The Software Testing Automation Framework

by C. Rankin

Software testing is an integral, costly, and time-consuming activity in the software development life cycle. As is true for software development in general, reuse of common artifacts can provide a significant gain in productivity. In addition, because testing involves running the system being tested under a variety of configurations and circumstances, automation of execution-related activities offers another potential source of savings in the testing process. This paper explores the opportunities for reuse and automation in one test organization, describes the shortcomings of potential solutions that are available "off the shelf," and introduces a new solution for addressing the questions of reuse and automation: the Software Testing Automation Framework (STAF), a multiplatform, multilanguage approach to reuse. It is based on the concept of reusable services that can be used to automate major activities in the testing process. The design of STAF is described. Also discussed is how it was employed to automate a resource-intensive test suite used by an actual testing organization within IBM.

In late 1997, the system verification test (SVT) and function verification test (FVT) organizations with which I worked recognized a need to reduce per-project resources in order to accommodate new projects in the future. To this end, a task force was created to examine ways to reduce the expense of testing. This task force focused on improvement in two primary areas, reuse and automation. For us, *reuse* refers to the ability to share libraries of common functions among multiple tests. For purposes of this paper, a *test* is a program executed to validate the behavior of another program. *Automation* refers to the removal of human interaction with a process and placing it under machine or program control. In our case, the process in question was software testing. Through reuse and automation, we planned to reduce or remove the resources (i.e., hardware, people, or time) necessary to perform our testing.

To help illustrate the problems we were seeing and the solution we produced, I use a running example of one particular product for which I was the SVT lead. This product, the IBM OS/2 WARP* Server for e-Business, encompassed not only the base operating system (OS/2*—Operating System/2*) but also included the file and print server for a local area network (LAN) (known as LAN Server), Web server, Java** virtual machine (JVM), and much more. Testing such a product is a daunting, time-consuming task. Any improvements we could make to reduce the complexity of the task would make it more feasible.

For our purposes, a *test suite* is a collection of tests that are all designed to validate the same area of a product. I discuss one test suite in particular, known affectionately as "Ogre." This test suite was designed to perform load and stress testing of LAN Server and the base OS/2. Ogre is a notoriously resource-intensive test suite, and we were looking at automation to help reduce the hardware, number of individuals, and time necessary to execute it.

With a focus on reducing the complexity of creating and automating our testing, we looked at existing solutions within IBM and the test industry. None of these solutions met our needs, so we developed a new one, the Software Testing Automation Framework (STAF). This paper explores the design of STAF, explains how STAF addresses reuse, and details how STAF was used to automate and demonstrably improve the Ogre test suite. The solution provided by STAF is quite flexible. The techniques presented here could be used by most test groups to enhance the efficiency of their testing process.

The problem

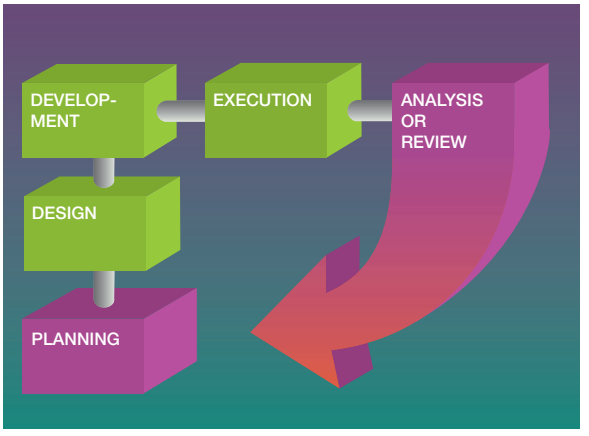
Figure 1 depicts the software testing cycle. Planning consists of analyzing the features of the product to be tested and detailing the scope of the test effort. Design includes documenting and detailing the tests that will be necessary to validate the product. Development involves creating or modifying the actual tests that will be used to validate the product. Execution is concerned with actually exercising the tests against the product. Analysis or review consists of evaluating the results and effectiveness of the test effort; the evaluation is then used during the planning stage of the next testing cycle.

Reuse is focused on improving the development, and to a lesser extent the design, portions of the testing cycle. Automation is focused on improving the execution portion of the testing cycle. Although every product testing cycle is different, generally, most person-hours are spent in execution, followed by development, then design, planning, and analysis or review. By improving our reuse and automation, we could positively influence the areas where the most effort is expended in the testing cycle.

The following subsections look individually at the areas of reuse and automation and delineate the problems we faced in each of these areas.

Reuse. This subsection provides some examples from the OS/2 WARP Server for e-Business SVT team that motivated the desire for reuse. Within the team, there were numerous smaller groups that were focused on developing and executing tests for different areas of the entire project. We wanted to ensure that each of these groups could leverage common sets of testing routines. To better understand this desire for reuse, consider some of the potential problems surrounding the seemingly simple task of logging textual messages to a file from within a test.

Figure 1 Software testing cycle



Several issues arise when this activity is left to be reinvented by each tester or group of testers, instead of using a common reusable routine. The problems are:

- Log files are stored in different places: Some groups create log routines that store the log files in the directory in which the test is run. Others create log routines that store them in a central directory. This discrepancy makes it difficult to determine where all the log files for tests run on a given system are stored. Ultimately, you have to scour the whole system looking for log files.
- Log file formats are different: Different groups order the data fields in a log record differently. This difference makes it difficult to write scripts that parse the log files looking for information.
- Message types are different: One group might use "FATAL" messages where another would use "ERROR," or one group might use "TRACE" where another would use "DEBUG." This variation makes it difficult to parse the log files. It also increases the difficulty in understanding the semantic meaning of a given log record.

None of these problems is insurmountable, and many could be handled sufficiently well through a "standards" document indicating where log files should be stored, the format of the log records, and the meaning, and intended use, of message types. Nonetheless, this list provides justification for our desire for common and consistent reusable routines. Also, additional problems exist that cannot be addressed by adhering to standards.

Multiple programming languages. Our testers write a wide variety of tests in a variety of programming languages. When testing the C language APIs (application programming interfaces) of the operating system, they write tests in C. When testing the command line utilities of the operating system or applications with command line interfaces, they write tests in scripting languages such as REXX (which is the native scripting language of OS/2). When testing the Java virtual machine of the operating system, they write tests in the Java language. In order for our testers to use common reusable routines to perform such tasks as logging, described above, the routines needed to be accessible from all the languages they use.

Multiple codepages. OS/2 WARP Server for e-Business was translated into 14 different languages, among them English, Japanese, and German. It is not uncommon for problems to exist in one translated version but not in another. Therefore, we were responsible for testing all of these versions. Testing multiple versions introduces additional complexities in our tests, and in particular to any set of reusable components we wanted our testers to use. One specific aspect of this situation is the use of different codepages by different translated versions. A *codepage* is the encoding of a set of characters (such as those used in English or Japanese) into a binary form that the computer can interpret. Using different codepages means that one codepage can encode the letter "A" in one binary form and another can encode it in a different binary form. Hence, care must be taken when manipulating the input and output of programs that use different codepages—a situation our testers would frequently encounter when testing across multiple translated versions of our product. If our testers were going to use a common set of routines for reading and writing log files, those routines had to be able to handle messages not only in an English codepage, but also in the codepages used by the other 13 languages into which our product was translated.

Multiple operating systems. While we were directly testing OS/2 WARP Server for e-Business, it was essential for us to run tests on other operating systems, such as Windows** and AIX* (Advanced Interactive Executive*) to perform interoperability and compatibility testing with our product. If we wanted our testers to use common reusable routines to perform such tasks as logging, described above, the routines needed to be accessible from all the operating systems we used.

Existing automation components. As we examined the types of components that were continually being recreated by our teams, as well as those that would need to exist to support the types of automation we wanted to put in place (as described in the following subsection), we realized that we would need a substantial base of automation components. Some of these components included process execution, file transfer, synchronization, logging, remote monitoring, resource management, event management, data management, and queuing. Additionally, these components had to be available both locally and in a remote fashion across the network. If the solution did not provide these components, we would have to create them. Therefore, we wanted a solution that provided a significant base of automation components.

Automation. This subsection provides some examples, using the Ogre test suite, to motivate the need for automation. As was mentioned, this test suite was designed to test the LAN Server and base OS/2 products under conditions of considerable load and stress, where *load* means a sustained level of work and *stress* means pushing the product beyond defined limits. The test suite consists of a set of individual tests focused on a specific aspect of the product (such as transferring files back and forth between the client and server). These tests are executed in a looping pseudorandom fashion on a set of client systems. The set of client systems is typically large, ranging upwards of 128 systems. The set of servers that are being tested is usually very small, typically no more than three. The test suite executes on the client systems for an extended period of time, typically 24 to 72 hours. The combination of the number and configuration of clients and servers and the amount of run time represents a *scenario*. If all the clients and servers are still operational after the prescribed amount of time, the scenario is considered to be successful. Multiple scenarios are executed during a given SVT cycle.

Figure 2 shows the basic procedure flow used to execute a given Ogre scenario. Note the areas in red. These areas indicate which steps in the procedure are currently done manually. The following subsections describe these areas in more detail.

Test suite execution. Our existing mechanism for starting or stopping a scenario was to have one or more individuals walk up to each client and start or stop the test suite. Given the situation of 128 clients spread throughout a large laboratory, this exercise is expensive, both in time and human resources. This

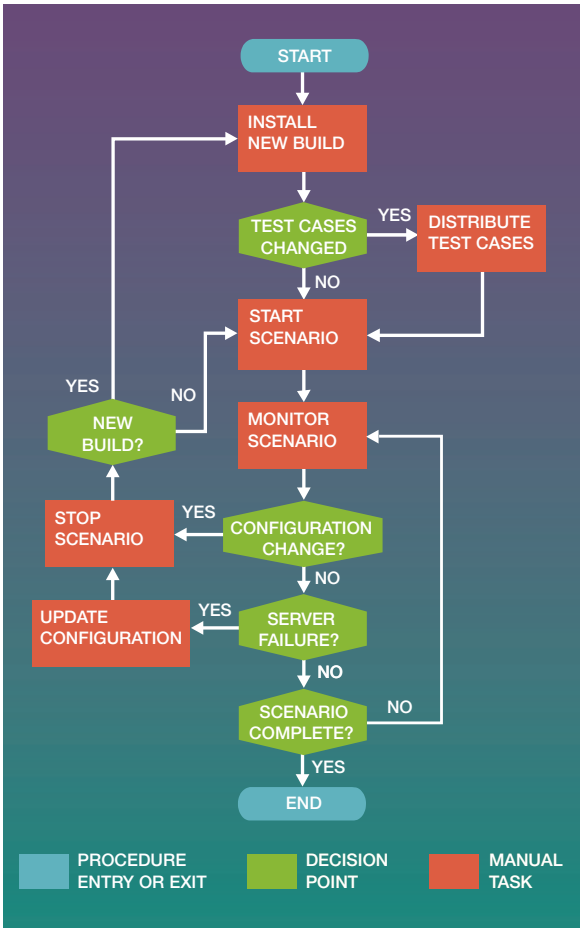
method also introduces the potential of skipping one or more clients, which can have a significant impact on the scenario (such as not uncovering a defect due to insufficient load or stress). Therefore, we wanted a solution that would allow us to start and stop the scenario from a central "management console."

Test suite distribution. As new tests were created or existing tests were modified, they needed to be distributed to all the client systems. Our existing mechanism consisted of one or more individuals walking around to each client copying the tests from diskettes. This method was complicated by the fact that the tests did not always exist in the exact same location on each client. Like the previous problem of test suite execution, this mechanism was very wasteful of time and human resources. It also introduced another potential point of failure whereby one or more clients do not receive updated tests, resulting in false errors. Therefore, we wanted a solution that provided a mechanism for distributing our tests to our clients correctly and consistently.

Test suite monitoring. While a scenario was running, we were responsible for continually monitoring it to ensure that no failures had occurred. Our existing mechanism consisted of one or more individuals walking around to each client system to look for errors on the system screen. Such monitoring was partially alleviated by the fact that the tests would emit audible beeps when an error occurred. The beeps generally made it possible to simply walk into the laboratory and "listen" for errors. Unfortunately, we still had to monitor the scenario after standard work hours and on the weekend, which meant having individuals periodically drive into work and walk around the laboratory looking and listening for errors. Again, this method was very wasteful of time and human resources. It was also a negative morale factor, since it was considered "grunt" work. Therefore, we wanted a solution that provided a remote monitoring mechanism so that the status of the scenario could be evaluated from an individual's office or by telneting in from home.

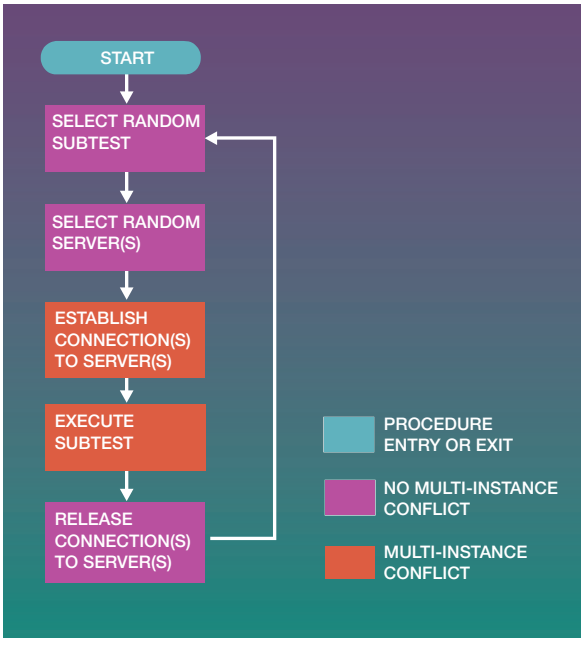
Test suite execution dynamics. The Ogre test suite was already very configurable. An extensive list of properties was defined in a configuration file that was read during test suite initialization (and cached in environment variables for faster access). These properties manipulated many aspects of the scenario, such as which resources were available on which servers, which servers were currently off line, and the ratios defining the frequency with which the servers were

Figure 2 Ogre scenario flow before automation



accessed relative to one another. This configurability allowed us, for example, to make a one-line change that would prevent the clients from accessing a given server (in case a problem was currently being investigated on it) or increase or decrease the stress one server received in relation to another. However, the only viable way to modify these parameters was to stop and start the entire scenario. As an example, assume that 36 hours into a 72-hour scenario, we found a problem with one of the servers. We could stop the scenario, change the configuration file to make the server unavailable, and then restart the scenario, which allowed us to exercise the remaining servers while the problem was being analyzed. Then, 12 hours later, when a fix for the problem had been created, we needed to bring the newly fixed server back into the mix. In order to do this,

Figure 3 Single Ogre instance before multi-instance support



we had to stop and start the entire scenario, which effectively negated all of the run time we had accumulated on the other servers at that point. Similar situations arose when we needed to change server stress ratios or other configuration parameters. Therefore, we wanted a solution that would allow us to change configuration information dynamically during the execution of a scenario.

Another long-standing issue with Ogre was that we were only able to execute one instance of the test suite at a time on any given client. It was felt that the ability to execute multiple instances of the test suite on the same client at the same time would allow us to produce equivalent stress with fewer clients. Figure 3 shows the basic procedure flow of a single instance of the Ogre test suite executing on a given system. Note the areas in red. These areas indicate places where running multiple instances of Ogre on the same system creates conflicts. The following two subsections describe these areas in more detail.

Test suite resource management. In order to make a connection to a server, the client must specify a drive letter (in the case of a file resource) or a printer port (in the case of a printer resource) through which the resource will be accessed. When running multiple instances of the test suite, race conditions arise surrounding which drive letter or printer port to specify at any given time. Therefore, we wanted a solution that allowed us to manage the drive letter and printer port assignments among multiple instances of the test suite.

Test suite synchronization. Some of our tests have strict, nonchangeable dependencies on being the only process on the system running that particular test. When running multiple instances of the test suite, we needed a way to avoid having multiple instances executing the same test simultaneously. Therefore, we wanted a solution that allowed us to synchronize access to individual tests.

Existing solutions

Because we had two separate problems (reuse and automation), we realized we might need to find two separate solutions. However, we were hoping to find a single solution that would address both problems. Our preferences, in order, were:

1. A single solution designed to solve both problems
2. Two separate solutions designed to work together
3. A solution to reuse, which provided components designed to support automation, from which we could build an automation solution
4. Two separate, disjoint solutions

In the following subsections, I describe existing solutions that we explored, how they addressed the problems of reuse and automation, and how they related to our solution preferences.

Scripting languages. Scripting languages such as Perl, Python, Tcl, and Java (although Java would not technically be considered a scripting language, since it does require programs to be compiled) are very popular in the programming industry as a whole, as well as within test organizations, since they facilitate a rapid development cycle.[1] As programming languages, scripting languages are not intended to directly solve either reuse or automation. Additionally, they are not directly targeted at the test environment, although their generality does not preclude their use in a test environment. Despite these limitations, we felt that given the wide popularity of scripting languages and the almost fanatical devotion of their proponents, we should examine their potential for solving our problems.

Although scripting languages are not a direct solution to reuse or automation, scripting languages do have some general applicability to the problem of reuse. To begin with, they are available on a wide variety of operating systems. They also have large well-established sets of extensions. Although not complete from a test perspective, these extensions would provide a solid base from which to build. Additionally, some languages (notably Tcl and Java) provide support for dealing with multiple codepages.

The benefits of scripting languages would clearly place them in category 3 of our preferences. Unfortunately, these benefits are only available if one is willing to standardize on one language exclusively. As was mentioned earlier, our testers create tests in many different programming languages, and it would have been tremendously difficult to force them to switch to one common programming language. Even if we could have convinced all of the testers on our team, we could never have convinced all the testers in our entire organization (much less those in other divisions, or at other sites), with whom we hoped to share our solution. Therefore, we were unable to rely on scripting languages for our solution.

Test harnesses. A test harness is an application that is used to execute one or more tests on one or more systems. In effect, test harnesses are designed to automate the execution of individually automated tests.

A variety of different test harnesses are available. Each is geared toward a particular type of testing. For example, many typical UNIX** tests are written in shell script or the C language. These tests are generally stand-alone executables that return zero on success and nonzero on error. Harnesses such as the Open Group's Test Environment Toolkit (TET, also known as TETware) are designed to execute these types of tests on one or more systems.[2] In contrast, a harness such as Sun's Java Test leverages the underlying Java programming language to create a harness that is geared specifically to tests written in the Java language. It would not be uncommon for a test team to use both of these harnesses. Additionally, it is not uncommon for test teams to create custom harnesses geared toward specialized areas they test, such as I/O subsystems and protocol stacks.

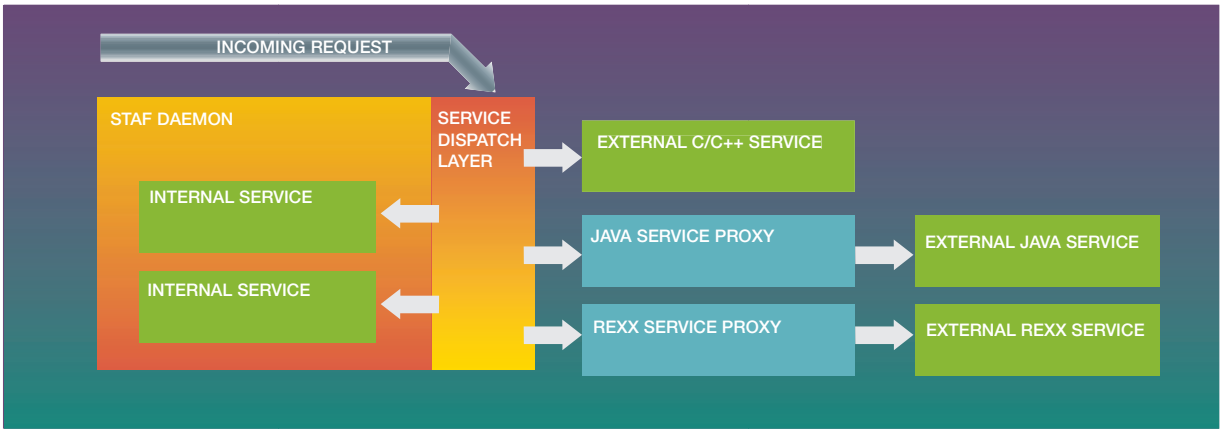
It is clear that test harnesses have direct applicability to the problem of automation. However, as a general rule, test harnesses only solve the execution part of the automation problem. This solution still leaves areas such as test suite distribution, test suite monitoring, and test suite execution dynamics unsolved. Additionally, test harnesses have no direct or general applicability to the problem of reuse. Thus, test harnesses are, at best, only part of the solution to category 4 of our preferences. That having been said, the proximity of test harnesses to the test environment made it likely that one or more test harnesses would play a role in our ultimate solution. However, we still needed to find a solution for reuse and determine which, if any, of the existing test harnesses we would use and extend to fill in the rest of the automation gaps.

CORBA. At a very basic level, CORBA** (Common Object Request Broker Architecture) is a set of industry-wide specifications that define mechanisms that allow applications running on different operating systems, and written in different programming languages, to communicate.[3] CORBA also defines a set of higher-level services, sitting on top of this communication layer, that provide functionality deemed beneficial by the programming community at large (such as naming, event, and transaction services). It is important to understand that CORBA itself is not a product; it is a set of specifications. For any given set of operating systems, languages, and services, it is necessary to either find a vendor who has implemented CORBA for that environment, or, much less desirably, implement it oneself.

CORBA is not intended to directly solve the problems of reuse and automation. However, CORBA does have some general applicability to the problem of reuse. First, CORBA is supported on a wide variety of operating systems. Second, there is CORBA support for a wide variety of programming languages. Thus, CORBA solves two of our key reuse problems. In contrast, CORBA has no direct support for multiple codepages. Additionally, the set of available CORBA services is not geared toward a test environment, which is understandable given the general applicability of CORBA to the computer programming industry as a whole.

Given the above, CORBA would clearly fit in category 3 of our preferences, although significant work would be necessary to provide the missing support in terms of multiple codepages and existing automation components. Additionally, as we mentioned above, there is no one company that produces a product called "CORBA." What this means is that for a complete solution one must frequently obtain products from multiple vendors and attempt to configure them to work together. This attempt has been

Figure 4 STAF service types



notoriously difficult in the past,[4] and, although the situation is improving, we would rather have avoided this layer of complication. All told, we felt that a CORBA solution was not worth the expense necessary to implement and maintain it.

The design of STAF

Having exhausted other avenues, we decided to create our own solution. We had a two-phased approach to the development of STAF. The first phase addressed the issue of reuse. This phase by itself would give us a solution that fell into category 3 of our solution preferences. The second phase tackled the problem of automation. In this phase we would build on top of the reuse solution and extend it to solve our automation problem. This two-step approach provided a solution that fell into category 1 of our solution preferences. The result of that work was the Software Testing Automation Framework, or STAF.

In the subsections that follow, I present the underlying design ideas surrounding STAF and how they helped provide a reuse solution. A subsequent section will then address how we built and extended this solution to solve the problem of automation.

Services. STAF was designed around the idea of reusable components. In STAF, we call these components *services*. Each service in STAF exposes a specialized set of functionality, such as logging, to users of STAF and other services. STAF, itself, is fundamentally a daemon process that provides a thin dispatching mechanism that routes incoming requests (from local and remote processes) to these services. STAF has two "flavors" of services, internal and external. Internal services are coded directly into the daemon process and provide the core services, such as data management and synchronization, upon which other services build. External services are accessed via shared libraries that are dynamically loaded by STAF. These external libraries represent either the service itself, in the case of languages like C or C++, which ultimately generate native executable object code, or a proxy interface to other languages, such as the Java or REXX languages, which do not generate native executable object code. The differentiation of service "flavors" and proxy handling can be seen in Figure 4.

This ability to provide services externally from the STAF daemon process allowed us to keep the core of STAF very small, while allowing users to pick and choose which additional pieces they wanted. It minimizes the infrastructure necessary to run STAF. Additionally, the small STAF core makes it easy to provide support on multiple platforms, and also to port STAF to new platforms.

Request-result format. Fundamentally, every STAF request consists of three parameters, all of which are strings. The first parameter is the name of the system to which the request should be sent. This parameter is analyzed by the local STAF daemon to determine whether the request should be handled locally or should be directed to another STAF system. Once the request has made it to the system that will handle it, the second parameter is analyzed to

determine which service is being invoked. Finally, the third parameter, which contains data for the request itself, is passed into the request handler of the service to be processed.

After processing the request, the service returns two pieces of data. The first is a numeric return code, which denotes the general result of the request. The second is a string that contains request-specific information. If the request was successful, this information contains the data, if any, which were asked for in the request. If the request was unsuccessful, this information typically contains additional diagnostic information.

By dealing primarily with strings, we have been able to simplify many facets of STAF. First, there is only one primary function used to interface with STAF from any given programming language. This function is known as STAFSubmit( ), and its parameters are the three strings described above. Because of the simplicity of this interface, requests look essentially identical across all supported programming languages, which makes using STAF from multiple programming languages much easier. Adding support for a new programming language is also trivial, because only a very small API set must be exposed in the target language. Had we chosen to use custom APIs for each service, the work to support a new programming language would be significant, since we would be faced with providing interfaces to a much, much larger set of APIs.

Strings also make it easier to create and interface with external services. The primary interface for communicating with an external service consists of a method to pass the requisite strings in and out of the service. Additionally, by restricting ourselves to strings we are able to provide to services a common set of routines to parse the incoming request strings. Common routines allow service providers to simply define the format of their request strings and pass them to this common parser for validation and data retrieval, which helps ease the creation of reusable components. This leads to benefits in the user space as well, since all service request strings follow a common lexical format, which provides a level of commonality to all services. It also makes it easier to use services when switching from one programming language or operating system to another, because the request strings are identical regardless of the environment. Commonality has the added benefit of hiding the programming language choice of the caller and the service provider from one another.

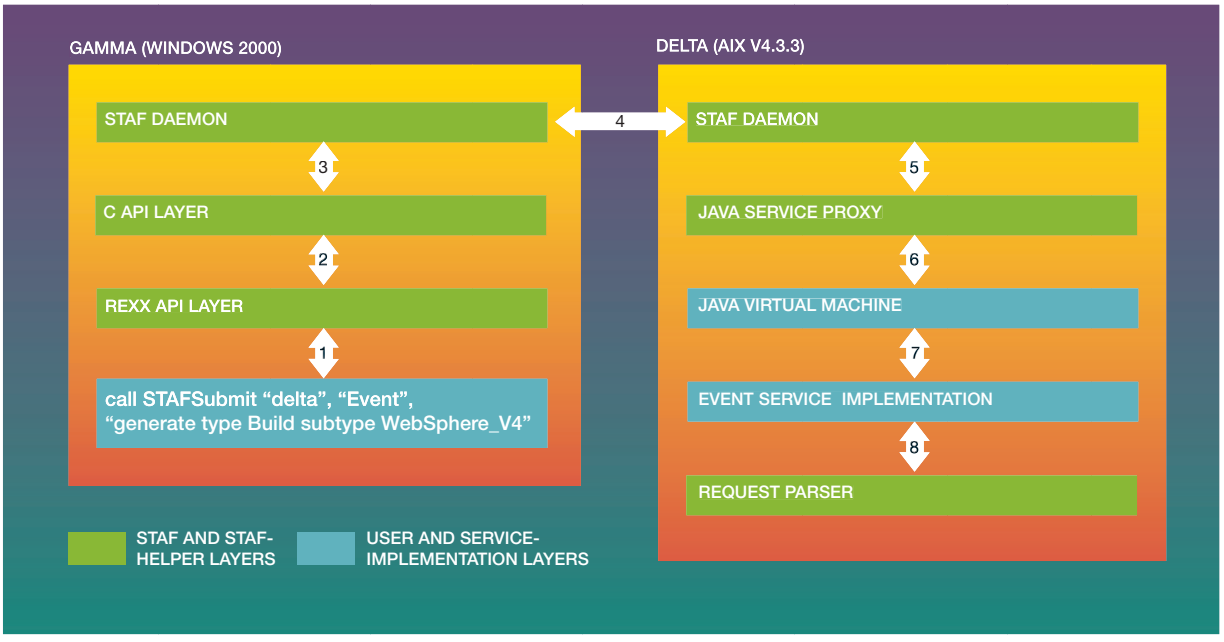
Figure 5 details the concepts just described. A STAF request is initiated by the REXX program running on machine gamma (running Windows 2000). It is submitting the request "generate type Build subtype WebSphere_V4" to the event service on machine delta. In step 1 the REXX interpreter passes the request to the REXX API layer of STAF. In step 2, the REXX API layer passes the request to the C API layer. In step 3 the C API layer makes the interprocess communication (IPC) request to the STAF daemon process. At this point the STAF daemon determines that the request is destined for another system, which initiates step 4, a network IPC request to the STAF daemon on machine delta (running AIX Version 4.3.3). The STAF daemon on machine delta determines that the request is bound for the event service. This leads to step 5 where the request is passed to the Java service proxy layer, the layer responsible for communicating directly with the JVM, which is step 6. In step 7, the JVM invokes the corresponding method on the event service object. Upon receiving the request, step 8 shows the event service passing the request string to the common request parser of STAF for validation. At this point the event service would perform the indicated request and steps 1 through 7 would be reversed as the result was passed back to the REXX program on machine gamma.

There are a number of things to note about this request flow. First, it was quite easy to specify a network-oriented request from the point of view of the REXX program. Second, the machines in question are running different operating systems on different hardware architectures, and neither the REXX program nor the event service need be aware of this difference. Third, neither the REXX program nor the Java-based event service need be concerned with the language the other was using.

The decision to have STAF deal only with strings was the most crucial and beneficial decision we made while designing STAF. It has allowed us to keep STAF simple and flexible at the same time.

Unicode. Because we focus predominantly on strings and were concerned with codepage issues, STAF was designed to use Unicode** internally. When a call to STAFSubmit( ) is made, the input strings are converted to Unicode. All further processing is carried out in Unicode. Data are only converted out of Unicode when a result is passed back from STAFSubmit( ), or if STAF is forced to interact with the operating system or some other entity that does not

Figure 5 STAF service request flow



accept Unicode strings. By processing data in Unicode, we keep the integrity of the data intact. For example, if a system using a Japanese codepage sends a request to log some data containing Japanese codepage characters to a system using an English codepage, the data are initially converted to Unicode (which maintains the integrity of the data) when the STAFSubmit( ) call is issued. The data are maintained in Unicode until another STAFSubmit( ) call is issued to retrieve the data. If the same system running the Japanese codepage requests the data, the data will be converted from Unicode back to the Japanese codepage, which preserves the integrity of the data, since the data were originally in the same codepage. The data retrieved will be the same data initially logged even though, for some indeterminate length of time, the data were being stored or maintained on a system using an English codepage. Thus, by using Unicode throughout STAF, we solved our problem of handling multiple codepages.

Available services. In order to solve our automation problems, we needed a set of components on which to build. As we built STAF, we kept this foremost in our minds and ensured that the services we developed included these essential automation components. Here we describe some of the services that STAF provides. We will see these services again later when we examine how they were used to create the solution to our automation problems.

Three core services in STAF are the handle, variable, and queue services. These services provide fundamental capabilities that are common across all services and provide a foundation from which to build. Unsurprisingly, these services expose the capabilities of handles, variables, and queuing in STAF.

Handles are used to identify and encapsulate application data in the STAF environment. When an application wishes to use STAF, it obtains a handle by calling a registration API. The handle returned is tied specifically to the registering application. In general, this is a 1-to-*N* mapping between applications and handles. An application may have more than one handle, but any given handle belongs to a single application. However, STAF does support special "static" handles that can be shared among applications. Each STAF handle has an associated message queue. This queue allows an application to receive data from other applications and services. It also forms the basis for local and network-oriented interprocess communication in STAF. Many services deliver data to an application via its queue. These queues allow ap-

plications to work in an event-driven manner similar to the approach used by many windowing systems.

STAF provides data management facilities through STAF variables. These STAF variables are used by STAF applications in much the same way that variables are used in a programming language. When a STAF request is submitted, any STAF variables in the request are replaced with their values. One of the powerful capabilities of STAF variables is that they can be changed outside of the scope of the running application. This capability provides the ability to dynamically alter the behavior of an application. For example, an application designed to apply a specific percentage of load on a system might allow the percentage to be specified through an environment variable or as a command line argument. In this case, once the application is running, the only way to change the load percentage is to stop the application and restart it with the altered environment variable or command line argument. Using STAF variables allows the value to be changed without stopping the application. The only change to the application would be to periodically reevaluate the value of the STAF variable. These STAF variables are stored in variable pools. Each STAF handle has a unique variable pool that is specific to that application. There is also a global variable pool that is common across all handles on a given STAF system. Commonality allows default values to be specified in the global variable pool, which can then be overridden on a handle-by-handle basis.

STAF provides several other services in addition to handle, variable, and queue. STAF provides synchronization facilities through the semaphore and resource pool services. The semaphore service provides named mutual exclusion (mutex) and event semaphores. Compared with native semaphores commonly provided by an operating system, STAF semaphores have two advantages. One, they are available remotely across the network. Two, they are more visible, meaning it is much easier, for example, to determine who owns a mutex semaphore and who is waiting on an event semaphore. The resource pool service provides a means to manage named pools of resources, such as machines, user identifiers, and licenses. In particular, it provides features for managing the content of the pools as well as synchronizing access to the elements in the pools.

STAF provides process execution facilities through the process service. This service allows processes on STAF systems to be started, stopped, and queried. It provides detailed control over the execution of processes including specification of environment variables, the working directory, input/output redirection, and effective user identification. The process service can also, at user request, deliver notifications when processes end. These notifications are delivered via the queuing facilities described earlier.

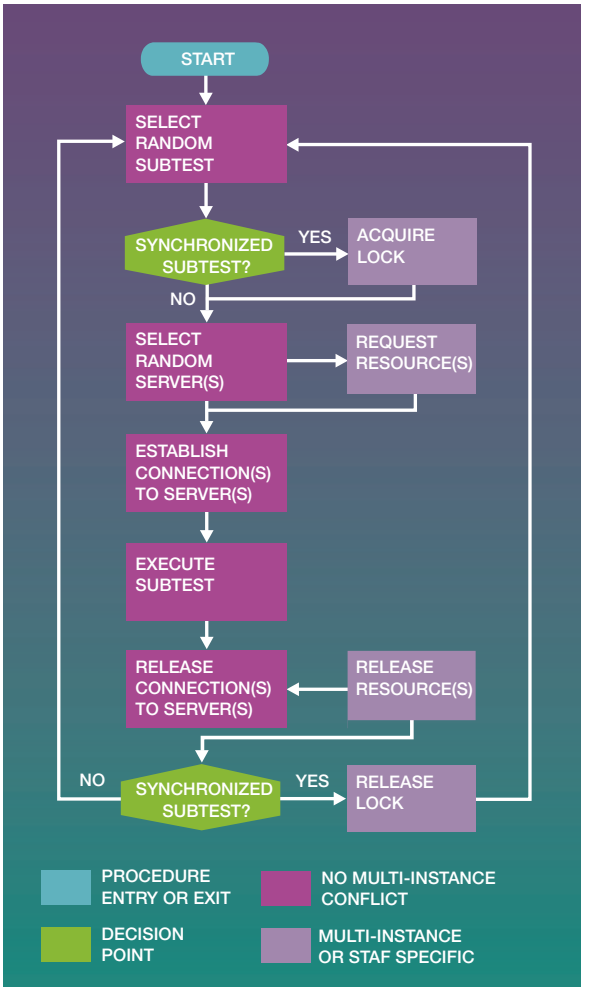
STAF provides file system facilities through the file system service. Currently, this service provides mechanisms for transferring files and accessing file content. Future versions of STAF will expand the capabilities of this service into file and directory management, such as directory creation and enumeration and file or directory deletion.

STAF provides logging facilities through the log service. At its most basic layer, this service provides time-stamped message logging based on levels, such as "FATAL," "ERROR," "WARNING," and "DEBUG." A variety of higher-level facilities are built on top of this foundation, including local and centralized logging, log sharing between applications, dynamic level-masking, and maintenance on active logs. The dynamic level-masking is of particular interest. Level-masking refers to the ability of the user to determine which logging levels will be stored in a log file. Messages with logging levels not included in the level-mask will be discarded. The fact that this feature is dynamic means that the level-mask can be changed while an application is running. For example, this ability would allow a user to "switch on" debug messages when a problem is encountered, without needing to stop and restart the application.

STAF provides remote monitoring facilities through the monitor service. This service provides a lightweight publish-query mechanism. Applications publish their state, which then allows other applications to remotely query it. The published state is a simple time-stamped string, yet this has proven sufficiently robust for monitoring the progress of typical tests and applications.

STAF provides event-handling facilities through the event service. This service provides standard publish-subscribe semantics. Applications register for specific types and, possibly subtypes, of events. Other applications generate events based on a type, subtype, and sets of properties (which are attribute/value pairs). The events are delivered via the queuing facilities described earlier.

Figure 6 Single Ogre instance after multi-instance support



In addition to the services described above, STAF makes it quite easy for groups to develop their own services to meet specific needs. These services can then become part of the set of service components available for use with STAF. The modular service-based nature of the platform provides the infrastructure for evolution and growth.

From reuse to automation

Having addressed reuse, we next focused on automation. Our plan was to build a solution on top of STAF by leveraging the automation components that it provides.

The first area we tackled was the execution of the Ogre test suite. Instead of trying to retrofit an existing test harness onto STAF, we chose to create a new one that was STAF-aware from the ground up. What we came up with was a program called the Generic WorkLoad processor or, in abbreviated form, GenWL (pronounced JEN-wall). This harness allows us to create a text file defining the configuration data for the scenario, the processes to be executed, and the systems on which they should be executed. This text file is called the *workload file*. Using GenWL, we are able to start or stop the entire workload with a single command from a central management console, which was our desired goal. GenWL also played an important role in solving other aspects of the automation problem, which are discussed below.

Next, we looked to solve the problems associated with executing more than one instance of Ogre on a given system. The two most pressing issues were test suite synchronization and resource management. To handle synchronized access to tests, we relied on the STAF semaphore service, in particular, its mutex semaphore support. This service allowed one instance of the test suite to gain exclusive access to a test and then release control once execution of that test was complete. To manage the drive letters and printer ports, we relied on the resource pool service of STAF. This service allowed us to set up separate pools for the drive letters and printer ports. The service then manages the access to entries within the pool. Thus, when one instance of the test suite requests a drive letter, we can be sure that no other instance of the test suite will obtain that drive letter until the first instance releases control of it back to the resource pool service. With these problems solved, we were able to run multiple instances of Ogre on our systems. These changes to the test suite are illustrated in Figure 6. In particular, the light purple areas of Figure 6 represent where STAF was used to solve the test suite synchronization and resource management problems.

While making the synchronization and resource management changes described above, we found ourselves redistributing the test suite more often than usual, so in conjunction with the above changes, we also set out to solve the test suite distribution problem. Here we were able to leverage the file system and variable services of STAF. Using these two services, we wrote a small script that iterated through a list of clients in a file and used the file system ser-

vice to copy each file. The variable service was used to deal with mapping the abstract destination defined in the copy command to the actual destination on each client. With the list of clients maintained in a file, we were assured the updated test suite was consistently distributed to all the clients.

STAF has allowed teams to focus on directly solving their problems instead of inventing infrastructure.

With the problems of test suite distribution and execution solved, we next addressed the test suite monitoring problem. Here we leveraged the monitor service of STAF. Our test suite published its state to the monitor service every time it entered a subtest or when an error or warning occurred. Given the published information, we next developed a way to view this information using the GenWL execution harness. The workload file read by GenWL defines all the test suite instances; thus it is trivial for GenWL to interact with the monitor service to retrieve the published state for all the test suite instances. GenWL then displays this information on a system-by-system basis. With a single command from our management console, we were able to ascertain the current state of the entire Ogre scenario.

Although GenWL and the monitor service allowed us to determine the state of the scenario at any given point in time, this capability was not sufficient for us to determine what had transpired over extended periods of time (e.g., from one evening until the following morning). With GenWL and the monitor service, we could see the state as we left and when we came in, but we were still unaware as to any problems that had occurred in between.

To solve this problem we simply exchanged our current logging mechanism with calls to the log service of STAF. This exchange allowed us to use an approach similar to the one used to solve the test suite distribution problem. We created a simple script that iterated over a list of clients in a file and used the facilities of the log service to retrieve all the error and warning messages that had been logged over a given period of time. We were then able to ascertain which, if any, of those errors and warnings were true problems or simply artifacts of temporarily pushing a server beyond its capacity. Remember, Ogre is a load and stress test, so we expect to occasionally push the servers beyond their limits.

Finally, we were left with the problem of execution dynamics. To solve this problem, we leveraged GenWL again. As mentioned above, the workload file contains the configuration information for the scenario. As the workload file is processed, this configuration information is stored on each of the client systems using the STAF variable service. As the test suite executes, it retrieves the configuration information from the variable service. By using the variable service, we were able to update the configuration information dynamically. Thus, if we needed to change the configuration information, such as to reintroduce a server or change server stress ratios, we simply updated the appropriate values in the workload file and directed GenWL to push that value out to all the clients.

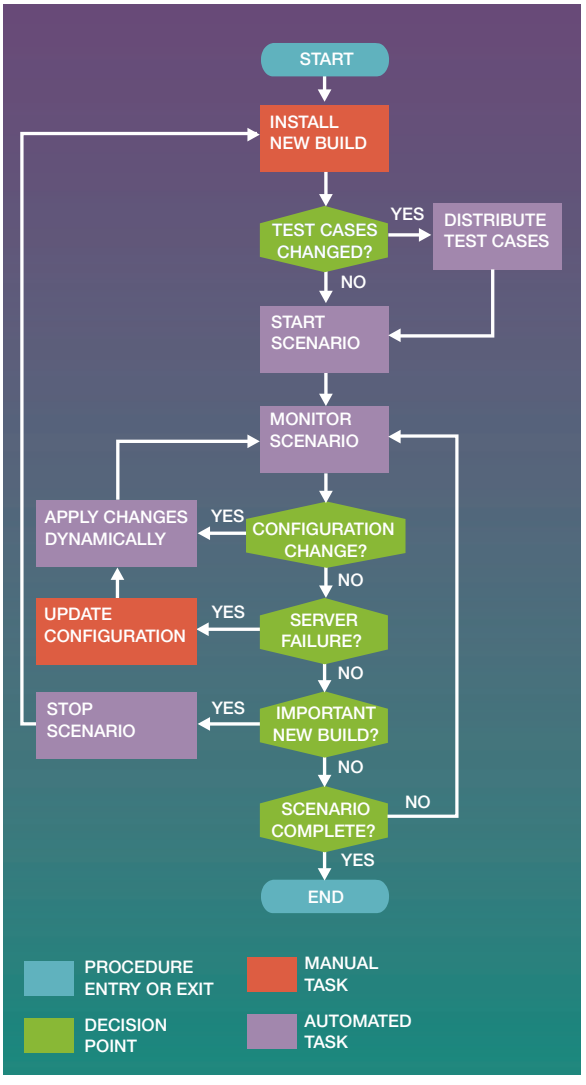
Figure 7 illustrates the flow of an Ogre scenario after our automation changes. In comparison to Figure 2, note that the majority of the steps are now automated. The only two steps that are not automated are updating the configuration and installing a new build. Updating the configuration consists of manually updating the workload file with the configuration changes. This updating effectively requires human intervention. Installing a new build is something that we have automated in other areas but was not deemed useful for the Ogre scenarios.

Issues

We have received surprisingly few complaints about STAF from our users. The vast majority of user issues concern clarifying the documentation or requesting new features (such as new services or extensions to existing services). We have also found and fixed isolated performance issues. For example, the log service was originally written in REXX, which proved to be unacceptably slow. We have since ported the log service to C++, which significantly improved its performance.

With respect to general overall performance, STAF requests do incur a minimal amount of overhead since they require an IPC request to go from the requesting process to the STAF daemon, plus the user's request string must be parsed (as opposed to dealing

Figure 7 Ogre scenario flow after automation



directly with raw data). This means STAF would not be appropriate for extremely low-latency requests. To date, we have not encountered this problem.

Benefits

By providing a reusable framework and reusable services, STAF has allowed teams to focus on directly solving their problems instead of inventing infrastructure. This advantage is illustrated with the tools developed for automating Ogre. The test distribution script and the log-querying script were both less than 50 lines of code. The scripts were so small because they were able to depend on the underlying STAF infrastructure and the services it provides. The GenWL program relies on a number of STAF services to perform its tasks. By reusing these services, GenWL is free to concern itself with the fundamental activities of parsing the command line parameters and the workload file. The remainder of the work is handled by STAF and includes setting the configuration information, starting and stopping the processes, and monitoring the test progress. This work is done with only nine commands in the GenWL program. We have found this type of usage to be fairly typical.

If we look at the application of STAF to our automation problem, we see significant savings arise. By overcoming our test suite synchronization and resource management problems, we were able to reduce the required number of client systems by approximately 33 percent, which in the largest case meant a reduction of 48 client systems. This reduction represents a very large savings in the hardware required to run the test suite.

By overcoming our test suite execution and test suite distribution problems, we were able to reduce the time it takes to restart a scenario based on a new build by roughly 50 percent. Our old manual procedure took us approximately eight hours. Our new automated procedure takes us approximately four hours. This difference is a significant reduction in time and is amplified even more when builds are received late in the day, e.g., 4:00 P.M. Because it previously took eight hours to start the scenario, we would typically begin working with the new build at approximately 8:00 A.M. the following morning. Thus the scenario was not actually running until 5:00 P.M. of that following day. However, with a reduction to four hours, someone can stay and have the scenario running by 8:00 P.M. the same night, which is an even more significant cycle-time reduction of 21 hours. In addition, it used to take several people to perform this work. Now one person can perform the work because we can manage everything from a central console. Thus, there are personnel savings as well.

A major benefit of overcoming our test suite monitoring problems was finding a number of defects in the product that would have gone undetected otherwise. Detecting problems before they reach the customer is a very significant source of savings, because problems found by customers are much more costly to fix than those found during testing.[5] In ad-

dition, our new monitoring capabilities improved morale by removing the "grunt" work of performing periodic monitoring check-ins at night and on the weekend. If a problem was uncovered while monitoring remotely, we were sometimes able to perform remote diagnostics and solve the problem without coming to the site.

Finally, by overcoming our test suite execution dynamics problems, we were able to save time and personnel by reducing the frequency of scenario restarts. This reduction in restarts was yet another morale boosting item, since we no longer felt like we were "twiddling our thumbs" when running the scenario in a configuration that we knew would have to be restarted in mid-run.

Many times our group had contemplated fixing some of the problems in the Ogre test suite. We had elaborated a list of items that we would need to create in order to solve the problem. Evaluating this list in hindsight, we realized that what we actually needed was STAF. Had we addressed our list of items earlier, we would have ended up with a solution that was centered around our particular test suite, as opposed to the general solution, which is STAF. Instead, the reuse philosophy of STAF allowed us to pick up the reusable components it provides and solve our test suite problems.

Conclusion

To improve the efficiency and effectiveness of the testing process, groups need to find ways to improve their reuse and automation. As a solution to help address these issues, we created STAF. It was designed to solve our reuse problems and was then leveraged to solve our automation problems. Using STAF, we have generated considerable savings with respect to the people, time, and hardware necessary to perform testing.

Since its inception, STAF has been adopted by numerous test groups throughout IBM, and it is being used to create a variety of innovative testing solutions. In my organization alone, we have developed a pluggable solution that drives automated testing from build through results collection. When a new build becomes available, the test systems are automatically set up and installed. Then the test suites are executed automatically, and the results are collected for analysis. These types of solutions would be tremendously more difficult, if not impossible, to create without a solution such as STAF from which to build.

After a long incubation period, STAF is available in an open-source form on the SourceForge Web site (http://staf.sourceforge.net). It is my hope that the availability of this flexible framework will lead to sustained advances in the testing efficiency and effectiveness of many software organizations.

Acknowledgments

I would like to thank Clay Williams, Karen Rosengren, Sharon Lucas, David Bender, and Don Randall for reviewing draft versions of this paper. I would like to express my sincere gratitude to Peri Tarr for helping me organize my thoughts and for keeping this paper flowing in a consistent manner. Her assistance has been invaluable in making this paper available to readers.

Cited references

1. J. Ousterhoust, "Scripting: Higher Level Programming for the 21st Century," *Computer* **31**, No. 3, 23–30 (March 1998).
2. The Open Group, http://tetworks.opengroup.org/.
3. The Object Management Group, http://www.corba.org.
4. R. Bastide, P. Palanque, O. Sy, and D. Navarre, "Formal Specification of CORBA Services: Experience and Lessons Learned," *OOPSLA Conference Proceedings* (2000), pp. 105–117.
5. R. S. Pressman, *Software Engineering: A Practitioner's Approach, 3rd Edition*, McGraw Hill, New York (1992), p. 559.

Accepted for publication September 18, 2001.

Charles Rankin *IBM Server Group, 11401 Burnet Road, Austin, Texas 78758 (electronic mail: rankinc@us.ibm.com).* Mr. Rankin is an advisory software engineer in the IBM Austin Development Laboratory. He graduated with a B.S. degree in electrical engineering from the University of Florida in 1993, after which he joined IBM in Austin. He has worked extensively with IBM's PC-oriented operating systems and networking products. He was the system test lead for IBM's Directory and Security Server for OS/2 and IBM's OS/2 WARP Server for e-Business. He is currently the lead developer for STAF.