

FSA Modernization Partner

United States Department of Education

Federal Student Aid



Integrated Technical Architecture Release 3.0
Build & Test Report
Appendix

Task Order #69

Deliverable # 69.1.5

Version 2.0

October 28, 2002



Table of Contents

1	RCS – Web Conversation Framework	12
1.1	PURPOSE	12
1.2	APPROACH	12
1.3	SUMMARY	12
1.4	TEST HARNESS DESIGN	13
1.4.1	TESTING ENVIRONMENT	13
1.4.1.1	Testing Criteria	13
1.4.1.2	Testing Configuration	14
1.4.1.3	JProbe Configuration File	14
1.4.1.4	UNIX Server Settings	15
1.4.1.5	WebSphere Application Server Configuration	15
1.4.1.6	struts-config.xml File	16
1.4.1.7	Additional Required Components	16
1.4.1.8	Directory Structure	16
1.5	TESTING SCENARIO	18
1.5.1	TEST PREPARATION	18
1.5.2	TEST SCENARIO	18
1.6	RESULTS AND ANALYSIS	19
1.6.1	HEAP SNAPSHOT (MEMORY USAGE)	19
1.6.1.1	Heap Graph Analysis	19
1.6.1.2	Instance Summary	20
1.6.2	PERFORMANCE SNAPSHOT (CODE EFFICIENCY)	23
1.6.2.1	Number of Calls	25
1.6.2.2	Method Time	25
1.6.2.3	Cumulative Time	26
1.6.2.4	Method Object Count	27
1.6.2.5	Cumulative Object Count	28
1.6.2.6	Average Method Time	28
1.6.2.7	Average Cumulative Time	29



1.6.2.8	Average Method Object.....	30
1.6.2.9	Average Cumulative Object Count.....	31
1.6.3	TEST CONCLUSIONS.....	33
1.7	APPENDIX A.....	35
1.7.1	JPROBE CONFIGURATION FILE.....	35
1.7.2	STRUTS-CONFIG.XML.....	37
1.8	RESOURCES.....	40
2	RCS – FTP Framework.....	41
2.1	PURPOSE.....	41
2.2	APPROACH.....	41
2.2.1	UNIT TESTING.....	41
2.2.2	PERFORMANCE PROFILING.....	41
2.3	BACKGROUND.....	41
2.4	42	
2.5	UNIT TESTING.....	42
2.5.1	SUMMARY.....	42
2.5.2	TEST HARNESS DESIGN.....	42
2.5.2.1	Environment.....	42
2.5.3	CONFIGURATION.....	44
2.5.3.1	struts-config.xml.....	44
2.5.3.2	properties files.....	45
2.5.3.3	Test Scenario.....	45
2.5.4	AUTOMATED TESTING CONDITIONS.....	48
2.5.5	MANUAL TESTING CONDITIONS.....	48
2.5.5.1	Cycle 1 – Normal.....	48
2.5.5.2	Cycle 2 – Connection Exception.....	51
2.5.5.3	Cycle 3 – Transfer Exception.....	52
2.6	PERFORMANCE PROFILING.....	55
2.6.1	SUMMARY.....	55



2.6.2	TEST HARNESS DESIGN	55
2.6.2.1	Environment	55
2.6.2.2	Configuration.....	57
2.6.2.3	Scenario setup.....	59
2.6.3	HEAP ANALYSIS	59
2.6.3.1	Instance Summary.....	60
2.6.4	PERFORMANCE ANALYSIS.....	62
2.6.4.1	Top ten FTP Framework related cumulative method time.....	64
3	RCS – XML Helper Framework	65
3.1	PURPOSE	65
3.2	APPROACH.....	65
3.3	SUMMARY	65
3.4	TEST HARNESS DESIGN	66
3.4.1	TESTING ENVIRONMENT.....	66
3.4.2	TESTING CRITERIA	66
3.4.3	TESTING CONFIGURATION.....	66
3.4.4	JPROBE CONFIGURATION FILE.....	66
3.4.5	UNIX SERVER SETTINGS	67
3.4.5.1	rules.properties:	67
3.4.5.2	queues.properties:.....	67
3.4.5.3	vhosts.properties:.....	67
3.4.6	WEBSHERE APPLICATION SERVER CONFIGURATION.....	68
3.4.6.1	Command line arguments:	68
3.4.6.2	Environment:.....	68
3.4.7	DIRECTORY STRUCTURE	69
3.5	TESTING SCENARIO	71
3.6	RESULTS AND ANALYSIS.....	72
3.7	HEAP SNAPSHOT (MEMORY USAGE)	72
3.7.1.1	Heap Graph Analysis.....	72



3.7.2	INSTANCE SUMMARY	73
3.7.2.1	DomTest.jsp.....	73
3.7.2.2	SaxTest.jsp.....	73
3.7.2.3	BindTest.jsp.....	73
3.8	PERFORMANCE SNAPSHOT (CODE EFFICIENCY).....	75
3.8.1	DOMTEST.JSP SCENARIO	75
3.8.1.1	Number of Calls.....	75
3.8.1.2	Method Time.....	76
3.8.1.3	Cumulative Time	77
3.8.1.4	Method Object Count.....	78
3.8.1.5	Cumulative Object Count	78
3.8.1.6	Average Method Time.....	79
3.8.1.7	Average Cumulative Time.....	80
3.8.1.8	Average Method Object.....	81
3.8.1.9	Average Cumulative Object Count.....	81
3.8.2	SAXTEST.JSP SCENARIO	82
3.8.2.1	Number of Calls.....	82
3.8.2.2	Method Time.....	83
3.8.2.3	Cumulative Time	84
3.8.2.4	Method Object Count.....	84
3.8.2.5	Cumulative Object Count	85
3.8.2.6	Average Method Time.....	86
3.8.2.7	Average Cumulative Time.....	87
3.8.2.8	Average Method Object.....	87
3.8.2.9	Average Cumulative Object Count.....	88
3.8.3	BINDTEST.JSP SCENARIO	88
3.8.3.1	Number of Calls.....	89
3.8.3.2	Method Time.....	89
3.8.3.3	Cumulative Time	91
3.8.3.4	Method Object Count.....	91
3.8.3.5	Cumulative Object Count	92
3.8.3.6	Average Method Time.....	92
3.8.3.7	Average Cumulative Time.....	93
3.8.3.8	Average Method Object.....	94
3.8.3.9	Average Cumulative Object Count.....	94
3.9	GENERAL PERFORMANCE TEST SUMMARY	95
3.10	APPENDIX A.....	96
3.10.1	JPROBE CONFIGURATION FILE	96



3.11	RESOURCES	99
4	RCS – Scheduler Framework.....	100
4.1	TEST HARNESS DESIGN	100
4.1.1	TESTING ENVIRONMENT.....	100
4.1.1.1	Testing Criteria.....	100
4.1.2	TESTING CONFIGURATION.....	100
4.1.2.1	JProbe Configuration File	100
4.1.2.2	UNIX Server Settings	101
4.1.3	WEBSHERE APPLICATION SERVER CONFIGURATION.....	101
4.1.3.1	Command line arguments:	102
4.1.3.2	Environment:.....	102
4.1.4	DIRECTORY STRUCTURE.....	103
4.2	TESTING SCENARIO	105
4.3	RESULTS AND ANALYSIS.....	106
4.4	HEAP SNAPSHOT (MEMORY USAGE)	106
4.4.1	HEAP GRAPH ANALYSIS.....	106
4.4.2	INSTANCE SUMMARY	107
4.4.2.1	onetime.jsp.....	107
4.4.2.2	recurs.jsp.....	107
4.5	PERFORMANCE SNAPSHOT (CODE EFFICIENCY).....	108
4.5.1	ONETIME.JSP SCENARIO	108
4.5.1.1	Number of Calls.....	108
4.5.1.2	Method Time.....	109
4.5.1.3	Cumulative Time	110
4.5.1.4	Method Object Count.....	110
4.5.1.5	Cumulative Object Count	111
4.5.1.6	Average Method Time.....	112
4.5.1.7	Average Cumulative Time.....	112
4.5.1.8	Average Method Object.....	113
4.5.1.9	Average Cumulative Object Count.....	114
4.5.2	RECURS.JSP SCENARIO.....	115
4.5.2.1	Number of Calls.....	115



4.5.2.2	Method Time.....	115
4.5.2.3	Cumulative Time	116
4.5.2.4	Method Object Count.....	117
4.5.2.5	Cumulative Object Count	117
4.5.2.6	Average Method Time.....	118
4.5.2.7	Average Cumulative Time.....	119
4.5.2.8	Average Method Object.....	119
4.5.2.9	Average Cumulative Object Count.....	120
4.6	GENERAL PERFORMANCE TEST SUMMARY	121
4.7	APPENDIX A.....	123
4.7.1	<i>JPROBE CONFIGURATION FILE</i>	123
4.8	RESOURCES	125
5	RCS – Session Framework.....	125
5.1	UNIT TEST REPORT	125
5.2	PURPOSE.....	125
5.3	APPROACH.....	125
5.4	BACKGROUND.....	126
5.5	TEST DESIGN	126
5.5.1	<i>TESTING ENVIRONMENT</i>	126
5.5.2	<i>TESTING CYCLES</i>	126
5.5.3	<i>TESTING CONFIGURATION</i>	127
5.5.3.1	UNIX Server Settings.....	127
5.5.3.2	WebSphere Application Server – Session Manager Configuration.....	127
5.5.3.3	Directory Structure.....	131
5.5.4	<i>TESTING CONDITIONS AND RESULTS</i>	133
5.5.4.1	Test Cycle 1	135
5.5.4.2	Test Cycle 2	136
5.5.4.3	Test Cycle 3	137
5.5.4.4	Test Cycle 4	138
5.5.4.5	Test Cycle 5	139
5.5.4.6	Test Cycle 6	140
5.6	PERFORMANCE ANALYSIS.....	141



5.6.1	PURPOSE.....	141
5.6.2	APPROACH.....	141
5.6.3	SUMMARY.....	141
5.6.4	TEST HARNESS DESIGN.....	142
5.6.4.1	Testing Environment.....	142
5.6.4.2	Testing Configuration.....	142
5.6.5	TESTING SCENARIO.....	145
5.6.6	RESULTS AND ANALYSIS.....	146
5.6.6.1	Heap Snapshot (Memory Usage).....	146
5.6.6.2	Performance Snapshot (Code Efficiency).....	149
5.6.6.3	General Performance Metrics.....	160
5.6.7	APPENDIX A.....	161
5.6.7.1	JProbe Configuration File.....	161
5.6.8	RESOURCES.....	164
6	RCS – Web Services (SOAP) Framework.....	164
6.1	PURPOSE.....	164
6.2	APPROACH.....	165
6.3	SUMMARY.....	165
6.4	TEST HARNESS DESIGN.....	166
6.4.1	TESTING ENVIRONMENT.....	166
6.4.1.1	Testing Criteria.....	166
6.4.1.2	JProbe Configuration File.....	166
6.4.1.3	WebSphere Application Server Configuration.....	169
6.4.1.4	Additional Required Components.....	169
6.5	TESTING SCENARIO.....	170
6.5.1	TEST PREPARATION.....	170
6.5.2	TEST SCENARIO.....	170
6.6	RESULTS AND ANALYSIS.....	170
6.6.1	HEAP SNAPSHOT (MEMORY USAGE).....	170



6.6.1.1	Heap Graph Analysis.....	170
6.6.1.2	Instance Summary.....	171
6.7	TEST CONCLUSIONS	172
6.8	RESOURCES	173
6.9	JAVABEANS ACTIVATION FRAMEWORK WEBSITE	173
6.10	APACHE XERCES WEBSITE.....	173
6.11	BEAN SCRIPTING FRAMEWORK WEBSITE.....	173
6.12	RHINO WEBSITE	173
6.13	HTTP://WWW.MOZILLA.ORG/RHINO/	173
7	RCS – Configuration Framework.....	174
7.1	PURPOSE.....	174
7.2	APPROACH.....	174
7.3	BACKGROUND.....	174
7.4	TESTING ENVIRONMENT.....	174
7.4.1	<i>XML FILES.....</i>	<i>175</i>
7.4.2	<i>DATABASE TABLES</i>	<i>175</i>
7.4.3	<i>WEBSHERE APPLICATION SERVER – CONFIGURATION FRAMEWORK CONFIGURATION</i>	<i>176</i>
7.5	AUTOMATED TESTING CONDITIONS.....	177
7.6	PERFORMANCE TESTING.....	183
7.6.1	<i>APPROACH.....</i>	<i>183</i>
7.6.2	<i>SUMMARY.....</i>	<i>183</i>
7.7	TEST HARNESS DESIGN	183
7.7.1	<i>TESTING ENVIRONMENT.....</i>	<i>183</i>
7.7.2	<i>TEST CONFIGURATON</i>	<i>183</i>
7.7.3	<i>WEBSHERE APPLICATION SERVER CONFIGURATION.....</i>	<i>184</i>



7.7.3.1	Command line arguments:	184
7.7.3.2	Environment:	184
7.8	TESTING SCENARIOS:	184
7.9	ANALYSIS:	184
7.9.1	MEMORY (HEAP) USAGE:	184
7.9.2	HEAP GRAPH ANALYSIS:	185
7.10	INSTANCE SUMMARY	185
7.11	GARBAGE COLLECTIONS	186
7.12	RESOURCES	187
7.12.1	GRNDS FRAMEWORK:	187
7.12.2	SUN JAVA WEBSITE:	187
8	RCS - JSP Custom Tag Library Framework	187
8.1	JSP CUSTOM TAG LIBRARY UNIT TEST REPORT	187
8.1.1		187
8.1.1.1	Purpose	187
8.1.1.2	Approach	187
8.1.1.3	Background	187
8.1.2	TEST DESIGN:	188
8.1.2.1	Testing Environment	188
8.1.2.2	Testing Configuration	188
8.1.3	TESTING CONDITIONS AND RESULTS	191
8.1.3.1	Test Cycle 1 - Jakarta Struts Bean Taglib	193
8.1.3.2	Test Cycle 2 - Jakarta Struts HTML Taglib	195
8.1.3.3	Test Cycle 3 - Jakarta Struts Logic Taglib	197
8.1.3.4	Test Cycle 4 - Jakarta Struts Template Taglib	198
8.1.3.5	Test Cycle 5 - Jakarta DateTime Taglib	199
8.1.3.6	Test Cycle 6 - Jakarta I18N Taglib	200
8.1.3.7	Test Cycle 7 - Jakarta Input Taglib	201
8.1.3.8	Test Cycle 8 - Logging Taglib	203
8.1.3.9	Test Cycle 9 - Jakarta Page Taglib	205
8.1.4	RESOURCES:	207
8.2	JSP CUSTOM TAG LIBRARY PERFORMANCE ANALYSIS	207



8.2.1	207	
8.2.2	PURPOSE.....	207
8.2.3	APPROACH.....	207
8.2.4	SUMMARY.....	207
8.2.5	TEST HARNESS DESIGN.....	209
8.2.5.1	Testing Environment.....	209
8.2.5.2	Testing Configuration.....	209
8.2.6	TESTING SCENARIO.....	212
8.2.7	RESULTS AND ANALYSIS.....	213
8.2.7.1	Heap Snapshot (Memory Usage).....	213
8.2.7.2	Performance Snapshot (Code Efficiency).....	215
8.2.7.3	General Performance Metrics.....	225
8.2.8	APPENDIX A.....	227
8.2.8.1	JProbe Configuration File.....	227
8.2.9	RESOURCES.....	230



1 RCS – Web Conversation Framework

1.1 Purpose

This section of the Performance Analysis Report documents the results of utilizing JProbe to analyze the ITA R3.0 Reusable Common Services (RCS) Web Conversation framework. This section provides an in-depth analysis of the results gathered from the JProbe and documents performance issues. The Detailed Design, User Guide, and the Performance Analysis documents for the Web Conversation framework will enable developers to quickly build applications using the Web Conversation framework within the ITA environment architecture.

1.2 Approach

To ensure program efficiency and to detect possible bottlenecks, ITA used JProbe to analyze the Web Conversation framework. JProbe is a performance-profiling tool and it was utilized to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming.

Two key groups of statistics are collected from the JProbe Profiler: The memory (heap) usage and the time spent on each method within the program (performance detail). This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow detailed information to be gathered on individual methods and define the calling relationship between methods.

1.3 Summary

This section of the report contains the performance test harness design, performance analysis, and resulting performance metrics for the Web Conversation framework. The example application used for testing maintains user registration and subscription information. The most commonly used Web Conversation classes and tag library methods (e.g. perform(), findForward(), and html: FormTag) were profiled using the example application. The actual results were compared against the results of how this framework is expected to function. Overall, this framework does not produce any loitering objects that remain in the heap after its useful life.

1.4 Test Harness Design

1.4.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance analysis is to identify loitering objects and time spent on each method relative to other methods within the Web Conversation Framework. The diagram below is representative of the environment configuration used for the performance analysis.

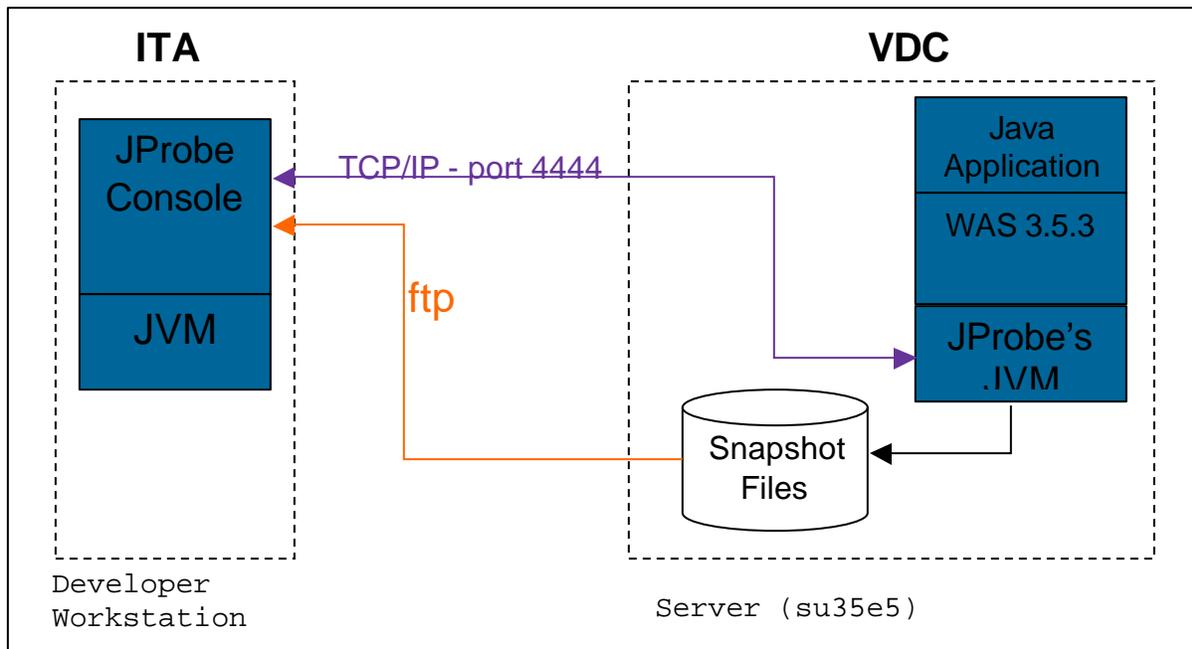


Figure 1: JProbe Application Analysis Environment

1.4.1.1 Testing Criteria

The ITA team has identified the most commonly used and most complicated Web Conversation framework classes where potential for code bottlenecks exist. Since the Web Conversation framework is an API, the example application packaged with the framework distribution was used as a test harness to profile and analyze the performance of the various methods.

The most commonly used methods identified by the ITA team that were tested as part of the example application included methods within:

Package	Class
org.apache.struts.action	Action
org.apache.struts.action	ActionError
org.apache.struts.action	ActionForm
org.apache.struts.action	ActionForward



Package	Class
org.apache.struts.action	ActionMapping
org.apache.struts.action	ActionServlet
org.apache.struts.util	MessageResources

Package	Tag
org.apache.struts.taglib.bean	MessageTag
org.apache.struts.taglib.bean	WriteTag
org.apache.struts.taglib.html	BaseTag
org.apache.struts.taglib.html	ButtonTag
org.apache.struts.taglib.html	CancelTag
org.apache.struts.taglib.html	CheckboxTag
org.apache.struts.taglib.html	ErrorsTag
org.apache.struts.taglib.html	FormTag
org.apache.struts.taglib.html	HiddenTag
org.apache.struts.taglib.html	HtmlTag
org.apache.struts.taglib.html	ImgTag
org.apache.struts.taglib.html	LinkTag
org.apache.struts.taglib.html	OptionsTag
org.apache.struts.taglib.html	PasswordTag
org.apache.struts.taglib.html	RadioTag
org.apache.struts.taglib.html	ResetTag
org.apache.struts.taglib.html	SubmitTag
org.apache.struts.taglib.html	TextareaTag
org.apache.struts.taglib.html	TextTag
org.apache.struts.taglib.logic	EqualTag
org.apache.struts.taglib.logic	IterateTag
org.apache.struts.taglib.logic	NotEqualTag

1.4.1.2 Testing Configuration

A new Web Sphere Application Server instance (JPROBE) was created to profile the Web Conversation example application using JProbe. The command line in the new Application Server references a JProbe configuration file specially created for this test. Additional settings and configurations were updated on the server and a struts-config.xml file was added to assist the Controller in determining where to direct an incoming request.

1.4.1.3 JProbe Configuration File

The JProbe configuration file has a file extension of .jpl. This file contains all of the settings that JProbe requires to profile an application, applet, or serverside component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options.



The example application test was conducted on the Solaris machine with the output directed to a remote Windows NT workstation. Performance and heap snapshots were taken before the Application Server was stopped. The configuration in the actual file used to conduct the test can be found in [Appendix A](#).

1.4.1.4 UNIX Server Settings

The Web Conversation framework is closely tied to the configuration of the WebSphere Application Server. Implementing this framework within a WebSphere environment removes the need to use a web.xml file that is read when the JSP container starts. The web.xml file would typically define which requests should be mapped to the ActionServlet. In a WebSphere environment, these resources and paths that would typically be defined in the web.xml file are spread across multiple Application Server and Web Application settings. Several WebSphere ApplicationServer *.properties files had to be updated to profile the example application. Refer to the Web Conversation Framework User Guide document for the *.properties files definition.

1.4.1.5 WebSphere Application Server Configuration

The WebSphere Command Line was configured with the JProbe configuration file used to ensure that the correct JVM was used. Two Environment Variables were added to the Application Server and two servlets were added to the Web Application.

1.4.1.5.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/06102002_test_struts.jpl -Xnoclassgc -  
Djava.compiler=NONE -ms128m -mx128m
```

1.4.1.5.2 Environment:

```
EXECUTE=YES  
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```

1.4.1.5.3 Action Servlet:

Servlet: action
Description: Struts Action Servlet
Servlet Class Name: org.apache.struts.action.ActionServlet
Servlet Web Path List: default_host/JPROBEWebApp/*.do
Init Parameters:

Init Param Name	Value
detail	2
debug	2
validate	true
config	/struts-config.xml
application	Resource

Debug Mode: False
Load at Startup: True



1.4.1.5.4 Database Servlet:

Servlet: database

Description: refers to database.xml

Servlet Class Name: org.apache.struts.webapp.example.DatabaseServlet

1.4.1.6 struts-config.xml File

In the Web Conversation framework, the struts-config.xml file is used to determine how to process incoming requests. A struts-config.xml file is required for each instance of the Application Server and can contain definitions for more than one Web Application. This analysis was conducted with the struts-config.xml file packaged with the example application. The struts-config.xml file was modified to look for the Document Type Definition (.dtd) file on the local server instead of on the Internet. The contents of the struts-config.xml file used in the test can be found in [Appendix A](#).

1.4.1.7 Additional Required Components

The following java archive files are required to run the example application:

- struts.jar
- jaxp.jar
- parser.jar

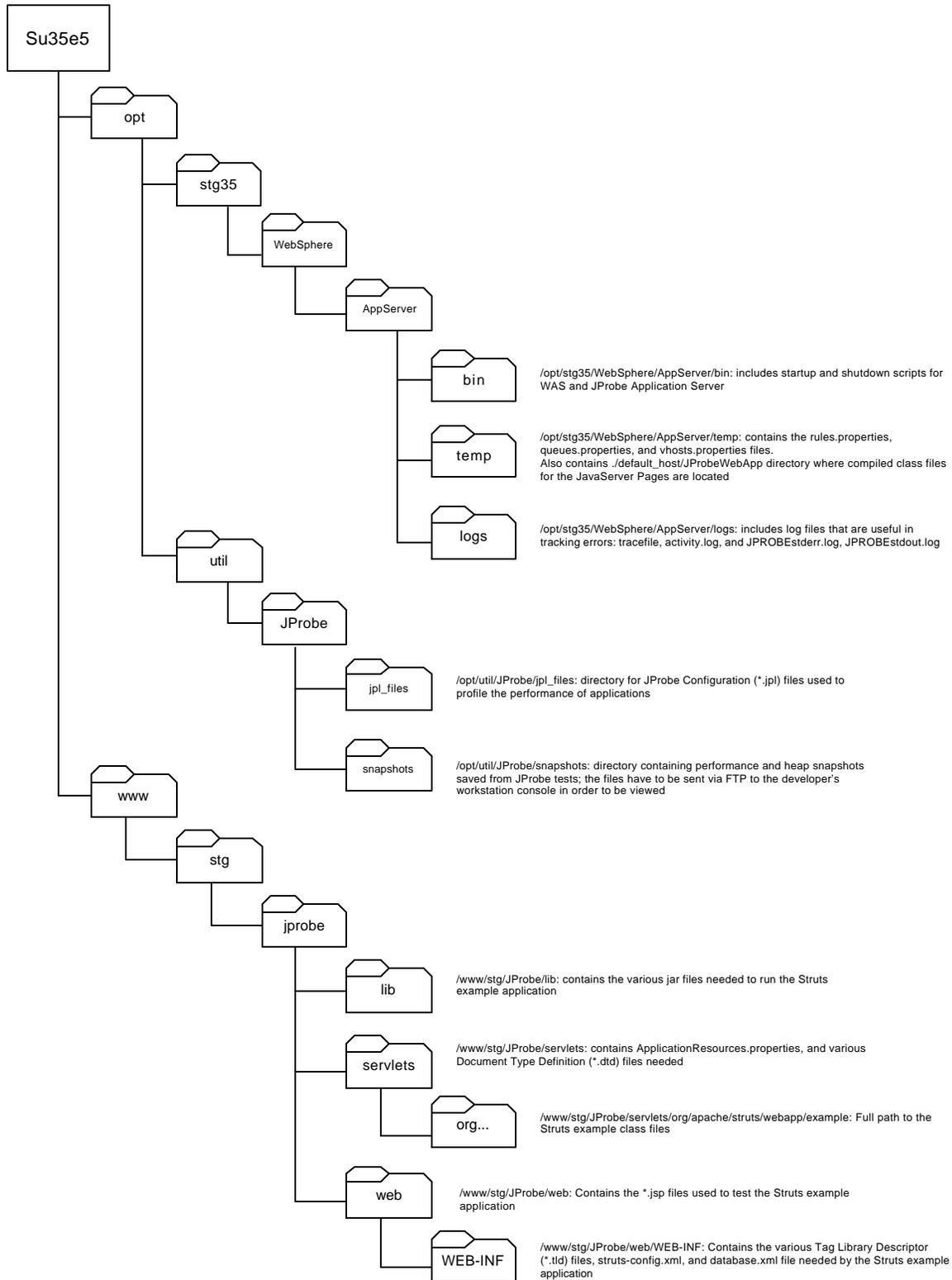
The following Document Type Definition files are required:

- struts-config_1_0.dtd
- web-app_2_2.dtd
- web-app_2_3.dtd

The following Tag Library Descriptors are required:

- struts.tld
- struts-bean.tld
- struts-form.tld
- struts-logic.tld
- struts-template.tld

1.4.1.8 Directory Structure





1.5 Testing 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.

1.5.1 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.

1.5.2 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"

1.6 Results and Analysis

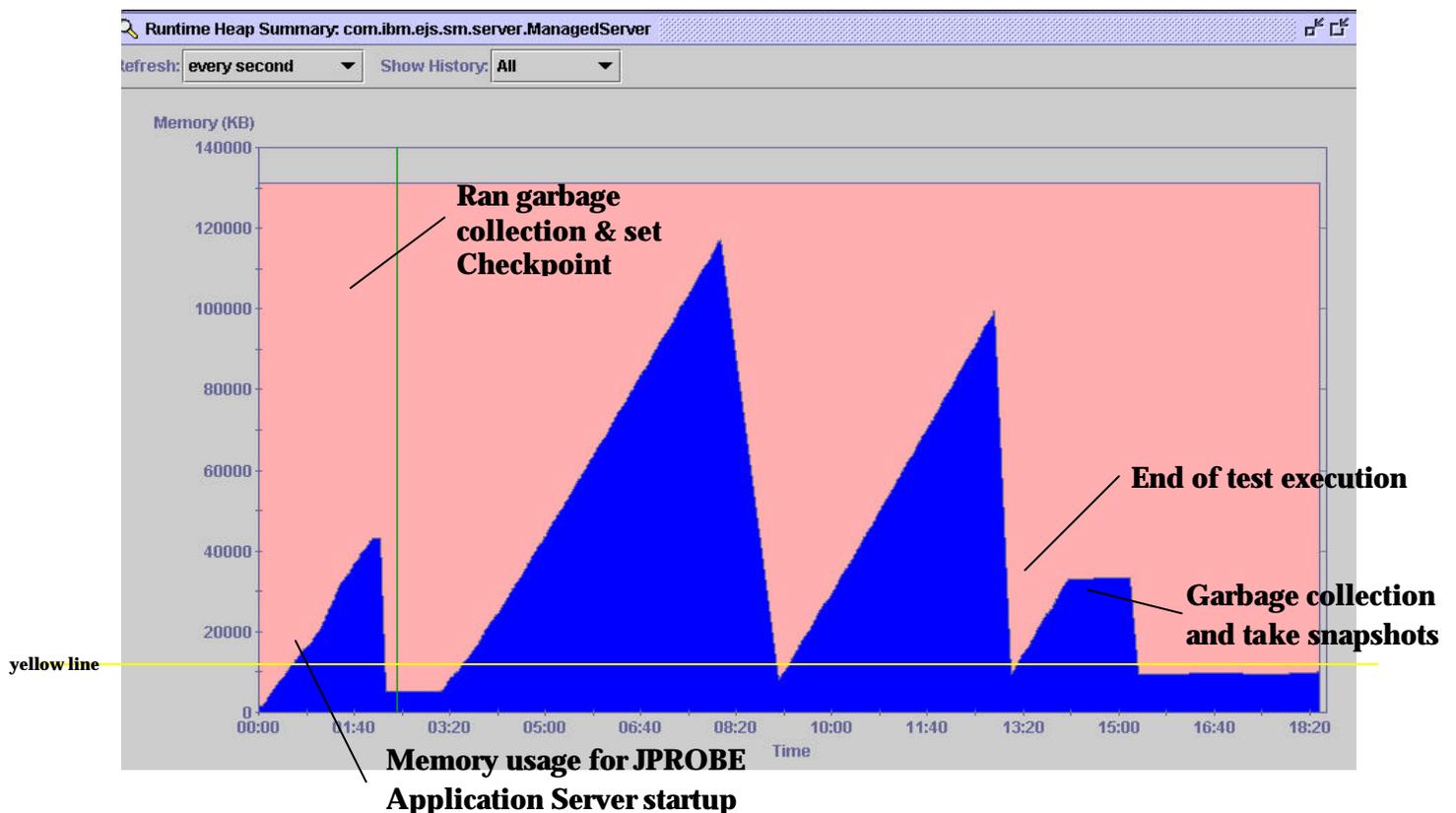
The JProbe Profiler with Memory Debugger application was used to trace both the memory usage and performance measurement of the example application. Two snapshots were taken: A heap snapshot and a performance snapshot.

1.6.1 Heap Snapshot (Memory Usage)

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

1.6.1.1 Heap Graph Analysis

The screenshot below is obtained from executing Scenario 2 twenty-five times. The spiked lines demonstrate that temporary objects are being allocated and garbage collected. The yellow horizontal line has been added to assist the reader in better gauging the depth of the trough. The yellow line, was used to determine 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 Scenario 2 is creating new user and subscription objects in the scenario. 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 instance summary to determine if these are loitering objects.

1.6.1.2 Instance Summary

The table below is a section of the Instance Summary result associated with running Scenario 2. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory those instances consume.

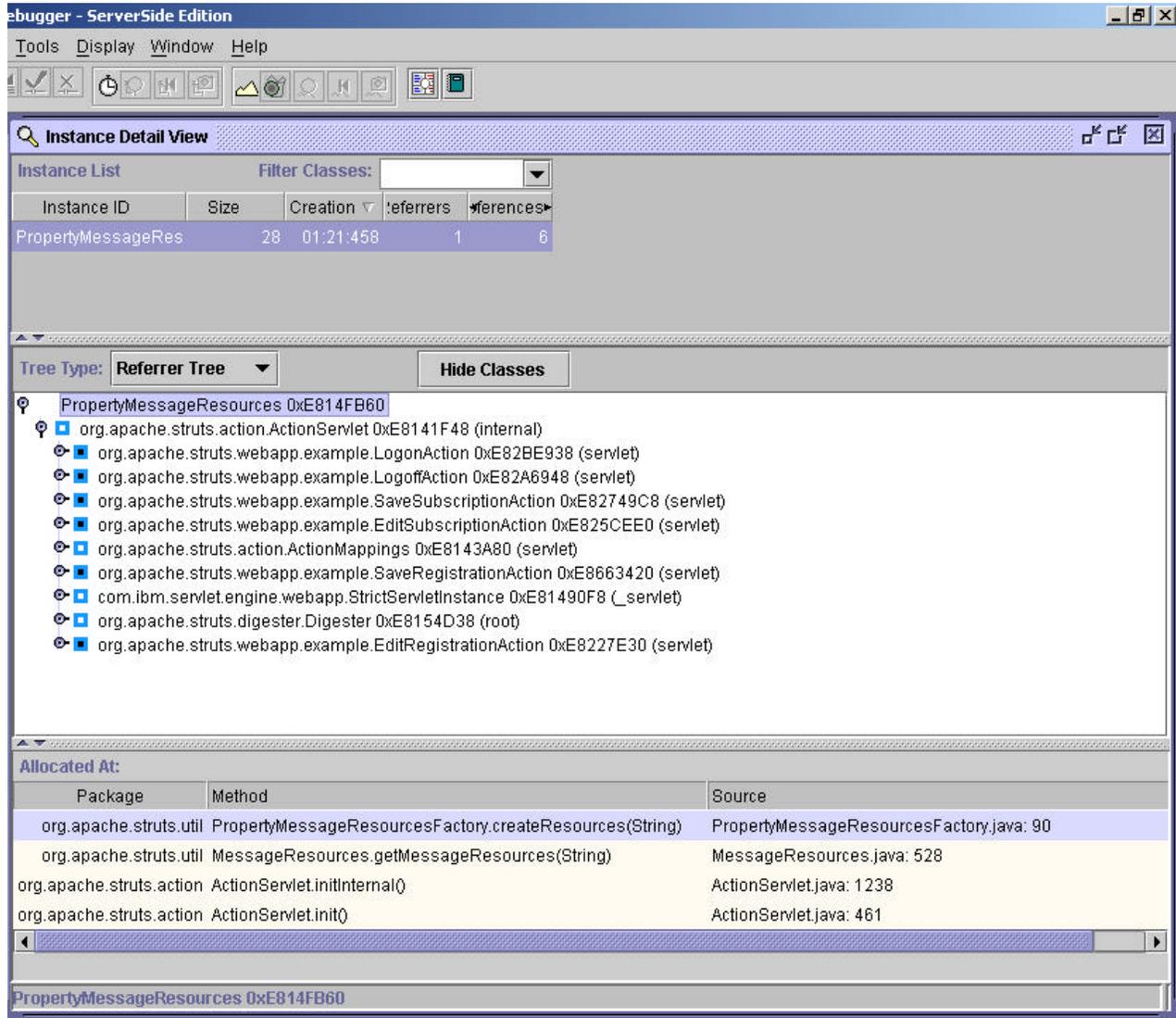
In the heap graph in the previous section, there is a green vertical line that shows where the Checkpoint was set. The Checkpoint tells JProbe to tag all subsequently created objects as “new.” The Count Change and Memory Change columns show data regarding new instances (created after the checkpoint) that are currently in the heap.

Package	Class	Count	Count Change	Memory	Memory Change
org.apache.struts.webapp.example	User	27 (18.1%)	+25	0.756 (18.3%)	+0.7
org.apache.struts.util	PropertyMessageResources	23 (15.4%)	+21	0.644 (15.6%)	+0.588
org.apache.struts.webapp.example	EditRegistrationAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004
org.apache.struts.webapp.example	EditSubscriptionAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004
org.apache.struts.webapp.example	LogoffAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004
org.apache.struts.webapp.example	LogonAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004
org.apache.struts.webapp.example	SaveRegistrationAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004
org.apache.struts.webapp.example	SaveSubscriptionAction	1 (0.7%)	+1	0.004 (0.1%)	+0.004

These results were gathered after the test scenario has finished executing and garbage collection has occurred. The results were filtered for ‘org.apache.struts.*’ since those are the classes this report is concerned with. The Count Change column was used to sort the data to determine which class had the most objects remaining in the heap after the scenario has been completed.

In the first row, the count change for the User class is +25; this number coincides with the number of times the scenario was executed. From the Package name column, it is possible to see that this class is part of the Struts example application and not actually part of the Struts 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.

The next class, `PropertyMessageResources`, is a class from the Web Conversation API. The `PropertyMessageResources` class is used to read message keys and their strings from the property resources file. It was possible to determine where this class was allocated (initiated) and any referrer objects by drilling down to the Instance Detail View.



The screenshot shows the JProbe ServerSide Edition interface. The main window is titled "Instance Detail View" and displays the following information:

Instance List

Instance ID	Size	Creation	Referrers	References
PropertyMessageRes	28	01:21:458	1	6

Tree Type: Referrer Tree

Hide Classes

PropertyMessageResources 0xE814FB60

- org.apache.struts.action.ActionServlet 0xE8141F48 (internal)
- org.apache.struts.webapp.example.LogonAction 0xE82BE938 (servlet)
- org.apache.struts.webapp.example.LogoffAction 0xE82A6948 (servlet)
- org.apache.struts.webapp.example.SaveSubscriptionAction 0xE82749C8 (servlet)
- org.apache.struts.webapp.example.EditSubscriptionAction 0xE825CEE0 (servlet)
- org.apache.struts.action.ActionMappings 0xE8143A80 (servlet)
- org.apache.struts.webapp.example.SaveRegistrationAction 0xE8663420 (servlet)
- com.ibm.servlet.engine.webapp.StrictServletInstance 0xE81490F8 (_servlet)
- org.apache.struts.digester.Digester 0xE8154D38 (root)
- org.apache.struts.webapp.example.EditRegistrationAction 0xE8227E30 (servlet)

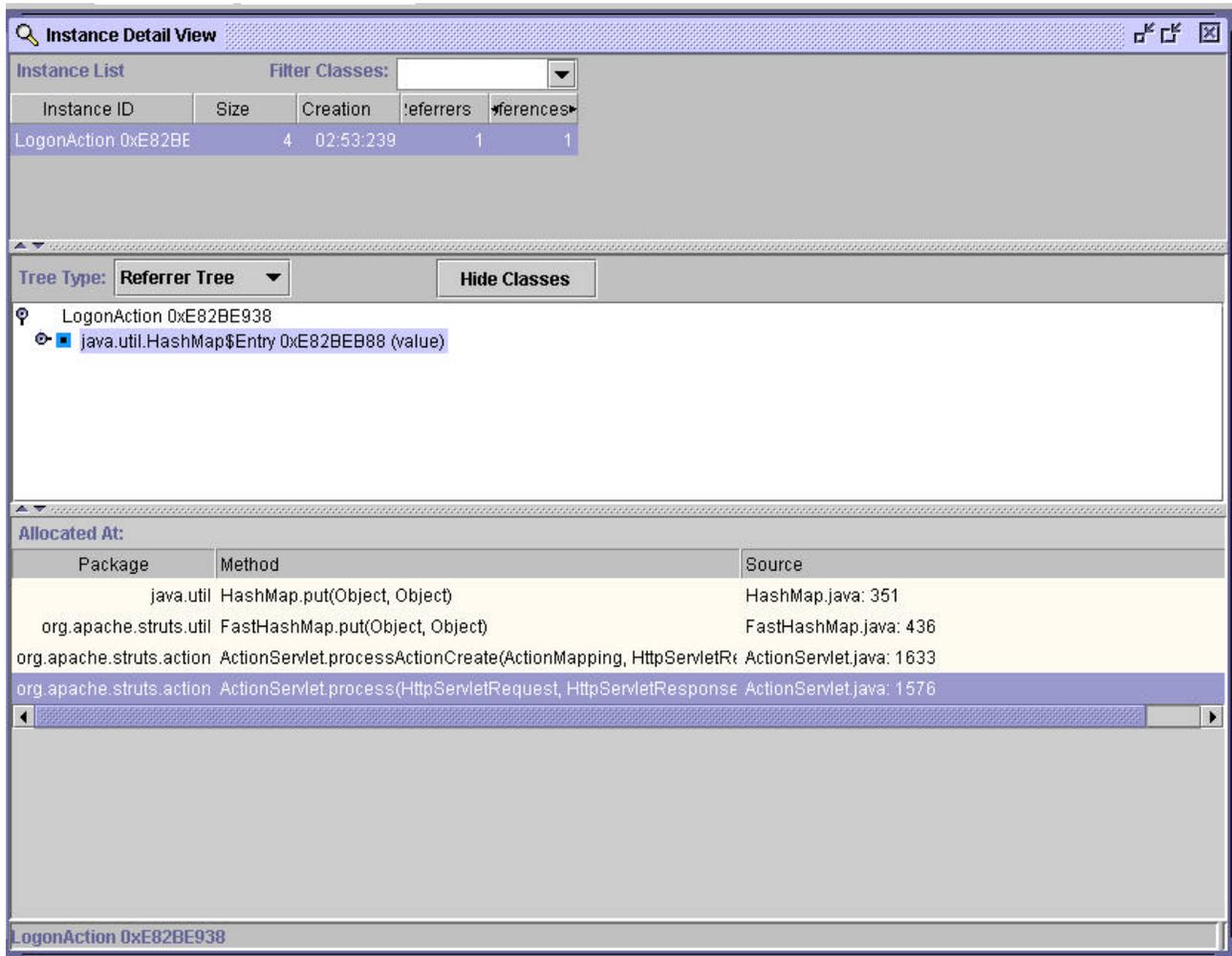
Allocated At:

Package	Method	Source
org.apache.struts.util	PropertyMessageResourcesFactory.createResources(String)	PropertyMessageResourcesFactory.java: 90
org.apache.struts.util	MessageResources.getMessageResources(String)	MessageResources.java: 528
org.apache.struts.action	ActionServlet.initInternal()	ActionServlet.java: 1238
org.apache.struts.action	ActionServlet.init()	ActionServlet.java: 461

PropertyMessageResources 0xE814FB60

The `ActionServlet` class, which is loaded upon startup of the Web Application, initiates this `PropertyMessageResources` instance. Since `ActionServlet` stores the `PropertyMessages` in the form of a `HashMap`, it will not release these messages from this cache during the life of this `ActionServlet`. 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.

Analyzing the remaining six classes that extend the Action class, it is possible to see that each has an extra instance remaining in the heap after the end of the performance analysis. Using the Instance Detail View for LogonAction below it is possible to trace the instance back to the ActionServlet class.



Instance Detail View

Instance List

Instance ID	Size	Creation	Referrers	References
LogonAction 0xE82BE	4	02:53:239	1	1

Tree Type: Referrer Tree

- LogonAction 0xE82BE938
 - java.util.HashMap\$Entry 0xE82BEB88 (value)

Allocated At:

Package	Method	Source
java.util	HashMap.put(Object, Object)	HashMap.java: 351
org.apache.struts.util	FastHashMap.put(Object, Object)	FastHashMap.java: 436
org.apache.struts.action	ActionServlet.processActionCreate(ActionMapping, HttpServletRequest)	ActionServlet.java: 1633
org.apache.struts.action	ActionServlet.process(HttpServletRequest, HttpServletResponse)	ActionServlet.java: 1576

LogonAction 0xE82BE938

Even though the different Action classes were called multiple times, only one instance was ever created. This leads to the conclusion that these objects are reachable and not loitering; therefore, these objects can still be reused.

By examining the source code, it is determined that once the Action instance is created, it is placed in a HashMap, which will not be 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. 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.

1.6.2 Performance Snapshot (Code Efficiency)

There are nine efficiency metrics that can be collected using JProbe – five basic metrics and four compound metrics. The basic metrics include: Number of calls, method time, cumulative time, method object count, and cumulative object count. The compound metrics are averages per number of calls, including: average method time, average cumulative time, average method object count, and average cumulative object count. Time is measured as elapsed time in milliseconds.

The following sections will describe each metric and display the top results for each measurement for the performance assessment of the Web Conversation framework. These metrics are basic indicators of process resource utilization. The detailed graphs associated with each method can be reviewed for unexpected activity or optimization opportunities.

All performance metric results were first filtered by `org.apache.struts.*` to obtain only the classes within the Web Conversation framework which is what the test is looking for. Then for each section, the results were sorted by the metric under investigation to obtain the top ten results for each metric.



1.6.2.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Package	Name	Calls	Source
org.apache.struts.util	FastHashMap.get(Object)	14,961	FastHashMap.java
org.apache.struts.util	PropertyUtils.getPropertyDescriptors(Object)	12,705	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getPropertyDescriptor(Object, String)	12,630	PropertyUtils.java
org.apache.struts.util	ResponseUtils.write(PageContext, String)	8,325	ResponseUtils.java
org.apache.struts.util	PropertyUtils.getAccessibleMethod(Method)	8,240	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getPropertySimpleProperty(Object, String)	5,375	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getReadMethod(PropertyDescriptor)	5,175	PropertyUtils.java
org.apache.struts.util	MessageResources.localeKey(Locale)	3,771	MessageResources.java
org.apache.struts.util	RequestUtils.message(PageContext, String, String, String, Object[])	3,675	RequestUtils.java
org.apache.struts.util	MessageResources.getMessage(Locale, String, Object[])	3,627	MessageResources.java

The chart above lists the top ten most frequently called methods. The classes from the org.apache.struts.util package were used the most. This is a part of the Web Conversation framework design in the separation of logic and presentation. The example JavaServer Pages used to analyze this framework heavily utilized resource files to obtain the strings to display to the end users.

1.6.2.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Package	Name	Method Time	Source
org.apache.struts.util	PropertyUtils.getPropertyDescriptor(Object, String)	785.63 (12.8%)	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getPropertySimpleProperty(Object, String)	378.39 (6.2%)	PropertyUtils.java
org.apache.struts.util	MessageResources.messageKey(Locale, String)	267.11 (4.4%)	MessageResources.java
org.apache.struts.util	ResponseUtils.filter(String)	261.27 (4.3%)	ResponseUtils.java



Package	Name	Method Time	Source
org.apache.struts.util	PropertyUtils.setSimpleProperty(Object, String, Object)	220.33 (3.6%)	PropertyUtils.java
org.apache.struts.taglib.html	BaseFieldTag.doStartTag()	199.35 (3.3%)	BaseFieldTag.java
org.apache.struts.taglib.bean	MessageTag.doStartTag()	197.44 (3.2%)	MessageTag.java
org.apache.struts.util	PropertyUtils.getAccessibleMethod(Method)	175.19 (2.9%)	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getPropertyDescriptors(Object)	171.29 (2.8%)	PropertyUtils.java
org.apache.struts.taglib.html	BaseHandlerTag.prepareEventHandlers()	170.40 (2.8%)	BaseHandlerTag.java

From the results of the top ten methods with the highest method times, it is possible to see that methods from the PropertyUtils class have the largest impact on the overall program execution time. Since this is a class that is intrinsic to the Web Conversation framework and should not be changed by the ITA team, developers should be aware of the amount of time it takes to execute methods from this class and be prepared for any impact that it may have to their program.

1.6.2.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Package	Name	Cumulative Time	Source
org.apache.struts.action	ActionServlet.process(HttpServletRequest, HttpServletResponse)	2,225.03 (36.3%)	ActionServlet.java
org.apache.struts.action	ActionServlet.doPost(HttpServletRequest, HttpServletResponse)	1,561.41 (25.5%)	ActionServlet.java
org.apache.struts.action	ActionServlet.processActionPerform(Action, ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	1,096.33 (17.9%)	ActionServlet.java
org.apache.struts.util	PropertyUtils.getPropertyDescriptor(Object, String)	1,080.64 (17.6%)	PropertyUtils.java
org.apache.struts.util	PropertyUtils.getSimpleProperty(Object, String)	993.60 (16.2%)	PropertyUtils.java
org.apache.struts.util	PropertyUtils.copyProperties(Object, Object)	854.67 (13.9%)	PropertyUtils.java
org.apache.struts.taglib.html	BaseFieldTag.doStartTag()	790.89 (12.9%)	BaseFieldTag.java



Package	Name	Cumulative Time	Source
org.apache.struts.taglib.bean	MessageTag.doStartTag()	726.13 (11.9%)	MessageTag.java
org.apache.struts.action	ActionServlet.processPopulate(ActionForm, ActionMapping, HttpServletRequest)	720.24 (11.8%)	ActionServlet.java
org.apache.struts.util	PropertyUtils.getProperty(Object, String)	719.52 (11.7%)	PropertyUtils.java

The example application spent the most time executing methods from the org.apache.struts.action.ActionServlet class and the org.apache.struts.util.PropertyUtils class. The results indicate that these classes lie in the critical path and have an impact on the system performance.

1.6.2.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Package	Name	Method Objects	Source
org.apache.struts.util	PropertyUtils.getSimpleProperty(Object, String)	5,829 (14.8%)	PropertyUtils.java
org.apache.struts.util	ResponseUtils.filter(String)	3,900 (9.9%)	ResponseUtils.java
org.apache.struts.util	MessageResources.messageKey(Locale, String)	3,629 (9.2%)	MessageResources.java
org.apache.struts.taglib.bean	MessageTag.doStartTag()	3,602 (9.1%)	MessageTag.java
org.apache.struts.util	PropertyUtils.setSimpleProperty(Object, String, Object)	3,344 (8.5%)	PropertyUtils.java
org.apache.struts.taglib.html	BaseHandlerTag.prepareEventHandlers()	2,650 (6.7%)	BaseHandlerTag.java
org.apache.struts.taglib.html	BaseHandlerTag.prepareStyles()	2,650 (6.7%)	BaseHandlerTag.java
org.apache.struts.util	BeanUtils.populate(Object, Map)	1,423 (3.6%)	BeanUtils.java
org.apache.struts.taglib.html	BaseFieldTag.doStartTag()	1,218 (3.1%)	BaseFieldTag.java
org.apache.struts.action	ActionServlet.processActionForm(ActionMapping, HttpServletRequest)	1,135 (2.9%)	ActionServlet.java

The above results indicate that the methods from the org.apache.struts.util package allocate the greatest number of objects. The main culprits are the methods responsible for obtaining the messages located in the properties file. Since the properties files are integral to the operation of the framework, developers need to be aware that loading the properties file utilizes system resources.



1.6.2.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Package	Name	Cumulative Objects	Source
org.apache.struts.action	ActionServlet.process(HttpServletRequest, HttpServletResponse)	10,968 (27.8%)	ActionServlet.java
org.apache.struts.action	ActionServlet.doPost(HttpServletRequest, HttpServletResponse)	7,623 (19.3%)	ActionServlet.java
org.apache.struts.taglib.bean	MessageTag.doStartTag()	7,434 (18.8%)	MessageTag.java
org.apache.struts.taglib.html	BaseFieldTag.doStartTag()	7,070 (17.9%)	BaseFieldTag.java
org.apache.struts.util	PropertyUtils.getSimpleProperty(Object, String)	5,931 (15.0%)	PropertyUtils.java
org.apache.struts.action	ActionServlet.processActionPerform(Action, ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	5,117 (13.0%)	ActionServlet.java
org.apache.struts.util	RequestUtils.message(PageContext, String, String, String, Object[])	3,955 (10.0%)	RequestUtils.java
org.apache.struts.util	ResponseUtils.filter(String)	3,900 (9.9%)	ResponseUtils.java
org.apache.struts.util	MessageResources.getMessage(Locale, String, Object[])	3,898 (9.9%)	MessageResources.java
org.apache.struts.util	PropertyUtils.getNestedProperty(Object, String)	3,752 (9.5%)	PropertyUtils.java

The ActionServlet.process() and ActionServlet.doPost() methods create the most objects themselves or through their descendants. These are expected since doPost() calls the process() method, which processes an HTTP request and performs the bulk of the operations.

1.6.2.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls.

Package	Name	Average Method Time	Source
org.apache.struts.action	ActionServlet.initDigester(int)	20.16 (0.3%)	ActionServlet.java
org.apache.struts.action	ActionServlet.<init>()	9.78 (0.2%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initInternal()	3.80 (0.1%)	ActionServlet.java
org.apache.struts.digester	Digester.getParser()	3.07 (0.1%)	Digester.java



Package	Name	Average Method Time	Source
org.apache.struts.action	ActionServlet.initDataSources()	2.93 (0.0%)	ActionServlet.java
org.apache.struts.digester	Digester.addCallMethod(String, String, int)	2.85 (0.0%)	Digester.java
org.apache.struts.util	ConvertUtils.<clinit>()	2.81 (0.0%)	ConvertUtils.java
org.apache.struts.taglib.html	SubmitTag.<clinit>()	2.36 (0.0%)	SubmitTag.java
org.apache.struts.digester	Digester.addSetTop(String, String)	1.61 (0.0%)	Digester.java
org.apache.struts.taglib.html	LinkTag.<clinit>()	1.55 (0.0%)	LinkTag.java

The above result shows that the ActionServlet initialization methods take the longest time to execute on average. This will not interfere with the overall performance of the Web Conversation framework as the ActionServlet initialization is performed only once on startup of the Application Server.

1.6.2.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls.

Package	Name	Average Cumulative Time	Source
org.apache.struts.action	ActionServlet.init()	189.90 (3.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initMapping()	163.34 (2.7%)	ActionServlet.java
org.apache.struts.digester	Digester.parse(InputStream)	49.40 (0.8%)	Digester.java
org.apache.struts.action	ActionServlet.initDigester(int)	30.60 (0.5%)	ActionServlet.java
org.apache.struts.webapp.example	DatabaseServlet.init()	21.92 (0.4%)	DatabaseServlet.java
org.apache.struts.webapp.example	DatabaseServlet.load()	21.29 (0.3%)	DatabaseServlet.java
org.apache.struts.action	ActionServlet.<init>()	10.37 (0.2%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initInternal()	8.45 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initServlet()	6.74 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initApplication()	6.64 (0.1%)	ActionServlet.java



Again, the ActionServlet initialization methods, together with their descendants, took the longest time to execute on average. These results will not impact the system performance once the ActionServlet has been started.

1.6.2.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Identifies the highest number of objects created for the least number of calls.

Package	Name	Avg. Method Object	Source
org.apache.struts.action	ActionServlet.initDigester(int)	56 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.<init>()	44 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initServlet()	20 (0.1%)	ActionServlet.java
org.apache.struts.util	ConvertUtils.<clinit>()	17 (0.0%)	ConvertUtils.java
org.apache.struts.action	ActionServlet.initMapping()	16 (0.0%)	ActionServlet.java
org.apache.struts.webapp.example	DatabaseServlet.load()	15 (0.0%)	DatabaseServlet.java
org.apache.struts.action	ActionServlet.initApplication()	12 (0.0%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initOther()	12 (0.0%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initUpload()	8 (0.0%)	ActionServlet.java
org.apache.struts.digester	Digester.resolveEntity(String, String)	7 (0.0%)	Digester.java

The ActionServlet initialization methods all created the most objects for only one call to that ActionServlet method. These findings will not affect the overall performance of the Web Conversation framework as it only creates these objects on initialization of the ActionServlet when the Application Server or Web Application is first started.



1.6.2.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls.

Package	Name	Average Method Object	Source
org.apache.struts.action	ActionServlet.init()	809 (2.0%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initMapping()	676 (1.7%)	ActionServlet.java
org.apache.struts.digester	Digester.parse(InputStream)	208 (0.5%)	Digester.java
org.apache.struts.webapp.example	DatabaseServlet.init()	112 (0.3%)	DatabaseServlet.java
org.apache.struts.webapp.example	DatabaseServlet.load()	106 (0.3%)	DatabaseServlet.java
org.apache.struts.action	ActionServlet.initDigester(int)	105 (0.3%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initApplication()	55 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.<init>()	52 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.initServlet()	45 (0.1%)	ActionServlet.java
org.apache.struts.action	ActionServlet.doPost(HttpServletRequest, HttpServletResponse)	30 (0.1%)	ActionServlet.java

Again, the ActionServlet initialization methods created the greatest number of cumulative objects per number of calls. These objects will not affect the overall performance of the Web Conversation framework.



1.6.3 Test Conclusions

From analyzing the results of the performance analysis of the example application packaged with the Struts distribution, it is concluded that the Web Conversation framework does not produce any loitering objects. Developers will need to keep in mind that 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. Developers will also have to be cautious about the use of Message Resources as those consume the most memory while utilizing this framework.



1.7 Appendix A

1.7.1 JProbe Configuration File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
      <classpath/>
    </application>
    <applet
      working_dir=""
      source_dir=""
      htmlfile=""
      main_package="">
      <classpath>
        <classpath.path location="%CLASSPATH%"/>
      </classpath>
    </applet>
    <serverside
      suggested_filters=""
      id="Other server"
      server_dir="/opt/stg35/WebSphere/AppServer"
      prepend_to_vm_args=""
      source_dir=""
      classname="com.ibm.ejs.sm.util.process.Nanny"
      main_package="org.apache.struts"
      exclude_server_classes="true"
      args=""
      working_dir="/opt/stg35/WebSphere/AppServer/servlets"
      prepend_to_classpath="">
      <classpath>
        <classpath.path location="%CLASSPATH%"/>
      </classpath>
    </serverside>
  </program>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.74:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
      timing="elapsed"
      track_natives="true"
      final_snapshot="true"
      granularity="method">
```



```
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask="*"
  time="ignore"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask="org.apache.struts.*"
  time="track"
  granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask="*/>
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask=".*"/>
</threadalyzer>
<coverage
  record_from_start="true"
  final_snapshot="true"
  granularity="line">
```



```
                <coverage.filter
                    visibility="invisible"
                    methodmask="*"
                    enabled="true"
                    classmask="*" />
                <coverage.filter
                    visibility="visible"
                    methodmask="*"
                    enabled="true"
                    classmask=".*" />
            </coverage>
        </analysis>
</jpl>
```

1.7.2 struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config SYSTEM
    "/www/stg/jprobe/servlets/struts-config_1_0.dtd">
<!--
This is the Struts configuration file for the example application,
using the proposed new syntax.

NOTE: You would only flesh out the details in the "form-bean"
declarations if you had a generator tool that used them to create
the corresponding Java classes for you. Otherwise, you would
need only the "form-bean" element itself, with the corresponding
"name" and "type" attributes.
-->

<struts-config>

<!-- ===== Form Bean Definitions ===== -->
<form-beans>

    <!-- Logon form bean -->
    <form-bean name="logonForm"
        type="org.apache.struts.webapp.example.LogonForm"/>

    <!-- Registration form bean -->
    <form-bean name="registrationForm"
        type="org.apache.struts.webapp.example.RegistrationForm"/>

    <!-- Subscription form bean -->
    <form-bean name="subscriptionForm"
        type="org.apache.struts.webapp.example.SubscriptionForm"/>

</form-beans>

<!-- ===== Global Forward Definitions ===== -->
<global-forwards>
    <forward name="logoff" path="/logoff.do"/>
    <forward name="logon" path="/logon.jsp"/>
</global-forwards>
```



```
<forward name="success"      path="/mainMenu.jsp"/>
</global-forwards>

<!-- ===== Action Mapping Definitions ===== -->
<action-mappings>

  <!-- Edit user registration -->
  <action  path="/editRegistration"
           type="org.apache.struts.webapp.example.EditRegistrationAction"
           name="registrationForm"
           scope="request"
           validate="false">
    <forward name="success"      path="/registration.jsp"/>
  </action>

  <!-- Edit mail subscription -->
  <action  path="/editSubscription"
           type="org.apache.struts.webapp.example.EditSubscriptionAction"
           name="subscriptionForm"
           scope="request"
           validate="false">
    <forward name="failure"      path="/mainMenu.jsp"/>
    <forward name="success"      path="/subscription.jsp"/>
  </action>

  <!-- Process a user logoff -->
  <action  path="/logoff"
           type="org.apache.struts.webapp.example.LogoffAction">
    <forward name="success"      path="/index.jsp"/>
  </action>

  <!-- Process a user logon -->
  <action  path="/logon"
           type="org.apache.struts.webapp.example.LogonAction"
           name="logonForm"
           scope="request"
           input="/logon.jsp">
  </action>

  <!-- Save user registration -->
  <action  path="/saveRegistration"
           type="org.apache.struts.webapp.example.SaveRegistrationAction"
           name="registrationForm"
           scope="request"
           input="/registration.jsp"/>

  <!-- Save mail subscription -->
  <action  path="/saveSubscription"
           type="org.apache.struts.webapp.example.SaveSubscriptionAction"
           name="subscriptionForm"
           scope="request"
           input="/subscription.jsp">
    <forward name="success"      path="/editRegistration.do?action=Edit"/>
  </action>

  <!-- Display the "walking tour" documentation -->
```



```
<action path="/tour"
  forward="/tour.htm">
</action>

<!-- The standard administrative actions available with Struts -->
<!-- These would be either omitted or protected by security -->
<!-- in a real application deployment -->
<action path="/admin/addFormBean"
  type="org.apache.struts.actions.AddFormBeanAction"/>
<action path="/admin/addForward"
  type="org.apache.struts.actions.AddForwardAction"/>
<action path="/admin/addMapping"
  type="org.apache.struts.actions.AddMappingAction"/>
<action path="/admin/reload"
  type="org.apache.struts.actions.ReloadAction"/>
<action path="/admin/removeFormBean"
  type="org.apache.struts.actions.RemoveFormBeanAction"/>
<action path="/admin/removeForward"
  type="org.apache.struts.actions.RemoveForwardAction"/>
<action path="/admin/removeMapping"
  type="org.apache.struts.actions.RemoveMappingAction"/>

</action-mappings>

</struts-config>
```



1.8 Resources

- Struts Homepage
 - <http://jakarta.apache.org/struts>
- Struts Documentation - Apache Struts Framework (Version 1.0)
 - <http://jakarta.apache.org/struts/api-1.0/index.html>
- Struts, an open-source MVC implementation
 - <http://www-106.ibm.com/developerworks/ibm/library/j-struts/>
- Strut Your Stuff with JSP Tags: Use and extend the open source Struts JSP tag library
 - <http://www.javaworld.com/javaworld/jw-12-2000/jw-1201-struts.html>
- Introduction to Jakarta Struts Framework – Parts 1 – 3
 - http://www.onjava.com/lpt/a//onjava/2001/09/11/jsp_servlets.html
 - <http://www.onjava.com/pub/a/onjava/2001/10/31/struts2.html>
 - http://www.onjava.com/pub/a/onjava/2001/11/14/jsp_servlets.html
- Building a Web Application: Strut by Strut
 - <http://husted.com/about/scaffolding/strutByStrut.htm>
- Java Developer's Journal
 - <http://www.sys-con.com/java/article.cfm?id=1175>

Lennart Jörelid. J2EE FrontEnd Technologies: A Programmer's Guide to Servlets, JavaServer Pages, and Enterprise JavaBeans. New York. Apress, 2002.



2 RCS – FTP Framework

2.1 Purpose

The RCS FTP Framework Build & Test Report documents testing configuration, unit testing and performance profiling of Integrated Technical Architecture (ITA) Reusable Common Services (RCS) FTP Framework. The report provides readers with detailed information on ITA's testing approach, testing conditions for unit testing and analysis on performance profiling. The intended audience is developers and testers who have interests in test conditions and profile of the framework. For applying the framework, please refer to RCS FTP Framework User Guide.

2.2 Approach

The FTP Framework first went through unit testing to ensure proper functioning in both API and web application aspects. The framework was then profiled to show its memory usage and performance.

2.2.1 Unit Testing

Unit testing of the framework was done in two separate approaches since the framework can be applied in two different ways. The unit testing of the API was done in an automated fashion using JUnit automated testing tool. As for the web interface, the framework was tested manually.

2.2.2 Performance Profiling

FTP Framework was performance profiled using JProbe. The profile captures heap utilization and application efficiency. By profiling, loitering objects that cause memory leak can be identified and performance bottleneck can be located. Profiling also offers an overall look of an application and provides developer with performance matrixes as references in his or hers development work.

2.3 Background

In the past, FSA applications had looked into a file transfer solution for batched files to be transferred between systems. The need for an enterprise wide FTP service became apparent during ITA Release 2 Strategic Assessment. The ITA responded to the need and came up with a Java based FTP solution that can be run in a WebSphere application Server (WAS) environment.

The FTP Framework provides developers with an API and a generic graphical user interface. Developers can use FtpClient as the interface to the framework and custom build a FTP client. The graphical user interface provides a working example of using



RCS Web Conversation Framework for building JSP and Servlets that uses FtpClient as the back end.

In particular the FTP Framework offers the following features:

- API for building customized FTP client.
- JSP and Servlet based front-end graphical user interface.
- Secure Socket Layer data transfer option.
- Active and passive data transfer modes.

2.4 Unit Testing

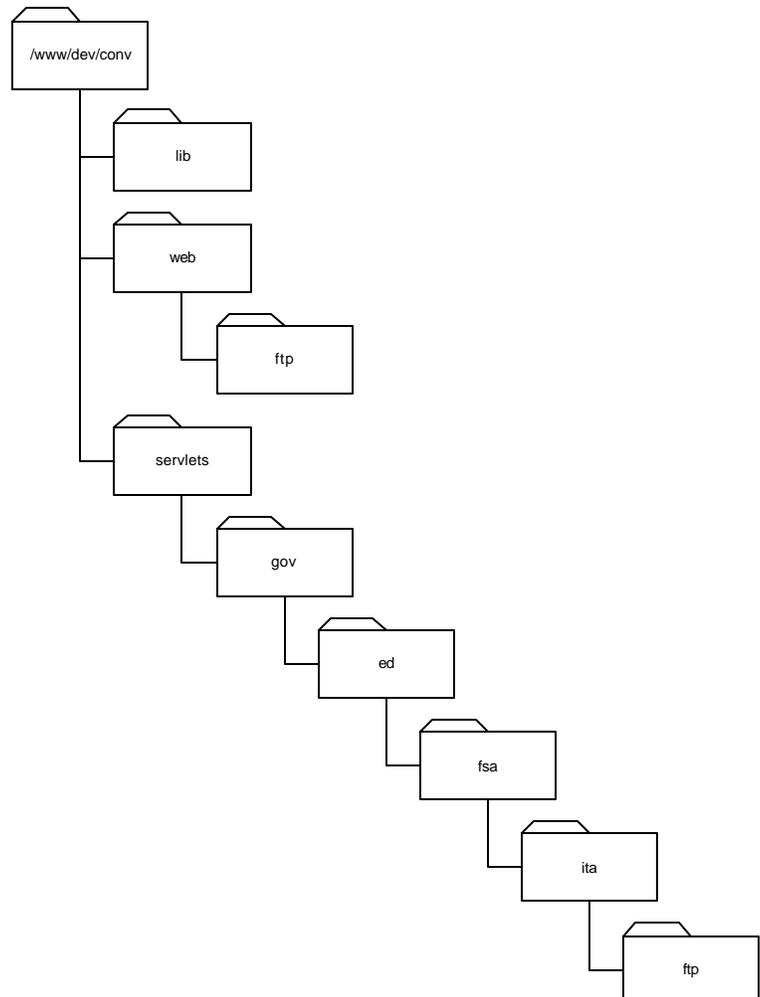
2.4.1 Summary

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

2.4.2 Test Harness Design

2.4.2.1 Environment

For automated unit testing, the code was tested on local development machine. The main reason for this setup was for testing SSL connection. For SSL connection, SSL capable FTP server needs to be installed. The current FSA development servers do not have this type of FTP server available for development and testing purposes. The development environment is shared across FSA application development teams. Installing a SSL FTP server on the shared environment might affect other teams negatively. Further more, software installation needs to be reviewed by CSC and the testing schedule could not afford the lengthy process. Thus, A demo version of the SSL FTP server software “Surge FTP” was installed on the local development machine and used during the automated testing.





ITA Release 3.0 Build & Test Report

FTP Framework application was placed under /www/dev/conv/ directory on su35e5 development application server. The file structure is shown on the right. Two major branches were setup for housing JSP and Servlets. Under ./web/ftp directory, all the front end user interface JSP pages were kept. As for FTP Framework classes, they were served out of the ./servlets/gov/ed/fsa/ita/ftp directory. RCS Web Conversation Framework configuration file, struts-config.xml was placed in the ../web/WEB-INF directory. The application properties files were in the ./servlets directory.



2.4.3 Configuration

2.4.3.1 struts-config.xml

The FTP Framework uses RCS Web Conversation Framework for its front end user interface. Web Conversation Framework uses a configuration file that serves as the switchboard for the application and directs HTTP request and response traffic accordingly. The configuration file is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config SYSTEM
"/www/stg/jprobe/servlets/struts-config_1_0.dtd">

<struts-config>

<!-- ===== Form Bean Definitions ===== -->
<form-beans>

<!-- FTP Connection Form Bean -->
<form-bean name="connForm" type="gov.ed.fsa.ita.ftp.FTPConnectForm"/>

<!-- FTP File Transfer Form Bean -->
<form-bean name="fileForm" type="gov.ed.fsa.ita.ftp.FTPMoveFileForm"/>

</form-beans>

<!-- ===== Global Forward Definitions ===== -->
<global-forwards>

</global-forwards>

<!-- ===== Action Mapping Definitions ===== -->
<action-mappings>

<!-- FTP -->
<action
  path="/login"
  type="gov.ed.fsa.ita.ftp.FTPConnectAction"
  name="connForm"
  validate="true"
  input="/ftp/FTPConnection.jsp">
  <forward name="moveFiles" path="/ftp/FTPMoveFile.jsp"/>
  <forward name="fail" path="/ftp/fail.jsp"/>
</action>

<action
  path="/move"
  type="gov.ed.fsa.ita.ftp.FTPMoveFileAction"
  name="fileForm"
  validate="true"
  input="/ftp/FTPMoveFile.jsp">
  <forward name="fail" path="/ftp/fail.jsp"/>
  <forward name="movePage" path="/ftp/FTPMoveFile.jsp"/>
</action>
```



```
<forward name="quitPage" path="/ftp/quit.jsp"/>
</action>

</action-mappings>

</struts-config>
```

2.4.3.2 properties files

Two properties files were used in the FTP Framework. Resource.properties file was for RCS Web Conversation Framework error messages and errorMessages.properties was for RCS Exception Handling Framework error messages. RCS Exception Handling Framework was used in Servlets for error catching purpose.

Resource.properties

```
error.hostname.required=<font color="red">Host name is required</font>
error.username.required=<font color="red">User name is required</font>
error.password.required=<font color="red">Password is required</font>
error.clientChangeDir.null=<font color="red">Directory location is required</font>
error.client.notDir=Selection was not a valid Directory
error.serverChangeDir.null=<font color="red">Directory location is required</font>
error.server.netDir=Selection was not a valid Directory
error.get.file.notSelected=<font color="red">Please select a server side file</font>
error.put.file.notSelected=<font color="red">Please select a client side file</font>
```

errorMessages.properties

```
# RCS Exception Handling Messages
# This file contains mapping information from error codes to error messages

# 1000-1100 Errors in the FTP Framework:
msg1001=Could not find host
msg1002=Could not create socket
msg1003=Could not create server socket
msg1004=Could not create input/output streams
msg1005=Could not set socket timeout
msg1006=Unexpected response from FTP server read
msg1007=Could not read response from input stream
msg1008=Could not close streams
msg1009=Could not close socket
```

2.4.3.3 Test Scenario

Two test scenarios were used for both automated and manual testing. Automated testing was done in 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. The automated testing script is provided below:



```
package gov.ed.fsa.ita.ftp;

/**
 * <p>Title: RCS FTP Framework</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Accenture</p>
 * @author Chi-Yen Yang
 * @version 1.0
 */

import junit.framework.*;
import gov.ed.sfa.ita.exception.*;

public class TestFtpClient extends TestCase {

    public TestFtpClient(String name)
    {
        super(name);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite();

        suite.addTest(new TestFtpClient("testNonSecureActiveFtpClient"));
        suite.addTest(new TestFtpClient("testNonSecurePassiveFtpClient"));
        suite.addTest(new TestFtpClient("testSecureActiveFtpClient"));
        suite.addTest(new TestFtpClient("testSecurePassiveFtpClient"));

        return suite;
    }

    public void testNonSecureActiveFtpClient() {
        try {
            FtpClient client = new FtpClient("170.248.222.113", 21, "active", false);
            client.login("chi-yen_yang", "123456");
            String currentPath = client.pwd();
            client.changeDirectory("/temp");
            String[] list = client.dir("/temp");
            client.setTransferMode("ASCII");
            client.getFile("test.log", "/home/Chi-Yen_Yang/");
            client.setTransferMode("BIN");
            client.putFile("test.mdb", "/temp/");
            client.quit();
        }
        catch (SFAException ex) {
            Assert.fail(ex.getAddlInfo());
        }
    }

    public void testNonSecurePassiveFtpClient() {
        try {
            FtpClient client = new FtpClient("170.248.222.113", 21, "passive", false);
            client.login("chi-yen_yang", "123456");
            String currentPath = client.pwd();
            client.changeDirectory("/temp");
            String[] list = client.dir("/temp");
            client.setTransferMode("ASCII");
        }
    }
}
```



```
client.getFile("test.log", "/");
client.setTransferMode("BIN");
client.putFile("test.mdb", "/temp/");
client.quit();
}
catch (SFAException ex) {
    Assert.fail(ex.getAddlInfo());
}
}

public void testSecureActiveFtpClient() {
    try {
        FtpClient client = new FtpClient("170.248.222.113", 990, "active", true);
        client.login("chi-yen_yang", "123456");
        String currentPath = client.pwd();
        client.changeDirectory("/temp");
        String[] list = client.dir("/temp");
        client.setTransferMode("ASCII");
        client.getFile("test.log", "/");
        client.setTransferMode("BIN");
        client.putFile("test.mdb", "/temp/");
        client.quit();
    }
    catch (SFAException ex) {
        Assert.fail(ex.getAddlInfo());
    }
}

public void testSecurePassiveFtpClient() {
    try {
        FtpClient client = new FtpClient("170.248.222.113", 990, "passive", true);
        client.login("chi-yen_yang", "123456");
        String currentPath = client.pwd();
        client.changeDirectory("/temp");
        String[] list = client.dir("/temp");
        client.setTransferMode("ASCII");
        client.getFile("test.log", "/");
        client.setTransferMode("BIN");
        client.putFile("test.mdb", "/temp/");
        client.quit();
    }
    catch (SFAException ex) {
        Assert.fail(ex.getAddlInfo());
    }
}
}
```



2.4.4 Automated Testing Conditions

Condition Number	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	The ftp client logs user in and navigates into appropriate directory for downloading and uploading test files in Active mode and non-secure channel.	TestFtpClient	TestNonSecureActiveFtpClient()	FtpClient	FtpClient() Login() Pwd() ChangeDirectory() Dir() SetTransferMode() GetFile() PutFile() Quit()	Get test.log and put test.mdb files.	Test.log Test.mdb
2	The ftp client logs user in and navigates into appropriate directory for downloading and uploading test files in Passive mode and non-secure channel.	TestFtpClient	TestNonSecurePassiveFtpClient()	FtpClient	FtpClient() Login() Pwd() ChangeDirectory() Dir() SetTransferMode() GetFile() PutFile() Quit()	Get test.log and put test.mdb files.	Test.log Test.mdb
3	The ftp client logs user in and navigates into appropriate directory for downloading and uploading test files in Active mode and secure channel.	TestFtpClient	TestSecureActiveFtpClient()	FtpClient	FtpClient() Login() Pwd() ChangeDirectory() Dir() SetTransferMode() GetFile() PutFile() Quit()	Get test.log and put test.mdb files.	Test.log Test.mdb
4	The ftp client logs user in and navigates into appropriate directory for downloading and uploading test files in Passive mode and secure channel.	TestFtpClient	TestSecurePassiveFtpClient()	FtpClient	FtpClient() Login() Pwd() ChangeDirectory() Dir() SetTransferMode() GetFile() PutFile() Quit()	Get test.log and put test.mdb files.	Test.log Test.mdb

2.4.5 Manual Testing Conditions

2.4.5.1 Cycle 1 – Normal

Component Name	FTP Framework	Version #	1
Prepared by	Chi-Yen Yang	Date Prepared	12-Jul-02
Tested by	Chi-Yen Yang	Date Tested	12-Jul-02
Reviewed by		Date Reviewed	



**ITA Release 3.0
Build & Test Report**

Step Number	Detailed Condition	Class Name	Method Name	JSP Name	Expected Results
1	Go to URL: http://dev.conv.sfa.ed.gov:8531/CONVWebApp/ftp/FTPConnection.jsp Enter the following information in form: Host Name: 4.20.14.132 User Name: chyang Password: ***** Press 'Reset'	FTPConnectForm	reset(ActionMapping, HttpServletRequest) validate(ActionMapping, HttpServletRequest)	FTPConnection.jsp	Form clears.
2	Enter the following information in form: Host Name: 4.20.14.132 User Name: chyang Password: ***** Press 'Create Connection'	FTPConnectForm FTPConnectAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPConnection.jsp	User logs in and returned with FTPMoveFiles.jsp page. Client directory should show: / and list of files and directories under this path. Server directory should show: /opt/home/chyang and list of files and directories under this path.
3	Click Change Client Directory radio button. Type /opt/home/chyang /clientTest in corresponding text field. Press 'Reset'	FTPMoveFilesForm	reset(ActionMapping, HttpServletRequest) validate(ActionMapping, HttpServletRequest)	FTPMoveFiles.jsp	Form clears.
4	Click Change Client Directory radio button. Type /opt/home/chyang /clientTest in corresponding text field. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	Client Directory changes to /opt/home/chyang /clientTest and selection box shows test.log.



ITA Release 3.0 Build & Test Report

5	<p>Click Change Server Directory radio button. Type /opt/home/chyang/serverTest in corresponding text field. Press 'Just do it'</p>	FTPMoveFilesForm FTPMoveFilesAction	<p>validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)</p>	FTPMoveFiles.jsp	Server Directory changes to /opt/home/chyang/serverTest and selection box shows test.log.
6	<p>Click Get File radio button. Click BIN mode radio button. Select test.doc from server selection box. Press 'Just do it'</p>	FTPMoveFilesForm FTPMoveFilesAction	<p>validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)</p>	FTPMoveFiles.jsp	FTPMoveFiles.jsp refreshes with test.doc showing on both selection boxes.
7	<p>Click Put File radio button. Click ASCII mode radio button. Select test.log from server selection box. Press 'Just do it'</p>	FTPMoveFilesForm FTPMoveFilesAction	<p>validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)</p>	FTPMoveFiles.jsp	FTPMoveFiles.jsp refreshes with test.doc showing on both selection boxes.
8	<p>Click Quit radio button. Press 'Just do it'</p>	FTPMoveFilesForm FTPMoveFilesAction	<p>validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)</p>	FTPMoveFiles.jsp	quit.jsp appears.



2.4.5.2 Cycle 2 – Connection Exception

Component Name	FTP Framework	Version #	1
Prepared by	Chi-Yen Yang	Date Prepared	12-Jul-02
Tested by	Chi-Yen Yang	Date Tested	12-Jul-02
Reviewed by		Date Reviewed	

Step Number	Detailed Condition	Class Name	Method Name	JSP Name	Expected Results
1	Go to URL: http://dev.conv.sfa.ed.gov:8531/CONVWebApp/ftp/FTPConnection.jsp Enter the following information in form: Host Name: User Name: Password: Press 'Connect'	FTPConnectForm	validate(ActionMapping, HttpServletRequest)	FTPConnection.jsp	error messages appear next to each text field. Prompting users to enter required information.
2	Hit Back button. Enter the following information in form: Host Name: abcdefg User Name: chyang Password: ***** Press 'Connect'	FTPConnectForm	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPConnection.jsp	error.jsp appears and shows unknown host exception.
3	Hit Back button. Enter the following information in form: Host Name: 4.20.14.132 User Name: adf Password: *****	FTPConnectForm	reset(ActionMapping, HttpServletRequest) validate(ActionMapping, HttpServletRequest)	FTPConnection.jsp	error.jsp appears and shows User not Log In message.



	Press 'Connect'				
4	Hit Back button. Enter the following information in form: Host Name: 4.20.14.132 User Name: chyang Password: * Press 'Connect'	FTPConnectForm	reset(ActionMapping, HttpServletRequest) validate(ActionMapping, HttpServletRequest)	FTPConnection.jsp	error.jsp appears and shows User not Log In message.

2.4.5.3 Cycle 3 – Transfer Exception

Component Name	FTP Framework	Version #	1
Prepared by	Chi-Yen Yang	Date Prepared	12-Jul-02
Tested by	Chi-Yen Yang	Date Tested	12-Jul-02
Reviewed by		Date Reviewed	

Step Number	Detailed Condition	Class Name	Method Name	JSP Name	Expected Results
1	Enter the following information in form: Host Name: 4.20.14.132 User Name: chyang Password: ***** Press 'Create Connection'	FTPConnectForm FTPConnectAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPConnection.jsp	User logs in and returned with FTPMoveFiles.jsp page. Client directory should show: / and list of files and directories under this path. Server directory should show: /opt/home/chyang and list of files and directories



**ITA Release 3.0
Build & Test Report**

					under this path.
2	Click Change Client Directory radio button. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	error message appears indicating no client directory path typed in the text field.
3	Click Change Server Directory radio button. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	error message appears indicating no server directory path typed in the text field.
4	Click Put File radio button. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	error message appears indicating no client file selected.



**ITA Release 3.0
Build & Test Report**

5	Click Gut File radio button. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	error message appears indicating no server file selected.
8	Click Quit radio button. Press 'Just do it'	FTPMoveFilesForm FTPMoveFilesAction	validate(ActionMapping, HttpServletRequest) perform(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)	FTPMoveFiles.jsp	quit.jsp appears.



2.5 Performance Profiling

2.5.1 Summary

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 shown below.

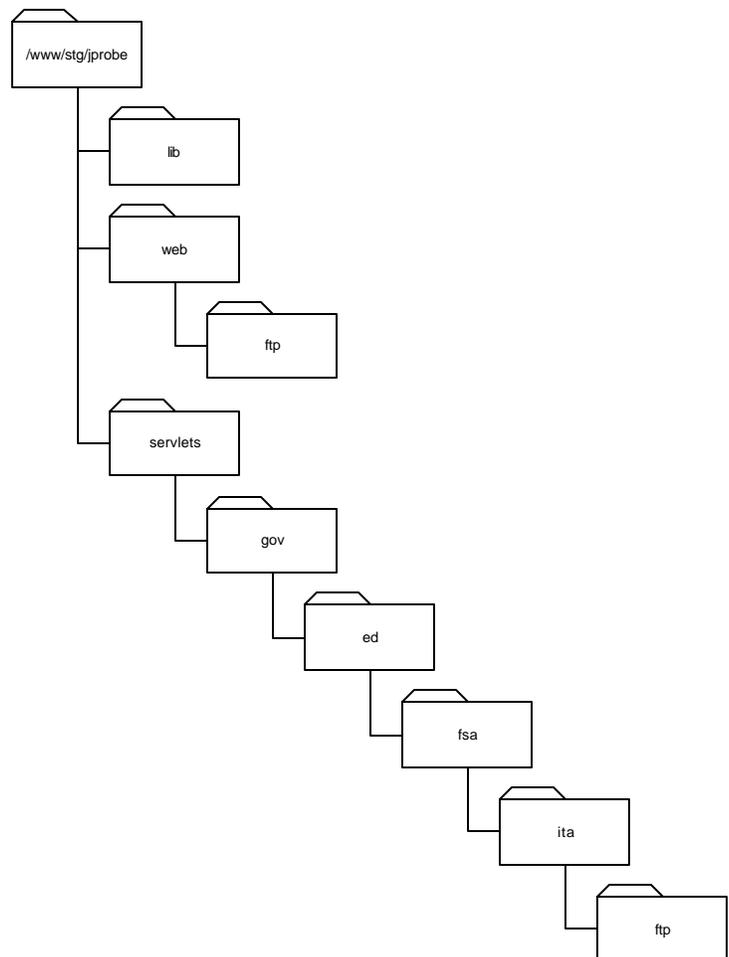
During the heap analysis, one object was found loitering in the heap. One instance of FtpControlSocket was not garbage collected at the end of the profiling session. Code changes were applied and the problem was fixed. Application performance did not pose to be an issue. FtpClient.putFile() did appear to be high on execution time, however, it was due to the file transfer time instead of actual execution time. A table of the top ten method time is provided for developer to reference back when including the framework in his or hers application.

2.5.2 Test Harness Design

2.5.2.1 Environment

The performance profiling was done in an isolated environment on su35e5 application server. By running only one application in the environment, it can be made sure that the statistics captured are from the application.

In the performance profiling environment, the FTP Framework application was placed under /www/stg/jprobe/ directory on su35e5 application server. The file structure is shown on the right. Two major branches were setup for housing JSP and Servlets. Under ./web/ftp directory, all the front end JSP pages were kept. As for FTP Framework Java classes, they were served out of the ./servlets/gov/ed/fsa/ita/ftp directory. RCS Web Conversation Framework configuration file, struts-config.xml was placed in the ../web/WEB-INF directory. The





ITA Release 3.0 Build & Test Report

application properties files were in the ./servlets directory. Required jar files such as jsse.jar, jnet.jar and jcert.jar were in the ./lib directory.



2.5.2.2 Configuration

Configuration setting for FTP Framework was the same as it was in unit testing. As for JProbe, it uses its own configuration file, “.jpl” file. The .jpl file lets JProbe know what type of statistics to collect and what Java classes to monitor specifically. The following .jpl file was used for the profiling session:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
    </application>
    <applet
      working_dir=""
      source_dir=""
      htmlfile=""
      main_package="">
    </applet>
    <serverside
      suggested_filters=""
      id="Other server"
      server_dir="/opt/stg35/WebSphere/AppServer"
      prepend_to_vm_args=""
      source_dir=""
      classname="com.ibm.ejs.sm.util.process.Nanny"
      main_package="gov.ed.fsa.ita.ftp"
      exclude_server_classes="true"
      args=""
      working_dir="/opt/stg35/WebSphere/AppServer/servlets"
      prepend_to_classpath="">
    </serverside>
  </program>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.113:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
    </performance>
  </analysis>
</jpl>
```



```
    timing="elapsed"
    track_natives="true"
    final_snapshot="true"
    granularity="method">
  <performance.filter
    visibility="visible"
    methodmask="*"
    enabled="true"
    classmask="*"
    time="ignore"
    granularity="method"/>
  <performance.filter
    visibility="visible"
    methodmask="*"
    enabled="true"
    classmask="gov.ed.fsa.ita.ftp.*"
    time="track"
    granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask="*/>
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask="."/>
</threadalyzer>
<coverage
```



```
record_from_start="true"  
final_snapshot="true"  
granularity="line">  
<coverage.filter  
  visibility="invisible"  
  methodmask="*"   
  enabled="true"  
  classmask="*" />  
<coverage.filter  
  visibility="visible"  
  methodmask="*"   
  enabled="true"  
  classmask="*" />  
</coverage>  
</analysis>  
</jpl>
```

To run FTP Framework in JProbe's JVM, several configuration changes were made on the application server.

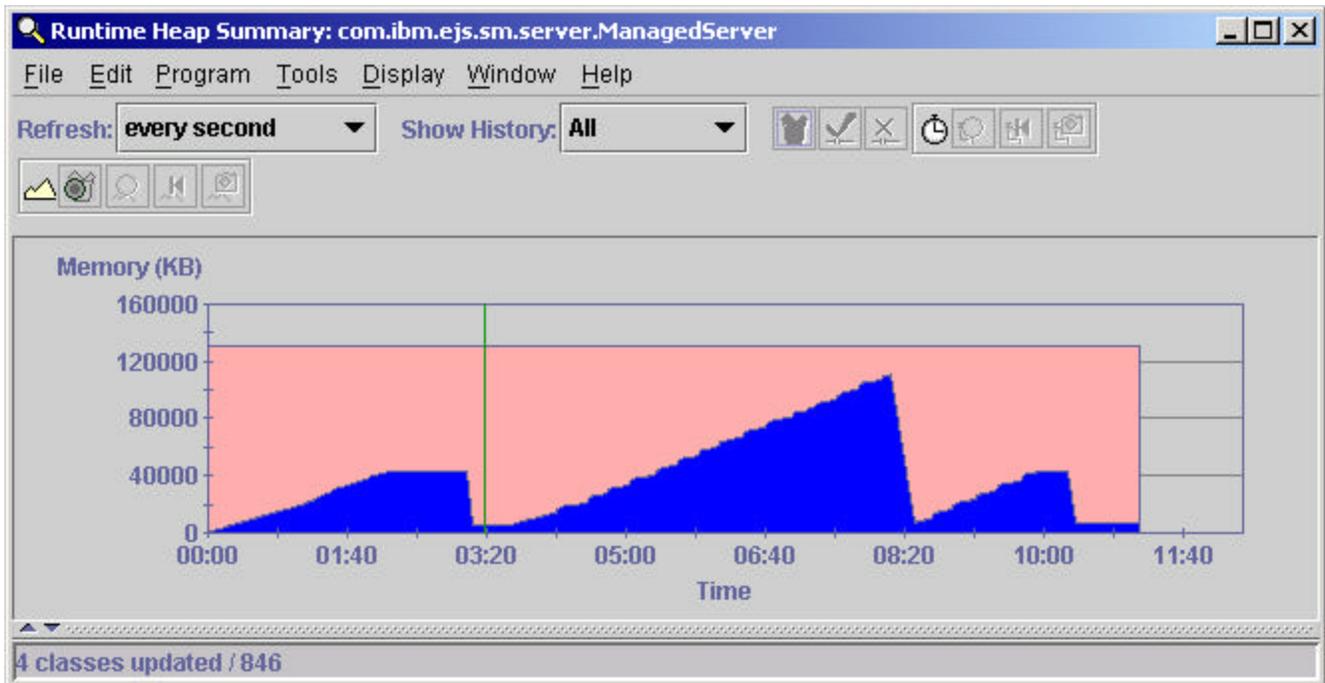
- Under application command line arguments:
 - Added `-jp_input=/opt/util/JProbe/jpl_files/07152002_test_ftp.jpl`
 - Added `-Djava.compiler=NONE`
- Under Environment:
 - Added `EXECUTABLE=/opt/util/JProbe/profiler/jprun`
 - `EXECUTE=YES`

2.5.2.3 Scenario setup

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 for 20 times. By running the same process multiple times, better statistics could be collected.

2.5.3 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.



The above heap graph shows the memory usage throughout the profiling session. The pink portion of the graph indicates the maximum allocated memory for the application server. In this case, the maximum memory allocated for the application server is 128 MB. The blue region shows the memory used within the allocated amount. The green line is a base for comparison (number of objects remain in the heap) at the end of the profiling session.

As the graph indicates, three garbage collections were done during the profiling session. The first garbage collection was manually requested to establish a base line for later comparison. The second garbage collection was done by the JRE as the application approached its allocated memory size. The third garbage collection was also requested manually to identify remaining objects in the memory after the scenario finished its 20 iterations.

2.5.3.1 Instance Summary

The table below is a section of the Instance Summary. An instance summary shows objects that are currently in the heap. The Count column displays the number of instances of an object currently exist in the heap and the Memory column shows the amount of memory those instances consume.



The screenshot shows the Heap Browser interface for snapshot_1. The main table lists classes with their counts and memory usage. The FSAFtpControlSocket class is highlighted, showing 30 instances and 360 bytes of memory. Below the table, the class details for gov.ed.fsa.ita.ftp.FSAFtpControlSocket are shown, including a list of instances with their IDs, sizes, and creation times. The instance summary for FSAFtpControlSocket 0xE82C8DE0 shows it was allocated at <root> Statics FSAFtpClient.

Name	Count	Count Change	Memory	Memory Change
FTPConnectAction	1 (0.0%)	+1	4 (0.0%)	+4
FSAFtpControlSocket	30 (0.0%)	+1	360 (0.0%)	+12
FTPMoveFileAction	1 (0.0%)	+1	4 (0.0%)	+4
FTPConnectForm	30 (0.0%)	-	1,080 (0.0%)	-
FSAFtpClient	30 (0.0%)	-	360 (0.0%)	-
FSAFtpDataSocket	120 (0.0%)	-	1,440 (0.0%)	-
FTPMoveFileForm	30 (0.0%)	-	1,080 (0.0%)	-

Instance ID	Size	Creation	Referrers
FSAFtpControlSocket	12	11:17:795	
FSAFtpControlSocket	12	11:36:743	
FSAFtpControlSocket	12	12:43:602	

The above matrix was sorted based on the Count Change column. Count Change column was calculated based on the “base” (green vertical line) at the beginning of the test scenarios. A positive number in this column means this number of objects is left in heap after the final garbage collection. A positive number is significant because it is a good indication, but not “THE” indication, that the associated class creates loitering objects and causes memory leak.

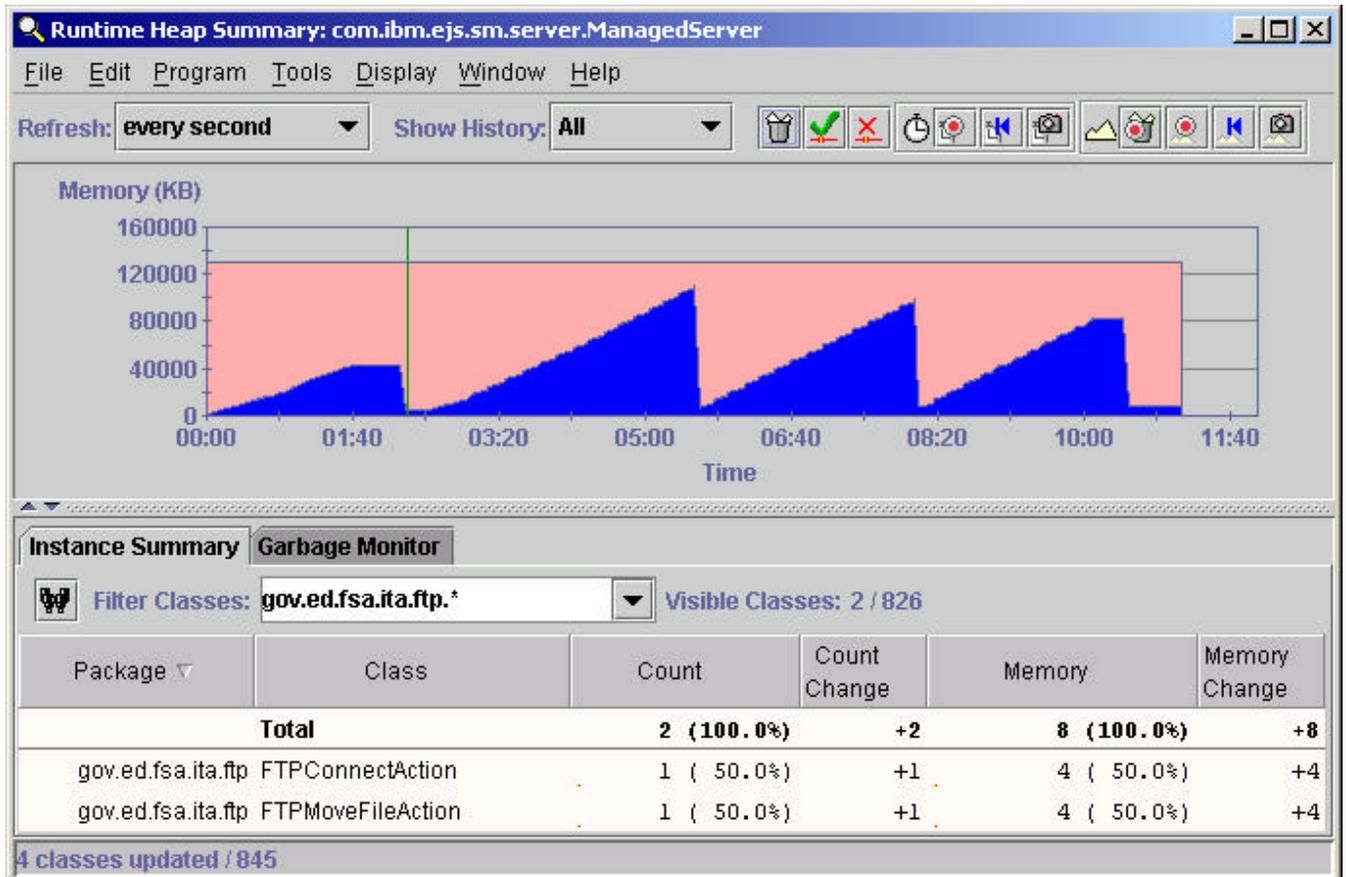
As one can see, three FTP Framework classes, FTPConnectAction, FTPMoveFileAction and FtpControlSocket, left one object each in the heap after the third garbage collection. As shown in RCS Web Conversation Performance Profiling report, *Action classes are reused through out the life of the application. 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.

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



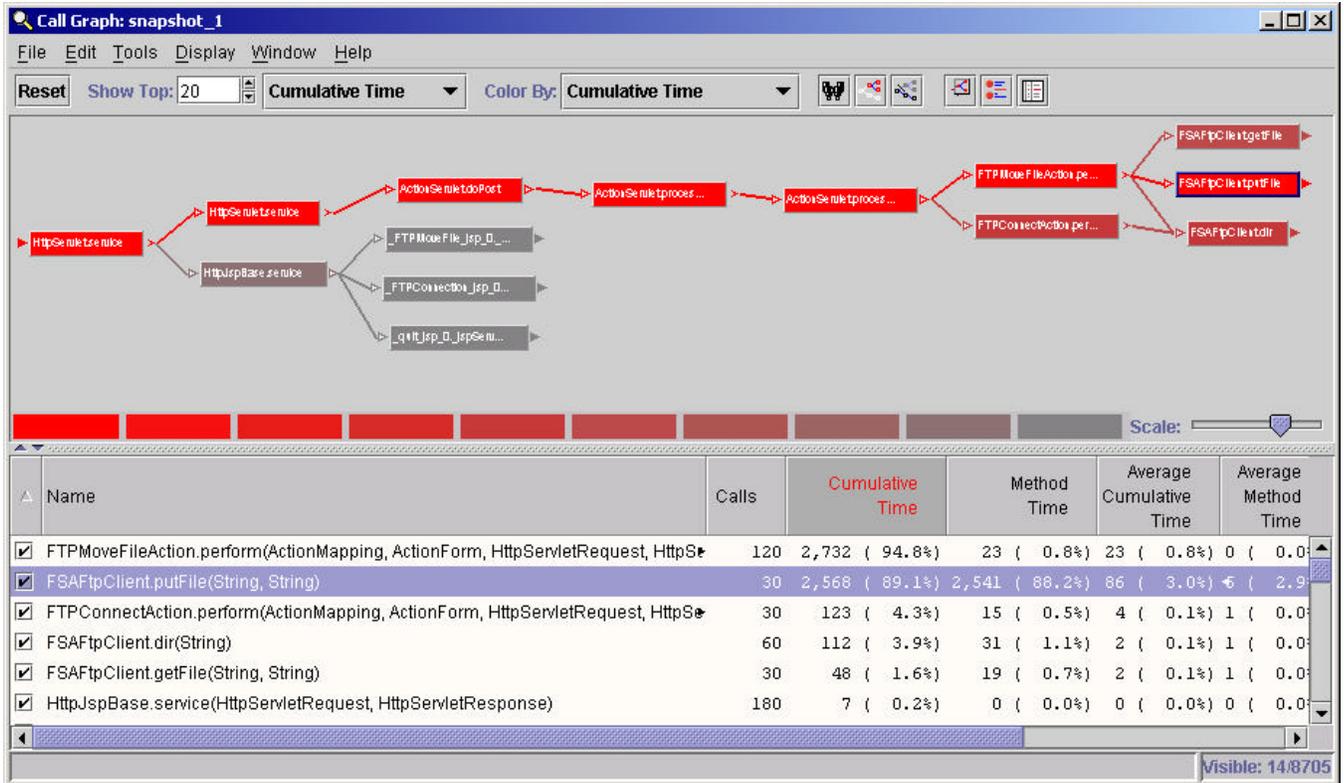
input/output streams and control socket itself. However, the method does not set FtpControlSocket to null after the end of the execution. Thus, one class object remains.

Code changes were made to include this new discovery. The graph below shows the new heap snapshot after 40 iterations. The only classes that remained in the heap were the two *Action classes.



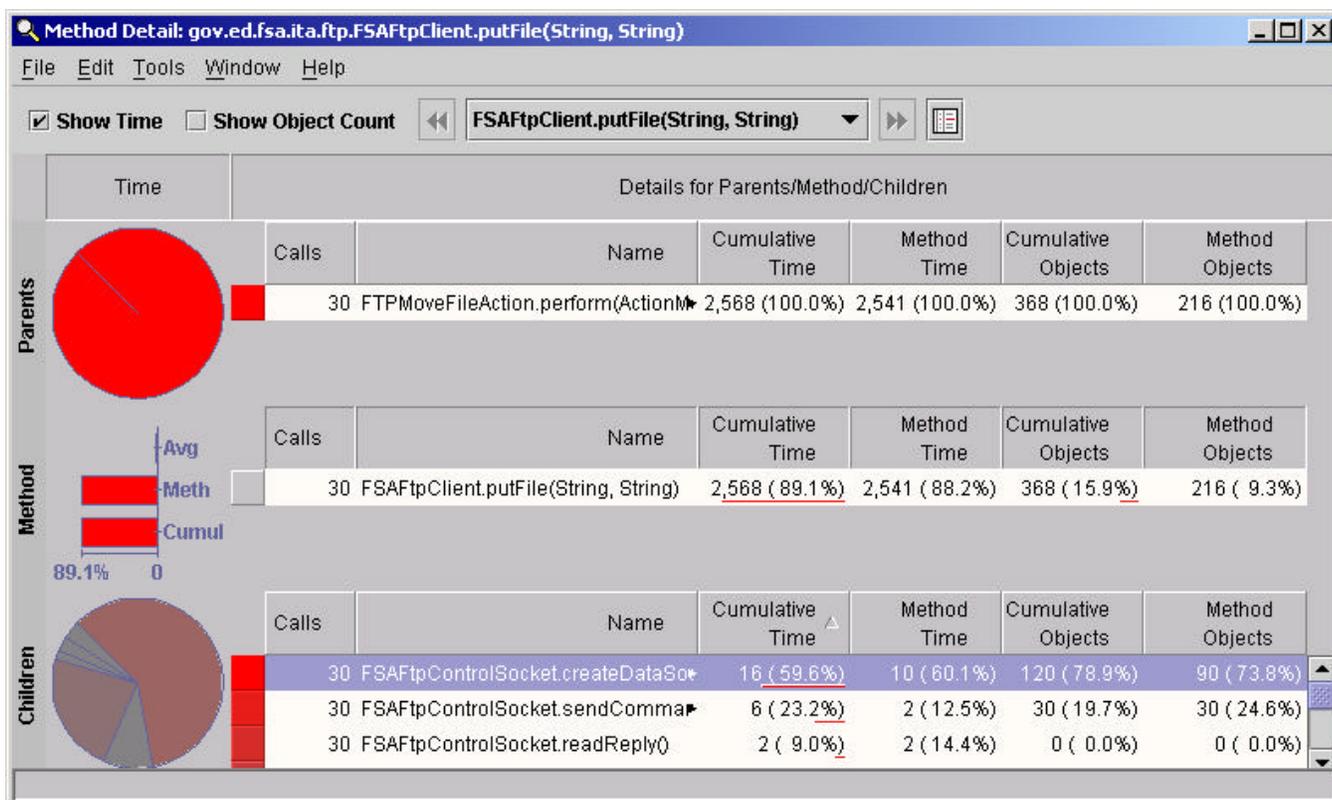
2.5.4 Performance Analysis

The following tree graph shows a list of methods that were used during the profiling session. The root of the graph is `Javax.servlets.http.HttpServlet.service` and from there on, it is divided into JSP and Servlets services. The graph is color-coded based on cumulative time. The darker the color, the more time spent in a class or method. Only the top methods are shown in the graph.

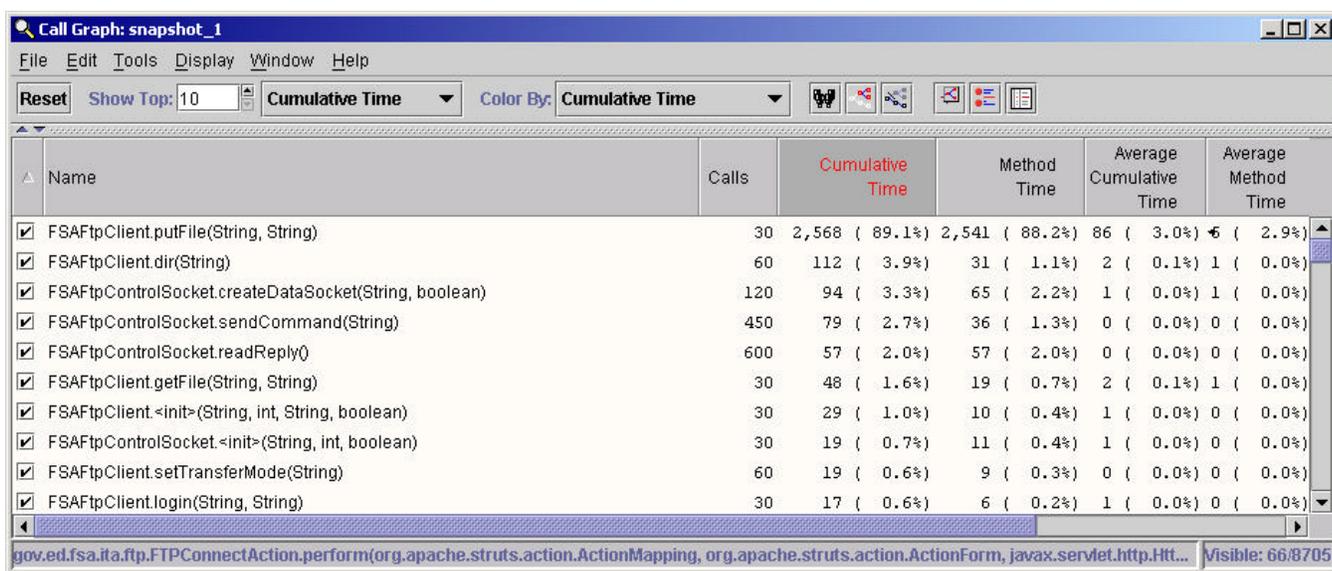


As indicated in the graph, most time was spent in the servlets branch. In particular, the `FtpClient.putFile()` method took the most execution time. Thus, the path that led to `FtpClient.putFile()` was the critical path of this framework. To improve performance, the critical path should be looked at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

The graph below shows the references of the `put()` method. As can be seen, the top reference to the `put()` method was the `createDataSocket()` method with a cumulative method time of 10 milliseconds. Furthermore, 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.



2.5.4.1 Top ten FTP Framework related cumulative method time





The above table shows the top ten FTP Framework related cumulative method time. The table should be used as reference in application development that includes the framework.

3 RCS – XML Helper Framework

3.1 Purpose

This Performance Analysis Report documents the results of utilizing JProbe to test the ITA R3.0 Reusable Common Services (RCS) XMLHelper framework. This report provides an in-depth analysis of the results gathered from the JProbe application profiling and documents any performance issues and suggests resolutions. The Detailed Design, User Guide, Unit Test Report, and the Performance Analysis documents for the XMLHelper framework documentation will enable developers to quickly build applications using the XMLHelper framework within the ITA environment architecture.

3.2 Approach

To ensure program efficiency and to detect possible bottleneck, ITA used JProbe to analyze the XMLHelper framework. JProbe is a performance-profiling tool and it was used to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming. In order to profile this framework, portions of the unit test scripts were used to conduct this test. The performance analysis of this framework is documented in this report.

Two key groups of statistics are collected from the JProbe Profiler: the memory (heap) usage and performance detail usage which include detailed method times, average method times, detailed object counts within methods and average method counts. This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow gathering detailed information on individual methods and the interaction between them.

3.3 Summary

This report contains the background information, performance test harness design, performance analyses, and resulting performance metrics for the framework. Profiling the XMLHelper framework using the test scripts will test the code performance of the framework. The actual results will be compared against the results of how this framework is expected to function. Overall, this framework does not produce any loitering objects or create an excessive amount of objects. Of course memory used is in direct relationship to the size of the XML document but for most XML documents used by FSA developers memory usage per document is tiny. This framework is a robust API that should not cause any performance issues for calling applications.



3.4 Test Harness Design

3.4.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance test is to identify loitering objects and time spent on each method relative to each other in the XMLHelper framework.

3.4.2 Testing Criteria

The three main components of the XMLHelper framework will be tested:

- Parsing XML documents using the DOM parser.
- Parsing XML documents using the SAX parser and a custom developed SAX parsing class.
- Instantiating a Java object from an XML document using a Data-Bind parser.

Since the XMLHelper framework is an API, the JavaServer Pages developed for the unit test will serve as a test harness to profile and analyze the performance of the various methods.

3.4.3 Testing Configuration

In order to profile the XMLHelper framework with JProbe, the JPROBE Application Server configured in WebSphere was used and some of the configurations were changed. In the command line reference of the Application Server, there is a reference to the JProbe configuration file. The file used to conduct this performance analysis is: `/opt/util/JProbe/jpl_files/08282002_test_xmlhelper.jpl`. Due to the fact that some of the applications use STRUTS there are several servlets that are present in the configuration that are not needed for this test. Thus the action, database, and HelloWorld servlets were all disabled.

3.4.4 JProbe Configuration File

The JProbe configuration file has a file extension of `.jpl`. This file contains all of the settings that JProbe requires to profile an application, applet, or server side component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options. The user will be able to specify the activity of the Profiler. For example, the file can be configured to cause JProbe Profiler to take a heap snapshot before it exits and the directory to save the snapshots in.

The example application test will be conducted on the Solaris machine with the output being sent to a remote Windows NT workstation. The configuration in the actual file used to conduct the test can be found in [Appendix A](#). A filter for the main package, `gov.ed.sfa.ita.xmlhelper`, was added to narrow the scope of the test to this package.



3.4.5 UNIX Server Settings

The current methodology that ITA uses to do performance testing of RCS packages is to run custom built test harnesses off a Application Server called JPROBE that is running in the stage environment of the development Solaris server. The server URI that is configured in the JPROBE Application server is **stg.jprobe.fsa.ed.gov**. The WebSphere JSP servlet Web path to call the test harnesses is JPROBEWebApp. So the following URL's should execute the three performance tests.

http:\\Stg.jprobe.fsa.ed.gov/JPROBEWebApp/xmlhelper/domTest.jsp

http:\\Stg.jprobe.fsa.ed.gov/JPROBEWebApp/xmlhelper/saxTest.jsp

http:\\Stg.jprobe.fsa.ed.gov/JPROBEWebApp/xmlhelper/databindTest.jsp

To accomplish the above URLs, the following WebSphere configuration files (located in **/opt/stg35/WebSphere/AppServer/temp**) were configured as documented.

3.4.5.1 rules.properties:

```
default_host/JPROBEWebApp/*.do=ibmoselink4
default_host/JPROBEWebApp/*.jsp=ibmoselink4
default_host/JPROBEWebApp/*.jsv=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/=ibmoselink4
default_host/JPROBEWebApp/ErrorReporter=ibmoselink4
default_host/JPROBEWebApp/servlet=ibmoselink4
default_host/JPROBEWebApp=ibmoselink4
```

3.4.5.2 queues.properties:

```
ose.srvgrp.ibmoselink4.clone1.port=8241
ose.srvgrp.ibmoselink4.clone1.type=remote
ose.srvgrp.ibmoselink4.clonescount=1
ose.srvgrp.ibmoselink4.type=FASTLINK
ose.srvgrp=ibmoselink3,ibmoselink2,ibmoselink4,ibmoselink17
```

3.4.5.3 vhosts.properties:

```
stg.jprobe.fsa.ed.gov=default_host
```



3.4.6 WebSphere Application Server Configuration

The WebSphere Command Line will identify the JProbe configuration file to use and ensure that the correct JVM is used. Two Environment Variables will be added to the Application Server to enable it to run with JPROBE.

3.4.6.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/08282002_test_xmlhelper.jpl -Xnoclassgc -  
Djava.compiler=NONE -ms128m -mx128m
```

3.4.6.2 Environment:

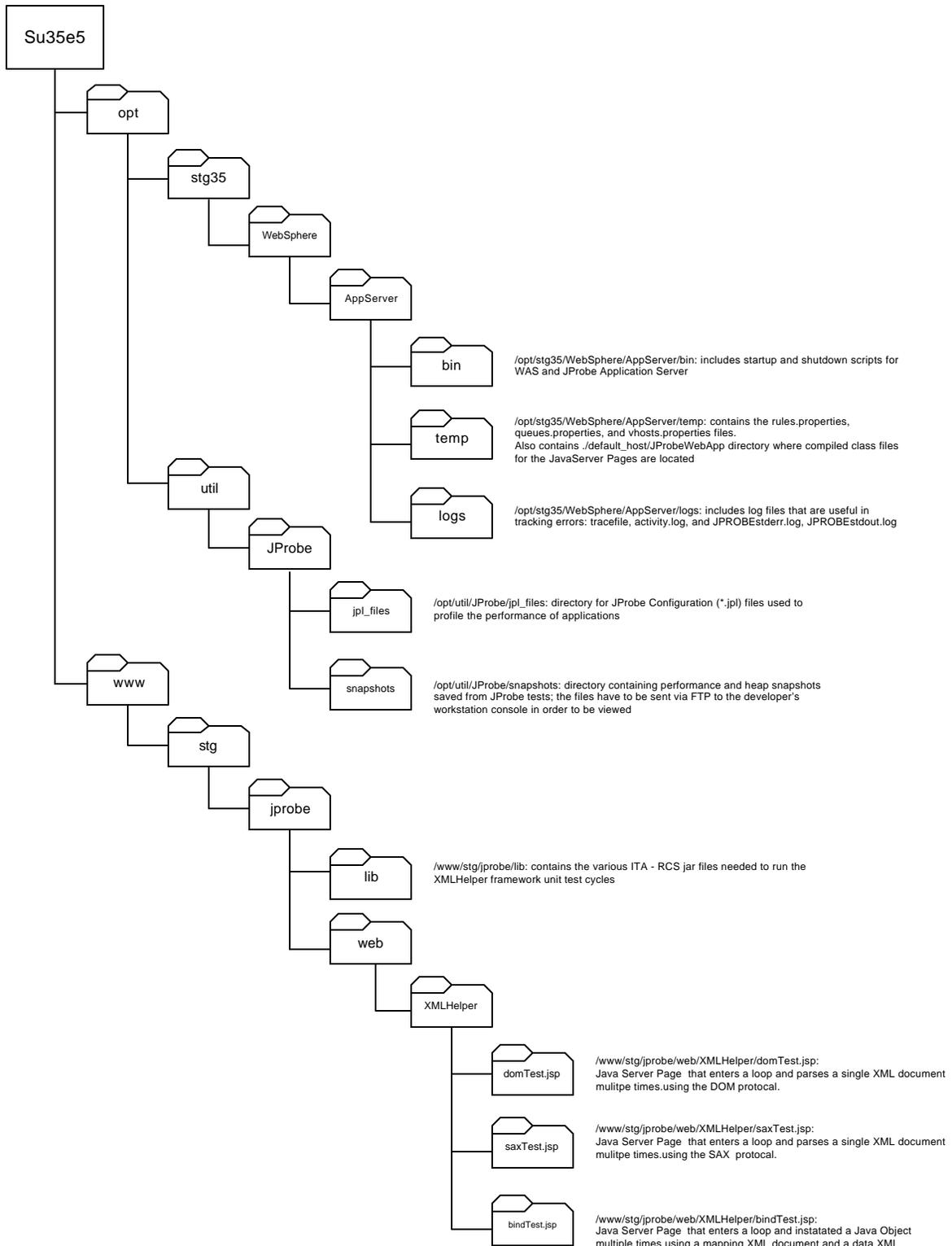
```
EXECUTE=YES  
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```



3.4.7 Directory Structure



ITA Release 3.0 Build & Test Report





3.5 Testing Scenario

Test applications created for the unit test of the RCS framework will be used to execute the performance analysis. These test applications are actually Java Server Pages that access the XMLHelper framework to do work.

DomTest.jsp—Currently configured to build a DOM tree of an example XML document located at */www/stg35/jprobe/properties/example.xml* and then search for a specific element, using the XMLHelper *searchDom* method, which returns the elements value. Once the value is returned, the value is checked against what was configured and will output whether the value matches or doesn't. 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 located at */www/stg35/jprobe/properties/example.xml* and then using and then search for a specific element, using the XMLHelper *searchSAX* method, which returns the elements value. Once the value is returned, the value is checked against what was configured and will output whether the value matches or doesn't. 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.

BindTest.jsp—Configured to instantiate a Java Object from two XML documents. The first document located at */www/stg35/jprobe/properties/mapping.xml* defines the attributes of the Java object that the parser is trying to build. The second XML document located at */www/stg35/jprobe/properties/schedule.xml* holds the objects attribute values. This 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.

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.

3.6 Results and Analysis

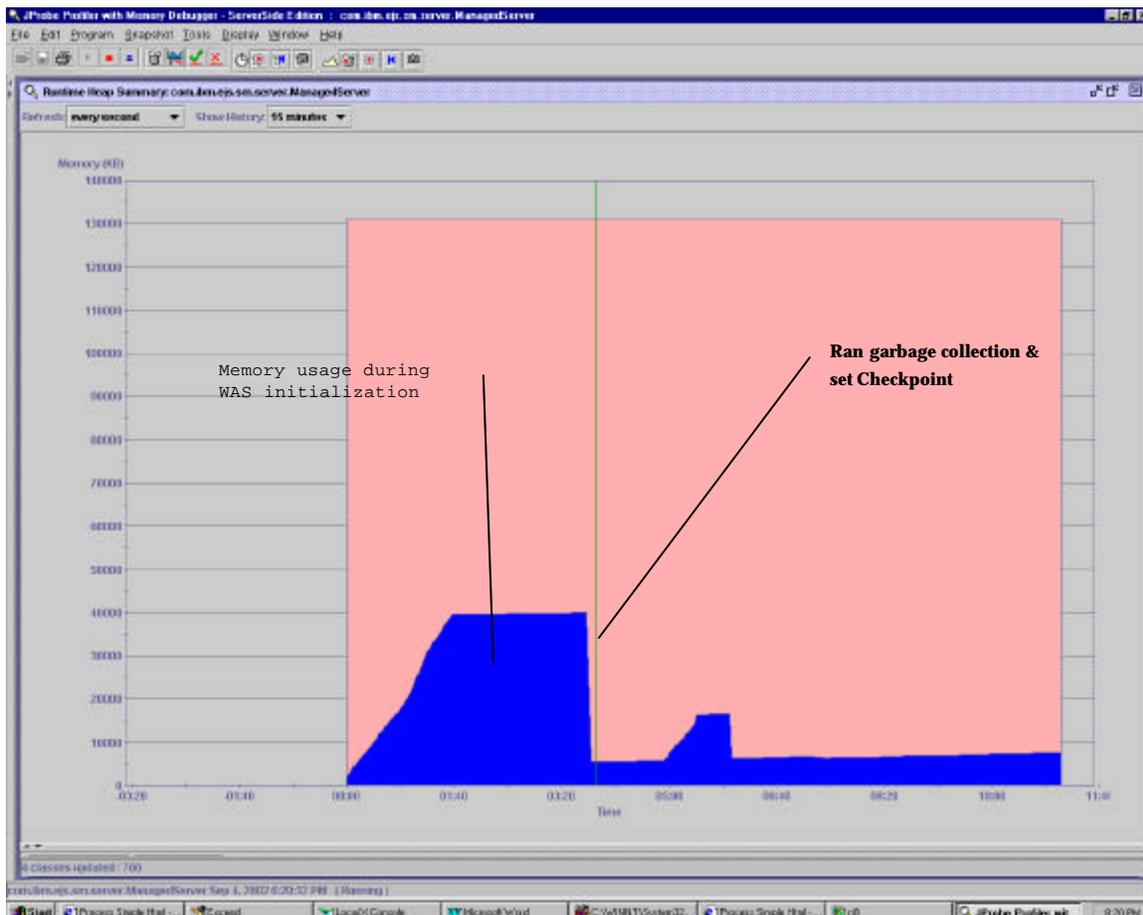
The JProbe Profiler with Memory Debugger application is used to trace both the memory usage and performance measurement of the XMLHelper framework API. Two snapshots are taken for each test scenario: a heap snapshot and a performance snapshot. Each snapshot provides different information regarding our test.

3.7 Heap Snapshot (Memory Usage)

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.

3.7.1.1 Heap Graph Analysis

The screenshot below is obtained from executing domTest.jsp. It is the only heap graph screenshot depicted in this report since the heap graphs from executing other test cycle exhibit the same pattern.





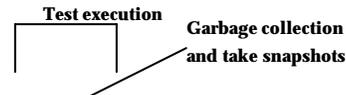
In the graph above, it is possible to see that when the Application Server is initialized, a great deal of memory is consumed. Once the App Server has finished initializing, the memory usage levels off to a flat line. JProbe will call the Garbage Collector to remove objects that are no longer being referenced 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. By reading the graph, it can be determined that the overall memory usage for the XMLHelper framework is very low and will not result in huge increase to the overhead of calling applications.

3.7.2 Instance Summary

The tables below represent Instance Summary result's associated with conducting the different test scenarios. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory (in bytes) those instances consume.

3.7.2.1 DomTest.jsp



Package	Class	Count	Memory
gov.ed.sfa.ita.xmlhelper	DomXml	40 (0.0%)	1,760 (0.0%)
gov.ed.sfa.ita.xmlhelper	PpKey	80 (0.0%)	960 (0.0%)
gov.ed.sfa.ita.xmlhelper	PpValue	80 (0.0%)	960 (0.0%)
gov.ed.sfa.ita.xmlhelper	DomXml\$	40 (0.0%)	480 (0.0%)
gov.ed.sfa.ita.xmlhelper	DomXml\$	40 (0.0%)	480 (0.0%)
gov.ed.sfa.ita.xmlhelper	DomXml\$3	10 (0.0%)	120 (0.0%)
gov.ed.sfa.ita.xmlhelper	XMLHelper	1 (0.0%)	44 (0.0%)

3.7.2.2 SaxTest.jsp

Package	Class	Count	Memory
gov.ed.sfa.ita.xmlhelper	SaxSearchHandler	10 (0.0%)	120 (0.0%)
gov.ed.sfa.ita.xmlhelper	XMLHelper	1 (0.0%)	44 (0.0%)
gov.ed.sfa.ita.xmlhelper	SaxXml	10 (0.0%)	40 (0.0%)

3.7.2.3 BindTest.jsp

Package	Class	Count	Memory
org.exolab.castor.xml.util	XMLFieldDescriptorImpl	551 (0.1%)	46,284 (0.4%)
org.exolab.castor.util	List	941 (0.1%)	18,820 (0.2%)
org.exolab.castor.xml	UnmarshalState	310 (0.0%)	13,640 (0.1%)



ITA Release 3.0 Build & Test Report

Package	Class	Count	Memory
org.exolab.castor.mapping.xml	FieldMapping	110 (0.0%)	9,240 (0.1%)
org.exolab.castor.xml	FieldValidator	420 (0.1%)	8,400 (0.1%)
org.exolab.castor.mapping.loader	FieldHandlerImpl	110 (0.0%)	8,360 (0.1%)
org.exolab.castor.mapping.loader	FieldDescriptorImpl	110 (0.0%)	4,840 (0.0%)
org.exolab.castor.xml.validators	NameValidator	90 (0.0%)	3,960 (0.0%)
org.exolab.castor.mapping.loader	TypeInfo	110 (0.0%)	3,960 (0.0%)
org.exolab.castor.xml.validator	StringValidator	90 (0.0%)	3,240 (0.0%)

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 than 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 DataBind API uses a DataBinding Framework called CASTOR to accomplish the marshalling and demarshalling activities.



3.8 Performance Snapshot (Code Efficiency)

There are nine efficiency metrics that can be collected using JProbe – five basic metrics and four compound metrics. The basic metrics include: number of calls, method time, cumulative time, method object count, and cumulative object count. The compound metrics are averages per number of calls, including: average method time, average cumulative time, average method object count, and average cumulative object count. Time is measured as elapsed time in milliseconds.

The following sections will describe each metric and display the top results for each measurement for the performance assessment of the XMLHelper framework. These metrics are basic indicators of process resource utilization. The detailed graphs associated with each method can be reviewed for unexpected activity or optimization opportunities.

All performance metric results were first filtered by *xmlhelper* to obtain only the classes within the XMLHelper framework which is what the test is looking for. For the DataBind API we also filtered for *exolab*. Then for each section, the results were sorted by the metric under investigation to obtain the top ten results for each metric.

3.8.1 DomTest.jsp Scenario

3.8.1.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Name	Calls	Source
DomXml.getRootNamespaceURI()	728	DomXml.java
PpKey.hasDefaultNamespaceURI()	727	PpKey.java
PpValue.isSingleItem()	600	PpValue.java
PpKey.hashCode()	430	PpKey.java
PpValue.getSingleItem()	320	PpValue.java
PpKey.equals(Object)	297	PpKey.java
PpValue.isString()	240	PpValue.java
PpValue.isAttribute()	160	PpValue.java
DomXml.uncheckedPut(Object, Object)	110	DomXml.java



Name	Calls	Source
PpKey.setPropertiesPlus(DomXml)	110	PpKey.java

For every DOM element encounter, the XMLHelper framework must decide if there is a namespace associated with that element and if the root namespace has changed. Thus the method *getRootNamespaceURI()* is the most called method.

3.8.1.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Name	Method Time	Source
DomXml.elementToObject(Element)	18 (18.3%)	DomXml.java
PpKey.hashCode()	11 (11.1%)	PpKey.java
DomXml.privPut(PpKey, Object, boolean)	6 (5.7%)	DomXml.java
DomXml.elementKeys()	5 (5.5%)	DomXml.java
DomXml.attributeKeys()	5 (5.3%)	DomXml.java
PpKey.hasDefaultNamespaceURI()	5 (5.1%)	PpKey.java
DomXml.fromXML(InputStream)	4 (4.1%)	DomXml.java
PpKey.equals(Object)	4 (4.0%)	PpKey.java
PpValue.isString()	2 (2.3%)	PpValue.java
XMLHelper.searchDom(String, DomXml)	2 (2.2%)	XMLHelper.java

The results above show the longest running method is *elementToObject(Element)*. This is the method that builds the DOM tree and places elements within a hash table for quick retrieval. Method times are below 50 milliseconds and no one method is dominating the times.



3.8.1.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Name	Cumulative Time	Source
XMLHelper.parse(String)	49 (50.3%)	XMLHelper.java
DomXml.loadXML(InputStream)	48 (48.4%)	DomXml.java
DomXml.fromXML(InputStream)	41 (41.8%)	DomXml.java
DomXml.elementToObject(Element)	37 (37.6%)	DomXml.java
XMLHelper.searchDom(String, DomXml)	34 (34.4%)	XMLHelper.java
XMLHelper.<init>()	15 (15.3%)	XMLHelper.java
PpKey.hashCode()	15 (15.3%)	PpKey.java
DomXml.privPut(PpKey, Object, boolean)	13 (13.6%)	DomXml.java
DomXml.privAdd(PpKey, Object)	13 (13.4%)	DomXml.java
DomXml.attributeKeys()	12 (12.6%)	DomXml.java

The framework entry point is the *parse(String)* method. It would be expected that *parse(String)* would be the longest cumulative time method.



3.8.1.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Name	Method Objects	Source
DomXml.elementToObject(Element)	206 (26.1%)	DomXml.java
PpKey.hashCode()	144 (18.3%)	PpKey.java
DomXml.attributeKeys()	122 (15.5%)	DomXml.java
DomXml.elementKeys()	122 (15.5%)	DomXml.java
DomXml.privPut(PpKey, Object, boolean)	84 (10.6%)	DomXml.java
DomXml.<init>(String, String)	42 (5.3%)	DomXml.java
XMLHelper.parse(String)	24 (3.0%)	XMLHelper.java
DomXml.keys()	12 (1.5%)	DomXml.java
DomXml.fromXML(InputStream)	8 (1.0%)	DomXml.java
XMLHelper.<init>()	8 (1.0%)	XMLHelper.java

Since the elementToObject(Element) method is the method that builds the DOM tree it had the most number of objects.

3.8.1.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Name	Cumulative Objects	Source
XMLHelper.parse(String)	442 (56.0%)	XMLHelper.java
DomXml.loadXML(InputStream)	418 (53.0%)	DomXml.java
DomXml.fromXML(InputStream)	386 (48.9%)	DomXml.java
DomXml.elementToObject(Element)	378 (47.9%)	DomXml.java
XMLHelper.searchDom(String, DomXml)	324 (41.1%)	XMLHelper.java
DomXml.elementKeys()	152 (19.3%)	DomXml.java



Name	Cumulative Objects	Source
DomXml.attributeKeys()	152 (19.3%)	DomXml.java
PpKey.hashCode()	144 (18.3%)	PpKey.java
DomXml.privPut(PpKey, Object, boolean)	114 (14.4%)	DomXml.java
DomXml.privAdd(PpKey, Object)	90 (11.4%)	DomXml.java

Again very similar to Cumulative Time and Cumulative object count, *parse(String)* is the entry method to the framework so it would be expected to have the most objects. The interesting fact about this chart is that if a developer traced the code from *parse()* hash table loading, the trace would look very similar to the above chart with each successive method adding a few more objects but not one adding more than 50%.

3.8.1.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Name	Avg. Method Time	Source
XMLHelper.<init>()	2 (1.6%)	XMLHelper.java
DomXml.fromXML(InputStream)	0 (0.4%)	DomXml.java
DomXml.elementToObject(Element)	0 (0.3%)	DomXml.java
XMLHelper.parse(String)	0 (0.2%)	XMLHelper.java
DomXml.elementKeys()	0 (0.1%)	DomXml.java
DomXml.attributeKeys()	0 (0.1%)	DomXml.java
DomXml.keys()	0 (0.1%)	DomXml.java
DomXml.<init>()	0 (0.1%)	DomXml.java
DomXml.mergeIn(DomXml)	0 (0.1%)	DomXml.java
DomXml.privPut(PpKey, Object, boolean)	0 (0.1%)	DomXml.java



3.8.1.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Name	Average Cumulative Time	Source
XMLHelper.<init>()	15 (15.3%)	XMLHelper.java
XMLHelper.parse(String)	5 (5.0%)	XMLHelper.java
DomXml.loadXML(InputStream)	5 (4.8%)	DomXml.java
DomXml.fromXML(InputStream)	4 (4.2%)	DomXml.java
XMLHelper.searchDom(String, DomXml)	1 (0.9%)	XMLHelper.java
DomXml.mergeIn(DomXml)	1 (0.6%)	DomXml.java
DomXml.elementToObject(Element)	1 (0.5%)	DomXml.java
DomXml.attributeKeys()	0 (0.3%)	DomXml.java
DomXml.elementKeys()	0 (0.3%)	DomXml.java
DomXml.privAdd(PpKey, Object)	0 (0.2%)	DomXml.java

The results above and below do not present any surprises and are consistent with the expected results based on evaluation of the previous performance metrics.



3.8.1.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Name	Avg. Method Object	Source
DomXml.elementToObject(Element)	206 (26.1%)	DomXml.java
PpKey.hashCode()	144 (18.3%)	PpKey.java
DomXml.attributeKeys()	122 (15.5%)	DomXml.java
DomXml.elementKeys()	122 (15.5%)	DomXml.java
DomXml.privPut(PpKey, Object, boolean)	84 (10.6%)	DomXml.java
DomXml.<init>(String, String)	42 (5.3%)	DomXml.java
XMLHelper.parse(String)	24 (3.0%)	XMLHelper.java
DomXml.keys()	12 (1.5%)	DomXml.java
XMLHelper.<init>()	8 (1.0%)	XMLHelper.java
DomXml.fromXML(InputStream)	8 (1.0%)	DomXml.java

3.8.1.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Name	Average Cumulative Object	Source
XMLHelper.parse(String)	44 (5.6%)	XMLHelper.java
DomXml.loadXML(InputStream)	41 (5.2%)	DomXml.java
DomXml.fromXML(InputStream)	38 (4.8%)	DomXml.java
XMLHelper.<init>()	23 (2.9%)	XMLHelper.java
XMLHelper.searchDom(String, DomXml)	8 (1.0%)	XMLHelper.java



Name	Average Cumulative Object	Source
DomXml.elementToObject(Element)	5 (0.6%)	DomXml.java
DomXml.attributeKeys()	3 (0.4%)	DomXml.java
DomXml.elementKeys()	3 (0.4%)	DomXml.java
DomXml.mergeIn(DomXml)	3 (0.4%)	DomXml.java
DomXml.privPut(PpKey, Object, boolean)	1 (0.1%)	DomXml.java

3.82 SaxTest.jsp Scenario

3.8.2.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Name	Calls	Source
SaxSearchHandler.characters(char[], int, int)	130	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	70	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	70	SaxSearchHandler.java
SaxSearchHandler.startElement(String, String, String, Attributes)	70	SaxSearchHandler.java
XMLHelper.parse(String, SaxHandlers)	10	XMLHelper.java
XMLHelper.searchSax(String, String)	10	XMLHelper.java
SaxHandlers.<init>()	10	SaxHandlers.java
SaxHandlers.endDocument()	10	SaxHandlers.java
SaxHandlers.startDocument()	10	SaxHandlers.java
SaxSearchHandler.<init>()	10	SaxSearchHandler.java

The Sax parser calls established methods depending upon what it is parsing within the XML document. For example the sax parser will call the method *startElement(String, String, String, Attributes)* if the parser is parsing the start of a element. The



method `characters(char[], int, int)` is called everytime a character is encountered in the XML document. This would explain why this method has the highest amount of calls

3.8.2.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Name	Method Time	Source
<code>XMLHelper.parse(String, SaxHandlers)</code>	5 (16.0%)	<code>XMLHelper.java</code>
<code>XMLHelper.searchSax(String, String)</code>	4 (13.5%)	<code>XMLHelper.java</code>
<code>SaxXml.parse(String, SaxHandlers)</code>	3 (10.3%)	<code>SaxXml.java</code>
<code>XMLHelper.<init>()</code>	2 (5.4%)	<code>XMLHelper.java</code>
<code>SaxSearchHandler.startElement(String, String, String, Attributes)</code>	1 (3.3%)	<code>SaxSearchHandler.java</code>
<code>SaxSearchHandler.characters(char[], int, int)</code>	1 (1.7%)	<code>SaxSearchHandler.java</code>
<code>SaxHandlers.startDocument()</code>	0 (1.4%)	<code>SaxHandlers.java</code>
<code>SaxHandlers.<init>()</code>	0 (1.3%)	<code>SaxHandlers.java</code>
<code>SaxXml.<init>()</code>	0 (1.3%)	<code>SaxXml.java</code>
<code>SaxSearchHandler.endElement(String, String, String)</code>	0 (0.7%)	<code>SaxSearchHandler.java</code>

Due to the fact that the Sax parser is dependent upon the developer in providing a handler class that provides the implementation of the parsing methods, the *parse(String, SaxHandlers)* method for the SAX API is a lot more active then the other two XMLHelper API parse methods. The *parse(String, SaxHandlers)* method takes the handler class as a argument and instantiates it. This will mean that *parse(String, SaxHandlers)* will be more active and more objects associated with it.



3.8.2.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Name	Cumulative Time	Source
XMLHelper.parse(String, SaxHandlers)	5 (43.8%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	3 (28.2%)	SaxXml.java
SaxSearchHandler.startElement(String, String, String, Attributes)	1 (9.0%)	SaxSearchHandler.java
SaxSearchHandler.characters(char[], int, int)	1 (4.7%)	SaxSearchHandler.java
SaxHandlers.startDocument()	0 (3.9%)	SaxHandlers.java
SaxXml.<init>()	0 (3.4%)	SaxXml.java
SaxSearchHandler.endElement(String, String, String)	0 (1.8%)	SaxSearchHandler.java
SaxHandlers.endDocument()	0 (1.8%)	SaxHandlers.java
SaxSearchHandler.getSearchName()	0 (1.7%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.3%)	SaxSearchHandler.java

The framework entry point is the *parse(String, SaxHandlers)* method. It would be expected that *parse(String, SaxHandlers)* would be the longest cumulative time method.

3.8.2.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Name	Method Objects	Source
XMLHelper.parse(String, SaxHandlers)	32 (41.0%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	32 (41.0%)	SaxXml.java
SaxHandlers.startDocument()	6 (7.7%)	SaxHandlers.java
SaxXml.<init>()	4 (5.1%)	SaxXml.java



Name	Method Objects	Source
SaxSearchHandler.startElement(String, String, String, Attributes)	2 (2.6%)	SaxSearchHandler.java
SaxHandlers.endDocument()	2 (2.6%)	SaxHandlers.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java

As explained in the method time area, SAX parsing uses a Handler class and where in DOM the *elementToObject(Element)* was the heaviest used method, in the SAX API most of the work is done in the parse method.

3.8.2.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Name	Cumulative Objects	Source
XMLHelper.parse(String, SaxHandlers)	78 (100.0%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	42 (53.8%)	SaxXml.java
SaxHandlers.startDocument()	6 (7.7%)	SaxHandlers.java
SaxXml.<init>()	4 (5.1%)	SaxXml.java
SaxSearchHandler.startElement(String, String, String, Attributes)	2 (2.6%)	SaxSearchHandler.java
SaxHandlers.endDocument()	2 (2.6%)	SaxHandlers.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java



3.8.2.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Name	Avg. Method Time	Source
XMLHelper.parse(String, SaxHandlers)	0 (4.4%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	0 (2.8%)	SaxXml.java
SaxHandlers.startDocument()	0 (0.4%)	SaxHandlers.java
SaxXml.<init>()	0 (0.3%)	SaxXml.java
SaxHandlers.endDocument()	0 (0.2%)	SaxHandlers.java
SaxSearchHandler.startElement(String, String, String, Attributes)	0 (0.1%)	SaxSearchHandler.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java



3.8.2.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Name	Average Cumulative Time	Source
XMLHelper.parse(String, SaxHandlers)	1 (10.0%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	1 (5.1%)	SaxXml.java
SaxXml.<init>()	0 (0.4%)	SaxXml.java
SaxHandlers.startDocument()	0 (0.4%)	SaxHandlers.java
SaxHandlers.endDocument()	0 (0.2%)	SaxHandlers.java
SaxSearchHandler.startElement(String, String, String, Attributes)	0 (0.2%)	SaxSearchHandler.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java

3.8.2.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Name	Avg. Method Object	Source
XMLHelper.parse(String, SaxHandlers)	3 (3.8%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	3 (3.8%)	SaxXml.java
SaxXml.<init>()	0 (0.0%)	SaxXml.java
SaxHandlers.startDocument()	0 (0.0%)	SaxHandlers.java
SaxHandlers.endDocument()	0 (0.0%)	SaxHandlers.java



Name	Avg. Method Object	Source
SaxSearchHandler.startElement(String, String, String, Attributes)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java

3.8.2.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Name	Average Cumulative Object	Source
XMLHelper.parse(String, SaxHandlers)	7 (9.0%)	XMLHelper.java
SaxXml.parse(String, SaxHandlers)	4 (5.1%)	SaxXml.java
SaxXml.<init>()	0 (0.0%)	SaxXml.java
SaxHandlers.startDocument()	0 (0.0%)	SaxHandlers.java
SaxHandlers.endDocument()	0 (0.0%)	SaxHandlers.java
SaxSearchHandler.startElement(String, String, String, Attributes)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.characters(char[], int, int)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.setSearchValue(String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.endElement(String, String, String)	0 (0.0%)	SaxSearchHandler.java
SaxSearchHandler.getSearchName()	0 (0.0%)	SaxSearchHandler.java

3.8.3 BindTest.jsp Scenario

The XMLHelper framework does include the open source CASTOR framework to accomplish marshalling XML document to a Java object. This means for this test, we filtered JPROBE on package names that included *xmlhelper* as well *exolab*.



3.8.3.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Name	Calls	Source
List.size()	7,590	List.java
List.get(int)	4,140	List.java
ValidationUtils.isLetter(char)	4,120	ValidationUtils.java
XMLFieldDescriptorImpl.getHandler()	3,700	XMLFieldDescriptorImpl.java
XMLFieldDescriptorImpl.isReference()	3,130	XMLFieldDescriptorImpl.java
List.add(Object)	2,340	List.java
XMLFieldDescriptorImpl.getValidator()	2,320	XMLFieldDescriptorImpl.java
FieldValidator.validate(Object, ClassDescriptorResolver)	2,320	FieldValidator.java
XMLFieldDescriptorImpl.isRequired()	2,270	XMLFieldDescriptorImpl.java
MarshalFramework.isPrimitive(Class)	2,240	MarshalFramework.java

The marshalling technology that CASTOR uses to instantiate an Object uses Lists to move attributes and data around. A high use of LIST methods is expected.

3.8.3.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Name	Method Time	Source
XMLHelper.parse(String, String)	8 (24.0%)	XMLHelper.java
DataBind.parse(String, String)	7 (21.2%)	DataBind.java
XMLHelper.<init>()	2 (5.9%)	XMLHelper.java
DataBind.<init>()	0 (1.2%)	DataBind.java
DomXml.<init>()	0 (0.5%)	DomXml.java



Name	Method Time	Source
List.size()	0 (0.0%)	List.java
List.get(int)	0 (0.0%)	List.java
ValidationUtils.isLetter(char)	0 (0.0%)	ValidationUtils.java
XMLFieldDescriptorImpl.getHandler()	0 (0.0%)	XMLFieldDescriptorImpl.java
XMLFieldDescriptorImpl.isReference()	0 (0.0%)	XMLFieldDescriptorImpl.java

The only methods available to XMLHelper using the DataBind API are *XMLHelper.parse(String, String)* and *XMLHelper.write(String, String)*. The marshalling and demarshalling from CASTOR are accomplished in the parse methods and thus are expected to be the high use methods in this test scenario



3.8.3.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Name	Cumulative Time	Source
XMLHelper.<init>()	18 (52.1%)	XMLHelper.java
XMLHelper.parse(String, String)	17 (47.9%)	XMLHelper.java
DataBind.parse(String, String)	8 (22.3%)	DataBind.java
DataBind.<init>()	0 (1.4%)	DataBind.java
Unmarshaller.unmarshal(InputSource)	0 (0.9%)	Unmarshaller.java
DomXml.<init>()	0 (0.5%)	DomXml.java
UnmarshalHandler.endElement(String)	0 (0.5%)	UnmarshalHandler.java
FieldHandlerImpl.setValue(Object, Object)	0 (0.5%)	FieldHandlerImpl.java
UnmarshalHandler.startElement(String, AttributeList)	0 (0.4%)	UnmarshalHandler.java
List.size()	0 (0.0%)	List.java

3.8.3.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Name	Method Objects	Source
DataBind.parse(String, String)	66 (49.6%)	DataBind.java
XMLHelper.parse(String, String)	40 (30.1%)	XMLHelper.java
XMLHelper.<init>()	8 (6.0%)	XMLHelper.java
DataBind.<init>()	4 (3.0%)	DataBind.java
DomXml.<init>()	1 (0.8%)	DomXml.java
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java
AccessMode.<init>(String)	0 (0.0%)	AccessMode.java



Name	Method Objects	Source
AccessMode.<clinit>()	0 (0.0%)	AccessMode.java
AccessType.toString()	0 (0.0%)	AccessType.java
AccessType.valueOf(String)	0 (0.0%)	AccessType.java

3.8.3.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Name	Cumulative Objects	Source
XMLHelper.parse(String, String)	110 (82.7%)	XMLHelper.java
DataBind.parse(String, String)	66 (49.6%)	DataBind.java
XMLHelper.<init>()	23 (17.3%)	XMLHelper.java
DataBind.<init>()	4 (3.0%)	DataBind.java
DomXml.<init>()	1 (0.8%)	DomXml.java
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java
AccessMode.<init>(String)	0 (0.0%)	AccessMode.java
AccessMode.<clinit>()	0 (0.0%)	AccessMode.java
AccessType.toString()	0 (0.0%)	AccessType.java
AccessType.valueOf(String)	0 (0.0%)	AccessType.java

3.8.3.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Name	Avg. Method Time	Source
XMLHelper.<init>()	2 (5.9%)	XMLHelper.java
XMLHelper.parse(String, String)	1 (2.4%)	XMLHelper.java



Name	Avg. Method Time	Source
DataBind.parse(String, String)	1 (2.1%)	DataBind.java
DomXml.<init>()	0 (0.5%)	DomXml.java
DataBind.<init>()	0 (0.1%)	DataBind.java
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java
AccessMode.<init>(String)	0 (0.0%)	AccessMode.java
AccessMode.<clinit>()	0 (0.0%)	AccessMode.java
AccessType.toString()	0 (0.0%)	AccessType.java
AccessType.valueOf(String)	0 (0.0%)	AccessType.java

3.8.3.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Name	Average Cumulative Time	Source
XMLHelper.<init>()	18 (52.1%)	XMLHelper.java
XMLHelper.parse(String, String)	2 (4.8%)	XMLHelper.java
DataBind.parse(String, String)	1 (2.2%)	DataBind.java
DomXml.<init>()	0 (0.5%)	DomXml.java
DataBind.<init>()	0 (0.1%)	DataBind.java
Unmarshaller.unmarshal(InputSource)	0 (0.0%)	Unmarshaller.java
FieldHandlerImpl.setValue(Object, Object)	0 (0.0%)	FieldHandlerImpl.java
UnmarshalHandler.endElement(String)	0 (0.0%)	UnmarshalHandler.java
UnmarshalHandler.startElement(String, AttributeList)	0 (0.0%)	UnmarshalHandler.java
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java



3.8.3.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Name	Avg. Method Object	Source
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java
UnmarshalHandler.startElement(String, AttributeList)	0 (0.0%)	UnmarshalHandler.java
UnmarshalHandler.endElement(String)	0 (0.0%)	UnmarshalHandler.java
FieldHandlerImpl.setValue(Object, Object)	0 (0.0%)	FieldHandlerImpl.java
Unmarshaller.unmarshal(InputSource)	0 (0.0%)	Unmarshaller.java
DataBind.<init>()	0 (0.0%)	DataBind.java
DomXml.<init>()	1 (0.8%)	DomXml.java
DataBind.parse(String, String)	6 (4.5%)	DataBind.java
XMLHelper.parse(String, String)	4 (3.0%)	XMLHelper.java
XMLHelper.<init>()	8 (6.0%)	XMLHelper.java

3.8.3.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Name	Average Cumulative Object	Source
XMLHelper.<init>()	23 (17.3%)	XMLHelper.java
XMLHelper.parse(String, String)	11 (8.3%)	XMLHelper.java
DataBind.parse(String, String)	6 (4.5%)	DataBind.java
DomXml.<init>()	1 (0.8%)	DomXml.java
DataBind.<init>()	0 (0.0%)	DataBind.java
Unmarshaller.unmarshal(InputSource)	0 (0.0%)	Unmarshaller.java
FieldHandlerImpl.setValue(Object, Object)	0 (0.0%)	FieldHandlerImpl.java



Name	Average Cumulative Object	Source
UnmarshalHandler.endElement(String)	0 (0.0%)	UnmarshalHandler.java
UnmarshalHandler.startElement(String, AttributeList)	0 (0.0%)	UnmarshalHandler.java
AccessMode.getAccessMode(String)	0 (0.0%)	AccessMode.java

3.9 General Performance Test Summary

All methods tested in the previous test scenarios executed very similar to each other and no one method stood out as being a performance problem or something that needed attention. Numbers of objects created per method were small and well distributed among the methods. No loitering objects or memory leaks were found in the heap at the end of each test cycle. Application groups using this RCS component should expect good performance low memory usage.



3.10 Appendix A

3.10.1 JProbe Configuration File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
    </application>
  </program>
  <applet
    working_dir=""
    source_dir=""
    htmlfile=""
    main_package="">
  </applet>
  <serverside
    suggested_filters=""
    id="Other server"
    server_dir="/opt/stg35/WebSphere/AppServer"
    prepend_to_vm_args=""
    source_dir=""
    classname="com.ibm.ejs.sm.util.process.Nanny"
    main_package="gov.ed.sfa.ita.xmlhelper"
    exclude_server_classes="true"
    args=""
    working_dir="/opt/stg35/WebSphere/AppServer/servlets"
    prepend_to_classpath="">
  </serverside>
  <classpath
    <classpath.path location="%CLASSPATH%"/>
  </classpath>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.52:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
      timing="elapsed"
    </performance>
  </analysis>
</jpl>
```



```
track_natives="true"
final_snapshot="true"
granularity="method">
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask="*"
  time="ignore"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask=" gov.ed.sfa.ita.xmlhelper.*"
  time="track"
  granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask="*" />
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask=".*" />
</threadalyzer>
```



ITA Release 3.0 Build & Test Report

```
<coverage
  record_from_start="true"
  final_snapshot="true"
  granularity="line">
  <coverage.filter
    visibility="invisible"
    methodmask="*"
    enabled="true"
    classmask="*" />
  <coverage.filter
    visibility="visible"
    methodmask="*"
    enabled="true"
    classmask=".*" />
</coverage>
</analysis>
</jpl>
```



3.11 Resources

- W3C Document Object Model specifications
 - <http://www.w3c.org/DOM/>
- IBM's Developer-Works
 - <http://www.ibm.com/developerworks/>
- XML Org
 - <http://www.xml.org/>
- Castor
 - <http://castor.exolab.org/>
- Sax Specifications
 - <http://www.saxproject.org/>



4 RCS – Scheduler Framework

4.1 Test Harness Design

4.1.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance test is to identify loitering objects and time spent on each method relative to each other in the schedule framework.

4.1.1.1 Testing Criteria

The main components of the Schedule framework that will be tested are:

- Adding a scheduled event that occurs at a specific time from a XML Document
- Adding a event that recursively occurs every minute from a XML document
- Checking that the event has been added using the schedule's framework method `containsAlarm()`.
- Removing the event using the schedule's framework method `removeAllAlarms()`.

Since the Schedule framework is an API, the JavaServer Pages developed for the unit test will serve as a test harness to profile and analyze the performance of the various methods.

4.1.2 Testing Configuration

In order to profile the Schedule framework with JProbe, the JPROBE Application Server configured in WebSphere was used and some of the configurations were changed. In the command line reference of the Application Server, there is a reference to the JProbe configuration file. The file used to conduct this performance analysis is: `/opt/util/JProbe/jpl_files/09122002_test_scheduler.jpl`. Due to the fact that some of the applications use STRUTS there are several servlets that are present in the configuration that are not needed for this test. Thus the action, database, and HelloWorld servlets were all disabled.

4.1.2.1 JProbe Configuration File

The JProbe configuration file has a file extension of `.jpl`. This file contains all of the settings that JProbe requires to profile an application, applet, or server side component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options. The user will be able to specify the activity of the Profiler. For example, the file can be configured to cause JProbe Profiler to take a heap snapshot before it exits and the directory to save the snapshots in.



The example application test will be conducted on the Solaris machine with the output being sent to a remote Windows NT workstation. The configuration in the actual file used to conduct the test can be found in [Appendix A](#). A filter for the main package, gov.ed.sfa.ita.schedule, was added to narrow the scope of the test to this package.

4.1.2.2 UNIX Server Settings

The current methodology that ITA uses to do performance testing of RCS packages is to run custom built test harnesses off a Application Server called JPROBE that is running in the stage environment of the development Solaris server. The server URI that is configured in the JPROBE Application server is **stg.jprobe.fsa.ed.gov**. The WebSphere JSP servlet Web path to call the test harnesses is JPROBEWebApp. So the following URL's should execute the three performance tests.

http:\\Stg.jprobe.fsa.ed.gov/JPROBEWebApp/scheduler/onetime.jsp
http:\\Stg.jprobe.fsa.ed.gov/JPROBEWebApp/scheduler/recurs.jsp

To accomplish the above URLs, the following WebSphere configuration files (located in **/opt/stg35/WebSphere/AppServer/temp**) were configured as documented.

4.1.2.2.1 rules.properties:

```
default_host/JPROBEWebApp/*.do=ibmoselink4
default_host/JPROBEWebApp/*.jsp=ibmoselink4
default_host/JPROBEWebApp/*.jsv=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/=ibmoselink4
default_host/JPROBEWebApp/ErrorReporter=ibmoselink4
default_host/JPROBEWebApp/servlet=ibmoselink4
default_host/JPROBEWebApp=ibmoselink4
```

4.1.2.2.2 queues.properties:

```
ose.srvgrp.ibmoselink4.clone1.port=8241
ose.srvgrp.ibmoselink4.clone1.type=remote
ose.srvgrp.ibmoselink4.clonescount=1
ose.srvgrp.ibmoselink4.type=FASTLINK
ose.srvgrp=ibmoselink3,ibmoselink2,ibmoselink4,ibmoselink17
```

vhosts.properties:

```
stg.jprobe.fsa.ed.gov=default_host
```

4.1.3 WebSphere Application Server Configuration

The WebSphere Command Line will identify the JProbe configuration file to use and ensure that the correct JVM is used. Two Environment Variables will be added to the Application Server to enable it to run with JPROBE.



4.1.3.1 Command line arguments:

-jp_input=/opt/util/JProbe/jpl_files/09122002_test_scheduler.jpl -Xnoclassgc -
Djava.compiler=NONE -ms128m -mx128m

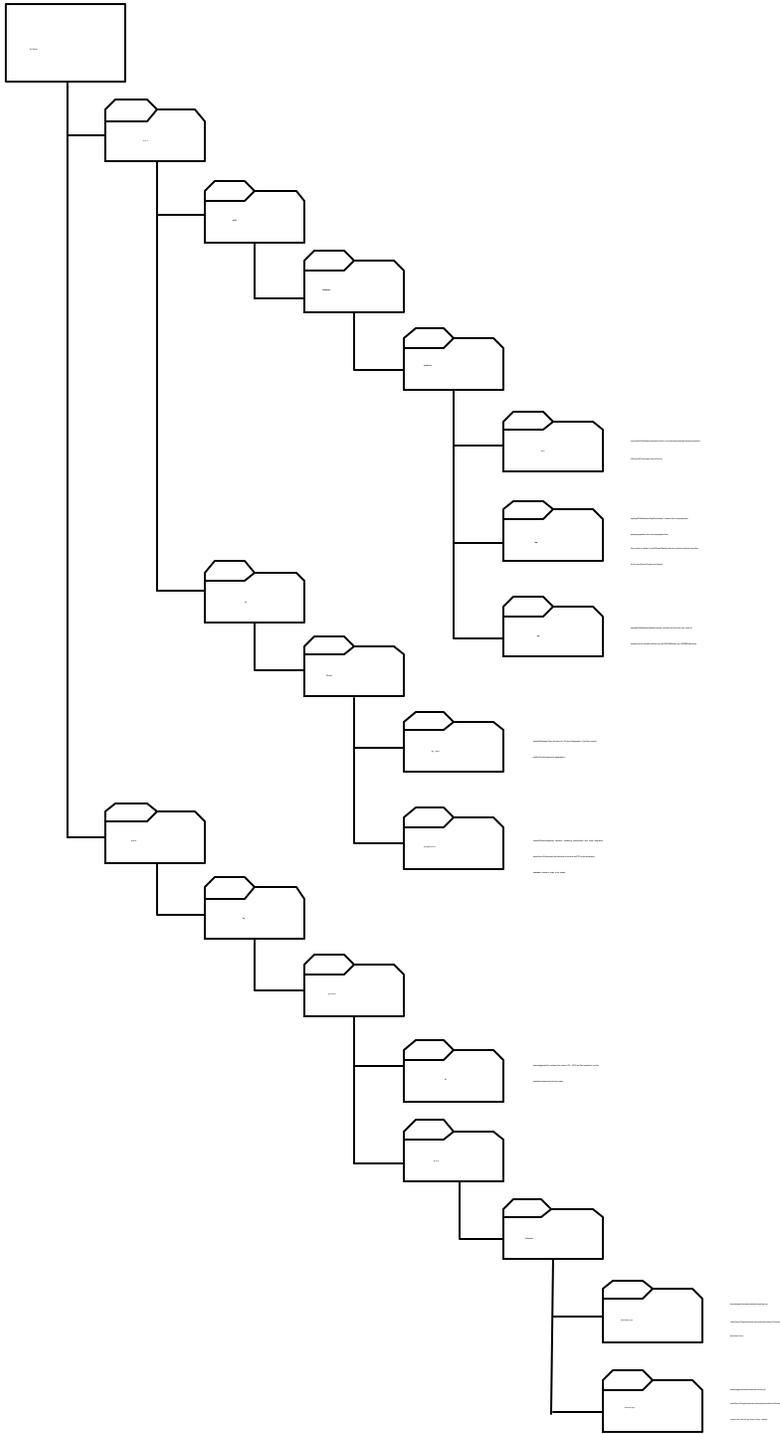
4.1.3.2 Environment:

EXECUTE=YES

EXECUTABLE=/opt/util/JProbe/profiler/jprun



4.1.4 Directory Structure





4.2 Testing Scenario

Test applications created for the unit test of the RCS framework will be used to execute the performance analysis. These test applications are actually Java Server Pages that access the Schedule framework to do work.

onetime.jsp—Currently coded to build a schedule object and then read in a specific timed schedule entry (launch method1() at 05:00:00) from two XML documents, onetimed.xml and onetimed.xml. The onetimed.xml has the mapping attribute parameters of the ScheduleEntry class and onetimed.xml has the data values for the ScheduleEntry object that the jsp builds. This JSP can be configured to loop multiple times so that multiple Schedule Entries can take place. In this test the loop was configured to 10 passes.

recurs.jsp— This JSP was coded to build a schedule object and then read in a schedule entry from two XML documents, recursm.xml and recursd.xml. that will recursively activate every minute. The recursm.xml has the mapping attribute parameters of the ScheduleEntry class and recursd.xml has the data values for the ScheduleEntry object that the jsp builds. By setting different data parameters within the data XML document (recursd.xml and onetimed.xml) that represents the data values for schedule entry is how the behavior is changed between recursive or one time type scheduling. This JSP can be configured to loop multiple times so that multiple Schedule Entries can take place. In this test the loop was configured to 10 passes.

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.

4.3 Results and Analysis

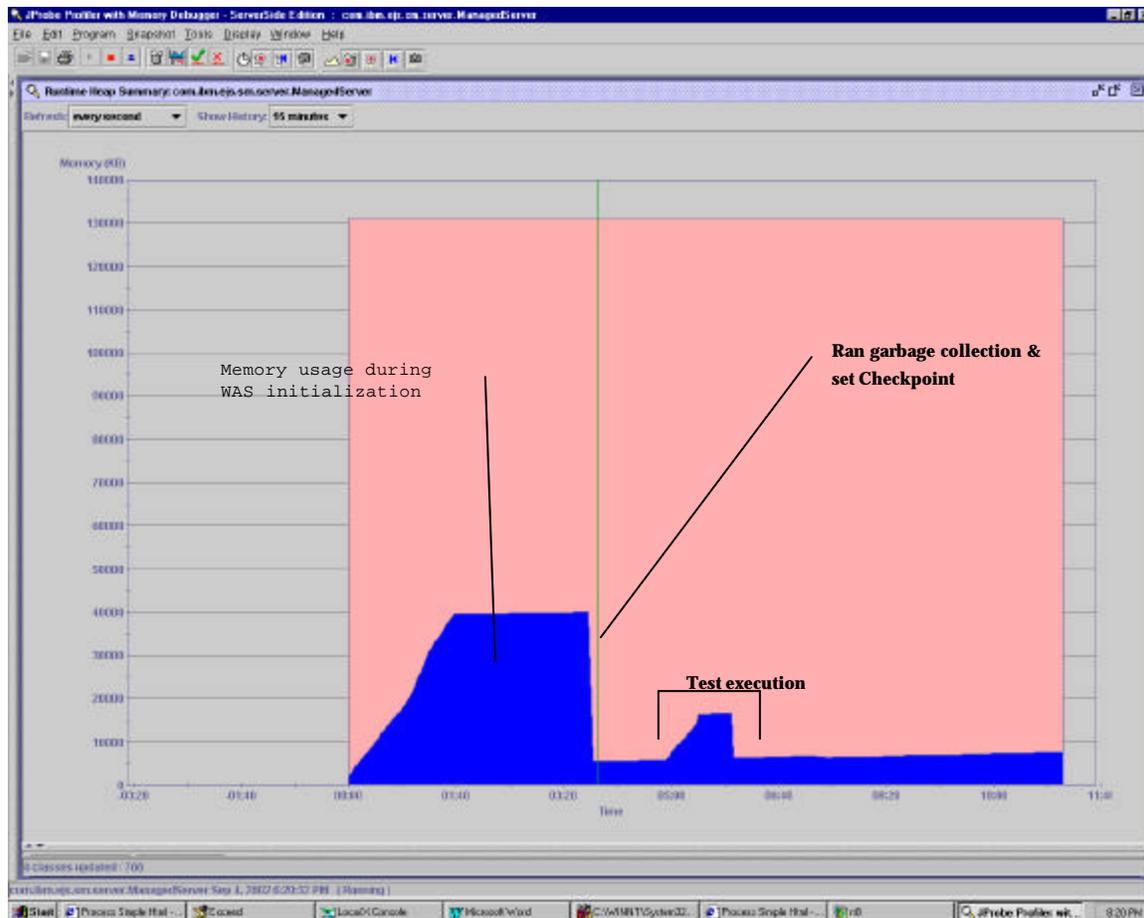
The JProbe Profiler with Memory Debugger application is used to trace both the memory usage and performance measurement of the schedule framework API. Two snapshots are taken for each test scenario: a heap snapshot and a performance snapshot. Each snapshot provides different information regarding our test.

4.4 Heap Snapshot (Memory Usage)

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.

4.4.1 Heap Graph Analysis

The screenshot below is obtained from executing recurs.jsp. It is the only heap graph screenshot depicted in this report since the heap graphs from executing other test cycle exhibit the same pattern.





In the graph above, it is possible to see that 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. By reading the graph, it can be determined that the overall memory usage for the schedule framework is very low and will not result in huge increase to the overhead of calling applications.

4.4.2 Instance Summary

The tables below represent Instance Summary result's associated with conducting the different test scenarios. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory (in bytes) those instances consume.

4.4.2.1 onetime.jsp

Package	Class	Count	Memory
gov.ed.sfa.ita.schedule	ScheduleEntry	40 (0.0%)	3,040 (0.0%)
fr.dyade.jdring	AlarmEntry	20 (0.0%)	1,040 (0.0%)
fr.dyade.jdring	AlarmWaiter	20 (0.0%)	560 (0.0%)
gov.ed.sfa.ita.schedule	Schedule	20 (0.0%)	240 (0.0%)
gov.ed.sfa.ita.schedule	Schedule\$2	20 (0.0%)	240 (0.0%)
scheduler_onetime_jsp_19	DomXml\$3	1 (0.0%)	20 (0.0%)

4.4.2.2 recurs.jsp

Package	Class	Count	Memory
gov.ed.sfa.ita.schedule	ScheduleEntry	20 (0.0%)	1,520 (0.0%)
fr.dyade.jdring	AlarmEntry	10 (0.0%)	520 (0.0%)
fr.dyade.jdring	AlarmWaiter	10 (0.0%)	280 (0.0%)
gov.ed.sfa.ita.schedule	Schedule	10 (0.0%)	120 (0.0%)



4.5 Performance Snapshot (Code Efficiency)

There are nine efficiency metrics that can be collected using JProbe – five basic metrics and four compound metrics. The basic metrics include: number of calls, method time, cumulative time, method object count, and cumulative object count. The compound metrics are averages per number of calls, including: average method time, average cumulative time, average method object count, and average cumulative object count. Time is measured as elapsed time in milliseconds.

The following sections will describe each metric and display the top results for each measurement for the performance assessment of the Schedule framework. These metrics are basic indicators of process resource utilization. The detailed graphs associated with each method can be reviewed for unexpected activity or optimization opportunities.

All performance metric results were first filtered by `*schedule*` to obtain only the classes within the Schedule framework which is what the test is looking for. Since the schedule also includes the package `jdrring` we also filtered for that. Then for each section, the results were sorted by the metric under investigation to obtain the top ten results for each metric.

4.5.1 onetime.jsp Scenario

4.5.1.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Name	Calls	Source
AlarmWaiter.debug(String)	80	AlarmWaiter.java
ScheduleEntry.<init>()	40	ScheduleEntry.java
AlarmManager.debug(String)	40	AlarmManager.java
AlarmEntry.debug(String)	40	AlarmEntry.java
_onetime_jsp_19._jspx_writeString(JspWriter, String)	36	_onetime_jsp_19.java



Name	Calls	Source
_onetime_jsp_19._jspx_writeString(JspWriter, char[])	36	_onetime_jsp_19.java
Schedule.configureXML(String, String)	20	Schedule.java
ScheduleEntry.setArg0(Object)	20	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	20	Schedule.java
Schedule.<init>()	20	Schedule.java

4.5.1.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Name	Method Time	Source
Schedule.configureXML(String, String)	22 (42.4%)	Schedule.java
ScheduleEntry.setArg0(Object)	5 (9.8%)	ScheduleEntry.java
ScheduleEntry.<init>()	5 (9.3%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	2 (4.3%)	Schedule.java
Schedule.<init>()	2 (4.3%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.9%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.8%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.4%)	Schedule.java
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_onetime_jsp_19.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java



4.5.1.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Name	Cumulative Time	Source
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	51 (100.0%)	_onetime_jsp_19.java
Schedule.configureXML(String, String)	32 (63.1%)	Schedule.java
Schedule.<init>()	15 (28.7%)	Schedule.java
ScheduleEntry.setArg0(Object)	5 (10.4%)	ScheduleEntry.java
ScheduleEntry.<init>()	5 (10.0%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	3 (5.0%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	1 (1.1%)	Schedule.java
Schedule.removeAllAlarms()	1 (1.0%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.4%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.1.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Name	Method Objects	Source
ScheduleEntry.<init>()	88 (31.0%)	ScheduleEntry.java
Schedule.configureXML(String, String)	78 (27.5%)	Schedule.java
ScheduleEntry.setArg0(Object)	68 (23.9%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	24 (8.5%)	Schedule.java



Name	Method Objects	Source
Schedule.<init>()	8 (2.8%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	2 (0.7%)	Schedule.java
Schedule.removeAllAlarms()	2 (0.7%)	Schedule.java
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_onetime_jsp_19.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.1.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Name	Cumulative Objects	Source
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	284 (100.0%)	_onetime_jsp_19.java
Schedule.configureXML(String, String)	210 (73.9%)	Schedule.java
ScheduleEntry.<init>()	88 (31.0%)	ScheduleEntry.java
ScheduleEntry.setArg0(Object)	68 (23.9%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	24 (8.5%)	Schedule.java
Schedule.<init>()	22 (7.7%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	2 (0.7%)	Schedule.java
Schedule.removeAllAlarms()	2 (0.7%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java



4.5.1.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Name	Avg. Method Time	Source
Schedule.configureXML(String, String)	1 (2.1%)	Schedule.java
ScheduleEntry.setArg0(Object)	0 (0.5%)	ScheduleEntry.java
ScheduleEntry.<init>()	0 (0.2%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	0 (0.2%)	Schedule.java
Schedule.<init>()	0 (0.2%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.0%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.0%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_onetime_jsp_19.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.1.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Name	Average Cumulative Time	Source
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	25 (50.0%)	_onetime_jsp_19.java



Name	Average Cumulative Time	Source
Schedule.configureXML(String, String)	2 (3.2%)	Schedule.java
Schedule.<init>()	1 (1.4%)	Schedule.java
ScheduleEntry.setArg0(Object)	0 (0.5%)	ScheduleEntry.java
ScheduleEntry.<init>()	0 (0.2%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	0 (0.2%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.1%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.1%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.1.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Name	Avg. Method Object	Source
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	25 (50.0%)	_onetime_jsp_19.java
Schedule.configureXML(String, String)	2 (3.2%)	Schedule.java
Schedule.<init>()	1 (1.4%)	Schedule.java
ScheduleEntry.setArg0(Object)	0 (0.5%)	ScheduleEntry.java
ScheduleEntry.<init>()	0 (0.2%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	0 (0.2%)	Schedule.java



Name	Avg. Method Object	Source
Schedule.containsAlarm(ScheduleEntry)	0 (0.1%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.1%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.1.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Name	Average Cumulative Object	Source
_onetime_jsp_19._jspService(HttpServletRequest, HttpServletResponse)	142 (50.0%)	_onetime_jsp_19.java
Schedule.configureXML(String, String)	10 (3.5%)	Schedule.java
ScheduleEntry.setArg0(Object)	3 (1.1%)	ScheduleEntry.java
ScheduleEntry.<init>()	2 (0.7%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	1 (0.4%)	Schedule.java
Schedule.<init>()	1 (0.4%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.0%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.0%)	Schedule.java
Schedule\$2.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java



4.5.2 recurs.jsp Scenario

4.5.2.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Name	Calls	Source
AlarmWaiter.debug(String)	40	AlarmWaiter.java
AlarmManager.debug(String)	20	AlarmManager.java
ScheduleEntry.<init>()	20	ScheduleEntry.java
_recurs_jsp_1._jspx_writeString(JspWriter, String)	18	_recurs_jsp_1.java
_recurs_jsp_1._jspx_writeString(JspWriter, char[])	18	_recurs_jsp_1.java
Schedule.<init>()	10	Schedule.java
Schedule.addAlarm(ScheduleEntry)	10	Schedule.java
Schedule.configureXML(String, String)	10	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	10	Schedule.java
Schedule.removeAllAlarms()	10	Schedule.java

4.5.2.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).



Name	Method Time	Source
Schedule.configureXML(String, String)	19 (45.7%)	Schedule.java
ScheduleEntry.setArg0(Object)	3 (7.1%)	ScheduleEntry.java
ScheduleEntry.<init>()	2 (5.6%)	ScheduleEntry.java
Schedule.<init>()	2 (5.3%)	Schedule.java
Schedule.addAlarm(ScheduleEntry)	1 (3.1%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.7%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.6%)	Schedule.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.3%)	Schedule.java
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_recurs_jsp_1.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Name	Cumulative Time	Source
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	42 (100.0%)	_recurs_jsp_1.java
Schedule.configureXML(String, String)	25 (60.1%)	Schedule.java
Schedule.<init>()	15 (34.6%)	Schedule.java
ScheduleEntry.setArg0(Object)	3 (7.4%)	ScheduleEntry.java
ScheduleEntry.<init>()	3 (6.0%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	2 (3.6%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.8%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.7%)	Schedule.java



Name	Cumulative Time	Source
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.3%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Name	Method Objects	Source
Schedule.configureXML(String, String)	58 (29.9%)	Schedule.java
ScheduleEntry.setArg0(Object)	48 (24.7%)	ScheduleEntry.java
ScheduleEntry.<init>()	48 (24.7%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	14 (7.2%)	Schedule.java
Schedule.<init>()	8 (4.1%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	2 (1.0%)	Schedule.java
Schedule.removeAllAlarms()	2 (1.0%)	Schedule.java
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_recurs_jsp_1.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.



Name	Cumulative Objects	Source
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	194 (100.0%)	_recurs_jsp_1.java
Schedule.configureXML(String, String)	140 (72.2%)	Schedule.java
ScheduleEntry.setArg0(Object)	48 (24.7%)	ScheduleEntry.java
ScheduleEntry.<init>()	48 (24.7%)	ScheduleEntry.java
Schedule.<init>()	22 (11.3%)	Schedule.java
Schedule.addAlarm(ScheduleEntry)	14 (7.2%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	2 (1.0%)	Schedule.java
Schedule.removeAllAlarms()	2 (1.0%)	Schedule.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Name	Avg. Method Time	Source
Schedule.configureXML(String, String)	2 (4.6%)	Schedule.java
ScheduleEntry.setArg0(Object)	0 (0.7%)	ScheduleEntry.java
Schedule.<init>()	0 (0.5%)	Schedule.java
Schedule.addAlarm(ScheduleEntry)	0 (0.3%)	Schedule.java
ScheduleEntry.<init>()	0 (0.3%)	ScheduleEntry.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.1%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.1%)	Schedule.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java



Name	Avg. Method Time	Source
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_recurs_jsp_1.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Name	Average Cumulative Time	Source
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	42 (100.0%)	_recurs_jsp_1.java
Schedule.configureXML(String, String)	3 (6.0%)	Schedule.java
Schedule.<init>()	1 (3.5%)	Schedule.java
ScheduleEntry.setArg0(Object)	0 (0.7%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	0 (0.4%)	Schedule.java
ScheduleEntry.<init>()	0 (0.3%)	ScheduleEntry.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.1%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.1%)	Schedule.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.



Name	Avg. Method Object	Source
Schedule.configureXML(String, String)	5 (2.6%)	Schedule.java
ScheduleEntry.setArg0(Object)	4 (2.1%)	ScheduleEntry.java
ScheduleEntry.<init>()	2 (1.0%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	1 (0.5%)	Schedule.java
Schedule.<init>()	0 (0.0%)	Schedule.java
Schedule.containsAlarm(ScheduleEntry)	0 (0.0%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.0%)	Schedule.java
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	0 (0.0%)	_recurs_jsp_1.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.5.2.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Name	Average Cumulative Object	Source
_recurs_jsp_1._jspService(HttpServletRequest, HttpServletResponse)	194 (100.0%)	_recurs_jsp_1.java
Schedule.configureXML(String, String)	14 (7.2%)	Schedule.java
ScheduleEntry.setArg0(Object)	4 (2.1%)	ScheduleEntry.java
Schedule.<init>()	2 (1.0%)	Schedule.java
ScheduleEntry.<init>()	2 (1.0%)	ScheduleEntry.java
Schedule.addAlarm(ScheduleEntry)	1 (0.5%)	Schedule.java



Name	Average Cumulative Object	Source
Schedule.containsAlarm(ScheduleEntry)	0 (0.0%)	Schedule.java
Schedule.removeAllAlarms()	0 (0.0%)	Schedule.java
Schedule\$1.<init>(Schedule, ScheduleEntry)	0 (0.0%)	Schedule.java
AlarmWaiter.debug(String)	0 (0.0%)	AlarmWaiter.java

4.6 General Performance Test Summary

All methods tested in the previous test scenarios executed very similar to each other and no one method stood out as being a performance problem or something that needed attention. Numbers of objects created per method were small and well distributed among the methods. No loitering objects or memory leaks were found in the heap at the end of each test cycle. Application groups using this RCS component should expect good performance low memory usage.



4.7 Appendix A

4.7.1 JProbe Configuration File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
      <classpath/>
    </application>
    <applet
      working_dir=""
      source_dir=""
      htmlfile=""
      main_package="">
      <classpath>
        <classpath.path location="%CLASSPATH%"/>
      </classpath>
    </applet>
    <serverside
      suggested_filters=""
      id="Other server"
      server_dir="/opt/stg35/WebSphere/AppServer"
      prepend_to_vm_args=""
      source_dir=""
      classname="com.ibm.ejs.sm.util.process.Nanny"
      main_package="gov.ed.sfa.ita.schedule"
      exclude_server_classes="true"
      args=""
      working_dir="/opt/stg35/WebSphere/AppServer/servlets"
      prepend_to_classpath="">
      <classpath>
        <classpath.path location="%CLASSPATH%"/>
      </classpath>
    </serverside>
  </program>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.52:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
      timing="elapsed"
      track_natives="true"
      final_snapshot="true"
      granularity="method">
```



```
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask="*"
  time="ignore"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask=" gov.ed.sfa.ita.schedule.*"
  time="track"
  granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask="*/>
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask=".*"/>
</threadalyzer>
<coverage
  record_from_start="true"
  final_snapshot="true"
  granularity="line">
```



```
<coverage.filter
  visibility="invisible"
  methodmask="*"
  enabled="true"
  classmask="*" />
<coverage.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask=".*" />
</coverage>
</analysis>
</jpl>
```

4.8 Resources

- IBM's Developer-Works
<http://www.ibm.com/developerworks/>
- JDRing Web Site
<http://webtools.dyade.fr/jdring/>
- Castor
<http://castor.exolab.org/>

5 RCS – Session Framework

5.1 Unit Test Report

5.2 Purpose

This Unit Test Report documents the test conditions and test script of the ITA R3.0 Reusable Common Services (RCS) User Session framework. This report also provides the expected results and actual results from running the test script.

5.3 Approach

To ensure quality of the RCS, the User Session framework went through extensive unit testing. ITA conducted manual unit testing of this framework.

Benefits to the unit test approach are:

- Standardize test conditions and cycles
- Increase code quality
- Increase consistency in the approach to testing
- Increase productivity
- Reduce time for regression testing
- More time available to spend on enhancements as less time is required for fixes



5.4 Background

The purpose of the ITA User Session framework is to provide a standard to simplify, standardize, and extend the use of user session/context information within the J2EE standard. The session framework will provide a common way to access session information. The framework will decouple session information from the request, session, and cookie contexts; and it will wrap WebSphere session extension classes.

5.5 Test Design

5.5.1 Testing Environment

The unit test for the User Session framework will be conducted manually. The unit test will be conducted on a Sun SPARC machine running Solaris 2.6 interacting with a client browser running on a Windows 2000 machine. Both Microsoft Internet Explorer 5.0.1 and Netscape Navigator 6.2 client browsers were used to conduct the tests scripts. The focus of this unit test is to identify that the User Session framework is functioning as designed.

5.5.2 Testing Cycles

The User Session framework is designed to retrieve user session information stored in cookies on the client (user's) machine or in the session object stored on the application server. There are many possible configurations to WebSphere's Session Manager that affects how the User Session framework will be used.

The following test cycles will be conducted to test the different scenarios the application developer could encounter:

Cycle Number	Type	Storage Type	Notes
1	Normal	Cookie	-
2	Normal	HttpSession	Variable
3	Normal	HttpSession	Persistent
4	Normal	IBMSession	Manual update method not called
5	Normal	IBMSession	Manual update method called
6	Exception	Session	Does not matter if HttpSession or IBMSession

Developers could encounter the scenarios above due to the complexity of managing user session state and using WebSphere Session Manager. The session object can be maintained in the server's cache (memory) or in a database depending on how WebSphere's Session Manager is configured. Storing user session information in a database (persistence) is required if the application is configured for cloning so that a user can be directed to different servers but still be able to access his or her session information. The Session Manager offers an IBMSession interface that extends HttpSession and provides the capability to manually request the session information be persisted to the database. If persistence is enabled and manual update is not enabled, the user information is persisted at the end of every servlet's service() method call.



5.5.3 Testing Configuration

In order to test the User Session framework, several JavaServer Pages had to be developed to utilize the User Session framework classes. An existing development Application Server (CONV) was used to conduct the tests, with some modification to the Session Manager settings and directory structure. The UNIX server settings were not modified and are listed below for reference only.

5.5.3.1 UNIX Server Settings

The usage of the User Session framework is closely tied to how the WebSphere Session Manager is configured. The WebSphere properties files have not been updated and the existing settings are used to run the test cycles.

The following sections list the properties related to the Web Application created to unit test the User Session framework. The configuration settings used in the Administration Console is defined in the next topic.

5.5.3.1.1 rules.properties:

```
default_host/CONVWebApp/*.activity=ibmoselink15
default_host/CONVWebApp/*.jsp=ibmoselink15
default_host/CONVWebApp/ErrorReporter=ibmoselink15
default_host/CONVWebApp/servlet/=ibmoselink15
default_host/CONVWebApp/servlet/messagerouter=ibmoselink15
default_host/CONVWebApp/servlet/rpcrouter=ibmoselink15
default_host/CONVWebApp/servlet=ibmoselink15
```

5.5.3.1.2 queues.properties:

```
ose.srvgrp.ibmoselink15.clone1.port=8400
ose.srvgrp.ibmoselink15.clone1.type=remote
ose.srvgrp.ibmoselink15.clonescount=1
ose.srvgrp.ibmoselink15.type=FASTLINK
ose.srvgrp=ibmoselink,ibmoselink1,ibmoselink2,ibmoselink3,ibmoselink4,ibmoselink5,ibmoselink6,ibmoselink7,ibmoselink8,ibmoselink9,ibmoselink10,ibmoselink11,ibmoselink12,ibmoselink13,ibmoselink14,ibmoselink15,ibmoselink16,ibmoselink17,ibmoselink18
```

5.5.3.1.3 vhosts.properties:

```
dev.conv.sfa.ed.gov:8531=default_host
```

5.5.3.2 WebSphere Application Server – Session Manager Configuration

The WebSphere Application Server level and Web Application level properties were not changed to conduct this unit test. The Session Manager settings had to be changed for each test according to the setup instructions for that test cycle.

The follow screenshots demonstrate the different configuration options for the Session Manager in the WebSphere Administration Console. The circled settings are options that will be

modified during the test cycles. If an option is not listed in the setup for the test cycle, then the current setting of that option is not important to the test and can be left as is.

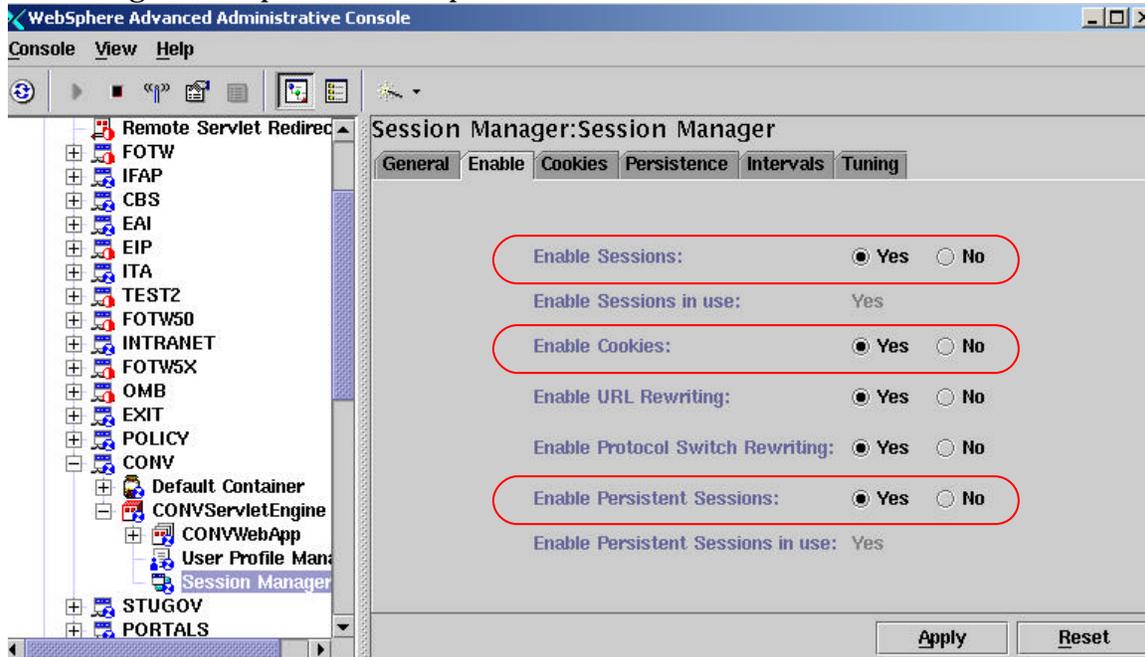


Figure 2: Enable Tab

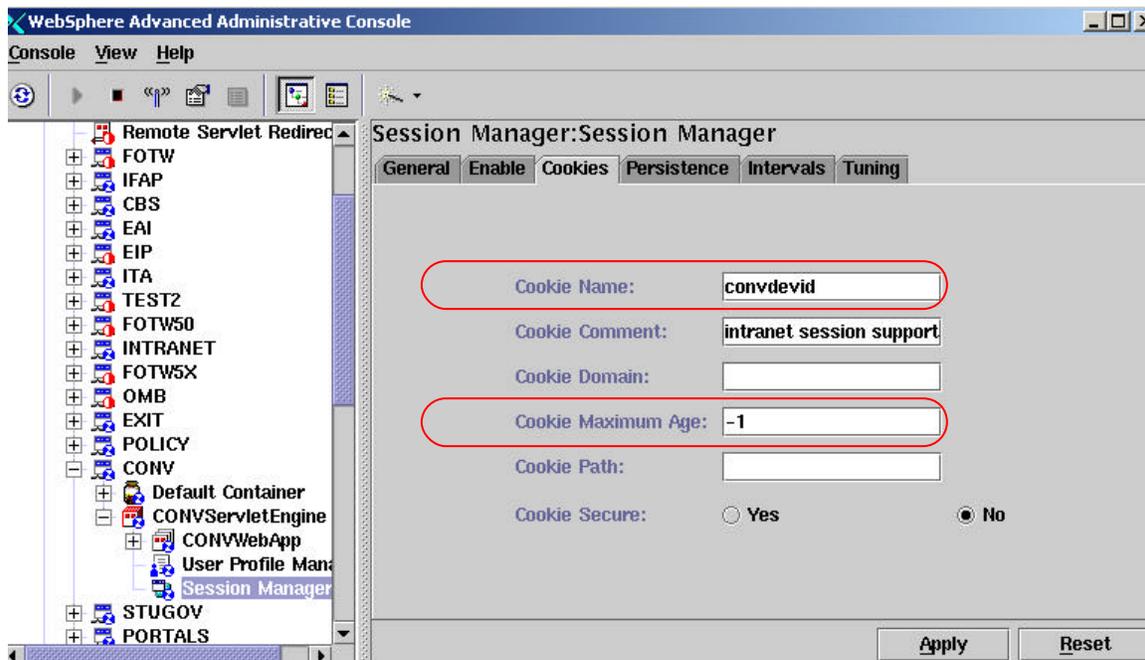


Figure 3: Cookies Tab

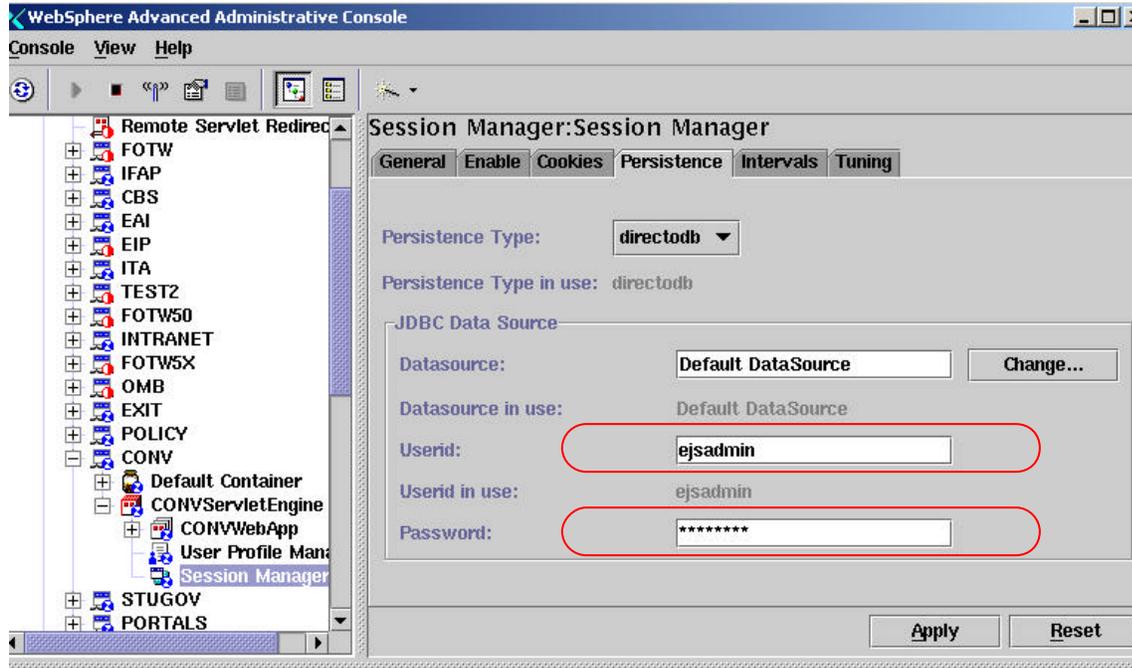


Figure 4: Persistence Tab

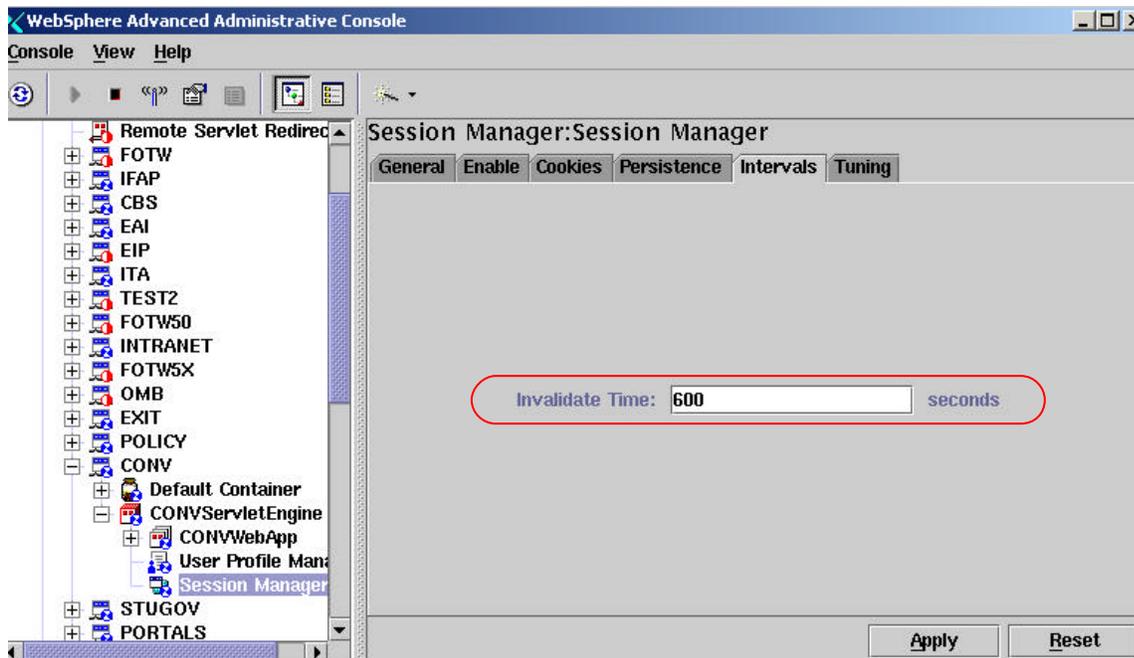


Figure 5: Intervals Tab

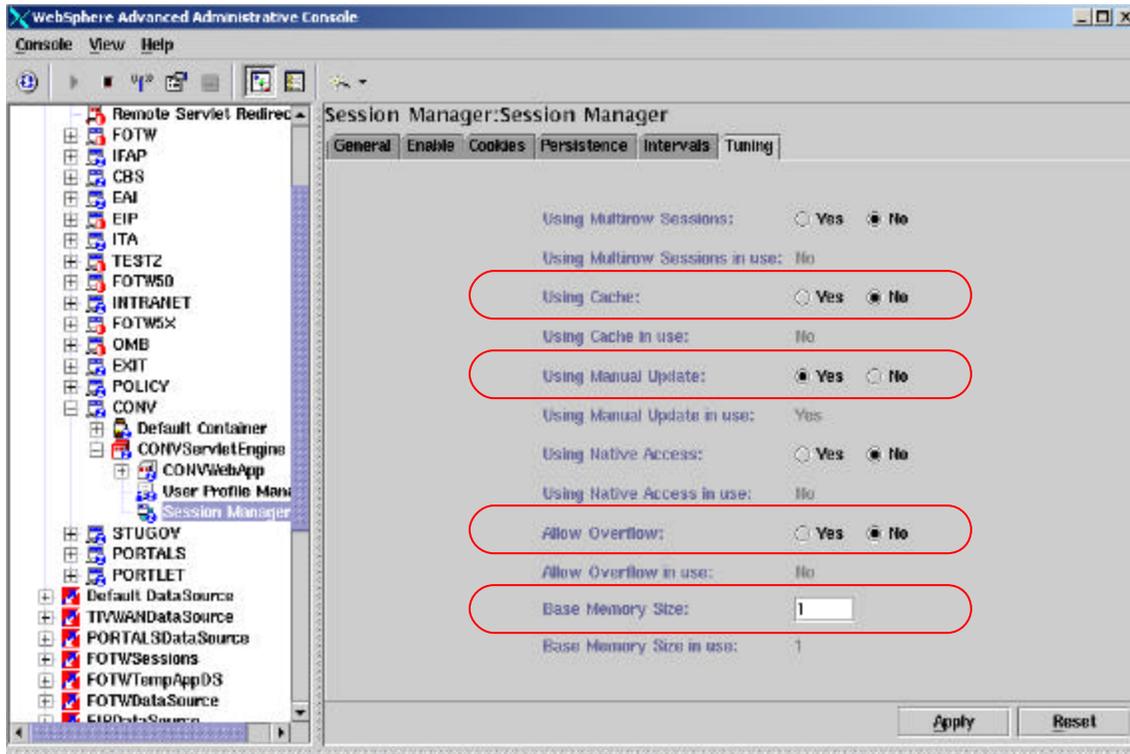
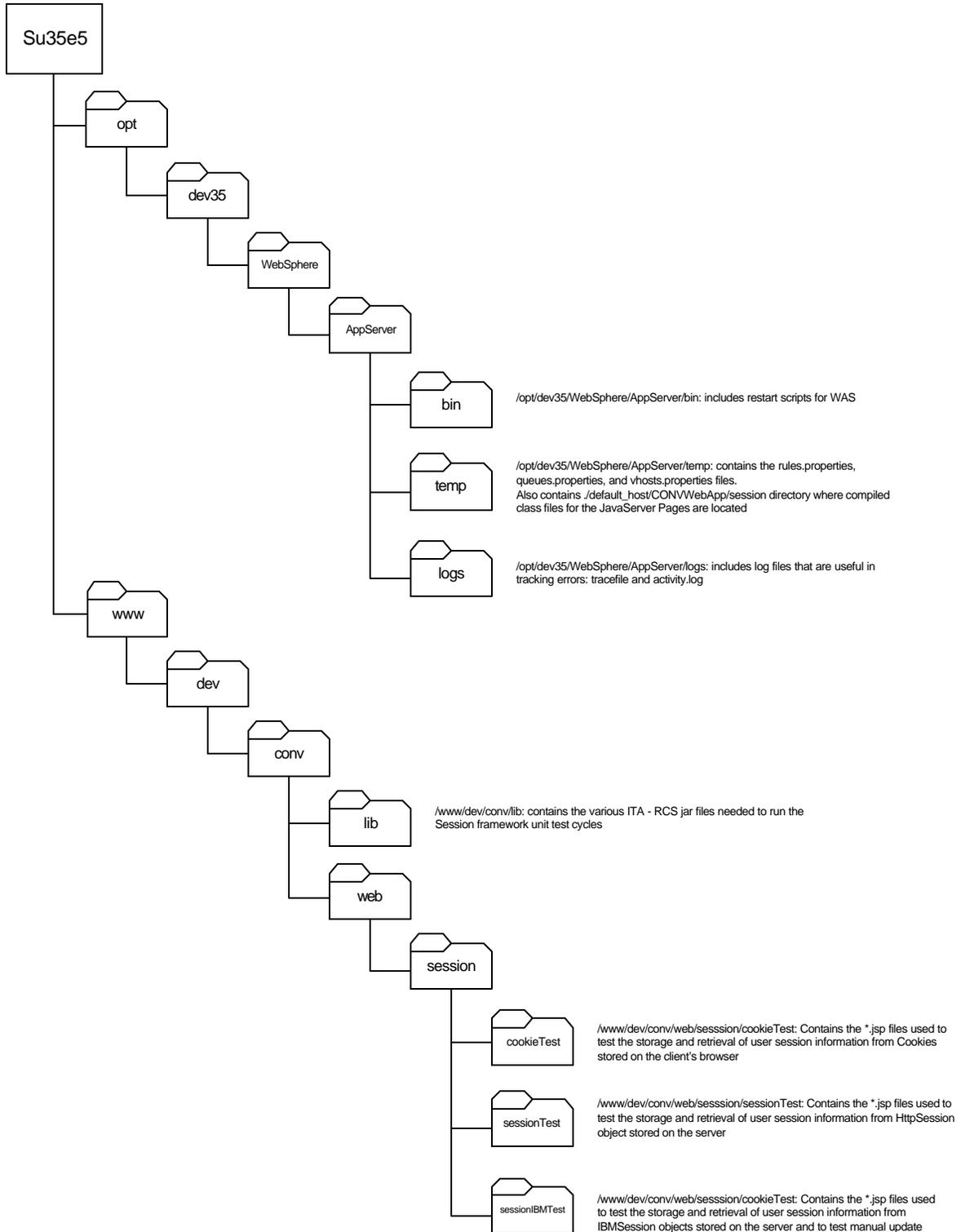


Figure 6: Tuning Tab

5.5.3.3 Directory Structure





5.5.4 Testing Conditions and Results

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 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, sessionIBMTTest. See the previous section for the complete path to the directory.

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

http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/cookieTest/*.jsp¹

http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/sessionTest/*.jsp

http://dev.conv.sfa.ed.gov:8531/CONVWebApp/session/sessionIBMTTest/*.jsp

All test cycles were conducted using Microsoft's Internet Explorer (IE). Netscape Navigator was used for testing cycles 1, 4, and 5 to ensure that the framework will behave as expected in another browser. Browser differences can occur though; such as when opening a completely new Internet Explorer browser (i.e. clicking the IE icon), the generated Session ID will be different in the new browser window; but launching a new window from within an existing browser window (i.e. Ctrl + N) will result in the same Session ID being used in the new browser window. Whereas, in Netscape, if one Navigator window is already open, opening a new browser in either method will still result in it using the same Session ID being used in the new browser window.

The test conditions are provided below and the configuration of the Session Manager settings and actual test scripts are provided in [Appendix A](#). To check if persistent session information has been stored in the database, a SQL query has to be executed. To access the database, type at the command prompt "sqlplus [ejsadmin@was35d](#)" without the " "s. Execute the SQL select statement "select id, maxinactivetime from sessions;" without the " "s. This will current sessions stored in the database.

¹ Where *.jsp refers to the different JavaServer Pages within each directory as listed in the test conditions and test scripts.

¹ Where *.jsp refers to the different JavaServer Pages within each directory as listed in the test conditions and test scripts.



5.5.4.1 Test Cycle 1

Tests the storage and retrieval of user session information from session cookies on the user's browser.

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Call the ContextManager constructor to access the cookie retrieval type	ContextManager, CookieRetrieval	ContextManager, CookieRetrieval	cookieTest/ input.jsp	First Name: Ben Last Name: Franklin Age: 100	Constructor is initialized and it in turn initializes the CookieRetrieval constructor
2	Save the information to a session cookie sent back to the client	ContextManager, CookieRetrieval	Constructor & setAttribute	cookieTest/ processInput.jsp	Information passed in from previous condition is stored in the cookie	Cookies are saved and values displayed should be identical to values entered in condition 1
3	Retrieve and display the information stored in the cookie	ContextManager, CookieRetrieval	Constructor, getNames, & getAttribute	cookieTest/ display.jsp	Information passed in from condition 1 is stored in the cookie	Obtain a list of all name and values saved in cookies. The list should match data entered in condition 1 plus display the Session ID
4	Delete information from a cookie	ContextManager, CookieRetrieval	Constructor & deleteAttribute	cookieTest/ deleteForm.jsp	Delete: Age	The cookie for age is deleted
5	Call the ContextManager to set cookies with a path (new browser)	ContextManager, CookieRetrieval	Constructor & setAttribute	cookieTest/pathTest/pathInput.jsp	First Name: George Last Name: W Age: 10	
6	Determine that the cookie set with a path can be displayed in the same directory	ContextManager, CookieRetrieval	Constructor, getNames, & getAttribute	cookieTest/pathTest/displayPath.jsp	Information passed in from previous condition is stored in the cookie	Information passed in from previous condition is displayed
7	Determine that the cookie set with a path can not be displayed in a parent directory	ContextManager, CookieRetrieval	Constructor, getNames, & getAttribute	cookieTest/ displayPath.jsp	none	No cookies except for the session cookie should be displayed
8	Determine that the cookie set with a path can be seen in a sub directory.	ContextManager, CookieRetrieval	Constructor, getNames, & getAttribute	cookieTest/pathTest/subDir/displayPath.jsp	Information passed in from condition 5 is stored in the cookie.	Information passed in from condition 5 is displayed.
9	Ensure cookies are not maintained across separate new browsers	ContextManager, CookieRetrieval	Constructor, getNames, & getAttribute	cookieTest/ display.jsp	No information should be persisted here from test condition 1	The Session ID from the test condition 1 should not be displayed and all values should be null
10	Ensure cookies are session cookies and not persistent cookies	none	none	none	none	No cookies should exist for dev.conv.sfa.ed.gov in the test computer's cookie folder.



5.5.4.2 Test Cycle 2

Tests the storage and retrieval of user session information stored in the HttpSession object in the application server's memory (cache).

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Call the ContextManager constructor to access the session retrieval type	ContextManager, SessionRetrieval	ContextManager, SessionRetrieval	sessionTest/input.jsp	First Name: Ben Last Name: Franklin Age: 100	Constructor is initialized and it in turn initializes the SessionRetrieval constructor
2	Bind the information to session object	ContextManager, SessionRetrieval	Constructor & setAttribute	sessionTest/processInput.jsp	Information passed in from previous condition is stored in the session variables	Session objects are saved in the server memory and attributes entered in condition 1 should be displayed
3	Retrieve and display the information bound to the session object	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/display.jsp	Information passed in from condition 1 is stored in the session variables	Obtain a list of all name and values saved in the session variables. The list should match data entered in condition 1 plus display the Session ID
4	Delete information bound to session object	ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionTest/deleteForm.jsp	Delete: Age	Delete the session variable for age
5	Introduce a second Session ID (open new browser) to the browser's memory to ensure objects bound to the session are not maintained across new browsers and also to ensure that the first session is still held in the memory cache	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/newSession.jsp	No information should be persisted here from test condition 1	The Session ID displayed should be different then the Session ID from test condition 1 and all other values should be null
		ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionTest/deleteForm.jsp	Delete: First Name	Delete the session variable for first name
6	Retrieve and display the information bound to the session object to show that the first session was still held in cache	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/display.jsp	Last name: Franklin	Last name from condition 1 will be the only value remaining



5.5.4.3 Test Cycle 3

Tests the storage and retrieval of user session information stored in the HttpSession object in the application server's database (persistent).

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Call the ContextManager constructor to access the session retrieval type	ContextManager, SessionRetrieval	ContextManager, SessionRetrieval	sessionTest/input.jsp	First Name: Ben Last Name: Franklin Age: 100	Constructor is initialized and it in turn initializes the SessionRetrieval constructor
2	Ensure session has been persisted to the sessions table in the database	ContextManager, SessionRetrieval	Constructor & setAttribute	sessionTest/processInput.jsp	Information passed in from previous condition is stored in the session object	A entry with the Session ID from test condition 1 should be listed with a maxinactive time of 600 and attributes entered in condition 1 should be displayed in the JSP Verify database population by using the SQL command: select id, maxinactive time from sessions;
3	Retrieve and display the information stored in the sessions table	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/display.jsp	Information passed in from condition 1 and stored in the sessions table	Obtain a list of all name and values saved in the session table. The list should match data entered in condition 1 plus display the Session ID
4	Delete information from session table	ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionTest/deleteForm.jsp	Delete: Age	Delete the session for age from the session table
5	Introduce a second session to ensure session information is not maintained across new browsers and to ensure that the first session is no longer held in the memory cache so that the data retrieved has to come from the database	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/newSession.jsp	No information should be persisted here from test condition 1	The Session ID displayed should be different then the Session ID from test condition 1 and all other values should be null
		ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionTest/deleteForm.jsp	Delete: First Name	Delete the session variable for first name
6	Retrieve and display the information stored in the session variables to show that the first session in the sessions table is still accessible	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionTest/display.jsp	Last name: Franklin	Last name from condition 1 will be the only value remaining



5.5.4.4 Test Cycle 4

Tests the storage and retrieval of user session information stored in the IBMSession object in the application server's database (persistent) without manually calling the method to persist the information to the database.

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Call the ContextManager constructor to access the session retrieval type	ContextManager, SessionRetrieval	ContextManager, SessionRetrieval	sessionIBMTTest/input.jsp	First Name: Ben Last Name: Franklin Age: 100	Constructor is initialized and it in turn initializes the SessionRetrieval constructor
2	Bind the objects to the session, does not call writeAttributes (sync)	ContextManager, SessionRetrieval	Constructor & setAttribute	sessionIBMTTest/processInput.jsp	Information passed in from previous condition is stored in the session variables	Bound objects are saved in the server memory cache and not the database since write was not called. Attributes entered in condition 1 should be displayed
3	Retrieve and display the objects bound to the session	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionIBMTTest/display.jsp	Information passed in from condition 1	Obtain a list of all name and values bound to the session. The list should match data entered in condition 1 plus display the Session ID. The attributes entered from condition 1 will be displayed even though writeAttributes was not called because it is stored in the server's memory cache
4	Delete object bound to the session, does not call writeAttributes (sync)	ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionIBMTTest/deleteForm.jsp	Delete: Age	Delete the age object bound to the session
5	Introduce a second session (new IE browser) to ensure session information is not maintained across new browsers and also to ensure that the first session is no longer held in the memory cache to ensure that the data retrieved has to come from the database	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionIBMTTest/input.jsp	No information should be persisted here from test condition 1	The Session ID displayed should be different than the Session ID from test condition 1 and all other values should be null
6	Access the original session by entering its session id in a new browser	none	none	sessionIBMTTest/newForm.jsp	Session ID: 0001 plus the Session ID saved from test condition 1	Pressing Submit will post the form to itself. If a value has been entered, a Hyperlink for next will appear. The current browser Session ID will be set to the Session ID created by the very first browser from test condition 1



Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
7	Display objects bound to the original session	ContextManager, SessionRetrieval	Constructor, getNames & getAttribute	sessionIBMTTest/newDisplay.jsp	No information should be persisted here from test condition 1	The values should be blank since writeAttributes (sync) was never called to persist the information to the database.

5.5.4.5 Test Cycle 5

Tests the storage and retrieval of user session information stored in the IBMSession object in the application server's database (persistent) with manually calling the method to persist the information to the database.

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Call the ContextManager constructor to access the appropriate retrieval type	ContextManager, SessionRetrieval	ContextManager, SessionRetrieval	sessionIBMTTest/inputW.jsp	First Name: Ben Last Name: Franklin Age: 100	Constructor is initialized and it in turn initializes the SessionRetrieval constructor
2	Bind the objects to the session, call writeAttributes (sync)	ContextManager, SessionRetrieval	Constructor & setAttribute	sessionIBMTTest/processInputW.jsp	Information passed in from previous condition is stored in the session variables	Session objects are saved in the server memory and to the sessions table; attributes entered in condition 1 should be displayed
3	Retrieve and display the objects bound to the session	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionIBMTTest/display.jsp	Information passed in from condition 1	Obtain a list of all name and values bound to the session. The list should match data entered in condition 1 plus display the Session ID.
4	Delete object bound to the session, call writeAttributes (sync)	ContextManager, SessionRetrieval	Constructor & deleteAttribute	sessionIBMTTest/deleteFormW.jsp	Delete: Age	Delete the age object bound to the session
5	Introduce a second session (new browser window) to ensure session information is not maintained across new browsers; also ensures the first session is no longer held in the memory cache so that the data retrieved has to come from the database	ContextManager, SessionRetrieval	Constructor, getNames, & getAttribute	sessionIBMTTest/inputW.jsp	No information should be persisted here from test condition 1	The Session ID displayed should be different then the Session ID from test condition 1 and all other values should be null



Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
6	Access the original session by entering its session id in a new browser	none	none	sessionIBMTTest/newForm.jsp	Session ID: 0001 plus the Session ID saved from test condition 1	Pressing Submit will post the form to itself. If a value has been entered, a Hyperlink for next will appear. The current browser Session ID will be set to the Session ID created by the very first browser from test condition 1
7	Display objects bound to the original session	ContextManager, SessionRetrieval	Constructor, getNames & getAttribute	sessionIBMTTest/newDisplay.jsp	Information passed in from condition 1	The values entered form test condition 1 will be displayed here since writeAttributes (sync) was called to persist the information to the database.

5.5.4.6 Test Cycle 6

Tests the exception handling of the User Session framework to ensure that appropriate exceptions are thrown when an invalid session has been detected while storing user session information on the server.

Condition Number	Detailed Condition	Class Name	Method Name	JSP Name	Form Data Input	Expected Results
1	Test setAttribute will throw an exception	ContextManager, SessionRetrieval	setAttribute	sessionIBMTTest/processInputW.jsp	First Name: Ben Last Name: Franklin Age: 100	An exception will be caught stating that the session is invalid.
2	Test getNames and getAttribute will throw exceptions	ContextManager, SessionRetrieval	getNames, & getAttribute	sessionIBMTTest/display.jsp	First Name: Ben Last Name: Franklin Age: 100	An exception will be caught stating that the session is invalid.
3	Test deleteAttribute will throw an exception	ContextManager, SessionRetrieval	deleteAttribute	sessionIBMTTest/deleteFormW.jsp	First Name: Ben Last Name: Franklin Age: 100	An exception will be caught stating that the session is invalid.



5.6 Performance Analysis

5.6.1 Purpose

This Performance Analysis Report documents the results of utilizing JProbe to test the ITA R3.0 Reusable Common Services (RCS) User Session framework. This report provides an in-depth analysis of the results gathered from the JProbe application profiling and documents any performance issues and suggests resolutions. The Detailed Design, User Guide, Unit Test Report, and the Performance Analysis documents for the User Session framework documentation will enable developers to quickly build applications using the User Session framework within the ITA environment architecture.

5.6.2 Approach

To ensure program efficiency and to detect possible bottleneck, ITA used JProbe to analyze the User Session framework. JProbe is a performance-profiling tool and it was used to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming. In order to profile this framework, portions of the unit test scripts were used to conduct this test. The performance analysis of this framework is documented in this report.

Two key groups of statistics are collected from the JProbe Profiler: the memory (heap) usage and the time spent on each method within the program (performance detail). This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow gathering detailed information on individual methods and the interaction between them.

5.6.3 Summary

This report contains the background information, performance test harness design, performance analyses, and resulting performance metrics for the framework. Profiling the User Session framework using the test scripts will test the code performance of the framework. The actual results will be compared against the results of how this framework is expected to function. Overall, this framework does not produce any loitering objects or create an excessive amount of objects. This framework is a robust API that should not cause any performance issues for calling applications.



5.6.4 Test Harness Design

5.6.4.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance test is to identify loitering objects and time spent on each method relative to each other in the User Session framework.

5.6.4.1.1 Testing Criteria

The two main components of the User Session framework will be tested: accessing session information stored in cookies and in the HTTP session object. Accessing information from the session object can be further divided into accessing the information from a variable or database, and the use of an IBMSession versus an HttpSession to store session objects. Since the User Session framework is an API, the JavaServer Pages developed for the unit test will serve as a test harness to profile and analyze the performance of the various methods.

5.6.4.2 Testing Configuration

In order to profile the User Session framework test applications for use with JProbe, the JPROBE Application Server configured in WebSphere was used and some of the configurations were changed. In the command line reference of the Application Server, there is a reference to the JProbe configuration file. The file used to conduct this performance analysis is: `/opt/util/JProbe/jpl_files/06052002_test_sessions.jpl`. The action, database, and HelloWorld servlets were all disabled. The Session Manager configurations were modified according to the settings required by the test scripts.

5.6.4.2.1 JProbe Configuration File

The JProbe configuration file has a file extension of .jpl. This file contains all of the settings that JProbe requires to profile an application, applet, or server side component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options. The user will be able to specify the activity of the Profiler. For example, the file can be configured to cause JProbe Profiler to take a heap snapshot before it exits and the directory to save the snapshots in.

The example application test will be conducted on the Solaris machine with the output being sent to a remote Windows NT workstation. The configuration in the actual file used to conduct the test can be found in [Appendix A](#). A filter for the main package, `gov.ed.fsa.ita.session`, was added to narrow the scope of the test to this package.

5.6.4.2.2 UNIX Server Settings

The usage of the User Session framework is closely tied to how the WebSphere Session Manager is configured. The WebSphere properties files have not been updated to run the test cycles.



The following sections list the properties related to the Web Application created to unit test the User Session framework. The configuration settings used in the Administration Console is defined in the next topic.

5.6.4.2.2.1 rules.properties:

```
default_host/JPROBEWebApp/*.do=ibmoselink4
default_host/JPROBEWebApp/*.jsp=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/=ibmoselink4
default_host/JPROBEWebApp/ErrorReporter=ibmoselink4
default_host/JPROBEWebApp/servlet=ibmoselink4
default_host/JPROBEWebApp=ibmoselink4
```

5.6.4.2.2.2 queues.properties:

```
ose.srvgrp.ibmoselink4.clone1.port=8241
ose.srvgrp.ibmoselink4.clone1.type=remote
ose.srvgrp.ibmoselink4.clonescount=1
ose.srvgrp.ibmoselink4.type=FASTLINK
ose.srvgrp=ibmoselink3,ibmoselink2,ibmoselink4,ibmoselink17
```

5.6.4.2.2.3 vhosts.properties:

```
stg.jprobe.fsa.ed.gov=default_host
```

5.6.4.2.3 WebSphere Application Server Configuration

The WebSphere Command Line will identify the JProbe configuration file to use and ensure that the correct JVM is used. Two Environment Variables will be added to the Application Server and two servlets will be added to the Web Application. The Session Manager configurations have to be updated, more information on how to update the settings can be found in the User Session Framework User Guide document.

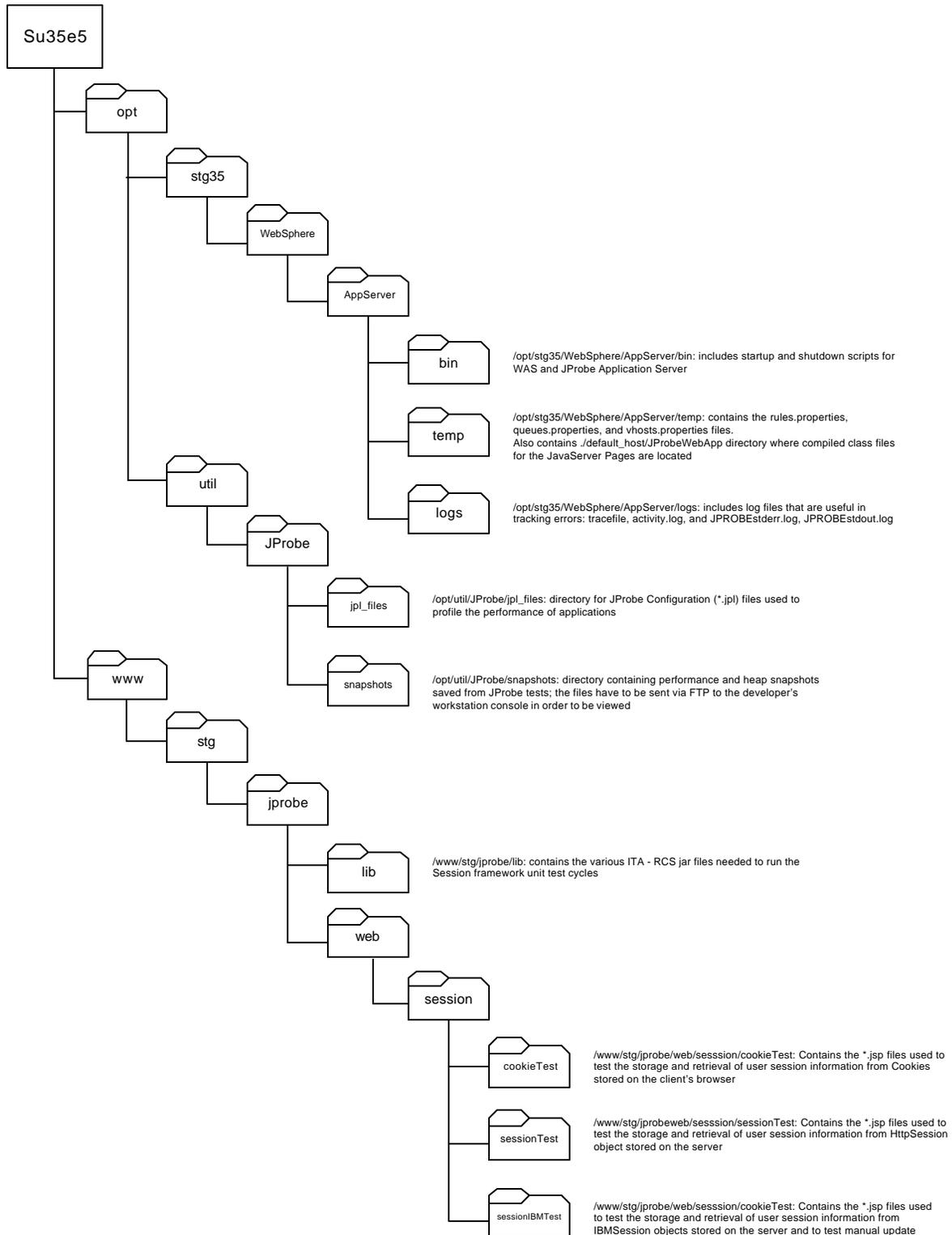
5.6.4.2.3.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/06052002_test_sessions.jpl -Xnoclassgc -
Djava.compiler=NONE -ms128m -mx128m
```

5.6.4.2.3.2 Environment:

```
EXECUTE=YES
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```

5.6.4.2.4 Directory Structure





5.6.5 Testing 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.

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.

5.6.6 Results and Analysis

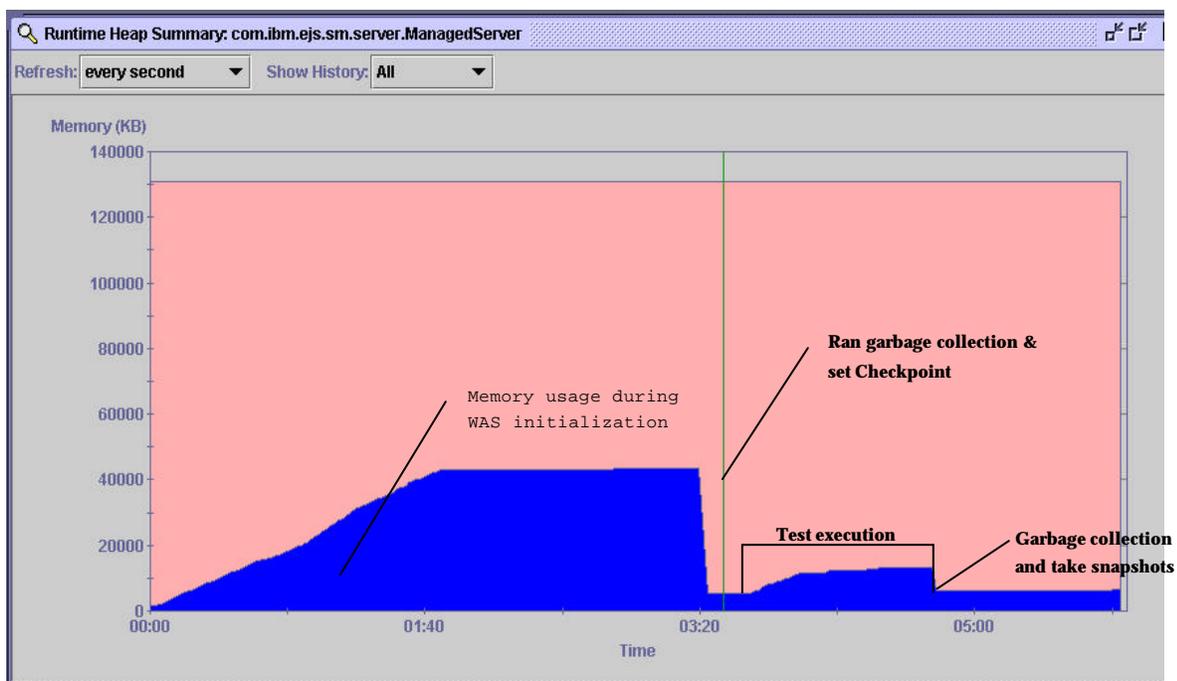
The JProbe Profiler with Memory Debugger application is used to trace both the memory usage and performance measurement of the User Session framework API. Two snapshots are taken: a heap snapshot and a performance Snapshot. Each snapshot provides different information regarding our test.

5.6.6.1 Heap Snapshot (Memory Usage)

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.

5.6.6.1.1 Heap Graph Analysis

The screenshot below is obtained from executing test cycle 3. It is the only heap graph screenshot depicted in this report since the heap graphs from executing other test cycle exhibit the same pattern.



In the graph above, it is possible to see that 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. By reading the graph, 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.

5.6.6.1.2 Instance Summary

The table below is a section of the Instance Summary result associated with conducting test cycle 3. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory those instances consume.

In the heap graph in the previous section, there is a green vertical line that shows where the checkpoint was set. The checkpoint tells JProbe to tag all subsequently created objects as “new.” The Count Change and Memory Change columns show data regarding new instances (created after the checkpoint) that are currently in the heap.

Package	Class	Count	Count Change	Memory	Memory Change
com.ibm.servlet.personalization.tracking	HashtableEntry	5 (14.7%)	+ 5	0.08 (5.9%)	+0.08
com.ibm.servlet.personalization.tracking	SessionDataList	4 (11.8%)	+ 4	0.048 (3.5%)	+0.048
com.ibm.servlet.personalization.tracking	SessionSimpleHashtable	4 (11.8%)	+ 4	0.048 (3.5%)	+0.012
com.ibm.servlet.personalization.tracking	DatabaseSessionData	2 (5.9%)	+ 2	0.36 (26.5%)	+0.36
com.ibm.servlet.personalization.tracking	SimpleHashtableEnumerator	2 (5.9%)	+ 2	0.04 (2.9%)	+0.04
com.ibm.servlet.personalization.tracking	BackedHastable	1 (2.9%)	+ 1	0.036 (2.6%)	+0.036
com.ibm.servlet.personalization.tracking	DatabaseSessionContext	1 (2.9%)	+ 1	0.124 (9.1%)	+0.124
com.ibm.servlet.personalization.tracking	SessionApplicationParameters	1 (2.9%)	+ 1	0.028 (2.1%)	+0.028
com.ibm.servlet.personalization.tracking	SessionTrackignEPMApplcationsData	1 (2.9%)	+ 1	0.028 (2.1%)	+0.028

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 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.



5.6.6.2 Performance Snapshot (Code Efficiency)

There are nine efficiency metrics that can be collected using JProbe – five basic metrics and four compound metrics. The basic metrics include: number of calls, method time, cumulative time, method object count, and cumulative object count. The compound metrics are averages per number of calls, including: average method time, average cumulative time, average method object count, and average cumulative object count. Time is measured as elapsed time in milliseconds.

The following sections will describe each metric and display the top results for each measurement for the performance assessment of the User Session framework. These metrics are basic indicators of process resource utilization. The detailed graphs associated with each method can be reviewed for unexpected activity or optimization opportunities.

All performance metric results were first filtered by *FSA* to obtain only the classes within the User Session framework which is what the test is looking for. Then for each section, the results were sorted by the metric under investigation to obtain the top ten results for each metric.

Only the test results from test cycle 1 and test cycle 5 are reported in this document. These two cycles were chosen since they represented two of the broadest uses of the ITA User Session framework.

5.6.6.2.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Cycle 1:

Name	Calls	Source
ContextManager.getAttribute(String)	15	ContextManager.java
CookieRetrieval.getAttribute(String)	15	CookieRetrieval
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	4	ContextManager.java
CookieRetrieval.<init>(HttpServletRequest, HttpServletResponse, String)	4	CookieRetrieval
ContextManager.getHttpRequest()	4	ContextManager.java
ContextManager.getHttpResponse()	3	ContextManager.java
ContextManager.setAttribute(String, Object)	3	ContextManager.java
CookieRetrieval.getHttpRequest()	3	CookieRetrieval



Name	Calls	Source
CookieRetrieval.getHttpResponse()	3	CookieRetrieval
CookieRetrieval.setAttribute(String, Object)	3	CookieRetrieval

Cycle 5:

Name	Calls	Source
ContextManager.getAttribute(String)	11	SessionRetrieval.java
SessionRetrieval.getAttribute(String)	11	ContextManager.java
ContextManager.getHttpRequest()	6	SessionRetrieval.java
ContextManager.getHttpResponse()	6	SessionRetrieval.java
SessionRetrieval.getHttpRequest()	6	ContextManager.java
SessionRetrieval.getHttpResponse()	6	ContextManager.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	4	SessionRetrieval.java
ContextManager.writeAttributes()	4	SessionRetrieval.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	4	ContextManager.java
SessionRetrieval.writeAttributes()	4	SessionRetrieval.java

From the results above, it is possible to see that method calls to the User Session framework APIs were behaving as expected. For every call to the ContextManager, there is a call to the CookieRetrieval or SessionRetrieval method to access user session information from the chosen storage context. There were no excessive calls since the framework is designed to provide a single set of APIs that developers could call which will then interact with the chosen storage context. The number of calls to the ContextManager could be reduced based on how the calling application chooses to utilize the framework. Since the test application is a series of JSP, an ContextManager object was created in each page.



5.6.6.2.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Cycle 1:

Name	Method Time	Source
CookieRetrieval.getNames()	2.89 (40.0%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	2.37 (32.9%)	ContextManager.java
CookieRetrieval.getAttribute(String)	0.67 (9.3%)	CookieRetrieval.java
CookieRetrieval.setAttribute(String, Object)	0.50 (6.9%)	CookieRetrieval.java
CookieRetrieval.class\$(String)	0.16 (2.2%)	CookieRetrieval.java
ContextManager.getAttribute(String)	0.13 (1.8%)	ContextManager.java
CookieRetrieval.deleteAttribute(String)	0.12 (1.6%)	CookieRetrieval.java
CookieRetrieval.<init>(HttpServletRequest, HttpServletResponse, String)	0.08 (1.2%)	CookieRetrieval.java
ContextManager.setAttribute(String, Object)	0.05 (0.7%)	ContextManager.java
ContextManager.getNames()	0.05 (0.6%)	ContextManager.java

Cycle 5:

Name	Method Time	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	5.08 (55.6%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	2.74 (29.9%)	SessionRetrieval.java
SessionRetrieval.writeAttributes()	0.26 (2.8%)	SessionRetrieval.java
SessionRetrieval.class\$(String)	0.17 (1.8%)	SessionRetrieval.java
SessionRetrieval.getAttribute(String)	0.13 (1.5%)	SessionRetrieval.java
ContextManager.getAttribute(String)	0.10 (1.1%)	ContextManager.java
SessionRetrieval.setAttribute(String, Object)	0.09 (1.0%)	SessionRetrieval.java
ContextManager.getHttpRequest()	0.07 (0.8%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0.07 (0.8%)	ContextManager.java
SessionRetrieval.getNames()	0.07 (0.7%)	SessionRetrieval.java

The results above show that the longest running methods are the initialization methods. The CookieRetrieval and SessionRetrieval methods have longer times compared to the ContextManager methods. This is due to the ContextManager methods calling an CookieRetrieval or SessionRetrieval method and the time spent executing the children methods are not being counted.



SessionRetrieval's writeAttributes() method has a longer execution time compared to other methods, which is expected since it has to access a database to perform a write to it. CookieRetrieval's getNames() method takes longer to execute compared to SessionRetrieval's getNames() method. This is an expected condition as it is more complicated to cycle through and obtain the names of all cookies then it is to retrieve all names from a session object.

5.6.6.2.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Cycle 1:

Name	Cumulative Time	Source
ContextManager.getNames()	3.10 (42.9%)	ContextManager.java
CookieRetrieval.getNames()	3.05 (42.3%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	2.46 (34.0%)	ContextManager.java
ContextManager.getAttribute(String)	0.81 (11.2%)	ContextManager.java
CookieRetrieval.getAttribute(String)	0.67 (9.3%)	CookieRetrieval.java
ContextManager.setAttribute(String, Object)	0.55 (7.6%)	ContextManager.java
CookieRetrieval.setAttribute(String, Object)	0.50 (6.9%)	CookieRetrieval.java
CookieRetrieval.class\$(String)	0.16 (2.2%)	CookieRetrieval.java
ContextManager.deleteAttribute(String)	0.15 (2.1%)	ContextManager.java
CookieRetrieval.deleteAttribute(String)	0.12 (1.6%)	CookieRetrieval.java

Cycle 5:

Name	Cumulative Time	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	7.99 (87.3%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	2.90 (31.7%)	SessionRetrieval.java
ContextManager.writeAttributes()	0.32 (3.5%)	ContextManager.java
SessionRetrieval.writeAttributes()	0.26 (2.8%)	SessionRetrieval.java
ContextManager.getAttribute(String)	0.24 (2.6%)	ContextManager.java
SessionRetrieval.class\$(String)	0.17 (1.8%)	SessionRetrieval.java
ContextManager.setAttribute(String, Object)	0.16 (1.8%)	ContextManager.java
SessionRetrieval.getAttribute(String)	0.13 (1.5%)	SessionRetrieval.java



Name	Cumulative Time	Source
ContextManager.getNames()	0.11 (1.2%)	ContextManager.java
ContextManager.getHttpRequest()	0.10 (1.1%)	ContextManager.java

In the results above, it is possible to see that ContextManager methods have a longer cumulative time since it includes in the count the time it takes for the ContextManager methods to call a sub-method to retrieve the data. Again, it takes considerably longer to execute the getNames() method in CookieRetrieval then it does in SessionRetrieval due to the complexity of the logic in that particular method.

5.6.6.2.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Cycle 1:

Name	Method Objects	Source
CookieRetrieval.getNames()	23 (51.1%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	10 (22.2%)	ContextManager.java
CookieRetrieval.setAttribute(String, Object)	9 (20.0%)	CookieRetrieval.java
CookieRetrieval.class\$(String)	2 (4.4%)	CookieRetrieval.java
CookieRetrieval.deleteAttribute(String)	1 (2.2%)	CookieRetrieval.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.deleteAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getHttpRequest()	0 (0.0%)	ContextManager.java
ContextManager.getHttpResponse()	0 (0.0%)	ContextManager.java

Cycle 5:

Name	Method Objects	Source
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	20 (45.5%)	SessionRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	18 (40.9%)	ContextManager.java
SessionRetrieval.writeAttributes()	4 (9.1%)	SessionRetrieval.java



Name	Method Objects	Source
SessionRetrieval.class\$(String)	2 (4.5%)	SessionRetrieval.java
ContextManager.writeAttributes()	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0 (0.0%)	ContextManager.java
SessionRetrieval.getAttribute(String)	0 (0.0%)	SessionRetrieval.java
ContextManager.getNames()	0 (0.0%)	ContextManager.java
ContextManager.getHttpRequest()	0 (0.0%)	ContextManager.java

The method `getNames()` from the `CookieRetrieval` class created several objects compared to the same method in `SessionRetrieval`. This is due to the design of the method, and which requires it to obtain all cookies from the request first, create a vector for the cookie name, iterate through all cookies, and then add the names to the vector. In future uses, if there is not enough memory to execute this framework then this method could be examined to see if less objects can be created. Presently, the `getNames()` method does not produce any significant performance problems that would require it to be redeveloped.

The method `setAttributes()` from the `CookieRetrieval` class compared to the same method in the `SessionRetrieval` class also created more objects. This is expected since a new `Cookie` object has to be created for each cookie to be added to the response object and set on the client browser. There is no alternative to the implementation of this method and developers will need to be aware that this method will create a new `Cookie` object for each cookie that has to be set.

5.6.6.2.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Cycle 1:

Name	Cumulative Objects	Source
<code>CookieRetrieval.getNames()</code>	25 (55.6%)	<code>CookieRetrieval.java</code>
<code>ContextManager.getNames()</code>	25 (55.6%)	<code>ContextManager.java</code>
<code>ContextManager.<init>(HttpServletRequest, HttpServletResponse)</code>	10 (22.2%)	<code>ContextManager.java</code>
<code>CookieRetrieval.setAttribute(String, Object)</code>	9 (20.0%)	<code>CookieRetrieval.java</code>
<code>ContextManager.setAttribute(String, Object)</code>	9 (20.0%)	<code>ContextManager.java</code>
<code>CookieRetrieval.class\$(String)</code>	2 (4.4%)	<code>CookieRetrieval.java</code>
<code>CookieRetrieval.deleteAttribute(String)</code>	1 (2.2%)	<code>CookieRetrieval.java</code>



Name	Cumulative Objects	Source
ContextManager.deleteAttribute(String)	1 (2.2%)	ContextManager.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java

Cycle 5:

Name	Cumulative Objects	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	40 (90.9%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	22 (50.0%)	SessionRetrieval.java
ContextManager.writeAttributes()	4 (9.1%)	ContextManager.java
SessionRetrieval.writeAttributes()	4 (9.1%)	SessionRetrieval.java
SessionRetrieval.class\$(String)	2 (4.5%)	SessionRetrieval.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.deleteAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getHttpRequest()	0 (0.0%)	ContextManager.java
ContextManager.getHttpResponse()	0 (0.0%)	ContextManager.java

The findings from this metric are similar to the previous results for Method Object Count. The count for objects created by methods in the ContextManager class increased since it now includes the count of objects created by sub-methods.

5.6.6.2.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Cycle 1:

Name	Avg. Method Time	Source
CookieRetrieval.getNames()	1.44 (20.0%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	0.59 (8.2%)	ContextManager.java
CookieRetrieval.setAttribute(String, Object)	0.17 (2.3%)	CookieRetrieval.java
CookieRetrieval.class\$(String)	0.16 (2.2%)	CookieRetrieval.java



Name	Avg. Method Time	Source
CookieRetrieval.deleteAttribute(String)	0.12 (1.6%)	CookieRetrieval.java
CookieRetrieval.getAttribute(String)	0.04 (0.6%)	CookieRetrieval.java
ContextManager.deleteAttribute(String)	0.04 (0.5%)	ContextManager.java
ContextManager.<clinit>()	0.03 (0.4%)	ContextManager.java
ContextManager.getNames()	0.02 (0.3%)	ContextManager.java
CookieRetrieval.<init>(HttpServletRequest, HttpServletResponse, String)	0.02 (0.3%)	CookieRetrieval.java

Cycle 5:

Name	Avg. Method Time	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	1.27 (13.9%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	0.68 (7.5%)	SessionRetrieval.java
SessionRetrieval.class\$(String)	0.17 (1.8%)	SessionRetrieval.java
SessionRetrieval.writeAttributes()	0.06 (0.7%)	SessionRetrieval.java
SessionRetrieval.deleteAttribute(String)	0.05 (0.5%)	SessionRetrieval.java
SessionRetrieval.getNames()	0.03 (0.4%)	SessionRetrieval.java
ContextManager.deleteAttribute(String)	0.03 (0.3%)	ContextManager.java
ContextManager.<clinit>()	0.03 (0.3%)	ContextManager.java
SessionRetrieval.setAttribute(String, Object)	0.03 (0.3%)	SessionRetrieval.java
ContextManager.getSession()	0.03 (0.3%)	ContextManager.java

The above results demonstrate that on average, the initialization methods, CookieRetrieval.getNames(), and FSASessionRetrieval.writeAttributes() takes longer to execute. These findings are expected and previously explored in the Method Time metric.

5.6.6.2.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Cycle 1:



Name	Average Cumulative Time	Source
ContextManager.getNames()	1.55 (21.5%)	ContextManager.java
CookieRetrieval.getNames()	1.53 (21.1%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	0.61 (8.5%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0.18 (2.5%)	ContextManager.java
CookieRetrieval.setAttribute(String, Object)	0.17 (2.3%)	CookieRetrieval.java
CookieRetrieval.class\$(String)	0.16 (2.2%)	CookieRetrieval.java
ContextManager.deleteAttribute(String)	0.15 (2.1%)	ContextManager.java
CookieRetrieval.deleteAttribute(String)	0.12 (1.6%)	CookieRetrieval.java
ContextManager.getAttribute(String)	0.05 (0.7%)	ContextManager.java
CookieRetrieval.getAttribute(String)	0.04 (0.6%)	CookieRetrieval.java

Cycle 5:

Name	Average Cumulative Time	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	2.00 (21.8%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	0.73 (7.9%)	SessionRetrieval.java
SessionRetrieval.class\$(String)	0.17 (1.8%)	SessionRetrieval.java
ContextManager.deleteAttribute(String)	0.08 (0.9%)	ContextManager.java
ContextManager.writeAttributes()	0.08 (0.9%)	ContextManager.java
SessionRetrieval.writeAttributes()	0.06 (0.7%)	SessionRetrieval.java
ContextManager.getNames()	0.05 (0.6%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0.05 (0.6%)	ContextManager.java
SessionRetrieval.deleteAttribute(String)	0.05 (0.5%)	SessionRetrieval.java
ContextManager.getHttpSession()	0.04 (0.4%)	ContextManager.java

The results above do not present any surprises and are consistent with the expected results based on evaluation of the previous performance metrics.

5.6.6.2.8 Average Method Object



Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Cycle 1:

Name	Avg. Method Object	Source
CookieRetrieval.getNames()	11 (24.4%)	CookieRetrieval.java
CookieRetrieval.setAttribute(String, Object)	3 (6.7%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	2 (4.4%)	ContextManager.java
CookieRetrieval.class\$(String)	2 (4.4%)	CookieRetrieval.java
CookieRetrieval.deleteAttribute(String)	1 (2.2%)	CookieRetrieval.java
ContextManager.getNames()	0 (0.0%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0 (0.0%)	ContextManager.java
ContextManager.deleteAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java

Cycle 5:

Name	Avg. Method Object	Source
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	5 (11.4%)	SessionRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	4 (9.1%)	ContextManager.java
SessionRetrieval.class\$(String)	2 (4.5%)	SessionRetrieval.java
SessionRetrieval.writeAttributes()	1 (2.3%)	SessionRetrieval.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.deleteAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.getHttpRequest()	0 (0.0%)	ContextManager.java
ContextManager.getHttpResponse()	0 (0.0%)	ContextManager.java
ContextManager.getHttpSession()	0 (0.0%)	ContextManager.java

These results serve to demonstrate that the methods that produce the most objects are called the most times. Designing the test application differently can eliminate some of the number of calls and objects created.



5.6.6.2.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Cycle 1:

Name	Average Cumulative Object	Source
ContextManager.getNames()	12 (26.7%)	ContextManager.java
CookieRetrieval.getNames()	12 (26.7%)	CookieRetrieval.java
ContextManager.setAttribute(String, Object)	3 (6.7%)	ContextManager.java
CookieRetrieval.setAttribute(String, Object)	3 (6.7%)	CookieRetrieval.java
ContextManager.<init>(HttpServletRequest, HttpServletResponse)	2 (4.4%)	ContextManager.java
CookieRetrieval.class\$(String)	2 (4.4%)	CookieRetrieval.java
ContextManager.deleteAttribute(String)	1 (2.2%)	ContextManager.java
CookieRetrieval.deleteAttribute(String)	1 (2.2%)	CookieRetrieval.java
ContextManager.<clinit>()	0 (0.0%)	ContextManager.java
ContextManager.getAttribute(String)	0 (0.0%)	ContextManager.java

Cycle 5:

Name	Average Cumulative Object	Source
ContextManager.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	10 (22.7%)	ContextManager.java
SessionRetrieval.<init>(HttpServletRequest, HttpServletResponse, boolean, boolean)	5 (11.4%)	SessionRetrieval.java
SessionRetrieval.class\$(String)	2 (4.5%)	SessionRetrieval.java
ContextManager.deleteAttribute(String)	0 (0.0%)	ContextManager.java
ContextManager.writeAttributes()	1 (2.3%)	ContextManager.java
SessionRetrieval.writeAttributes()	1 (2.3%)	SessionRetrieval.java
ContextManager.getNames()	0 (0.0%)	ContextManager.java
ContextManager.setAttribute(String, Object)	0 (0.0%)	ContextManager.java
SessionRetrieval.deleteAttribute(String)	0 (0.0%)	SessionRetrieval.java
ContextManager.getSession()	0 (0.0%)	ContextManager.java



The average cumulative object is a reflection of the Average Method Object Count metric but includes information for methods from the ContextManager class and calls to its children.

5.6.6.3 General Performance Metrics

The RCS User Session framework is tested on a Solaris 2.6 platform running JDK1.2.2 Reference Implementation. The test harness tested the major operations of the User Session framework independently and the system as a whole.

No memory leaks were found in the Session framework using the different test cycles as a test harness. No loitering objects were found in the heap at the end of the each test cycle.



5.6.7 Appendix A

5.6.7.1 JProbe Configuration File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
    </application>
    <applet
      working_dir=""
      source_dir=""
      htmlfile=""
      main_package="">
    </applet>
    <serverside
      suggested_filters=""
      id="Other server"
      server_dir="/opt/stg35/WebSphere/AppServer"
      prepend_to_vm_args=""
      source_dir=""
      classname="com.ibm.ejs.sm.util.process.Nanny"
      main_package="gov.ed.fsa.ita.session"
      exclude_server_classes="true"
      args=""
      working_dir="/opt/stg35/WebSphere/AppServer/servlets"
      prepend_to_classpath="">
    </serverside>
  </program>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.74:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
      timing="elapsed"
      track_natives="true"
    </performance>
  </analysis>
</jpl>
```



```
final_snapshot="true"
granularity="method">
<performance.filter
  visibility="visible"
  methodmask=""
  enabled="true"
  classmask=""
  time="ignore"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask=""
  enabled="true"
  classmask=" gov.ed.fsa.ita.session.*"
  time="track"
  granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask=""/>
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask=""/>
</threadalyzer>
<coverage
  record_from_start="true"
```



ITA Release 3.0 Build & Test Report

```
final_snapshot="true"  
granularity="line">  
<coverage.filter  
  visibility="invisible"  
  methodmask=""  
  enabled="true"  
  classmask="*/>  
<coverage.filter  
  visibility="visible"  
  methodmask=""  
  enabled="true"  
  classmask=".*"/>  
</coverage>  
</analysis>  
</jpl>
```



5.6.8 Resources

- Best Practices for Session Programming: WebSphere Application Server
 - <http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/0404010108.html>
- Building Business Solutions with WebSphere
 - <http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/06061100.html>
- Maintaining Session Data with the WebSphere Session Manager –
 - <http://www6.software.ibm.com/devtools/news0801/art26.htm>
- Session Manager Properties
 - <http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/06061100.html>
- WebSphere Application Server Best Practices using HTTP Sessions
 - http://www.106.ibm.com/developerworks/patterns/guidelines/HTTP_Session_Best_Practice.pdf

6 RCS – Web Services (SOAP) Framework

6.1 Purpose

This section of the Performance Analysis Report documents the results of utilizing JProbe to analyze the ITA R3.0 Reusable Common Services (RCS) SOAP framework. This section provides an in-depth analysis of the results gathered from the JProbe and documents performance issues. The Detailed Design, User Guide, and the Performance Analysis documents for the SOAP framework will enable developers to quickly build applications using the SOAP framework within the ITA environment architecture.



6.2 Approach

To ensure program efficiency and to detect possible bottlenecks, ITA used JProbe to analyze the SOAP framework. JProbe is a performance-profiling tool and it was utilized to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming.

Two key groups of statistics are collected from the JProbe Profiler: The memory (heap) usage and the time spent on each method within the program (performance detail). This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow detailed information to be gathered on individual methods and define the calling relationship between methods.

6.3 Summary

This section of the report contains the performance test harness design, performance analysis, and resulting performance metrics for the SOAP framework. 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. The actual results were compared against the results of how this framework is expected to function. Overall, this framework does not produce any loitering objects that remain in the heap after its useful life.



6.4 Test Harness Design

6.4.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance analysis is to identify loitering objects and time spent on each method relative to other methods within the SOAP Framework.

6.4.1.1 Testing Criteria

The Messaging portion of the Apache SOAP Framework is what needed to be performance tested. Since the SOAP Framework is an API, the example messaging application packaged with the framework distribution was used as a test harness to profile and analyse the performance of the various methods.

6.4.1.2 JProbe Configuration File

The JProbe Configuration file has a file extension of .jpl. This file contains all of the settings that JProbe requires to profile an application, applet, or serverside component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options.

The example application test was conducted on the Solaris machine with the output directed to a remote Windows NT workstation. Performance and heap snapshots were taken before the Application Server was stopped. The following is the actual file used to conduct the test.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname="">
    <classpath/>
  </application>
  <applet
    working_dir=""
    source_dir=""
    htmlfile=""
    main_package="">
  <classpath>
    <classpath.path location="%CLASSPATH%"/>
  </classpath>
</applet>
```



```
<serverside
  suggested_filters=""
  id="Other server"
  server_dir="/opt/stg35/WebSphere/AppServer"
  prepend_to_vm_args=""
  source_dir=""
  classname="com.ibm.ejs.sm.util.process.Nanny"
  main_package="org.apache.soap"
  exclude_server_classes="true"
  args=""
  working_dir="/opt/stg35/WebSphere/AppServer/servlets"
  prepend_to_classpath="">
  <classpath>
    <classpath.path
location="%CLASSPATH%"/>
  </classpath>
</serverside>
</program>
<vm
  snapshot_dir="/opt/util/JProbe/snapshots"
  location="/opt/util/jdk1.2.2/bin/java"
  args=""
  type="java2"
  use_jit="true"/>
<viewer
  socket="170.248.222.80:4444"
  type="remote"/>
<analysis type="profile">
  <performance
    record_from_start="true"
    timing="elapsed"
    track_natives="true"
    final_snapshot="false"
    granularity="method">
    <performance.filter
      visibility="visible"
      methodmask="*"
      enabled="true"
      classmask="*"
      time="ignore"
      granularity="method"/>
    <performance.filter
      visibility="visible"
      methodmask="*"
      enabled="true"
      classmask="org.apache.soap.*"
      time="track"
      granularity="method"/>
    <performance.filter
      visibility="visible"
      methodmask="*"
      enabled="true"
      classmask="samples.messaging.*"
```



```
                time="track"
                granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="false"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
  <deadlock_detection
    enabled="true"
    deadlock_and_exit="true"
    report_stalls="false"
    track_system_threads="false"
    block_can_stall="false"
    deadlock_threshold="2"/>
  <deadlock_prediction
    enable_hold_and_wait="false"
    enable_lock_order="false"
    lock_order_maintains_covers="true"/>
  <data_race
    ignore_volatile="false"
    enable_happens_before="false"
    no_stack_trace_limit="false"
    enable_lock_covers="false"
    max_stack_trace="1"
    instrument_elements="false"/>
  <visualizer
    enabled="true"
    visualization_level="1"/>
  <threadalyzer.filter
    visibility="invisible"
    enabled="true"
    classmask="*" />
  <threadalyzer.filter
    visibility="visible"
    enabled="true"
    classmask=".*" />
</threadalyzer>
<coverage
  record_from_start="true"
  final_snapshot="false"
  granularity="line">
  <coverage.filter
    visibility="invisible"
    methodmask="*"
    enabled="true"
    classmask="*" />
  <coverage.filter
    visibility="visible"
```



```
methodmask="*"
enabled="true"
classmask=".*"/>
</coverage>
</analysis>
</jpl>
```

6.4.1.3 WebSphere Application Server Configuration

The WebSphere Command Line was configured with the JProbe configuration file used to ensure that the correct JVM was used. One servlet was added to the Web Application to listen for SOAP messages coming in.

6.4.1.3.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/09232002_test_soap.jpl -Xnoclassgc -
Djava.compiler=NONE -ms128m -mx128m
```

6.4.1.3.2 Environment:

```
EXECUTE=YES
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```

6.4.1.3.3 Message Servlet:

Servlet: messageRouter

Description: SOAP Message Servlet

Servlet Class Name: org.apache.soap.servlet.http.MessageRouterServlet

Servlet Web Path List: default_host/JPROBEWebApp/

Init Parameters:

Init Param Name	Value
detail	2
debug	2
validate	true
config	/struts-config.xml
application	Resource

Debug Mode: False

Load at Startup: True

6.4.1.4 Additional Required Components

The following java archive files are required to run the example application:

- soap.jar



- mail.jar
- activation.jar
- xerces.jar
- bsf.jar
- js.jar

6.5 Testing Scenario

The example messaging application provided with the framework distribution was used as the test harness.

6.5.1 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.

6.5.2 Test Scenario

Run the following test script from the soap2_2/samples/messaging directory:

```
@echo off
echo This test assumes server URLs of http://stg.jprobe.fsa.ed.gov /JPROBEWebApp/servlet/rpcrouter
echo and http://stg.jprobe.fsa.ed.gov/JPROBEWebApp/servlet/messengerouter
java samples.messaging.SendMessage http://stg.jprobe.fsa.ed.gov/JPROBEWebApp/servlet/messengerouter msg1.xml
echo . after sent message
java samples.messaging.SendMessage http://stg.jprobe.fsa.ed.gov/JPROBEWebApp/servlet/messengerouter msg1.xml
echo . after sent message
java samples.messaging.SendMessage http://stg.jprobe.fsa.ed.gov/JPROBEWebApp/servlet/messengerouter msg1.xml
echo . after sent message
```

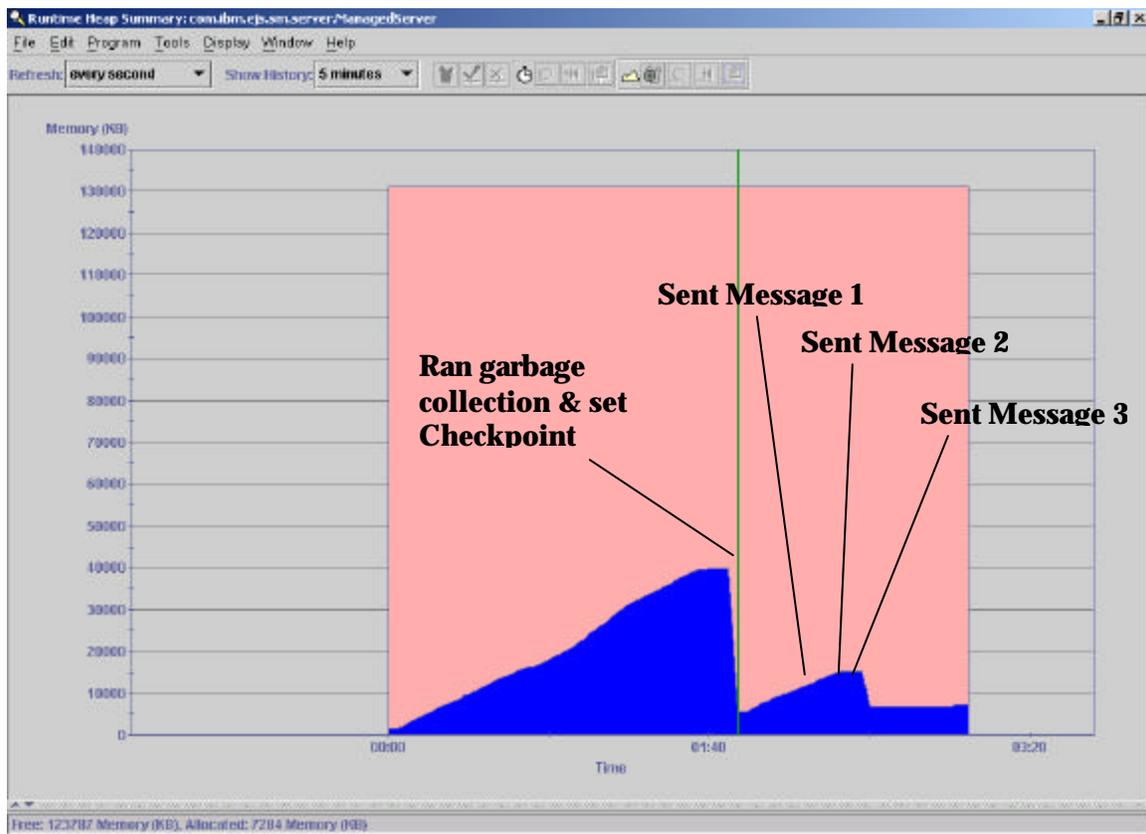
6.6 Results and Analysis

6.6.1 Heap Snapshot (Memory Usage)

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

6.6.1.1 Heap Graph Analysis

The screenshot below is obtained from sending three messages to the Message Servlet.



The spike is expected since a new message is being created to be sent back. You will notice that as the second and third messages are sent, the heap graph stays the same height. This tells us that no extra objects are being created as each message comes through the Messaging Servlet.

6.6.1.2 Instance Summary

The table below is a section of the Instance Summary results associated with running the three messages through the Messaging Servlet. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory those instances consume.

In the heap graph in the previous section, there is a green vertical line that shows where the Checkpoint was set. The Checkpoint tells JProbe to tag all subsequently created objects as “new.” The Count Change and Memory Change columns show data regarding new instances (created after the checkpoint) that are currently in the heap.



Runtime Heap Summary: com.ibm.js.srvr.ManagedServer

File Edit Program Tools Display Window Help

Refresh: EVERY SECOND Show History: 5 MINUTES

Instance Summary Garbage Monitor

Filter Classes: Visible Classes: 825 / 825

Package	Class	Count / %	Count Change	Cumulative Count	Memory	Mem. Chg.
Total		135,670 (100.0%)	+30,325	711,354 (100.0%)	5,824,392 (100.0%)	
	char[]	31,059 (22.9%)	+6,942	212,809 (29.9%)	2,462,940 (42.3%)	
java.lang	String	30,407 (22.4%)	+5,503	176,372 (24.8%)	304,884 (5.3%)	
	Object[]	13,676 (10.1%)	+5,131	53,978 (7.6%)	957,636 (16.4%)	
com.ibm.js.srvr.util	HashtableEntry	8,389 (6.2%)	+3	8,389 (1.2%)	167,780 (2.9%)	
com.ibm.js.util.cache	Bucket	8,190 (6.0%)	+4,096	8,190 (1.2%)	229,320 (3.9%)	
java.util	HashMapEntry	6,978 (5.1%)	+871	7,858 (1.1%)	139,560 (2.4%)	
java.util	HashtableEntry	5,945 (4.4%)	+1,147	10,207 (1.4%)	136,900 (2.0%)	
java.util.jar	AttributesName	5,019 (3.7%)	+603	5,020 (0.7%)	60,220 (1.0%)	
java.lang	Object	4,169 (3.1%)	+10	4,170 (0.6%)	16,676 (0.3%)	
java.lang	Class	2,519 (1.9%)	+612	2,519 (0.4%)	352,660 (6.1%)	
com.ibm.js.util	Bucket	2,167 (1.6%)	+317	2,167 (0.3%)	26,004 (0.4%)	
java.util	HashMap	1,705 (1.3%)	+206	2,036 (0.3%)	76,672 (1.4%)	
java.util.jar	Attributes	1,606 (1.2%)	+201	1,904 (0.3%)	6,744 (0.1%)	
	(int)	1,288 (0.9%)	+1,189	11,559 (1.6%)	87,528 (1.5%)	
java.lang	StringBuffer	869 (0.6%)	+864	79,018 (11.1%)	10,428 (0.2%)	
java.util	Vector	868 (0.6%)	+153	1,153 (0.2%)	17,960 (0.3%)	
	byte[]	644 (0.5%)	+392	39,701 (5.6%)	436,440 (7.5%)	
java.lang	Integer	600 (0.4%)	+204	3,118 (0.4%)	2,720 (0.0%)	
java.lang.reflect	Field	465 (0.3%)	+21	1,733 (0.2%)	13,020 (0.2%)	
java.util	Hashtable	436 (0.3%)	+35	901 (0.1%)	15,696 (0.3%)	
java.text	FieldPosition	428 (0.3%)	+428	1,493 (0.2%)	5,136 (0.1%)	
java.util	SimpleTimeZone	330 (0.2%)	-	334 (0.0%)	25,080 (0.4%)	
java.io	ObjectStreamField	312 (0.2%)	+52	794 (0.1%)	6,736 (0.1%)	

Free: 123787 Memory (KB), Allocated: 7284 Memory (KB)

These results were gathered after the test scenario has finished executing and garbage collection has occurred. The results were filtered for 'org.apache.soap.*' since those are the classes this report is concerned with. The Count Change column was used to sort the data to determine which class had the most objects remaining in the heap after the scenario has been completed.

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.

6.7 Test Conclusions

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.

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.



6.8 Resources

- Apache SOAP Toolkit Website
<http://xml.apache.org/soap/index.html>
- JavaMail Website
<http://java.sun.com/products/javamail/>

6.9 JavaBeans Activation Framework Website

<http://java.sun.com/products/beans/glasgow/jaf.html>

6.10 Apache Xerces Website

<http://xml.apache.org/xerces-j>

6.11 Bean Scripting Framework Website

<http://oss.software.ibm.com/developerworks/projects/bsf>

6.12 Rhino Website

6.13 <http://www.mozilla.org/rhino/>



7 RCS – Configuration Framework

7.1 Purpose

This Unit Test Report documents the test conditions and test script of the ITA R3.0 Reusable Common Services (RCS) Configuration framework. This report also provides the expected results and actual results from running the test script.

7.2 Approach

To ensure quality of the RCS, the Configuration framework went through extensive unit testing. ITA conducted automatic unit testing of this framework.

Benefits to the unit test approach are:

- Standardize test conditions and cycles
- Increase code quality
- Increase consistency in the approach to testing
- Increase productivity
- Reduce time for regression testing
- More time available to spend on enhancements as less time is required for fixes

7.3 Background

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

The ITA configuration framework is implemented using the Accenture's GRNDS (General and Reusable Netcentric Delivery Solution) configuration framework. The GRNDS code has been extended to meet FSA application development requirements. Specifically, the framework has been extended to:

- Use a static initializer to load the configuration files, instead of using the GRNDS bootstrap framework.
- Support configuration input from database tables.

7.4 Testing Environment

The unit test for the Configuration framework was automated by using JUnit. JUnit is a set of Java packages that allows developers to readily create Java test cases for Java



classes, and to then run these unit tests interactively or in batch mode. The unit test was conducted on a Sun SPARC machine running Solaris 2.6 interacting with a client browser running on a Windows 2000 machine. The focus of this unit test is to identify that the Configuration framework is functioning as designed.

errorMessages.properties

```
# RCS Exception Handling Messages
# This file contains mapping information from error codes to error messages

# 601 -700 Errors in the ConfigurationFramework:
msg601=Could not initialize {0}
msg602=Error occurred during {0} finalization
msg603=Error while reading properties file
msg604=Error accessing {0} class
msg605=Error instantiating {0} objects
msg606=I/O error occurred parsing configuration documents
msg607=Runtime exception occurred. Be sure xml resources are in classpath
```

masterBasic_app2.properties

```
app2_key=application2_text
name=firstName
```

7.4.1 XML Files

One xml file was used in the Configuration Framework. The file masterBasic_app1.xml was used to test the xml file portion of the Configuration Framework.

MasterBasic_app1.xml

```
<?xml version="1.0"?>
<App1>
  <name>Application1</name>
  <app1_key>Application1 key</app1_key>
</App1>
```

7.4.2 Database tables

Three database tables were used in the Configuration Framework. They were used to test the database table portion of the Configuration Framework.

CONFIG

PROPERTY_ID	DOMAIN_ID
1	1
2	2
3	1
4	3
4	1
5	3
6	4



7	4
4	5
6	6
7	6
8	6
5	5
9	5
10	5

PROPERTY_DOMAIN

DOMAIN_ID	DOMAIN_NAME	PARENT_ID
1	Resource	0
2	fr	1
3	Master	0
4	app1	3
5	MasterBasic	0
6	app3	5

PROPERTY

PROPERTY_ID	PROPERTY_KEY	PROPERTY_VALUE
1	button.ok	OK
2	button.ok	Yessir
3	test.try	Crazy
4	Name	Test db name
5	appDomain.App1	MasterBasic_app1.xml
6	app1_key	application1db_text
7	name	app1 db name
8	app3_key	application3_text
9	appDomain.App2	masterBasic_app2.properties
10	appDomain.App3	masterBasic_app3

7.4.3 WebSphere Application Server – Configuration Framework Configuration

The name of the master domain needs to be placed on the command line of the WebSphere console. The following is an example:



Application Server: CONV

General | Advanced | Debug

Application Server Name: CONV

Current State: Running

Desired State: Running

Start Time: Aug 8, 2002 1:09:46 PM

Executable in use: /opt/dev35/WebSphere/AppServer/java/jre/bin/java

Command line arguments: wutonomy.xml - DmasterConfig=master.properties

Environment: Environment...

Process ID: 4894

Working directory:

Standard input: /dev/null

Standard input in use: /dev/null

Console Messages

```
8/8/02 5:38 PM : Loading ...
8/8/02 5:39 PM : Console Ready.
```

7.5 Automated Testing Conditions

Component Name	Configuration Framework	Version #	1
File Name	FSADatabaseSource.java		



ITA Release 3.0 Build & Test Report

Prepared By	Kirsten Metzler	Date Prepared	4/30/2002
Tested By	Kirsten Metzler	Date Testing Finished	5/10/2002
Reviewed By	Wayne Chang	Date Reviewed	4/31/2002

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Database Table
1	Valid domain, no subdomains	TestDatabaseSource	testValidDomainNoSubdomain	FSADatabaseSource	getEnvironment	database domain data is loaded into the environment cache	config, property, property_domain
2	Valid domain, valid subdomain	TestDatabaseSource	testValidDomainValidSubdomain	FSADatabaseSource	getEnvironment	database domain data and subdomain data is loaded into the environment cache	config, property, property_domain
3	Invalid domain	TestDatabaseSource	testInvalidDomain	FSADatabaseSource	getEnvironment	database domain data is not loaded. A message is written to the logs that the file could not be found.	config, property, property_domain
4	Valid domain, invalid subdomain	TestDatabaseSource	testInvalidSubDomain	FSADatabaseSource	getEnvironment	database domain data is loaded into the environment cache, database subdomain data is not loaded and a message is written to the logs.	config, property, property_domain
5	Could not get database connection	TestDatabaseSource	testDatabaseConnection	FSADatabaseSource	getEnvironment	Error is written to the log, no data is loaded. An exception is thrown.	config, property, property_domain
6	Database tables do not exist	TestDatabaseSource	testNoDatabaseTables	FSADatabaseSource	getEnvironment	Error is written to the log, no data is loaded. An exception is thrown.	config, property, property_domain

Component Name	Configuration Framework	Version #	1
File Name	FSAXmlFileSource.java		
Prepared By	Kirsten Metzler	Date Prepared	4/30/2002
Tested By	Kirsten Metzler	Date Testing Finished	5/10/2002
Reviewed By	Wayne Chang	Date Reviewed	4/31/2002



#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Valid domain, no subdomains	TestXmlFileSource	testValidDomainNoSubdomain	FSAXmlFileSource	getEnvironment	properties domain file is loaded into the environment cache	master.xml
2	Valid domain, valid subdomain	TestXmlFileSource	testValidDomainValidSubdomain	FSAXmlFileSource	getEnvironment	properties domain file and subdomain file is loaded into the environment cache	master.xml, master_app1.xml
3	Invalid domain	TestXmlFileSource	testInvalidDomain	FSAXmlFileSource	getEnvironment	properties domain file is not loaded. A message is written to the logs that the file could not be found.	none.
4	Valid domain, invalid subdomain	TestXmlFileSource	testInvalidSubDomain	FSAXmlFileSource	getEnvironment	properties domain file is loaded into the environment cache, properties subdomain file is not loaded and a message is written to the logs.	master.xml
5	Relative path	TestXmlFileSource	testRelativePath	FSAXmlFileSource	getEnvironment	properties domain file is loaded into the environment cache.	master.xml
6	Absolute path	TestXmlFileSource	testAbsolutePath	FSAXmlFileSource	getEnvironment	properties domain file is loaded into the environment cache.	master.xml

Component Name	Configuration Framework	Version #	1
File Name	FSConfigurationSI.java		
Prepared By	Kirsten Metzler	Date Prepared	4/30/2002
Tested By	Kirsten Metzler	Date Testing Finished	5/10/2002
Reviewed By	Wayne Chang	Date Reviewed	4/31/2002



#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name	Database Table
1	Master config is a properties file	TestFSAConfigurationSIProp	TestMasterProperties	FSAConfigurationSI	init	master properties file is loaded into the environment cache.	master.properties	
2	Master config is an xml file	TestFSAConfigurationSIXml	TestMasterXml	FSAConfigurationSI	init	master xml file is loaded into the environment cache.	master.xml	
3	Master config is the database.	TestFSAConfigurationSIDb	TestMasterDatabase	FSAConfigurationSI	init	master data is loaded from the database into the environment cache	None.	config, property, property_domain
4	Master config properties file contains xml subdomain	TestFSAConfigurationSIProp	TestMasterPropertiesSubXml	FSAConfigurationSI	init	master properties file and subdomain xml file data is loaded into the environment cache.	master.properties, master_app1.xml	
5	Master config properties file contains properties subdomain	TestFSAConfigurationSIProp	TestMasterPropertiesSubProperties	FSAConfigurationSI	init	master properties file and subdomain properties file data is loaded into the environment cache	master.properties, master_app1.properties	
6	Master config properties file contains database subdomain	TestFSAConfigurationSIProp	TestMasterPropertisSubDatabase	FSAConfigurationSI	init	master properties file and subdomain database data is loaded into the environment cache.	master.properties	config, property, property_domain
7	Master config xml file contains xml subdomain	TestFSAConfigurationSIXml	TestMasterXmlSubXml	FSAConfigurationSI	init	master xml file and subdomain xml file data is loaded into the environment cache	master.xml, master_app1.xml	
8	Master config xml file contains properties subdomain	TestFSAConfigurationSIXml	TestMasterXmlSubProperties	FSAConfigurationSI	init	master xml file and subdomain properties file data is loaded into the environment cache	master.xml, master_app1.properties	



9	Master config xml file contains database subdomain	TestFSAConfigurationSIxml	TestMasterXmlSubDatabase	FSAConfigurationSI	init	master xml file and subdomain database data is loaded into the environment cache.	master.xml	config, property, property_domain
10	Master config database contains xm subdomain	TestFSAConfigurationSIDb	TestMasterDatabaseSubXml	FSAConfigurationSI	init	master database and subdomain xml data is loaded into the environment cache	master_app1.xml	config, property, property_domain
11	Master config database contains properties subdomain	TestFSAConfigurationSIDb	TestMasterDatabaseSubProperties	FSAConfigurationSI	init	master database and subdomain properties data is loaded into the environment cache	master_app1.properties	config, property, property_domain
12	Master config database contains database subdomain	TestFSAConfigurationSIDb	TestMasterDatabaseSubDatabase	FSAConfigurationSI	init	master database and subdomain database data is loaded into the environment cache		config, property, property_domain

Component Name	Configuration Framework	Version #	1
File Name	FSAConfigurationSI.java		
Prepared By	Kirsten Metzler	Date Prepared	4/30/2002
Tested By	Kirsten Metzler	Date Testing Finished	5/10/2002
Reviewed By	Wayne Chang	Date Reviewed	4/31/2002

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Master configuration file/database does not exist	TestMasterDoesNotExist	TestNoMaster	FSAConfigurationSI	init	Error is written to the log and no data is loaded.	None.



Component Name	Configuration Framework	Version #	1
File Name	FSAConfigurationSI.java		
Prepared By	Kirsten Metzler	Date Prepared	4/30/2002
Tested By	Kirsten Metzler	Date Testing Finished	5/10/2002
Reviewed By	Wayne Chang	Date Reviewed	4/31/2002

#	Detailed Condition	Test Class Name	Test Class Method	Class Name	Method Name	Results	Data File Name
1	Configuration type not properties, xml or database	TestBadConfigType	TestBadConfig	FSAConfigurationSI	init	Error is written to the log and no data is loaded.	None.



7.6 Performance Testing

7.6.1 Approach

To ensure program efficiency and to detect possible bottleneck, ITA used JProbe to analyze the Configuration framework. JProbe is a performance-profiling tool and it was used to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming. In order to profile this framework, portions of the unit test scripts were used to conduct this test. The performance analysis of this framework is documented in this report.

Two key groups of statistics are collected from the JProbe Profiler: the memory (heap) usage and the time spent on each method within the program (performance detail). This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow gathering detailed information on individual methods and the interaction between them.

7.6.2 Summary

This report contains the background information, performance test harness design, performance analyses, and resulting performance metrics for the framework. Profiling the Configuration framework using the test scripts will test the code performance of the framework. The actual results will be compared against the results of how this framework is expected to function.

7.7 Test Harness Design

7.7.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance test is to identify loitering objects and time spent on each method relative to each other in the Configuration framework.

7.7.2 Test Configuraton

There is very little configuration that needs to be done for the ITA Configuration framework itself. There are two system level properties that need to be configured within the WebSphere administartion console.

- A system variable “masterConfig” must be added to command line with value set to the master configuration domain
- The ServletInitializer system variable must be updated to contain a call run the servlet initializer with the FSAConfigurationSI class

JProbe needs to be configured to be able to gather metrics from the framework as it ran in the IBM Websphere environment. A .jpl file, which lists all the profiling parameters, must be created. These parameters are normally set in the JProbe GUI, but since a server is being monitored, they are set through a file interface. The text of the configuration file is provided below:



7.7.3 WebSphere Application Server Configuration

The WebSphere Command Line will identify the JProbe configuration file to use and ensure that the correct JVM is used. Two Environment Variables will be added to the Application Server and two servlets will be added to the Web Application. The Session Manager configurations have to be updated, more information on how to update the settings can be found in the User Session Framework User Guide document.

7.7.3.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/06052002_test_sessions.jpl -Xnoclassgc -  
Djava.compiler=NONE -ms128m -mx128m
```

7.7.3.2 Environment:

```
EXECUTE=YES  
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```

7.8 Testing Scenarios

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.

7.9 Analysis

The analysis consists of three parts:

1. Memory (Heap) Usage: Examines how the memory (heap) is used by the RCS Java code to identify loitering object and over-allocation of objects.
2. Garbage Collection: The garbage collector is a process that runs on a low priority thread. When the JVM attempts to allocate an object but the Java heap is full, the JVM calls the garbage collector. The garbage collector frees memory using some algorithm to remove unused objects. Examining the activities of the garbage collection will give a good indication of the performance impact of the garbage collector on the application.
3. Code Efficiency: To identify any performance bottlenecks due to inefficient code algorithms

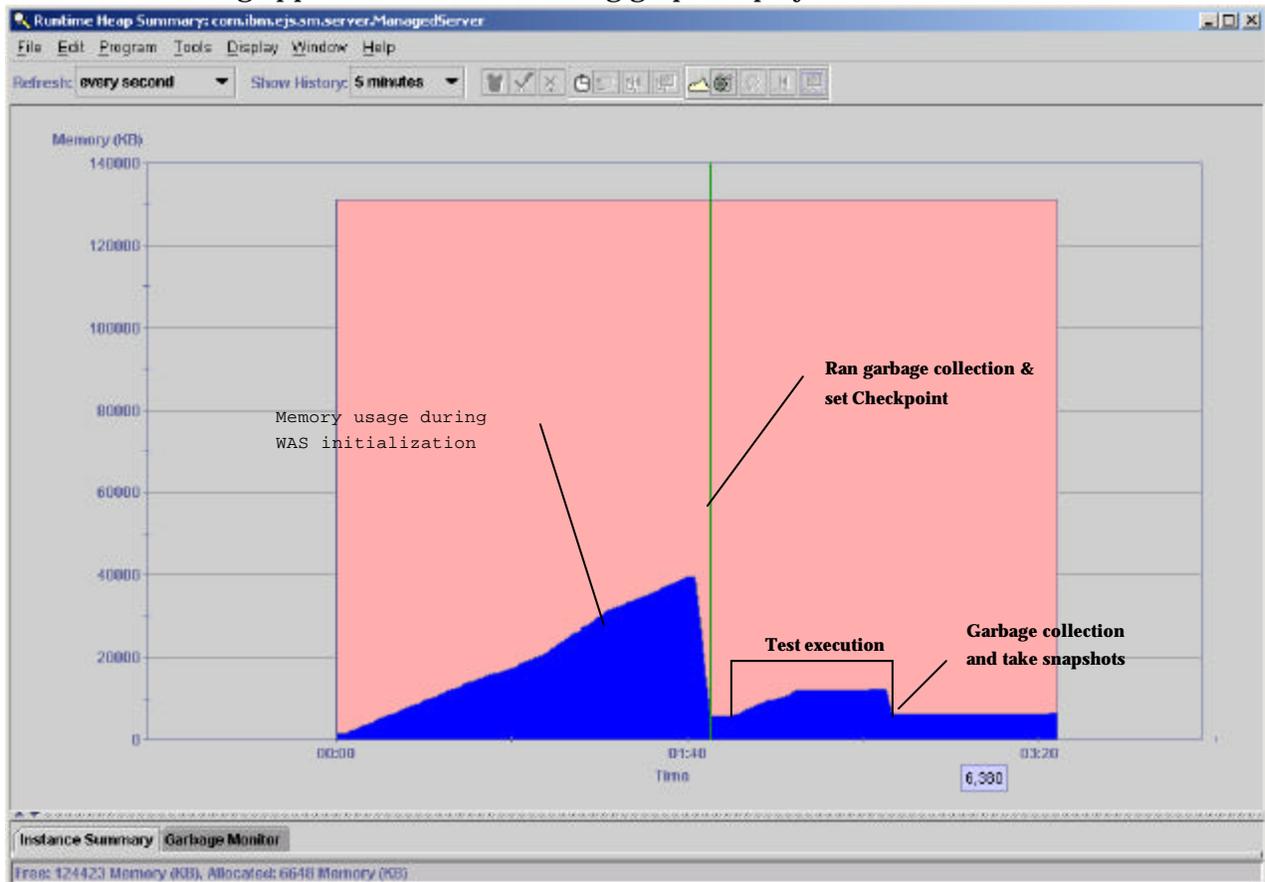
7.9.1 Memory (Heap) Usage

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.

7.9.2 Heap Graph 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. The overall memory usage for the Configuration framework is very low and will not result in huge increase to the overhead of calling applications. The following graph displays this information.



7.10 Instance Summary

The table below is a section of the Instance Summary result associated with conducting the test. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory those instances consume.

In the heap graph in the previous section, there is a green vertical line that shows where the checkpoint was set. The checkpoint tells JProbe to tag all subsequently created objects as “new.” The Count Change and Memory Change columns show data regarding new instances (created after the checkpoint) that are currently in the heap.



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. 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.

Package	Class	Count	Count Change	Cumulative Count	Memory
Total		124,314 (100.0%)	+20,796	678,636 (100.0%)	5,309,794 (100.0%)
com.ibm.ejs.util.cache	Bucket	8,190 (6.6%)	+4,096	8,190 (1.2%)	229,320 (4.3%)
	char[]	28,029 (22.5%)	+8,677	201,455 (30.0%)	2,167,804 (40.8%)
java.lang	String	27,563 (22.2%)	+9,018	166,214 (24.0%)	390,636 (6.2%)
java.lang	StringBuffer	644 (0.5%)	+639	76,462 (11.4%)	7,728 (0.1%)
java.util	Hashtable\$Entry	5,418 (4.4%)	+624	9,612 (1.4%)	100,360 (2.0%)
java.lang	Class	2,343 (1.9%)	+436	2,343 (0.3%)	328,020 (6.2%)
	byte[]	763 (0.6%)	+420	38,248 (5.7%)	441,506 (8.3%)
java.text	FieldPosition	356 (0.3%)	+356	1,348 (0.2%)	4,272 (0.1%)
com.ibm.ejs.util	Bucket	2,167 (1.7%)	+517	2,167 (0.3%)	26,004 (0.5%)
	short[]	242 (0.2%)	+239	245 (0.0%)	9,112 (0.2%)
	int[]	206 (0.2%)	+167	9,367 (1.4%)	32,176 (0.6%)
java.lang	Integer	629 (0.5%)	+153	3,018 (0.5%)	2,516 (0.0%)
java.util	Vector	840 (0.7%)	+141	1,075 (0.2%)	16,800 (0.3%)
com.ibm.service.util	ListEntry	120 (0.1%)	+100	145 (0.0%)	2,400 (0.0%)
java.util	Locale	120 (0.1%)	+98	120 (0.0%)	2,400 (0.0%)
com.ibm.service.util	RegexNode	112 (0.1%)	+75	132 (0.0%)	4,032 (0.1%)
com.ibm.ejs.ras	TraceComponent	287 (0.2%)	+70	287 (0.0%)	17,220 (0.3%)
java.math	BigInteger	70 (0.1%)	+70	70 (0.0%)	1,960 (0.0%)
oracle.jdbc.util	SQLStateRange	59 (0.0%)	+59	59 (0.0%)	708 (0.0%)
	double[]	61 (0.0%)	+58	61 (0.0%)	1,900 (0.0%)
oracle.jdbc.OC7	TTCItem	45 (0.0%)	+45	75 (0.0%)	1,620 (0.0%)
java.util	Date	40 (0.0%)	+41	173 (0.0%)	576 (0.0%)

7.11 Garbage Collections

The Garbage Monitor was used to identify the classes that are responsible for large allocations of short-lived objects. It shows the cumulative results of successive garbage collections during the session. The Garbage Monitor shows only the top ten classes, representing the classes with the most instances garbage collected. During the session, the top ten classes will change as the number of garbage collected objects accumulates.

There was not any unexpected activity shown, or activity that would indicate a performance problem. Most of the objects created are either strings, string buffers, or character arrays. These



numbers are in line with the framework requirements and expected behavior as it formats a large number of messages.

7.12 Resources

7.12.1 GRNDS Framework

<https://onesource.accenture.com>

7.12.2 Sun Java website

<http://java.sun.com>

8 RCS - JSP Custom Tag Library Framework

8.1 JSP Custom Tag Library Unit Test Report

8.1.1

8.1.1.1 Purpose

This Unit Test Report documents the test conditions and test script of the ITA R3.0 Reusable Common Services (RCS) JSP Custom Tag Library framework. This report also provides the expected results and actual results from running the test applications.

8.1.1.2 Approach

To ensure quality of the RCS, the JSP Custom Tag Library framework went through extensive unit testing. ITA conducted manual unit testing of this framework.

Benefits to the unit test approach are:

- Standardize test conditions and cycles
- Increase code quality
- Increase consistency in the approach to testing
- Increase productivity
- Reduce time for regression testing
- More time available to spend on enhancements as less time is required for fixes

8.1.1.3 Background

The purpose of the ITA RCS JSP Custom Tag Library framework is to provide a set of custom tags for developers to utilize to simplify, standardize, and extend the use of JSP tag libraries within the J2EE standard.



8.1.2 Test Design

8.1.2.1 Testing Environment

The unit test for the JSP Custom Tag Library framework was conducted manually. The unit test was conducted on a Sun SPARC machine running Solaris 2.6 interacting with a client browser running on a Windows 2000 machine. Both Microsoft Internet Explorer 5.0.1 and Netscape Navigator 6.2 client browsers were used to conduct the tests scripts. The focus of this unit test is to identify that the JSP Custom Tag Library framework is functioning as designed.

8.1.2.1.1 Testing Cycles

The RCS JSP Custom Tag Library framework is created using the Java programming language. The tag library framework provides a collection of commonly used JSP custom tag libraries for JSP developers to access. The JSP Tag Library framework is comprised of libraries leveraged from the Jakarta Struts framework, Apache Taglibs project, and custom developed libraries.

Eleven tag libraries were tested as part of this framework:

Test Cycle 1	Jakarta Struts Bean Taglib
Test Cycle 2	Jakarta Struts HTML Taglib
Test Cycle 3	Jakarta Struts Logic Taglib
Test Cycle 4	Jakarta Struts Template Taglib
Test Cycle 5	Jakarta DateTime Taglib
Test Cycle 6	Jakarta I18N Taglib
Test Cycle 7	Jakarta Input Taglib
Test Cycle 8	Logging Taglib
Test Cycle 9	Jakarta Page Taglib
Test Cycle 10	Jakarta XSL Taglib
Test Cycle 11	Jakarta XTags Taglib

8.1.2.2 Testing Configuration

In order to test the JSP Custom Tag Library framework, several JavaServer Pages had to be developed to utilize the different tags available within each library. An existing development Application Server (CONV) was used to conduct the tests, with some modification to the Session Manager settings and directory structure.

8.1.2.2.1 UNIX Server Settings

The WebSphere properties files have not been updated and the existing settings are used to run the test cycles.

The following sections list the properties related to the Web Application created to unit test the JSP Custom Tag Library framework. The configuration settings used in the Administration Console is defined in the next topic.



8.1.2.2.1.1 rules.properties:

```
default_host/CONVWebApp/*.activity=ibmoselink15
default_host/CONVWebApp/*.jsp=ibmoselink15
default_host/CONVWebApp/ErrorHandler=ibmoselink15
default_host/CONVWebApp/servlet/=ibmoselink15
default_host/CONVWebApp/servlet/messagerouter=ibmoselink15
default_host/CONVWebApp/servlet/rpcrouter=ibmoselink15
default_host/CONVWebApp/servlet=ibmoselink15
```

8.1.2.2.1.2 queues.properties:

```
ose.srvgrp.ibmoselink15.clone1.port=8400
ose.srvgrp.ibmoselink15.clone1.type=remote
ose.srvgrp.ibmoselink15.clonescount=1
ose.srvgrp.ibmoselink15.type=FASTLINK
ose.srvgrp=ibmoselink,ibmoselink1,ibmoselink2,ibmoselink3,ibmoselink4,ibmoselink5,ibmoselink6,ibmoselink7,ibmoselink9,ibmoselink8,ibmoselink10,ibmoselink12,ibmoselink13,ibmoselink14,ibmoselink15,ibmoselink16,ibmoselink17,ibmoselink18
```

vhosts.properties:

```
dev.conv.sfa.ed.gov\8531=default_host
```



8.1.2.2.2 Directory Structure



8.1.3 Testing Conditions and Results

Eleven sets of applications have been created to test the different functionality available within the JSP Custom Tag Library framework. The files associated with each set of the test applications have been placed into separate directories: Bean, datetime, html, i18n, input, logging, logic, page, template, xsl, and xtags. See the previous section for the complete path to the directory.

Of the eleven tag libraries tested, two tag libraries were removed from this framework due to incompatibilities found during testing. The versions of xerces.jar used by those tag libraries and the version currently used for FSA development are different. All frameworks must be regression tested in order to include these two tag libraries in this framework. The scope of such an effort is beyond the current bandwidth available to perform the regression testing, so these two tag libraries may be included in a future release.

The URLs to access the different pages of the test applications are in the following format:
http://dev.conv.sfa.ed.gov:8531/CONVWebApp/jsptags/<tag library>/*.jsp².

An example URL is:

<http://dev.conv.sfa.ed.gov:8531/CONVWebApp/jsptags/datetime/datetimeTest.jsp>

All test cycles were conducted using Microsoft's Internet Explorer (IE). Netscape Navigator was used for testing a few of the applications to ensure that the framework will behave as expected in another browser.

For each test cycle, there is a 'Test Findings' section, which contains the results gathered from running the test JavaServer Pages. Any attributes from leveraged tag libraries that did not working as documented by Jakarta will be noted. It is up to the developer to ensure that any tags that are used are thoroughly tested with his/her application and servlet container.

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



8.1.3.1 Test Cycle 1 – Jakarta Struts Bean Taglib

The "struts-bean" tag library contains JSP custom tags useful to defining new beans (in any desired scope), from a variety of possible sources, as well as a tag to render a particular bean (or bean property) to the output response.

Tags:

Class	Test Page
CookieTag* ³	bean-cookie.jsp
Define Tag*	bean-define.jsp, bean-define2.jsp
Header Tag*	bean-header.jsp, bean-header_multi.jsp
IncludeTag*	bean-include.jsp
MessageTag	bean-message.jsp
PageTag*	bean-page.jsp
ParameterTag*	bean-parameter-InputMulti.jsp, bean-parameter.jsp
ResourceTag*	bean-resource.jsp
SizeTag*	bean-size.jsp
StrutsTag*	bean-struts.jsp
WriteTag	bean-write.jsp, bean-write2.jsp

TEST FINDINGS:

CookieTag

Four JavaServer Pages were created to test the cookieTag in this tag library. The first JSP will set cookies to be displayed by the second JSP. The third JSP will set a new cookie with the same name as a cookie set in the first JSP but with a different path. The fourth JSP will be used to display it to be sure the 'multiple' attribute is working.

DefineTag

Jakarta's documentation does not note that to obtain the value for a bean originally defined with a scope, the scope in the define tag must be defined; otherwise, a 'null' will be returned. The code pertaining to the boolean and int in the second JSP have been commented out. To test the scope of the <bean:define> tag, uncomment one and an error message will result saying that the looked for bean is out of scope.

³ Note: * next to a tag indicates that the TagExtraInfo (TEI) implementation of the class is available (i.e.. CookieTag* means that there is a CookieTei. All TEI classes contain the method getVariableInfo, which returns information about the scripting variable to be created).



HeaderTag

The second test JSP is a modified version of the first JSP to test the 'multiple' attribute. The testing does not show that the result is displayed any differently between the two JSPs. This could be attributed to how the application is designed or the servlet container being used since as other tags have behaved differently when running different containers. It is the developer's responsibility to ensure the tag works appropriately for his/her application.

MessageTag

This jsp tests the use of the Resource.properties file located in the /www/dev/conv/servlets directory. The attributes: locale and bundle were not tested.

SizeTag

The test application used is an example application from the tag library distribution. The scope attribute was not tested in this sample application.

WriteTag

The first JSP tests that bean:write can output a bean. The second JSP tests the scope attribute. The JSP will display an error if it can't find the bean in the scope regardless of if the bean exists.



8.1.3.2 Test Cycle 2 - Jakarta Struts HTML Taglib

The "struts-html" tag library contains JSP custom tags for creating dynamic HTML user interfaces, including input forms.

Tags:

Class	Test Page
BaseTag	html-form.jsp, html-link.jsp
ButtonTag	html-form.jsp
CancelTag	html-form.jsp, html-select.jsp
CheckboxTag	html-form.jsp, html-link.jsp
ErrorsTag	html-form.jsp
FileTag	html-form.jsp
FormTag	html-form.jsp
HiddenTag	html-form.jsp
HtmlTag	html-form.jsp
ImageTag	html-form.jsp
ImgTag	html-form.jsp
LinkTag	html-link.jsp
MultiboxTag	html-form.jsp
OptionsTag	html-select.jsp
OptionTag	html-select.jsp
PasswordTag	html-form.jsp
RadioTag	html-form.jsp
ResetTag	html-form.jsp, html-select.jsp
RewriteTag	Not Tested
SelectTag	html-select.jsp
SubmitTag	html-form.jsp, html-select.jsp
TextareaTag	html-form.jsp
TextTag	html-form.jsp, html-link.jsp



TEST FINDINGS:

ButtonTag

In testing, the 'accesskey' attribute did not work. This could be due to the design or servlet container used. It is the developer's responsibility to ensure that this tag works with his/her application.

ImageTag

This tag did not work in testing until the WebSphere Session Manager - URL Rewriting option was disabled. Otherwise, the session identifier was inserted at the end of the URL, which will cause the image link to fail. The 'border' attribute for this tag was not working in the test application.

ImgTag

The 'page' attribute does not work for this tag, either the 'src' or 'srcKey' attributes should be used instead. This attribute does not work as it is a local reference and the images are stored on the web server and not the application server, which causes it to not find the image and fails.

LinkTag

The 'target' and 'transaction' attributes were not tested.



8.1.3.3 Test Cycle 3 - Jakarta Struts Logic Taglib

The "struts-logic" tag library contains tags that for managing conditional generation of output text, looping over object collections for repetitive generation of output text, and application flow management.

Tags:

Class	Test Page
EqualTag	createCookie.jsp, enterName.jsp, logic-equal.jsp
ForwardTag	logic-equal.jsp
GreaterEqualTag	createCookie.jsp, logic-greaterEqual.jsp
GreaterThanTag	logic-greaterEqual.jsp
IterateTag*	see /bean/bean-header_multi.jsp
LessEqualTag	logic-greaterEqual.jsp
LessThanTag	logic-greaterEqual.jsp
MatchTag	createCookie.jsp, enterAddress.jsp, logic-match.jsp
NotEqualTag	logic-equal.jsp
NotMatchTag	logic-match.jsp
NotPresentTag	logic-equal.jsp
PresentTag	logic-equal.jsp
RedirectTag	createCookie.jsp, logic-match.jsp

TEST FINDINGS:

EqualTag & NotEqualTag

The 'scope' attribute was not tested for either tag.

RedirectTag

The redirect tag contains a bug: <http://archive.covalent.net/jakarta/struts-dev/2001/12/0064.xml>. Developers should test this tag to ensure it works properly before using it in a production environment.

IterateTag

The iterate tag was tested when testing the bean tag library and is not tested again here.



8.1.3.4 Test Cycle 4 - Jakarta Struts Template Taglib

The "struts-template" tag library contains tags that are useful for creating dynamic JSP templates for pages that share a common format.

Tags:

Class	Test Page
GetTag	chapterTemplate.jsp
InsertTag	introduction.jsp
PutTag	introduction.jsp

TEST FINDINGS:

The template tag library works as intended and no additional results are reported.



8.1.3.5 Test Cycle 5 - Jakarta DateTime Taglib

The DateTime custom tag library contains tags, which can be used to handle date, and time related functions. Tags are provided for formatting a Date for output, generating a Date from HTML form input, using time zones, and localization.

Tags:

Class	Test Page
AmPmsTag*	datetimeTest.jsp
CurrentTimeTag	setzone.jsp
ErasTag*	datetimeTest.js
FormatTag	datetimeTest.jsp, setzone.jsp
MonthsTag*	datetimeTest.jsp
ParseTag	datetimeTest.jsp, setzone.jspn
TimeZonesTag*	datetimeTest.jsp
TimeZoneTag	datetimeTest.jsp, setzone.jsp
WeekdaysTag*	datetimeTest.jsp

TEST FINDINGS:

In the format tag, the 'date' attribute does not work when the JSP was tested in WebSphere, but did work when tested using Jakarta Tomcat. Developers should test the any applications that use this tag attribute thoroughly.



8.1.3.6 Test Cycle 6 - Jakarta I18N Taglib

The i18n custom tag library contains tags that help manage the complexity of creating multi-lingual web applications. These tags provide similar (though not identical) functionality to the internationalization available in the Struts framework, but do not require adopting the entire Struts framework.

Tags:

Class	Test Page
BundleTag*	i18nTest.jsp
FormatCurrencyTag	format-include.jsf
FormatDateTag*	format-include.jsf, formatLocale.jsf
FormatDateTimeTag*	format-include.jsf, formatLocale.jsf
FormatNumberTag*	format-include.jsf, formatLocale.jsf
FormatPercentTag*	format-include.jsf, formatLocale.jsf
FormatStringTag*	format.jsp
FormatTimeTag*	format-include.jsf, formatLocale.jsf
IfdefTag	ifdef.jsf
IfndefTag	ifndef.jsf
LocaleTag*	format.jsp
MessageArgumentsTag	message.jsf
MessageTag*	message.jsf

TEST FINDINGS:

The bundle base tag is the location to the *.properties files starting from the /www/dev/conv/servlets directory. The bundle:debug and message:debug attributes were not tested. The id attribute for the formatXXX tags did not work properly in testing during testing.



8.1.3.7 Test Cycle 7 - Jakarta Input Taglib

The input tag extension library features the presentment of HTML <form> elements that are tied to the ServletRequest calling the JSP page. Forms elements can be pre-populated with prior values that the user has chosen – or with default values for the first time user of a web page. This is useful when the same page needs to be presented to the user several times. Server-side validation is a good example of this process.

It is also possible to automatically build up <select> boxes, making it easier to build data-driven forms. Even if the same page is presented multiple times, and the form elements that have default values are desired, this library provides this functionality to free programmers from writing extensive code.

Tags:

Class	Test Page
Checkbox	inputTest.jsp
Radio	inputTest.jsp
Select	inputTest.jsp
Text	inputTest.jsp
TextArea	inputTest.jsp

TEST FINDINGS:

In the inputTest.jsp file for the select attribute, the line `map.put("multiple", null)` is commented out so the select display is a drop down list. If it is not commented out, the select display becomes a list.



8.1.3.8 Test Cycle 8 - Logging Taglib

The Log library allows embedding logging calls in JSP using the ITA RCS logging framework. This tag library is leveraged from the ITA RCSlogging framework. It has the ability to log messages and test if a given level can be logged based on the current settings.

Tags:

Class	Test Page
CanLogTag	testlog.jsp
CanNotLogTag	testlog.jsp
SyslogTag	testlog.jsp

TEST FINDINGS:

The current logging level in the rcs.xml file was changed to different levels to test this tag library. The testlog.jsp executed in the browser and the results displayed varied based on the current logging level set.



8.1.3.9 Test Cycle 9 - Jakarta Page Taglib

Used to access all of the information about the PageContext of a JSP page.

Tags:

Class	Test Page
AttributeTag	pageTest.jsp
AttributesTag*	pageTest.jsp
EqualsAttributeTag	pageTest.jsp
ExistsAttributeTag	pageTest.jsp
RemoveAttributeTag	pageTest.jsp
SetAttributeTag	pageTest.jsp

TEST FINDINGS:

<http://dev.conv.sfa.ed.gov:8531/CONVWebApp/jsptags/page/pageTest.jsp>



8.1.4 Resources

- Best Practices for Session Programming: WebSphere Application Server
 - <http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/0404010108.html>

8.2 JSP Custom Tag Library Performance Analysis

8.2.1

8.2.2 Purpose

This Performance Analysis Report documents the results of utilizing JProbe to test the ITA R3.0 Reusable Common Services (RCS) JSP Custom Tag Library framework. This report provides an in-depth analysis of the results gathered from the JProbe application profiling and documents any performance issues and suggests resolutions. The series of JSP Custom Tag Library framework documentation will enable developers to quickly build applications using the JSP Custom Tag Library within the ITA environment architecture.

8.2.3 Approach

To ensure program efficiency and to detect possible bottleneck, ITA used JProbe to analyze the JSP Custom Tag Library framework. Only the Custom Logging Tag Library (which utilizes RCS Logging framework) was profiled as it is built completely by the ITA team and not leveraged from external sources.

JProbe is a performance-profiling tool and it was used to detect performance issues such as loitering objects, unexpected references, and over-use of objects in Java based programming. In order to profile this framework, the unit test application was used to conduct this test. The performance analysis of this framework is documented in this report.

Two key groups of statistics are collected from the JProbe Profiler: the memory (heap) usage and the time spent on each method within the program (performance detail). This tool can be used to identify loitering objects and inefficiencies in code more easily. JProbe also contains the capabilities to drill-down and allow gathering detailed information on individual methods and the interaction between them.

8.2.4 Summary

This report contains the background information, performance test harness design, performance analyses, and resulting performance metrics for the framework. Profiling the JSP Custom Tag Library framework using the test JSP will test the code performance of the framework. The



actual results will be compared against the results of how this framework is expected to function.



8.2.5 Test Harness Design

8.2.5.1 Testing Environment

The performance test was conducted on a Sun SPARC machine running Solaris 2.6. The focus of this performance test is to identify loitering objects and time spent on each method relative to each other in the Logging Tag Library within the JSP Custom Tag Library framework.

8.2.5.1.1 Testing Criteria

The two main components of the Logging Tag Library will be tested: the ability to write data to a log file and also check whether a given level can or cannot be logged. Since the Logging Tag Library is an API, a JavaServer Page was developed for the unit test to serve as a test harness to profile and analyze the performance of the various methods.

8.2.5.2 Testing Configuration

In order to profile the Logging Tag Library using the test application and JProbe, the JPROBE Application Server was used and some of the configurations were changed. In the command line reference of the Application Server, there is a reference to the JProbe configuration file. The file used to conduct this performance analysis is:

`/opt/util/JProbe/jpl_files/08022002_test_jsptags.jpl`. The action, database, and HelloWorld servlets were all disabled.

8.2.5.2.1 JProbe Configuration File

The JProbe configuration file has a file extension of `.jpl`. This file contains all of the settings that JProbe requires to profile an application, applet, or server side component (such as JavaServer Pages and Servlets). The configuration file will determine which JVM is used to run JProbe and the monitoring options. The user will be able to specify the activity of the Profiler. For example, the file can be configured to cause JProbe Profiler to take a heap snapshot before it exits and the directory to save the snapshots in.

The example application test will be conducted on the Solaris machine with the output being sent to a remote Windows NT workstation. Performance and heap snapshots will be taken before the application is exited. The configuration in the actual file used to conduct the test can be found in [Appendix A](#). A filter for the main package, `gov.ed.fsa.ita.jsptags`, was added to narrow the scope of the test to this package.

8.2.5.2.2 UNIX Server Settings

The usage of the User Session framework is closely tied to how the WebSphere Session Manager is configured. The WebSphere properties files have not been updated to run the test cycles.

The following sections list the properties related to the Web Application created to unit test the User Session framework. The configuration settings used in the Administration Console is defined in the next topic.



8.2.5.2.2.1 rules.properties:

```
default_host/JPROBEWebApp/*.do=ibmoselink4
default_host/JPROBEWebApp/*.jsp=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/*.jsw=ibmoselink4
default_host/JPROBEWebApp/=ibmoselink4
default_host/JPROBEWebApp/ErrorHandler=ibmoselink4
default_host/JPROBEWebApp/servlet=ibmoselink4
default_host/JPROBEWebApp=ibmoselink4
```

8.2.5.2.2.2 queues.properties:

```
ose.srvgrp.ibmoselink4.clone1.port=8241
ose.srvgrp.ibmoselink4.clone1.type=remote
ose.srvgrp.ibmoselink4.clonescount=1
ose.srvgrp.ibmoselink4.type=FASTLINK
ose.srvgrp=ibmoselink3,ibmoselink2,ibmoselink4,ibmoselink17
```

8.2.5.2.2.3 vhosts.properties:

```
stg.jprobe.fsa.ed.gov=default_host
```

8.2.5.2.3 WebSphere Application Server Configuration

The WebSphere Command Line will identify the JProbe configuration file to use and ensure that the correct JVM is used. Two Environment Variables will be added to the Application Server.

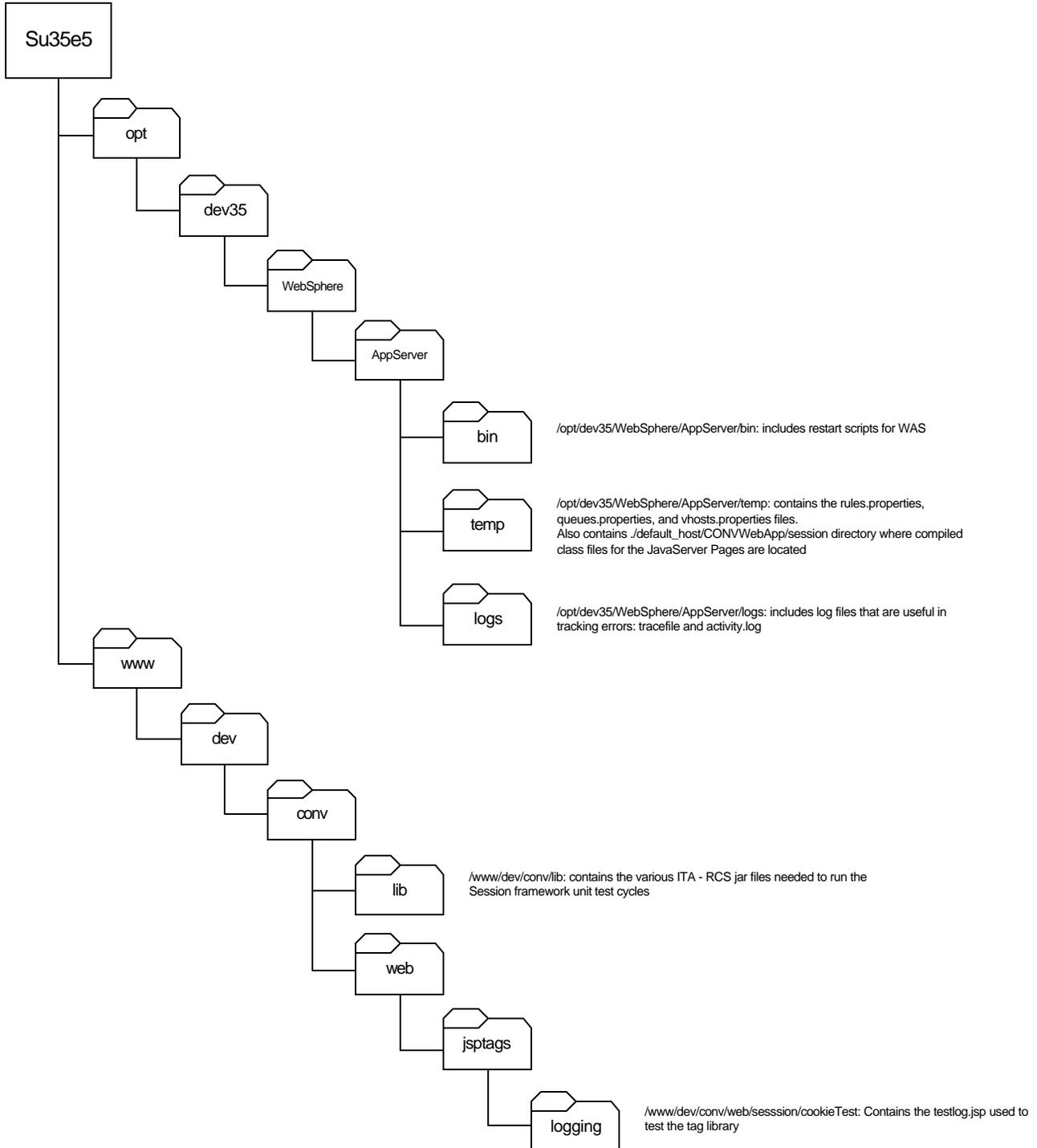
8.2.5.2.3.1 Command line arguments:

```
-jp_input=/opt/util/JProbe/jpl_files/08022002_test_jsptags.jpl -Xnoclassgc -
Djava.compiler=NONE -ms128m -mx128m
```

8.2.5.2.3.2 Environment:

```
EXECUTE=YES
EXECUTABLE=/opt/util/JProbe/profiler/jprun
```

8.2.5.2.4 Directory Structure





8.2.6 Testing Scenario

The test JSP created for the unit test was also used to execute the performance analysis. 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.

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.

8.2.7 Results and Analysis

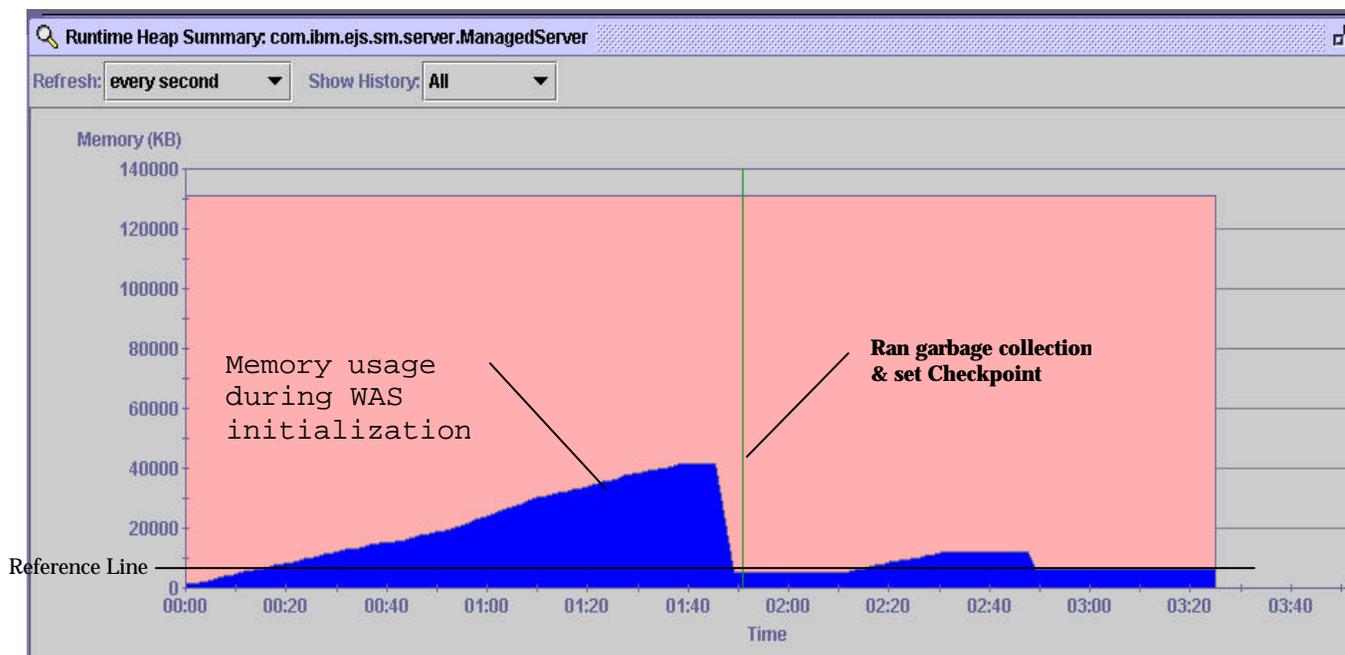
The JProbe Profiler with Memory Debugger application is used to trace both the memory usage and performance measurement of the Logging Tag Library API. Two snapshots are taken: a heap snapshot and a performance Snapshot. Each snapshot provides different information regarding our test.

8.2.7.1 Heap Snapshot (Memory Usage)

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.

8.2.7.1.1 Heap Graph Analysis

The screenshot below is obtained from executing the test JSP.



In the graph above, it is possible to see that 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 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.



8.2.7.1.2 Instance Summary

The table below is a section of the Instance Summary result associated with conducting test cycle 3. The Count column displays how many instances of the class currently exist in the heap and the Memory column shows how much memory those instances consume.

In the heap graph in the previous section, there is a green vertical line that shows where the checkpoint was set. The checkpoint tells JProbe to tag all subsequently created objects as “new.” The Count Change and Memory Change columns show data regarding new instances (created after the checkpoint) that are currently in the heap.

Package	Class	Count	Count Change	Memory	Memory Change
gov.ed.fsa.ita.jsptags	SyslogTag	11 (0.0%)	-	0.484 (0.0%)	-
gov.ed.fsa.ita.jsptags	CanNotLogTag	6 (0.0%)	-	0.216 (0.0%)	-
gov.ed.fsa.ita.jsptags	CanLogTag	6 (0.0%)	-	0.168 (0.0%)	-

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.



8.2.7.2 Performance Snapshot (Code Efficiency)

There are nine efficiency metrics that can be collected using JProbe – five basic metrics and four compound metrics. The basic metrics include: number of calls, method time, cumulative time, method object count, and cumulative object count. The compound metrics are averages per number of calls, including: average method time, average cumulative time, average method object count, and average cumulative object count. Time is measured as elapsed time in milliseconds.

The following sections will describe each metric and display the top results for each measurement for the performance assessment of the JSP Custom Logging Tag Library. These metrics are basic indicators of process resource utilization. The detailed graphs associated with each method can be reviewed for unexpected activity or optimization opportunities.

All performance metric results were first filtered by gov.ed.* to obtain only the classes within the JSP Custom Logging Tag Library which is what the test is looking for. Then for each section, the results were sorted by the metric under investigation to obtain the top ten results for each metric.



8.2.7.2.1 Number of Calls

Measures the number of times the method was invoked and shows the methods with the most calls. Helps to determine and streamline excessive method calls.

Package	Name	Calls	Source
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	23	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	12	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	12	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	12	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.setLevel(String)	12	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	11	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doEndTag()	11	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	11	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setChannel(String)	11	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setClassname(String)	11	SyslogTag.java

From the results above, it is possible to see that the setCallLevel() method in the SyslogTag class was called almost twice as often as other method calls for this test. This lead to the conclusion that the test application code and the library's API code should be examined to determine if the method is overused.

There were 23 calls made to the setCallLevel() method, and in the test application the tags from this tag library was used 23 times. This lead to the conclusion that every call to the tags from this library resulted in a call to the setCallLevel() method. This analysis was based on a complete understanding of the tag library's design and code and the high number of calls to this method was expected, as every tag in this library was designed to call this method to set the current logging level that the application developer wants to use for that tag. The tag library design will not be changed as a fundamental part of the RCS Logging framework, which this tag library implements, is the ability to have different messages be set to different levels in the same application.

The count of the number of calls to the remaining methods is also accurate based on the number of times the tags were called in the test application.



8.2.7.2.2 Method Time

Measures the amount of time (in milliseconds) spent executing the method, but it excludes the time spent in its descendants (sub-methods).

Package	Name	Method Time	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14.06 (71.8%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	3.19 (16.3%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0.73 (3.7%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0.32 (1.6%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0.22 (1.1%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0.21 (1.1%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0.15 (0.8%)	CanNotLogTag.java
gov.ed.sfa.ita.logging	Syslog.log(Object, Object, Object, Object, int)	0.10 (0.5%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	0.09 (0.5%)	CanLogTag.java
gov.ed.sfa.ita.logging	Syslog.canLog(int)	0.08 (0.4%)	Syslog.java

The results above show that the longest running method was the call to initialize the Syslog object from the RCS Logging framework. While the length of time seems excessive compared against the other methods, this method will only be called once during the life of the test application.

The second highest method time is for the CanLogTag.condition() method which is called by both the CanLogTag and CanNotLogTag classes. This method evaluates a given tag to see if the condition is equal to the current logging level and was expected to require more time to execute along with the other initialization and Logging framework class, which had to write the output to a file.



8.2.7.2.3 Cumulative Time

Measures the total amount of time (in milliseconds) spent executing the method and the time spent in its descendants, but excludes the time spent in recursive calls to its descendants.

Package	Name	Cumulative Time	Source
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	18.09 (92.4%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	17.89 (91.4%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	17.75 (90.7%)	CanLogTag.java
gov.ed.fsa.ita.logging	Syslog.<clinit>()	14.06 (71.8%)	Syslog.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0.73 (3.7%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0.59 (3.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0.25 (1.3%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0.22 (1.1%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0.21 (1.1%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0.19 (1.0%)	CanNotLogTag.java

The doStartTag() method for the various tags were expected to be in this list as all methods called for each tag was called as a result of doStartTag() executing. Both CanLogTag.condition() and CanNotLogTag.condition() methods call the CanLogTag.condition(boolean) method, which explains why the cumulative time for the condition(boolean) method is more than the other two methods when listed separately. The results do not contain any surprises to what the design specified.

It is also important to keep in mind while reviewing this analysis that the syslog class' initialization method in the RCS Logging framework is only called once and in this instance, the time for that method has been added to the CanLogTag methods' times since that is the first tag the JSP accessed. If the JSP accessed the SyslogTag first then the cumulative time displayed would be different.



8.2.7.2.4 Method Object Count

Measures the number of objects created during the method's execution, excluding those created by its descendants.

Package	Name	Method Objects	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14 (41.2%)	Syslog.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	10 (29.4%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	2 (5.9%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	2 (5.9%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	2 (5.9%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	2 (5.9%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	2 (5.9%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	0 (0.0%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	0 (0.0%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0 (0.0%)	CanNotLogTag.java

The Syslog class from the RCS logging framework creates the largest number of objects in its initialization methods. This method objects count refers to the number of objects created each time the method is called and is not reporting the total number of objects created by the method during the execution of the entire application (i.e. SyslogTag.setCallLevel() creates 10 objects each time it is called and it was called 23 times. The count reports 10 and not 230.)

Refer to the Performance Analysis Report for that framework for detail information regarding the Syslog class. The low number of objects created by the other methods should not lead to any performance impacts.



8.2.7.2.5 Cumulative Object Count

Measures the total number of objects created during the method's execution, including those created by its descendants.

Package	Name	Cumulative Objects	Source
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	26 (76.5%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	26 (76.5%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	26 (76.5%)	CanLogTag.java
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14 (41.2%)	Syslog.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	10 (29.4%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	2 (5.9%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	2 (5.9%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	2 (5.9%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	2 (5.9%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.setLevel(String)	0 (0.0%)	CanLogTag.java

The number of cumulative objects listed in the chart above show several methods with similar cumulative objects count. This is due to the sequence of method calls, with one method calling on the next and all of their created objects being added into the cumulative objects count. These results can be used to determine if the methods create an excessively high number of objects. Similar to the Method Object Count, the Cumulative Object Count represents the count for each call of the method and not a running count of objects created for all calls to the method.

When the RCS logging framework is initialized, 14 objects are created and those are objects are included in the count of 26. Taking that out leaves us with 12 objects that can be directly attributed to the CanLogTag class, out of which 10 of these objects can be attributed to the SyslogTag.setCallLevel() method. The number of objects created by this framework should not be considered excessive.



8.2.7.2.6 Average Method Time

Measures Method Time (in milliseconds) divided by the Number of Calls. Helps to identify individual methods that, on average, take a long time to execute.

Package	Name	Avg. Method Time	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14.06 (71.8%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	0.27 (1.4%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0.03 (0.2%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0.03 (0.1%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0.03 (0.1%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0.02 (0.1%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0.02 (0.1%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0.01 (0.0%)	CanNotLogTag.java
gov.ed.sfa.ita.logging	Syslog.log(Object, Object, Object, Object, int)	0.01 (0.0%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	0.01 (0.0%)	CanLogTag.java

The RCS Logging framework's initialization method takes the longest to execute on average. Just looking at these numbers alone does not provide any useful information since this number makes the situation appear worse than it actually is. The execution time on average does not automatically translate to a bad event here when taking into consideration that the method is only executed once during the life of the application.



8.2.7.2.7 Average Cumulative Time

Measures Cumulative Time (in milliseconds) divided by Number of Calls. Helps to identify methods that, together with their descendants, take a long time (on average) to execute.

Package	Name	Average Cumulative Time	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14.06 (71.8%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	2.96 (15.1%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	1.51 (7.7%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	1.49 (7.6%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0.05 (0.3%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0.04 (0.2%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0.03 (0.2%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0.03 (0.2%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0.02 (0.1%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0.02 (0.1%)	CanLogTag.java

The results above do not present any surprises and are consistent with the expected results based on evaluation of the previous performance metrics.



8.2.7.2.8 Average Method Object

Measures Method Object Count divided by Number of Calls. Highlights methods with the highest method object count per number of calls.

Package	Name	Avg. Method Object	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14 (41.2%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	0 (0.0%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	0 (0.0%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	0 (0.0%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0 (0.0%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0 (0.0%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0 (0.0%)	CanLogTag.java

The results of the average method object column displays mostly zeros, as the number is the results of the method object count divided by number of calls rounded down. Only the RCS Logging framework's initialization method resulted in a number greater than zero as the method was only called once. CanLogTag.condition() created 10 objects but was called 23 times which leads to an actual average method object count of 0.43478 which was rounded down to zero. These results highlight the fact that no classes from the JSP Custom Logging Tag library contain any methods that create many objects.



8.2.7.2.9 Average Cumulative Object Count

Measures Cumulative Object Count divided by Number of Calls. Highlights methods with the highest cumulative object count per number of calls.

Package	Name	Average Cumulative Object	Source
gov.ed.sfa.ita.logging	Syslog.<clinit>()	14 (41.2%)	Syslog.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition()	4 (11.8%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.doStartTag()	2 (5.9%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.condition(boolean)	2 (5.9%)	CanLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.doStartTag()	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.condition()	0 (0.0%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	CanNotLogTag.<init>()	0 (0.0%)	CanNotLogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.setCallLevel(String)	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	SyslogTag.<init>()	0 (0.0%)	SyslogTag.java
gov.ed.fsa.ita.jsptags	CanLogTag.<init>()	0 (0.0%)	CanLogTag.java

The average cumulative object count demonstrates that on average, these methods create the most number of objects. It should be noted that the cumulative counts includes objects created by other methods in this table so the numbers from this table should not be added. These results do not indicate that the tag library will create too many cumulative objects on average.



8.2.7.3 General Performance Metrics

The RCS JSP Custom Tag Library framework was tested on a Solaris 2.6 platform running JDK1.2.2 Reference Implementation. The test harness tested the major operations of the JSP Custom Logging Tag Library independently and the system as a whole.

No memory leaks were found in the Logging Tag Library using the test JSP as a test harness. No loitering objects were found in the heap at the end of the each test cycle.



8.2.8 Appendix A

8.2.8.1 JProbe Configuration File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jpl SYSTEM "jpl.dtd" >

<jpl version="1.5">
  <program type="application">
    <application
      args=""
      working_dir=""
      source_dir=""
      classname=""
    ></application>
    <applet
      working_dir=""
      source_dir=""
      htmlfile=""
      main_package=""
    ></applet>
    <serverside
      suggested_filters=""
      id="Other server"
      server_dir="/opt/stg35/WebSphere/AppServer"
      prepend_to_vm_args=""
      source_dir=""
      classname="com.ibm.ejs.sm.util.process.Nanny"
      main_package="gov.ed.fsa.ita.jsptags"
      exclude_server_classes="true"
      args=""
      working_dir="/opt/stg35/WebSphere/AppServer/servlets"
      prepend_to_classpath=""
    ></serverside>
  </program>
  <vm
    snapshot_dir="/opt/util/JProbe/snapshots"
    location="/opt/util/jdk1.2.2/bin/java"
    args=""
    type="java2"
    use_jit="true"/>
  <viewer
    socket="170.248.222.74:4444"
    type="remote"/>
  <analysis type="profile">
    <performance
      record_from_start="true"
      timing="elapsed"
      track_natives="true"
      final_snapshot="true"
    >
  </analysis>
</jpl>
```



```
granularity="method">
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask="*"
  time="ignore"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask=" gov.ed.fsa.ita.jsptags.*"
  time="track"
  granularity="method"/>
<performance.filter
  visibility="visible"
  methodmask="*"
  enabled="true"
  classmask=" gov.ed.sfa.ita.logging.*"
  time="track"
  granularity="method"/>
</performance>
<heap
  record_from_start="true"
  no_stack_trace_limit="false"
  final_snapshot="true"
  max_stack_trace="4"
  track_dead_objects="true"/>
<threadalyzer
  record_from_start="true"
  write_to_console="false">
<deadlock_detection
  enabled="true"
  deadlock_and_exit="true"
  report_stalls="false"
  track_system_threads="false"
  block_can_stall="false"
  deadlock_threshold="2"/>
<deadlock_prediction
  enable_hold_and_wait="false"
  enable_lock_order="false"
  lock_order_maintains_covers="true"/>
<data_race
  ignore_volatile="false"
  enable_happens_before="false"
  no_stack_trace_limit="false"
  enable_lock_covers="false"
  max_stack_trace="1"
  instrument_elements="false"/>
<visualizer
  enabled="true"
  visualization_level="1"/>
<threadalyzer.filter
  visibility="invisible"
  enabled="true"
  classmask="*" />
</threadalyzer.filter
```



```
visibility="visible"  
enabled="true"  
classmask=".*"/>  
</threadalyzer>  
<coverage  
  record_from_start="true"  
  final_snapshot="true"  
  granularity="line">  
  <coverage.filter  
    visibility="invisible"  
    methodmask=""  
    enabled="true"  
    classmask="*/>  
  <coverage.filter  
    visibility="visible"  
    methodmask=""  
    enabled="true"  
    classmask=".*"/>  
  </coverage>  
</analysis>  
</jpl>
```



8.2.9 Resources

- The Jakarta Taglibs Project
 - <http://jakarta.apache.org/taglibs/>
- Core Servlets and JavaServer Pages – Chapter 14: Creating Custom Tag Libraries
 - <http://developer.java.sun.com/developer/Books/javaserverpages/cservletsjsp/chapter14.pdf>
- The Struts Framework Project
 - <http://jakarta.apache.org/struts>
- Struts Framework API (Version 1.0)⁴
 - <http://jakarta.apache.org/struts/api-1.0/index.html>
- XTags is built on DOM4J
 - <http://DOM4J.org>

⁴ Version 1.0.1 is a patch release for version 1.0.