FSA Modernization Partner

United States Department of Education

Federal Student Aid

Integrated Technical Architecture Release 3.0 Build & Test Report

Task Order #69

Deliverable # 69.1.5

Version 2.0

October 28, 2002

Table of Contents

1 EXECUTIVE SUMMARY 3

2 APPROACH 3

2.1 TESTING 3

2.2 PERFORMANCE ANALYSIS 4

2.3 CONFIGURATION MANAGEMENT 5

2.4 CLEARCASE AND REUSABLE COMMON SERVICES 5

2.5 BUILD MANAGEMENT 5

3 TEST RESULTS 6

3.1 WEB CONVERSATION FRAMEWORK 6

3.2 FTP FRAMEWORK 9

3.3 XML HELPER FRAMEWORK 10

3.4 SCHEDULER FRAMEWORK 13

3.5 USER SESSION FRAMEWORK 16

3.6 WEB SERVICES (SOAP) FRAMEWORK 18

3.7 CONFIGURATION FRAMEWORK 19

3.8 JSP CUSTOM TAG LIBRARY FRAMEWORK 20

1 Executive Summary

The Integrated Technical Architecture Release (ITA) 3.0 Build and Test Report documents the build procedures, test conditions, and results of the ITA Release 3.0 Reusable Common Services (RCS). This report is divided into two seperate documents, the first is the Build & Test Report which is provides an abbreviated, executive summary of the development and testing of Reusable Common Services, and the second document (named Build & Test Report Appendix) provides in-depth coverage of the following topics:

- RCS test conditions and results
- RCS performance analysis
- RCS usage scenarios
- RCS repository and build approach

ITA Release 3.0 RCS includes:

- Web Conversation Framework
- File Transfer Protocol Framework
- XML Helper Framework
- Scheduler Framework
- User Session Framework
- Web Services (SOAP) Framework
- Configuration Framework
- JSP Tag Library

2 Approach

2.1 Testing

To ensure the quality of RCS, each service has undergone extensive unit testing. ITA relied on two methods in conducting unit tests: Automated and manual unit test. Wherever applicable, ITA utilized JUnit for automated testing. JUnit is a set of Java packages that allows developers to create Java test cases for Java classes, and to then run these unit tests interactively or in batch mode. The intended result is higher quality code, as well as avoidance of the cumbersome and repetitive task of reconstructing unit tests after all code has been written/modified.

For special cases where automated unit tests could not be performed by JUnit, ITA developers conducted manual tests. Test conditions for automated and manual testing are provided in this document and expected results are also provided for manual testing.

Benefits to the unit test approach are:

- Standardized test conditions and cycles
- Increased code quality
- Increased consistency in the approach to testing
- Increased productivity
- Reduced time for regression testing

- More time available to spend on enhancements as less time is required for fixes

Test conditions are documented in a tabular format with the following column headings:

Column	Description
Condition Number	Test condition number
Detailed Condition	Detailed description of the test condition/case
Test Class Name	Test class name in JUnit Java code
Test Class Method	Test method (test case) in the test class
Class Name	Specific Java source file of the RCS code to be tested by the JUnit test code
Method Name	Specific method in the Java source file of the RCS code to be tested by the JUnit test code
Results	Expected results
Data File Name	Any configuration and/or data files needed to execute this test case

2.2 Performance Analysis

To ensure program efficiency and to detect possible bottlenecks, ITA used JProbe (a performance profiling tool produced by Sitraka) to analyze each RCS component to identify common performance issues such as thread integrity and memory management.

Common thread integrity issues include:

- Data Race Conditions – occur when concurrent threads attempt to access a shared resource at the same time. One thread can be writing to a shared resource at the same time another thread attempts to read from or write to the same shared resource. This will result in unreliable data.
- Deadlocks – occur when one thread is holding a lock while attempting to acquire a lock held by another thread, while at the same time, the second thread needs the lock held by the first. Incorrect programming logic will cause the threads to never move forward and cause the program to terminate.

Common memory management issues include:

- Loitering objects – occur when the application will not use the objects again, but the developers fail to remove the reference to the objects, the objects will remain, or loiter, in memory indefinitely. This condition can consume a significant amount of memory and degrade the Java Virtual Machine (JVM) performance.
- Excessive object allocation – if the application creates an excessive number of objects, the Java heap (a type of memory) will grow larger and garbage collection activities will take longer because there are more objects to evaluate. This will also degrade the JVM performance.

The performance analysis for each service is documented in this report. The topics included in the performance analysis are:

- Background information

- Test harness design
  - Test environment
  - Test configuration
  - Test scenario
- Memory (Heap) analysis
- Performance analysis
- General Performance Metrics

2.3 Configuration Management

ITA uses Rational ClearCase for its configuration management system. ClearCase manages multiple variants of evolving software systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies.

These capabilities enable ClearCase to address the critical requirements of organizations that produce and release software:

- **Effective Development:** ClearCase enables developers to work efficiently, allowing them to fine-tune the balance between sharing each other's work and isolating themselves from destabilizing changes. ClearCase automatically manages the sharing of both source files and the files produced by a software build.
- **Effective Management:** ClearCase tracks the software build process, so that developers can determine what was built, and how it was built. Further, ClearCase can instantly recreate the source base from which a software system was built, allowing it to be rebuilt, debugged, and updated all without interfering with other programming work.
- **Enforcement Of Development Policies:** ClearCase enables project administrators to define development policies and procedures, and to automate their enforcement.

2.4 ClearCase and Reusable Common Services

ClearCase is a robust version control system that can manage large projects with highly interdependent code. There are two main capabilities that RCS will utilize within ClearCase:

- **Version Control:** Developers use ClearCase on a daily basis to maintain a complete history of their project files. This will aid developers, project managers, and build managers in maintaining a complete picture of the progression of a resource.
- **Common Directory Structure:** A common directory structure will be implemented and utilized for each common service. This standard will simplify the process of releasing major and minor versions.

2.5 Build Management

ANT is a Java-based build tool used for RCS Build Management. ANT is an open-source project from the Jakarta Project. It is a powerful scripting tool that lets developers execute the build processes around the code requirements using predefined tasks. A defined build process ensures that the software in the development project is built in the exact same manner each time a build is executed. As the build process becomes more complex it becomes increasingly necessary to achieve such consistency. ANT is a platform-independent scripting tool that lets

developers construct build scripts using a large number of built-in tasks with minimal customization.

The ANT tool builds the following distribution packages for each RCS framework:

Package	Description
Distribution Package	• Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.package.exe • Contents: Executables, Source, Documentation, Release Notes
Executable	• Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.exe The Executable package will be the distribution for any project that would just like the deliverable components, with the necessary libraries and class files present. The self-extracting WinZip file will only need to be extracted into the class path of the application for the RCS components to be utilized.
Source	• Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.src.exe This is the package of all of the source files.
Documentation	• Package naming format: EX. RCS.<Service>.<Major #>.<Minor #>.doc.exe The documentation package will contain the design documents, user guides, examples, configuration files, test documentation and the JavaDoc associated with a particular common service.
Release Notes	• Package naming format: EX. readme.<Service>.<Major #>.<Minor #>.txt The release notes will be used to provide the end user of RCS components with an overview of the capabilities and any installation and build instructions in order to utilize the components.

3 Test Results

3.1 Web Conversation Framework

3.1.1 Test Conclusions

A formal unit test was not conducted on the Web Conversation framework. It is leveraged from an established framework created by the Jakarta Group's as part of the Apache project. Apache frameworks are open source frameworks. They have been already unit tested by Apache. It is not necessary to reperform the unit test on an established open source framework.

ITA performed an analysis of the example application packaged with the Struts distribution. Analysis of the results led to the conclusion that the Web Conversation framework does not produce any loitering objects.

Action objects are loaded into a Hash Map that stays in memory once an ActionMapping has used it. Only one object is created for each action and it is reusable. These objects still remain in reachable memory during the life of the Web Application. This could impact the performance of the system if numerous Action objects are defined for that application.

3.1.2 Performance Testing

3.1.2.1 Scenario

The example application provided with the framework distribution was used as the test harness. LoadRunner was used to execute the scenario twenty-five times to obtain an accurate measurement of the test results on average.

3.1.2.1.1 Test Preparation

Refer to the JProbe Quick Start Guide for the test execution preparation information. This guide identifies the steps required to profile an application using JProbe.

3.1.2.1.2 Test Scenario

1. Open a web browser and connect to the site http://stg.jprobe.fsa.ed.gov/JPROBEWebApp/index.jsp
2. On index page, select Register with the MailReader Demonstration Application link

3. Create a new user:
   a. User: test
   b. Password: testing
   c. Full Name: testy tester
   d. From Address: test@test.com
   e. Reply-to: info@test.com
4. Save
5. Select "Edit your user registration profile"
6. Modify the From Address and Reply To Address to anything.
7. Press Reset
8. Select "Add"
9. Create a new subscription:
   a. Mail Server: mail.yahoo.com
   b. Mail Username: tt33
   c. ttt
10. Save
11. Edit the newly created subscription
12. Press reset
13. Edit Mail Server: smtp.yahoo.com
14. Save
15. Delete the subscription from the list
16. Confirm
17. Save
18. Select "Log off MailReader Demonstration Application"
19. Select "Log on to the MailReader Demonstration Application"
20. Username: amy
21. Submit

22. Error message - Password: pass
23. Select "Log off MailReader Demonstration Application"

3.1.2.2 Results

The JProbe Profiler with Memory Debugger application was used to trace both the memory usage and performance measurement of the example application. Two snapshots were taken: A heap snapshot and a performance snapshot.

3.1.2.2.1 Memory (Heap) Analysis

The heap graph showed that the level of the trough is getting higher over time, meaning that not all temporary objects are being garbage collected. The spikes are expected since new user and subscription objects are being created in the scenario and then garbage collected. The level change of the troughs is unexpected since the test was conducted with the assumption that all temporary objects will be removed from the heap. The next section will examine the instance summary to determine if these are loitering objects.

3.1.2.2.2 Instance Summary

The count change for the User class is +25 (which coincides with the number of times the scenario was executed). The User class is part of the Struts example application and not actually part of the Web Conversation framework . A new User object was created in each cycle during the execution of the scenario but the objects were never removed from the heap. These loitering objects are attributed to the design of the example application and not to the Web Conversation framework.

By examining the source code, it is determined that once the Action instance is created, it is placed in a HashMap, which is not removed from the heap until the Web Application has stopped. This means that any Action class defined by a mapping will be loaded into a FastHashMap when it is used. The Action object is not removed from the HashMap until the ActionServlet.destroy() or ActionServlet.reload() method is called. Since WebSphere loaded the ActionServlet, the destroy() and reload() methods will never be called within the scope of the JProbe performance analysis[1]. Although this is not a memory leak, it is necessary to keep in mind that overall system performance could be impacted if a tremendous number of Actions are defined per Application Server.

Although the Action objects will remain in the heap memory when not in use, they will be available without the additional overhead of being re-instantiated the next time the Action is invoked. This paradigm follows the theory that an object invoked will most likely be used again.

[1] Because the ActionServlet was loaded on startup of the Application Server, it will not be unloaded until the Web Application is stopped. This release will not show up in the profile of the example application as JProbe stops collecting data prior to the shutdown of the Application Server.

3.2 FTP Framework

3.2.1 Test Conclusion

Unit Testing - Both automated and manual tests went through with pass status.

Performance Profiling - Loitering object found in analysis. Code change was made to resolve the problem.

3.2.2 Unit Testing

3.2.2.1 Scenario

Two test scenarios were used for both automated and manual testing. Automated testing was done with the JUnit unit testing tool. In the automated scenario, a user establishes connections in the combination of Active / Passive transfer modes and SSL / NonSSL connect modes. By doing this test, the proper functioning of the FTP client can be assured. The manual testing had a different goal. Besides running through a typical FTP session, the scenario also tries to test the application's exception handling ability. FTP return code is checked after each file transfer in the framework. This ensures the integrity of the files uploaded or downloaded.

3.2.2.2 Results

Unit testing was done in both automated and manual fashions during the test session. Both automated and manual tests went through with pass status.

3.2.3 Performance Testing

3.2.3.1 Scenario

To profile the FTP Framework, LoadRunner was used to simulate real users stepping through the application. In the profiling scenario, the user logged onto a ftp server, navigated to the desired location and uploaded / downloaded files in both ASCII and BIN modes. The same process was repeated 20 times. By running the same process multiple times, better statistics could be collected.

3.2.3.2 Results

Performance profiling on RCS FTP Framework was done on JProbe profiling tool. Two sets of statistics were taken, memory (heap) usage and application performance. Analysis was done on these two sets of data; both heap analysis and performance analysis are included in the document.

During the heap analysis, one object was found loitering in the heap. One instance of FtpControlSocket was not garbage collected at the end of the profiling session. Code changes were applied and the problem was fixed. Application performance did not pose to be an issue. FtpClient.putFile() did appear to be high on execution time, however, it was due to the file transfer time instead of actual execution time. Section 2.6.4.1 of the Build and Test Report Appendix document includes a table of the top ten method times for the developer to reference when including the framework in his / her application.

3.2.3.2.1 Memory (Heap) Analysis

Three FTP Framework classes - FTPConnectAction, FTPMoveFileAction and FtpControlSocket, left one object each in the heap after the third garbage collection. As shown in the RCS Web Conversation performance results, *Action classes are reused throughout the life of the application. Thus, it was normal for the framework to leave one instance of FTPConnectAction and FTPMoveFileAction in its heap. However, FtpControlSocket was not expected to remain in the heap after the last garbage collection. This situation could potentially cause memory leak. The following section goes into more detail on this. A complete analysis of this issue is in the Build and Test Report Appendix document.

3.2.3.2.2 Instance Summary

From the instance summary, it can be observed that the FtpControlSocket class was called by FtpClient class. In the FtpClient class, the logout() method is supposed to close input/output streams and control socket itself. However, the method does not set FtpControlSocket to null after the end of the execution. Thus, one class object remains. Code changes were made to resolve this issue.

3.2.3.3 Results

Most time was spent in the servlets branch. In particular, the FtpClient.putFile() method took the most execution time. Thus, the path that led to FtpClient.putFile() was the critical path of this framework. To improve performance, the critical path should be looked at first.

In FtpClient.putFile, put() method had the most execution time. The top reference to the put() method was the createDataSocket() method with a cumulative method time of 10 milliseconds. Further more, the total method time for the reference methods did not add up to the put() method cumulative time. Also, the actual average execution time for the method was 85 milliseconds. Thus, it can be concluded that the majority time was spent in file transfer rather than method execution. The conclusion is reasonable since the file used during the profiling session was a large binary file.

3.3 XML Helper Framework

The XMLHelper framework is helpful to developers that need parsing of XML documents. The XMLHelper framework provides 3 different XML Parsing technologies. The framework provides a DOM level 2 parser that will build a DOM tree in memory and provide users with attribute and element values. The framework also provides a SAX parser that will allow a developer to pass a custom SAX parsing class so that SAX level 2 parsing can occur on a XML document. Finally the framework provides a data-binding parsing ability that will instantiate a Java object from a XML document or write a XML document that represents a Java object.

3.3.1 Test Conclusion

The XMLHelper framework was unit tested using the JUnit product suite. Two priorities were focused on during unit testing. The first area was to ensure that all parsing API's within the framework provided proper error messaging and exception handling if the XML documents

were missing or invalid in some way. The second priority was to ensure that a simple parse worked with each technology. The final priority was to execute several methods within a single test for each parsing API that would simulate real work that a developer would actually do.

The conclusion from the JUnit testing is that the XMLHelper framework properly parses example XML documents using all three parsing technologies. This framework could significantly help a development group in several areas. One, the framework standardizes all three parsing technologies within a single package and provides standard search and retrieval methods to the user for the different API's. Two, the framework provides documentation as to when the developer should use DOM parsing versus SAX parsing. This documentation lessens the learning curve a development group must go through to achieve XML understanding. Finally the framework provides simplified method access to actually do the parsing. All the developer has to do in some cases is parse and search for a value.

3.3.2 Unit Testing

3.3.2.1 Scenario

The unit test for the XMLFramework framework was automated by using JUnit. The test conditions matrix can be found in the Build and Test Report Appendix document.

In order to execute the unit test conditions for the XMLHelper Framework, some example XML documents are needed. These documents are shown in the Build and Test Report Appendix document and include the following:

Parser	File name	Description
DOM,SAX	Example.xml	Example XML Document that contains normal elements and attributes. This document also support Namespace attributes.
DataBind	Mapping.XML	Example attribute mapping document that contains attribute mapping information the Databind framework uses to instantiate a Java object from a XML document.
DataBind	Schedule.XML	Example data mapping document that provides data values for the Java Object.
DataBind	Mapping.dtd	Dtd library that provides support for the mapping XML document .
DataBind	New.XML	This document is actually built by the Databind section of the XMLHelper framework to show the example of a XML document being created that represents a Java object
Dom,Sax	Missing.xml	A non-existent XML document. It is referred to in the JUnit testing, but doesn't exist to test error recovery.
DOM,SAX	BadFormat.XML	This XML document is a copy of example.xml

		with a parsing error. Again provide to check error routines.

3.3.2.2 Results

All JUnit test scenarios passed.

3.3.3 Performance Testing

3.3.3.1 Scenario

The performance testing of the XMLHelper framework involved providing three JSP files with each file testing a different parsing technology. A listing of the JSP files can be found in the Build and Test Report Appendix document.

DomTest.jsp - Currently configured to build a DOM tree of an example XML document and then search for a specific element. Uses the XMLHelper ***searchDom*** method, which returns the element's value**.** Once returned, the value is checked against what was configured and will output whether the value's match status. This JSP can be configured to loop multiple times so that multiple DOM parses and multiple searches take place. In this test the loop was configured to 10 passes.

SaxTest.jsp - Currently configured to use the SAX protocol to parse the example XML document and then search for a specific element. Uses the XMLHelper ***searchSAX*** method, which returns the element's value. Once returned, the value is checked against what was configured and will output the value's match status. This JSP can be configured to loop multiple times so that multiple SAX parses and multiple searches take place. In this test the loop was configured to 10 passes.

BindTest.jsp - Configured to instantiate a Java Object from two XML documents. The first document, ***mapping.xml***, defines the attributes of the Java object the parser is trying to build. The second XML document called ***schedule.xml*** holds the objects attribute values. A Java Server Page will construct the Java object called ScheduleEntry from the scheduler framework. This JSP can be configured to loop multiple times so that multiple ScheduleEntry objects will be built. In this test the loop was configured to 10 passes.

3.3.3.2 Results

The results gathered from the application that are external to the XMLHelper Framework APIs will not be included in the performance profiling results. These results will be excluded since the purpose of profiling is to determine the performance of the application under normal conditions. The performance of the methods used to test the APIs has to be excluded to test just the behavior of the framework. The following sections 3.3.3.2.1 and 3.3.3.2.2 describe the results of the performance test.

3.3.3.2.1 Memory (Heap) Analysis

When the Application Server is initialized, approximately 4000 KB of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe

will call the Garbage Collector to remove objects that are no longer being referenced from the heap.

A Checkpoint will then be set to mark the starting count point of this performance analysis. The object count will be measured against the count at the checkpoint. The overall memory usage for the XMLHelper framework is very low and will not result in huge increase to the overhead of calling applications. A graph displaying this information can be found in the Build and Test Report Appendix document.

3.3.3.2.2 Instance Summary

These results were gathered after the test scenario finished executing and garbage collection occurred. The results were filtered for 'gov.ed.sfa.XMLHelper.*' since those are the classes the XMLHelper Framework is concerned with.

The SAX parser has the lightest footprint in terms of object count and memory. This is due to how the SAX parsing technology works. The SAX parser calls pre-specified methods that are part of a SAX handler class, when the parser encounters different element tokens within the XML document. For example, when the parser encounters a document start tag, the parser launches the method StartDocument(). When the parser encounters an element start tag, the parser launches StartElement(). During the performance test, the SAX parser was using the handler SAXSearchHandler class. Most of the tag methods are very simple so not much work is done. These handler classes are expected to be developed by development groups so the memory and object numbers will change with the development changes but in general the SAX parser has always been known to use the lightest load. The total memory used for the SAX API test is 208 bytes with the number of objects at 21.

Since the DOM API builds a DOM tree in memory that represents the entire XML document, the DOM parsing API will be much more expensive in memory then the SAX API. In comparison to the SAX API, the DOM tree uses almost 20% more objects and 400% more memory. The memory used in this test was still less then 5 Kbytes and the number of objects approached 300.

The DataBind API does take a bit more memory in comparison to the other XML API parsing technologies but the functionality that the DataBind API adds is much more complex then simply parsing the XML document. The DataBind API parses two XML documents and then builds a Java Object that reflects the data that is in the XML documents. The DataBind API also has the ability to build a XML document that reflects an existing Java Object. The total memory used for the Databind API test is 100 Kbytes with the number of objects approaching 2300.

3.4 Scheduler Framework

The Scheduler framework is a FSA Java package that provides Alarm scheduling that is very similar to UNIX cron and Windows AT daemons. This scheduling allows applications to set alarms that will launch Java methods to do work. The scheduling can occur on a recursive basis (ex. every minute, every hour) or can occur on a scheduled format (ex. Tuesdays at 10AM). There are also methods to remove and change alarms.

3.4.1 Test Conclusion

The Scheduler framework was unit tested using the JUnit product suite. The focus of the test was to ensure the Schedule framework could input scheduled data through a XML interface (using the XMLHelper framework) or inline programming. The conclusion from the JUnit testing is that the Scheduler Framework is capable of setting alarms that would happen on a recursive or a scheduled basis using both inputs.

3.4.2 Unit Testing

3.4.2.1 Scenario

The unit test for the Scheduler Framework was automated by using JUnit. The test conditions matrix can be found in the Build and Test Report Appendix document.

In order to execute the unit test conditions for the Schedule Framework, some example XML documents are needed. These documents are shown in the Build and Test Report Appendix document.

File name	Description
OneTimeShotMapping.XML	Example mapping XML document that shows the class ScheduleEntry and all its attributes for the onetime alarm scenario.
OneTimeShotSchedule.XML	Example data XML document that shows data for the Class ScheduleEntry for the onetime alarm scenario.
Mapping.dtd	Dtd library that provides support for the mapping XML documents
RecursMap.XML	Example mapping XML document that shows the class ScheduleEntry and all its attributes for the recursive alarm scenario.
Recursched.xml	Example data XML document that shows data for the Class ScheduleEntry for the recursive alarm scenario.

3.4.2.2 Results

All JUnit test scenarios passed.

3.4.3 Performance Testing

3.4.3.1 Scenario

The performance testing of the Scheduler framework involved providing two JSP files with their respective XML documentation.

onetime.jsp - Currently configured to add an alarm at a specified time to launch the method ***method1()*** that is part of the class ***gov.ed.sfa.ita.schedule.test.SchFire***. This JSP uses two XML documents, called onetimed.XML and onetimem.XML, as XML input to schedule the alarm. Once the alarm has been set, the JSP uses the method containsAlarm() to check if the alarm has been set and if true, the program uses removeAllAlarms to remove the alarm. This JSP can be configured to loop multiple times so that multiple alarms can be scheduled. In this test the loop was configured to 10 passes.

recurs.jsp - Currently configured to add an alarm to launch the method ***method5()*** that is part of the class ***gov.ed.sfa.ita.schedule.test.SchFire*** one minute after the scenario is executed. This alarm is set up to continue to execute in a recursive manner every minute after the scenario starts. This JSP uses two XML documents, called recursd.XML and recursm.XML, as XML input to schedule the alarm. Once the alarm has been set, the JSP uses the method containsAlarm() to check if the alarm has been set and then if true, the program uses removeAllAlarms() to remove the alarm. This JSP can be configured to loop multiple times so that multiple alarms can be scheduled. In this test the loop was configured to 10 passes.

3.4.3.2 Results

The results gathered from the application that are external to the Schedule Framework APIs will not be included in the performance profiling results. These results will be excluded since the purpose of profiling is to determine the performance of the application under normal conditions. The performance of the methods used to test the APIs has to be excluded to test just the behavior of the framework. The following sections 3.4.3.2.1 and 3.4.3.2.2 describe the results of the performance test.

3.4.3.2.1 Memory (Heap) Analysis

When the Application Server is initialized, approximately 4000 KB of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe will call the Garbage Collector to remove objects that are no longer being referenced from the heap.

A Checkpoint will then be set to mark the starting count point of this performance analysis. The object count will be measured against the count at the checkpoint. The overall memory usage for the Scheduler framework is very low and will not result in huge increase to the

overhead of calling applications. A graph displaying this information can be found in the Build and Test Report Appendix document.

3.4.3.2.2 Instance Summary

These results were gathered after the test scenario finished executing and garbage collection occurred. The results were filtered for 'gov.ed.sfa.schedule.*' since those are the classes the Scheduler Framework is concerned with.

Both the recursive alarm scenario and the one time alarm scenario show similar results in terms of memory used and object counts. The largest amounts of objects gathered for the framework is the ScheduleEntry class, which is expected because this object represents the Alarm. For each alarm we have one ScheduleEntry object. Since this test included running the one time scenario 10 times and the recursive scenario 10 times, we would expect the framework to build 20 ScheduleEntry objects, which jprobe confirmed. This is the most numerous object in this test with a total memory consumption of 1520 bytes.

3.5 User Session Framework

3.5.1 Test Conclusion

The Session framework passed unit testing without any problems. Developers can be confident that the framework works as expected.

From analyzing the results of the performance analysis for the test harness, it is concluded that the Session framework does not produce any loitering objects.

3.5.2 Unit Testing

3.5.2.1 Scenario

Three sets of applications have been created to test the different functionality available within the User Session framework. The basic design and flow of these applications are the same with slight changes in the constructor. The constructor is used to access the User Session framework's ContextManager to store and retrieve user session information.

The files associated with each set of the test applications have been placed into separate directories: cookieTest, sessionTest, sessionIBMTest. See the Build and Test Report Appendix document for the complete path to the directory.

Multiple test cycles were introduced for some of the test applications to test the different session manager parameter configuration (i.e. using a regular HttpSession versus using an IBMSession object).

The following URLs are used to access the different pages of the test applications:

http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/cookieTest/*.jsp[2]
http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/sessionTest/*.jsp
http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/sessionIBMTest/*.jsp

3.5.2.2 Results

The Session framework unit test was conducted manually. All test scenarios passed. A detailed account of the results can be found in the Build and Test Report Appendix document.

3.5.3 Performance Testing

3.5.3.1 Scenario

Test applications created for the unit test will be used to execute the performance analysis. Portions of Test Cycles: 1, 2, 3, and 5 will be executed to test the performance of the User Session framework in different scenarios.

Test Cycle 1 will be executed to profile the performance of methods used to access and store data from cookies. Test Cycle 2 and 3 will test the use of storing user data in HttpSession objects in either the application server memory or in a persistent database. Test Cycle 5 will be used to test how the API functions when using an IBMSession object instead of an HttpSession object.

3.5.3.2 Results

The results gathered from the application that are external to the User Session Framework APIs will not be included in the performance profiling results. These results will be excluded since the purpose of profiling is to determine the performance of the application under normal conditions. The performance of the methods used to test the APIs has to be excluded to test just the behavior of the framework. The following sections 3.5.3.2.1 and 3.5.3.2.2 describe the results of the performance test.

3.5.3.2.1 Memory (Heap) Analysis

When the Application Server is initialized, a great deal of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe will call the Garbage Collector to remove objects that are no longer being referenced from the heap.

A Checkpoint will then be set to mark the starting count point of this performance analysis. The object count will be measured against the count at the checkpoint. From analyzing the results, it can be determined that the overall memory usage for the User Session framework is very low and will not result in huge increase to the overhead of calling applications.

3.5.3.2.2 Instance Summary

These results were gathered after the test scenario has finished executing and garbage collection has occurred. We then filtered for "*session*" since those are the only results we are interested

[2] Where *.jsp refers to the different JavaServer Pages within each directory as listed in the test conditions and test scripts.

in. The Count Change column was used to sort the data to determine which objects remain loitering in the heap after the scenario has been completed.

None of the User Session framework objects remain in the memory heap after garbage collection has been called. This includes all calls to the ContextManager class, which in turn calls the CookieRetrieval or SessionRetrieval classes. From this we can determine that the User Session framework does not create any loitering objects once the browser has been exited or the session invalidated.

3.6 Web Services (SOAP) Framework

3.6.1 Test Conclusion

A formal unit test was not conducted on the SOAP Framework. It is leveraged from an established framework created by the Jakarta Group as part of the Apache project. Apache frameworks are open source frameworks. They have been already unit tested by Apache. It is not necessary to reperform the unit test on an established open source framework.

ITA performed an analysis of the example messaging application packaged with the SOAP distribution. Analysis of the results led to the conclusion that the SOAP Framework does not produce any loitering objects.

3.6.2 Performance Testing

3.6.2.1 Scenario

The example SOAP messaging application provided with the framework distribution was used as the test harness. The test was executed with one message and also with three messages.

3.6.2.2 Results

3.6.2.2.1 Memory (Heap) Analysis

The heap graph showed the level of the memory usage getting higher over time. This is expected since the first message invokes the Messaging Servlet. Once the servlet receives the message, an object is created to process that message and send a return message. When three messages are run, the memory usage reaches the same hight as a single message and levels off. This shows that processing a message does not create any loitering object. The graph displaying this information can be found in the Build and Test Report Appendix document.

3.6.2.2.2 Instance Summary

These results were gathered after the test scenario has finished executing and garbage collection has occurred. The results were filtered for ‘samples.messaging.*’ since those are the classes the SOAP Framework is concerned with.

None of the SOAP Framework objects remain in the memory heap after garbage collection has been called. All the message objects are destroyed as are the objects created by the servlet to process the messages. From this we can determine that the SOAP framework does not create any loitering objects once the messages have been processed.

3.7 Configuration Framework

The purpose of the ITA Configuration Framework is to provide a standard for application configuration input. The framework allows configuration information to be loaded from properties files, xml files, or database tables.

3.7.1 Test Conclusions

The Configuration Framework passed unit testing without any problems. Developers can be confident that the framework works as expected.

From analyzing the results of the performance analysis of the test configuration application, it is concluded that the Configuration framework does not produce any loitering objects. Developers will need to keep in mind that String objects are loaded into a DomXml object that stays in memory once an application has been started. These objects still remain in reachable memory during the life of the web application.

3.7.2 Unit Testing

3.7.2.1 Scenario

The unit test for the Configuration framework was automated by using JUnit. The test conditions matrix can be found in the Build and Test Report Appendix document.

In order to execute the unit test conditions for the Configuration Framework, some configuration data needed to be created. A properties file was created to test the properties file portion of the Configuration Framework. An xml file was created to test the xml file portion of the Configuration Framework. Three database tables with data were created to test the database table portion of the Configuration Framework.

3.7.2.2 Results

All JUnit test scenarios passed. A detailed account of the results can be found in the Build and Test Report Appendix document.

3.7.3 Performance Testing

3.7.3.1 Scenario

The configuration performance test focused on one usage scenario for its analysis: The creation of the configuration information in memory.

The test did a Class.forName("gov.ed.fsa.ita.config.FSAConfigurationSI") which runs the static initializer within the FSAConfigurationSI class. This static initializer loads all the configuration data within the application into a storage object.

3.7.3.2 Results

3.7.3.2.1 Memory (Heap) Analysis

When the Application Server is initialized, approximately 4000 KB of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe will call the Garbage Collector to remove objects that are no longer being referenced from the heap.

A Checkpoint will then be set to mark the starting count point of this performance analysis. The object count will be measured against the count at the checkpoint. The overall memory usage for the Configuration framework is very low and will not result in huge increase to the overhead of calling applications. A graph displaying this information can be found in the Build and Test Report Appendix document.

3.7.3.2.2 Instance Summary

These results were gathered after the test scenario has finished executing and garbage collection has occurred. The results were filtered for 'gov.ed.fsa.ita.config.*' since those are the classes the Configuration Framework is concerned with.

The count change for the String class is very high at 5,503. This is expected because the Configuration Framework must create a String object every time it loads a new configuration parameter. It creates a String object to read in the parameter, then places this parameter into the main PropertiesPlus object. When the static initializer has completed loading the configuration data, this PropertiesPlus object holds all the data. This data is stored for the life of the web application, so there should be String objects held in memory.

3.8 JSP Custom Tag Library Framework

3.8.1 Test Conclusion

Overall, the JSP Custom Tag Library framework performed well. All tags from the ITA custom built logging tag library passed Unit Testing and produced no loitering objects in the heap. Major tags and attributes of the leveraged custom tag libraries were tested. A few of the attributes failed to work as expected and details can be found in the Build and Test Report Appendix document.

3.8.2 Unit Testing

3.8.2.1 Scenario

JavaServer Pages were created to test each tag library. These JSPs contained the tags and attributes used in a manner similar to how developers would use them. Each JSP was then loaded to determine test the tag library.

3.8.2.2 Results

While the majority of the tags and attributes worked as expected, a few did not. The unexpected results can be mostly attributed to the container used (WAS 3.5.5) or the design of the JSP. It is the developer's responsibility to ensure the tags and attributes included in this framework will work appropriately with his/her framework. Refer to the Build and Test Report Appendix document for detail data on the test results.

3.8.3 Performance Testing

3.8.3.1 Scenario

The test JSP created for the unit test was also used to execute the performance analysis of the Logging tag library. The tags attempt to log different levels of messages to the log file. The test will also validate that an error message is logged when the user uses an incorrect/non-existent logging level. The tag will be used to test if the given logging level can be logged based on the current filtering criteria.

3.8.3.2 Results

The results gathered from the application that are external to the Custom Logging Tag Library APIs will not be included in the performance profiling results. These results will be excluded since the purpose of profiling is to determine the performance of the application under normal conditions. The performance of the methods used to test the APIs has to be excluded to test just the behavior of the framework. The following sections 3.8.3.2.1 and 3.8.3.2.2 describe the results of the performance test.

3.8.3.2.1 Memory (Heap) Analysis

The heap snapshot can be used to visualize how memory is being used in the heap, obtain information on objects allocated in the heap, and determine if there are any loitering objects at the end of the test.

3.8.3.2.2 Heap Graph Analysis

When the Application Server is initialized, approximately 4000 KB of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe asks the Garbage Collector to remove objects that are no longer being referenced from the heap.

A Checkpoint is then set to mark the starting count point of this performance analysis. The object count remaining in the heap at the end of the test is measured against the count at the checkpoint. By reading the graph, it can be determined that the overall memory usage for the JSP Custom Logging Tag Library is very low and will not result in huge increase to the overhead of calling applications.

3.8.3.2.3 Instance Summary

The above results were gathered after the test scenario has finished executing and garbage collection has occurred. We then filtered for "gov.ed.*" since those are the only results we are interested in. The Count Change column was used to sort the data to determine which objects remain loitering in the heap after the scenario has been completed.

None of the Logging Tag Library objects remain in the memory heap after garbage collection has been called. From this we can determine that the Logging Tag Library does not create any loitering objects.