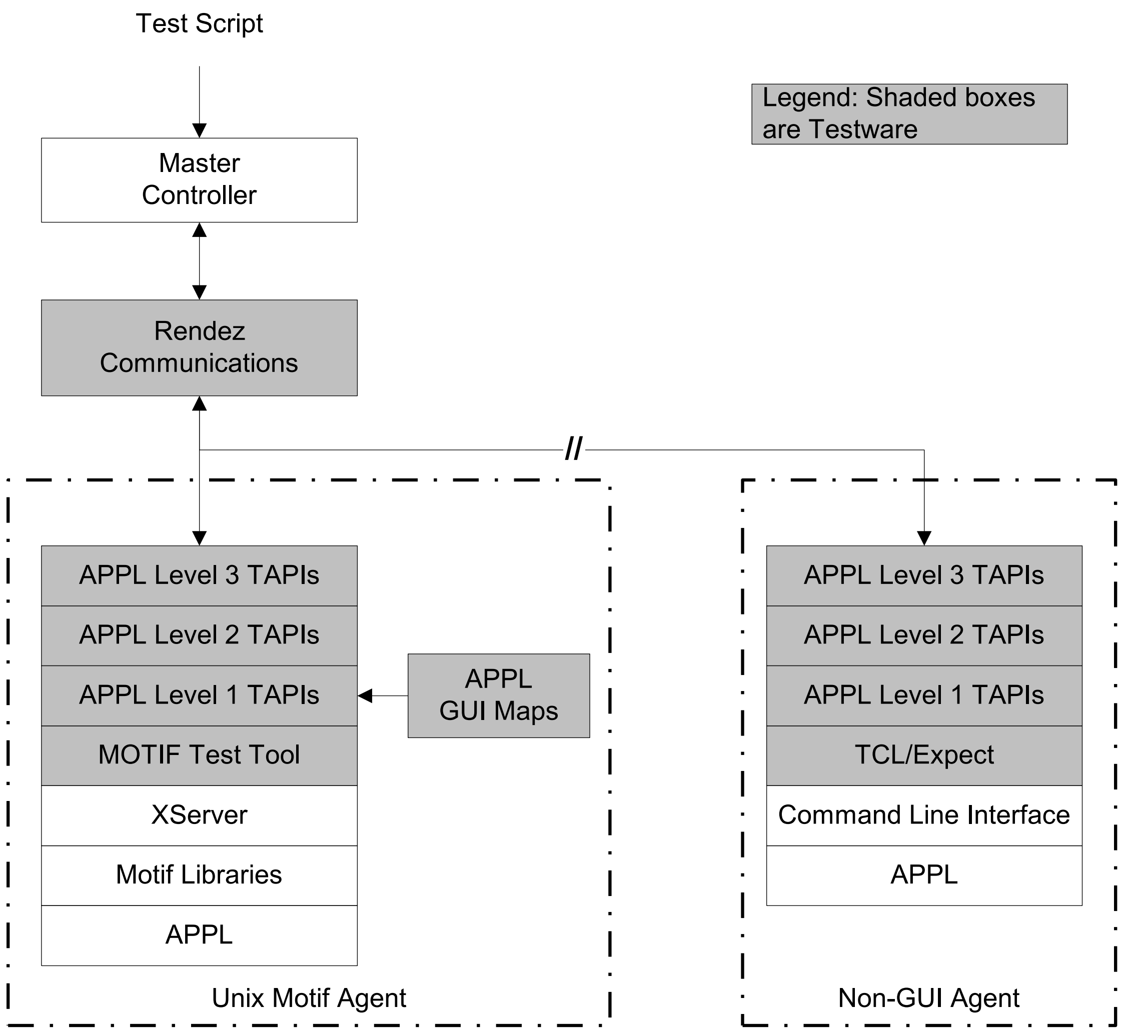

Figure 1: Agent View of Architecture

Test APIs (TAPIs)

The testware elements of a GUI Agent consist of TAPIs, GUI Maps, and a test tool to interact with the APPL. Business TAPIs are Level 3 TAPIs. These perform business action such as:

```
enterOrder BearStearns, Buy, IBM, 127 1/8, 2000
executeTrade IBM, 127 1/8, 500, BUY: BearStearns,
     SELL: MerillLynch
```

The purpose of a Level 3 TAPI is to test a business function [LW_87]. The TAPI encapsulates the function and helps testware deal with modifications to the APPL. Often, if the implementation of the APPL is changed but not the business function, only the Level 3 TAPI needs maintenance. For example, if a confirmation popup window is added to (or deleted from) an APPL, the Level 3 TAPI is modified to handle (or not to handle) the window. If text widgets are aggregated into a table widget, the Level 3 TAPI is modified to enter table data in lieu of text widget data. In these real examples, there was no need to modify test scripts. A new version of TAPIs was created for the new version of the APPL. When the test script was launched, first a version of the APPL was selected, then the proper version of TAPIs for that version of APPL was loaded.

Our users often wanted the Level 3 TAPI to provide a *sanity* check that the TAPI has performed its job, and that the test was worth continuing. As a result, a Level 3 TAPI typically contains a limited self-check capability. This capability is particularly useful during the debugging phase because errors are detected close to the test script code that exposes the problem. Once detected, using control panel flags and settings, the script may be automatically *paused* in order to investigate the problem and take the necessary corrective action.

An additional requirement of TAPIs, in general (and Level 3 TAPIs, in particular), is timing. Transactions are not processed instantaneously; one must wait a finite period of time before a reading can be taken to observe the effect of the transaction. The amount of time before taking a reading may vary greatly, depending on the system load at any particular point in time. How long should one wait? Timing issues must be resolved to create usable, robust test scripts. A TAPI must not wait too short a time or it will fail good APPLs, i.e., APPLs that are working properly. It must not wait too long or it will waste scarce test resources. When more than ten seconds is needed to observe an effect, some sort of trigger was identified (i.e., an observable phenomenon such as an event or property), than indicated that the TAPI should proceed. If the trigger condition is not immediately met, the TAPI is placed in a loop waiting for the trigger signal. As soon as the trigger is detected, the script exits the loop. If the trigger is not detected, the TAPI times out. The chosen timeout value is usually very conservative to ensure that a good APPL will always pass the test. The impact of a large timeout value is not a problem because the TAPI is fast when the check is good, and slow only when there is a failure or the system is truly slow.

To control an APPL, a Level 3 TAPI needs to generate keystrokes and mouse clicks. In the enterOrder example given above, BearStearns wishes to buy 2000 shares of IBM at 27 1/8. To accomplish this, we need to enter BearStearns in the Firm field, 2000 shares in the Quantity field, and 27 1/8 in the Price field. We use Level 2 TAPIs to express these actions. For example,

```
enterPrice 127 1/8
enterStock IBM
```

The test tool statements that generate the keystrokes and mouse clicks are embedded within a Level 2 TAPI. During the testware development phase, these embedded statements are often the source of errors. During formal test execution, low level statements are often the first place where problems in the test bed are detected. A Level 2 TAPI encapsulates and reduces the amount of low-level code, and provides a container to handle errors in a robust manner. When an error occurs, appropriate data can be generated and sent to the Master Controller and the user. Following error detection, the test script flow can be altered depending on flag settings and the specific task at hand. During a formal test run, the error can be logged and the test can either be bypassed or rerun. During integration, the program can be paused and diagnostic data dumped. The integrator can interpret the data and take action, as necessary.

GUI Map

When a GUI element is tested, the test script asks a GUI test tool (such as XRunner or Preview) to (1) apply a mouse click or keystroke, or (2) measure a property of a GUI element. The test script must contain sufficient information for the test tool to identify the unique GUI element. The GUI Map is a file correlating a logical name to the detailed GUI parameters needed by the GUI test tool to select the correct GUI element. A GUI Map allows a test script to be coded with logical business names in lieu of low-level GUI properties. A GUI Map improves the readability of test scripts, and even more important, makes a test script more maintainable. If the GUI element changes but the business function remains stable, often only the GUI Map and not the test script, needs to be modified.

Master Controller

Details of the Master Controller are illustrated in Figure 2. The Control Panel is used to control test execution and display status. Its initial primary purpose was to provide services needed to ensure user productivity. During implementation, we found that the panel had an additional very important benefit. The panel was the missing ingredient needed to solve the problem of dealing with test tools using different languages.

Several different approaches were considered to address the multiple languages issue. The first was a peer-to-peer architecture. In this architecture, each Agent involved in a test would have a script, written in the native language of the test tool. These peer-to-peer scripts would communicate with each other, signaling when to stop and when to continue. However, before any scripts were written based on this architecture, it was concluded that the scripts would be difficult to read and worse yet, to write. The code was fragmented into different files and test flow was very difficult to follow. Therefore, these scripts were judged to be unmaintainable.

The next approach was to code in a single language, providing filters to map calls to Level 3 TAPIs from one language to another. Note that the word, *map* and not *translate* was used. Mapping[1] can and is successfully performed. However, there are very few cases where automated translation between languages has been accurate or useful without manual intervention. From both a cost and quality viewpoint, we did not believe that translation was a viable solution. Although the filters for Level 3 TAPIs are working satisfactorily, we felt that providing a comprehensive set of filters for additional lower level TAPIs was likely to be a difficult task. We expected that (1) the cost would

[1] That is, a targeted substitution of procedure names and arguments from Controller to Agent.

be high, (2) the quality would be uneven, and (3) validation of the permutations of filters, TAPIs, and APPL versions would be difficult. Also, more functionality was needed than could be provided with just Level 3 TAPIs. Fortunately, a solution came from an unexpected quarter.

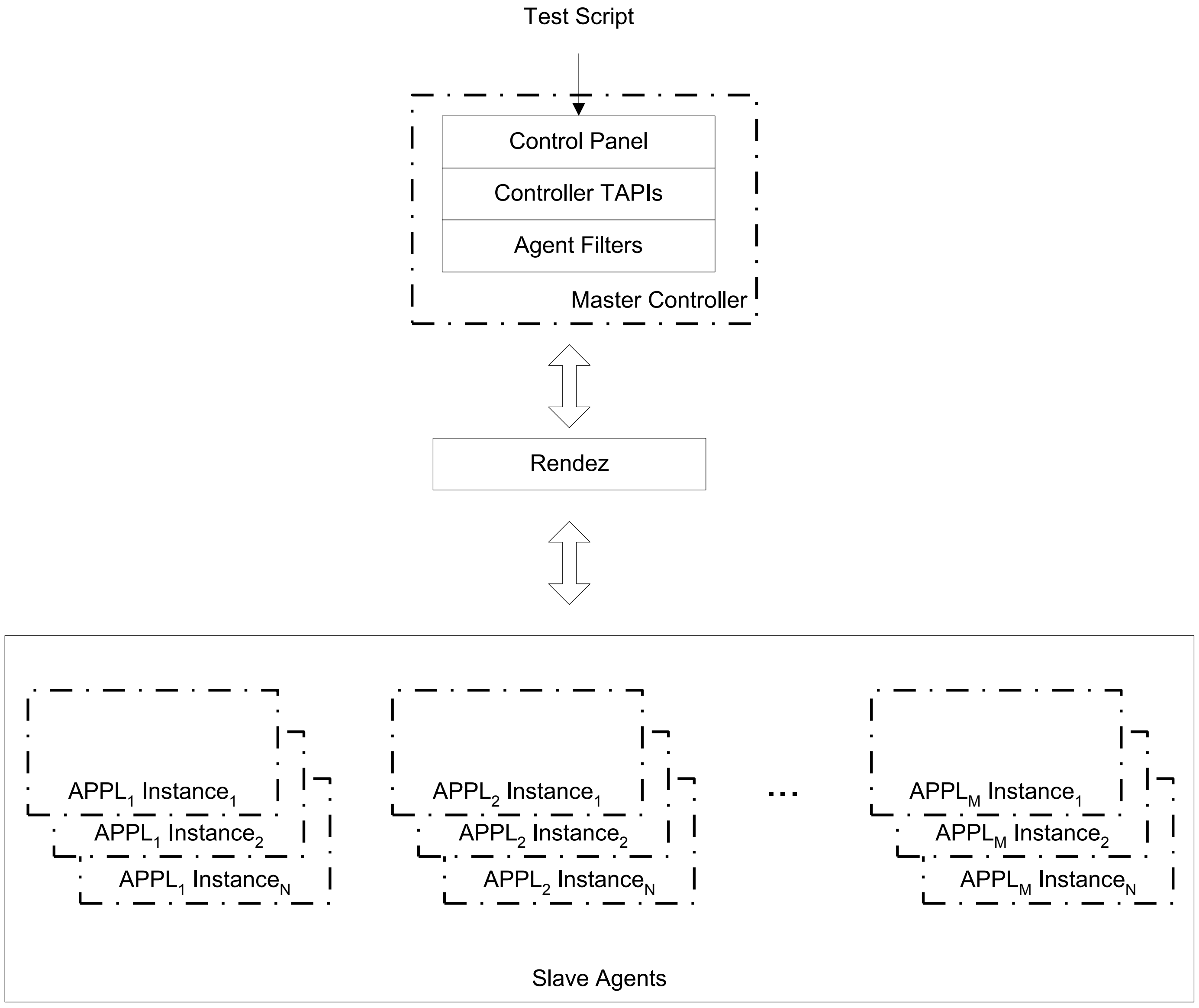

Figure 2: Master Slave Relationship

When we conducted the design review for the Control Panel, it was noticed that the panel read a script file line-by-line, and stored the lines in a database. At that time, the file was in a single language, Tcl, the preferred language for the testware. The database contained other information about the line that was used to control the run–time execution of the test. This information included a break point flag, execution and edit cursor flags, a selection flag, and a do–not–execute flag. The key point is that the database didn't care whether the lines of code were written in Tcl. We realized that if some sort of blocking mechanism could be provided, languages could be mixed in a file.

This is how we deal with a single file that contains multiple languages. An ideal blocking mechanism had already been developed, albeit for other purposes. The TAPIs, `startGroup`, `startTest`, and `startStep` had been developed to do a variety of functions[2]. The TAPI with the smallest scope, `startStep`, was given an additional responsibility. A parameter of the `startStep` TAPI defines the language of the block of code between two `startStep` statements. When a test script is executed, lines of code are extracted from the database, and the extraction routine knows the language of the block and where to send the block, i.e., it knows whether to send the lines to the Tcl interpreter or to another Agent interpreter. Thus, an Agent receives only code that it understands and can execute. This allows a single file to have code written in multiple languages without requiring complex software for its execution. The code is relatively easy to read because, in general, Level 3 TAPIs are employed, and Level 3 TAPIs are readable.

To be sure, an Agent may be the focus of a test and may require more control than can be had with Level 3 TAPIs. When the test tool for an Agent uses a non-preferred language (in our system, a language other than Tcl), then it is often necessary to code in the native language of the test tool. For example, a MOTIF Agent uses a test tool called XRunner whose native language is TSL. Hence, some scripts for a MOTIF Agent require the use of TSL. This means that, in general, the APPL test expert needs to read and write in the native language of the test tool associated with the APPL in addition to Tcl. In a perfect world, this is undesirable. However, we have found this to be one of our smallest problems, well worth the trade-off for the powerful functionality achieved. This permits us to select what we judge to be the best tool for interfacing with the APPL, and avoid compromises arising from language translation issues.

There was one additional issue that needed to be resolved to allow multiple languages to coexist in a single file. Often data needs to be shared between two blocks of code written in different languages. As a result, variable data needs to be translated from one language to another. To perform this function, we provide TAPIs for a limited number of data types, specifically, those that can be mapped and are straightforward to implement. We felt the user was better served by providing a limited capability that worked well, as opposed to a comprehensive set of functions that would be difficult to generate, validate, and maintain. We also felt that deployment of this functionality should be treated on a prototype basis, with the user advising if and where additional

[2] Some functions include:
- Control of program flow
- Set and reset pass/fail flags
- Hooks for debug
- Facilitate structured coding styles

support was needed. On the other hand, the API, the interface to the function, is extensible and support for additional data types can be provided.

OTHER KEY ELEMENTS AND LESSONS LEARNED

Test Script Maintenance can be Reduced, not Eliminated

The architecture discussed in this paper provides many ways to reduce Test Script maintenance. Clearly, maintenance can never be eliminated; the goal is to minimize it and thereby control costs. For a myriad of other reasons, test scripts need maintenance because business abstractions are not fully understood, and even if they were, the test code does not do a perfect job of implementing the test of the abstractions.

For example, initially the completion of a transaction was determined by checking that the size of a list was incremented. The test script called a TAPI, `getSizeOfList,` before and after the transaction. The script failed when multiple lines were written to the list, the transaction no longer wrote to the list, and when the list was changed to a table. Eventually, it was realized that the change in size of the list wasn't important; what was important was that the transaction had completed. The size of the list was of secondary importance; it was an implementation artifact of the APPL. What really was needed was a TAPI, `waitForTransactionToComplete`. When this *wait for TAPI* needed to know the APPL status just prior to the transaction, a TAPI, `getSyncData` was invoked prior to the transaction. These TAPIs allow the underlying data, used to determine transaction completion, to be removed from the test script. The actual data used to determine completion is private to the TAPIs, `getSyncData` and `waitForTransactionToComplete,` and can be changed as the APPL evolves.

There are many other examples of weak coding practices including hard coding paths, environmental variables, test bed assets, etc. While one should strive to get it right the first time, ferreting out good coding practices is an evolving process. Experience and lessons learned expose short sighted and naïve techniques and over time, superior techniques can be identified and institutionalized. Talented programmers, sound review processes, and style guides will minimize errors, and in the long run, maintenance. The bottom line is that while architectural elements described in this paper help control the cost of test script maintenance, they are not a panacea; other factors must also be addressed, and maintenance will always be necessary.

Tcl is an Excellent Test Language

In our system, Tcl is the language of choice. It is a powerful, extensible, public domain language. At the risk of slighting other important features of Tcl, several special features to be noted are:

- Tcl is designed to glue together building blocks written in any language [BW_97]. This feature has been a key to automate and integrate test scripts for the many diverse computers, languages, operating systems, and test tools in use at the stock exchange.

- Tcl is easily embedded, as a scripting language, in other tools. Tcl's introspection capabilities, i.e., the ability to examine the line of code being interpreted, allow easy operator intervention and modification of behavior.

- Tcl is an interpreter. The advantages of an interpreter versus a compiled language are a key to the high productivity with which we have implemented our testware infrastructure. An interpreter provides great flexibility for debugging and testing code, creating test scenarios to diagnose and clarify unusual behavior, implementing temporary patches, changing settings and variables, and a host of other functions critical to productivity, especially during the development phase.

Stabilizing the Test Environment

A major challenge was to stabilize the test lab environment. This task was so important that a separate group with this sole assignment was created. The group has been a major success, improving productivity in all development and test efforts. The manual nature of previous testing had hidden problems. When testing was performed manually, knowledgeable users recognized common errors in the test environment, taking corrective action, aborting and restarting the test, adjusting initialization parameters, etc. Automation stresses the test environment and code. Automated recovery is difficult and error-prone to implement. It is far more frustrating to run automated test scripts in an unstable test environment than to run manual test scripts. However, the difficulty of running automated test scripts proved to be a cloud with a silver lining. By exposing the magnitude of the problem, resources were allocated to resolve the problem.

Large Scope to Manage Growth and Expectations

When testing large systems, many elements are needed in order to be productive. Our architecture identified these elements and had a broad scope. This broad scope helped to create a roadmap that was used to initiate, justify, and coordinate modestly sized and scoped projects. Each of these sub-projects addressed a limited, well-defined and testable set of requirements. Priorities, schedules, budgets, and scope were created and tuned with the assistance and approval of users. Schedules identified when an element would be available and the inter-dependencies of elements. Just as important, the schedules highlighted the fact that requirements for many sub-projects were not budgeted, and would not be budgeted until a consensus to adjust priorities had been reached. Thus, users knew what was available and what was not.

The large scope of the effort led to what might seem like a non-intuitive, contradictory, and beneficial result. Namely, the large scope led to small, incremental tasks that grew little during implementation. To be sure, the lessons learned in implementation identified the need for expanded and modified functionality. However, it was frequently inappropriate to address the newly identified need immediately in the current task. Often the new need was seen to be a small part of a larger, more generic requirement. Rather than address a portion of the overall requirement with half measures, feature growth was managed and assigned to a future task, often already identified that would have adequate time and budget to more fully address the requirement. In any case, feature creep was minimized and schedules were met.

The large scope had another important benefit. It helped to manage user expectations. The architecture identified the need for many elements that would not be implemented in the near future. Identification of missing elements turned out to be an excellent mechanism to explain to the users that:

- An important function was not available, and would not be available for a period of time.
- There was a consensus about the importance of providing the function.

It was the users' job to influence our priorities so that we addressed their needs in an intelligent orderly manner. Often, our users prodded our sponsors to increase our budget so that critical test services could be provided. As it turned out, achieving a consensus on priorities with users was an easy, enjoyable task.

Testability Saves Money

As a result of this project, users became more aware of testability issues. Although a description of testable design techniques is beyond the scope of this paper, it is worth noting that designers were very open to suggestions for using alternative design techniques___ if, a very big if, the suggestion was offered early in the design process. In other words, if alternatives were offered prior to expenditures and commitments, designers were open to suggestions that reduced downstream cost, schedule, and risk. The lesson learned was that testability could be a *winner* if cost and schedule savings justified the requests for testability. If testability costs money, or is thought to cost money, testability suggestions are rarely followed. We found it useful to ask the following, "Why give money to us, a test group? If you do this [testability thing], we will need less money, and you can allocate more of your budget to design."

CONCLUSIONS

The New York Stock Exchange is implementing a new initiative to improve and automate testing of the complex software systems used to trade trillion of dollars in securities. Test of these heterogeneous software and hardware tools requires development of a robust testware infrastructure. The success of SIAC's efforts is proof that the goals of the initiative are sensible and achievable. Major assumptions have been proven. It is practical to mix programming languages in a single script, and within a single file. It is practical to write scripts that are robust and maintainable in the face of change introduced by continuous revisions to APPL systems.

The development and integration of many innovative concepts and elements were needed to achieve success. User opinions and lessons learned were constantly gathered and applied to the testware architecture and design. The close ties to the users as well as the adjustments based on lessons learned were important, perhaps as important as the innovations themselves, to the success of the initiative.

REFERENCES

BB_95 *Black-Box Testing, Techniques for Functional Testing of Software and Systems*, Beizer, Boris, John Wiley & Sons, Inc., New York, New York, 1995

BW_97 *Practical Programming in Tcl & TK*, Welch, Brent, Prentice Hall, Inc, NJ, 1997

DT_81 *Analyzing Discourse: Text and Talk*, Tannen, Deborah: editor, Georgetown University Round Table on Languages and Linguistics 1981, Georgetown University Press, Washington, DC, 1981.

IDA_88 *The Role of Concurrent Engineering in Weapons System Acquisition*, IDA Report R-38, IDA, Alexandria, VA, 1988.

JH_99 *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House; Highsmith, James, New York, New York, 1999.

LW_87 *Dictionaries: An Essential Concept for Implementing CAE Infrastructure*; Lowenstein, Al; Winter, Greg; ATE Seminar/Exhibit, June 1987 pp. 463-476; and Autotestcon 87, November 1987, pp. 153-160.

LW_89 *CAE Transport: A Discourse Analysis View*, Lowenstein, Al; Hall, Don; Schlosser, Steve; Winter, Greg, Autotestcon 89, September 1989, pp. 80-87.

email alowenst@siac.com