



Eusko Jaurlaritzaren Informatika Elkarte
Sociedad Informática del Gobierno Vasco

Quest JProbe Suite 7.0:

Manual rápido de usuario

Fecha: 21/07/2006

Referencia:

EJIE S.A.
Mediterráneo, 3
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Control de documentación

Título de documento: Manual rápido de usuario

Histórico de versiones

Código:

Versión: 1.0

Fecha:

Resumen de cambios: Versión inicial.

Versión: 1.1

Fecha: 28/11/2007

Resumen de cambios: Cambios en el apartado 3 sobre petición de realización de pruebas.

Versión: 1.2

Fecha: 17/12/2007

Resumen de cambios: Cambios en el apartado de solicitud de ejecución de Jprobe. Se añade ejemplo de fichero jpl a adjuntar en la petición. Comprobación de la ocupación de las licencias de cliente y servidor.

Cambios producidos desde la última versión

Tercera versión.

Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma:

Fecha: 21/07/2006

Distribución:

Referencias de archivo

Autor: Consultoría de áreas de conocimiento

Nombre archivo: JProbe. Manual rápido de usuario vn.n.doc

Localización: Sharepoint - Consultoría de áreas de conocimiento.

Contenido

Capítulo/sección	Página
1 Introducción	5
2 Conceptos básicos	5
3 Petición de realización de pruebas	¡Error! Marcador no definido.
3.1 Procedimiento	¡Error! Marcador no definido.
3.2 Configuración fichero JPL	¡Error! Marcador no definido.
4 Arranque del programa y conexión al servidor	5
4.1 Inicio de JProbe Profiler y Jprobe Memory Debugger	5
4.2 Conexión a un servidor que ya está previamente configurado con los archivos JPL introducidos en el arranque del servidor de aplicaciones.	¡Error! Marcador no definido.
5 JProbe Profiler	7
5.1 Ejecución de los casos de uso.	7
5.2 Análisis de los casos de uso.	8
6 JProbe Memory Debugger	11
6.1 Ejecución de los casos de uso.	12
6.2 Análisis de los casos de uso.	13
6.3 Resto de operativas asociadas a un caso de uso.	14

1 Introducción

El presente documento describe cuales son las tareas básicas que se pueden ejecutar en la explotación de la herramienta de testeo de software Quest JProbe Suite 7.0.

El contenido del documento integra, tanto los aspectos de uso en el entorno de EJIE como las características elementales de funcionamiento de la aplicación.

También se añade el procedimiento de solicitud de arranque de la instancia de Weblogic 8 con JProbe y un ejemplo de fichero jpl que hay que adjuntar en la petición.

2 Conceptos básicos

Quest JProbe Suite 7.0 es una herramienta para monitorizar el rendimiento de las aplicaciones, entre sus características destaca:

- Busca y encuentra usos excesivos de memoria y cuellos de botella, rápida y fácilmente.
- Genera gráficos de rendimiento, que muestran los cambios en el rendimiento o uso de la memoria en el tiempo.
- Crea informe personalizados para poder compartir información con otras personas.

La herramienta proporciona la visualización y testeo sobre de las típicas variables relacionadas con la optimización de código (consumo de memoria, de CPU, detección de memory leaks, garbage collector, tiempos de respuesta, etc.) así como la identificación del origen de los posibles problemas asociados a ellos.

La herramienta proporcionará informes de actividad y resultado de la ejecución.

Para obtener información adicional sobre el producto acceder a su página web:

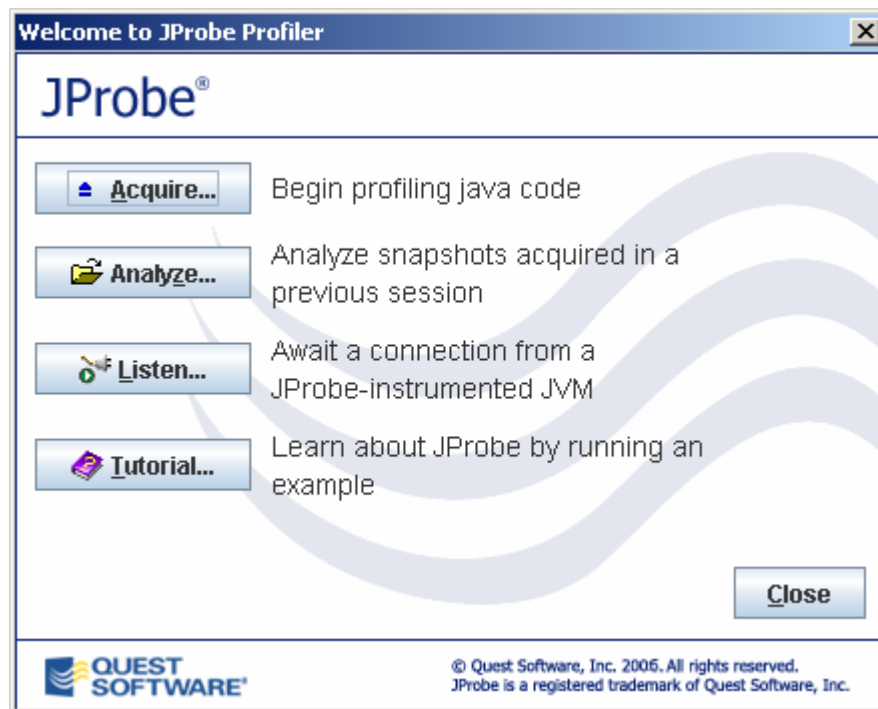
<http://www.quest.com/jprobe>

3 Arranque del programa y conexión al servidor

Vamos a ver el modo de conexión común de las dos aplicaciones JProbe al servidor Weblogic, una vez que el área de Soporte ha recibido nuestra solicitud, y que ha llegado la hora en que se solicitó hacer las pruebas.

3.1 Inicio de JProbe Profiler y Jprobe Memory Debugger

En el arranque de ambos programas, disponemos de cuatro formas de trabajar con la herramienta.



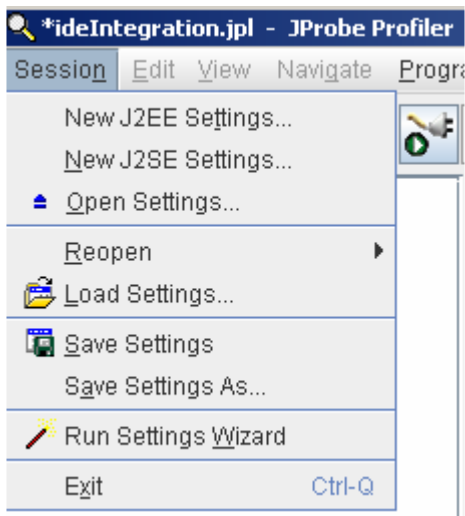
La primera opción, **Acquire**, permite la definición del tipo de aplicación que se desea testear y probar bajo la herramienta. Esta opción permite que mediante configuraciones previas de arranque de servidores web y plataformas JVM, el lanzar casos de uso sobre el servidor que JProbe Profiler ha inicializado

La segunda opción, **Analyze**, permite el análisis de los diferentes snapshot adquiridos en una sesión previa de análisis de JProbe Profiler.

La tercera opción, **Listen**, permite conectarse a un servidor que previamente ha sido arrancado, con las modificaciones previamente almacenadas mediante la ejecución de un archivo JPL, el cual se incorporará en el proceso de arranque del servidor.

La cuarta opción, **Tutorial**, permite el acceso al tutorial el cual enseña mediante ejemplos prácticos como trabajar con la herramienta.

Lo más habitual es el cancelar esta pantalla y acceder a las opciones existentes en el menú principal del aplicativo:



Se pueden crear nuevas sesiones de trabajo, pulsando sobre **New J2EE Settings** o **New J2SE Settings**.

Se puede acceder a configuraciones ya almacenadas en nuestras unidades locales mediante la opción de **Open Settings**.

Se puede volver a abrir un proyecto abierto recientemente mediante la opción de **Reopen**.

Se puede recargar en el sistema un archivo de configuraciones previamente creado mediante la opción de **Load Settings**.

Si existe un archivo de configuraciones cargado en el sistema y éste ha sido modificado, es posible grabarlo mediante la opción de **Save Settings**.

Si existe un archivo de configuraciones cargado en el sistema, es posible grabarlo con otro nombre mediante la opción de **Save Settings As**.

4 JProbe Profiler


El objetivo de JProbe Profiler es descubrir dónde se encuentran los cuellos de botella de nuestros programas Java. Para ello, el programa se encargará de analizar los tiempos de ejecución de los métodos, de identificar las clases de los mismos, contabilizar los objetos que se van creando, e identificar métodos ineficientes y toda la cadena de llamadas entre ellos a lo largo de nuestra navegación por una aplicación J2EE.

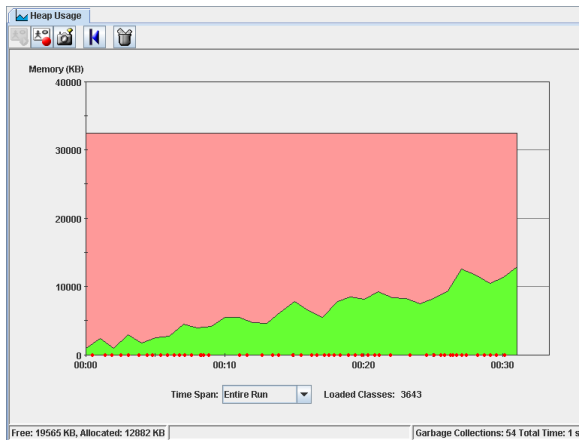
De esta forma, los programadores de aplicaciones J2EE podrán chequear el rendimiento global de una aplicación. Mediante este programa, se permite el análisis e identificación de cuellos de botella en la ejecución, identificando los métodos y clases implicados para su corrección por parte del equipo de desarrollo.

4.1 Ejecución de los casos de uso.


Una vez conectados al servidor de aplicaciones, procederemos a ejecutar el control del caso de uso,



 Run

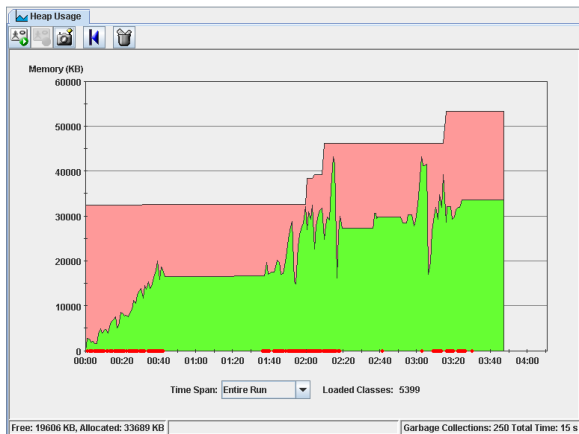
pulsando la opción de , que inicialmente nos visualizará los consumos de memoria en el momento de la conexión al servidor de aplicaciones.



Una vez arrancado el servidor, procederemos a ejecutar nuestro aplicativo y a ejecutar los casos de uso que se pretenden analizar. Para ello, indicaremos a la herramienta que vamos a comenzar nuestro caso de uso.

Esto se efectúa mediante el acceso al botón  ubicado en la parte superior izquierda de la pantalla de


resultados. Una vez finalizamos los mismos procederemos a finalizar el snapshot pulsando la opción de 

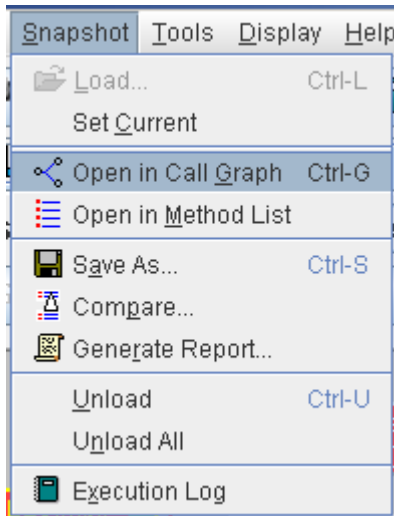


Como se puede observar en la anterior ilustración, se ven reflejados los diferentes cambios en la memoria mediante el acceso del HEAP USAGE.

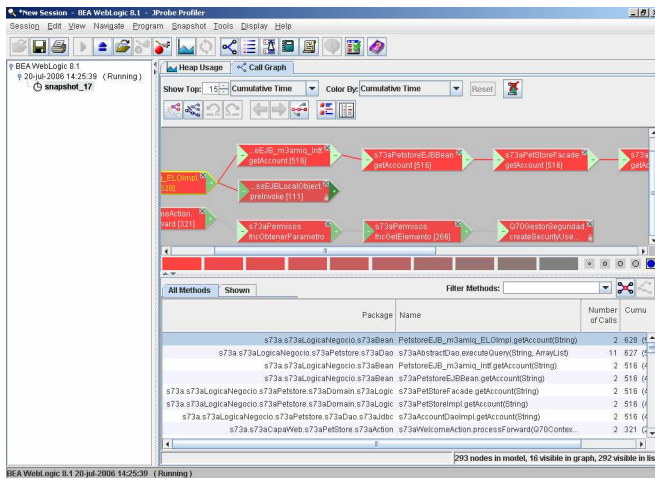
4.2 Análisis de los casos de uso.

Para ver la lista de llamadas efectuadas por el caso de uso ejecutado, una vez finalizado el Snapshot,

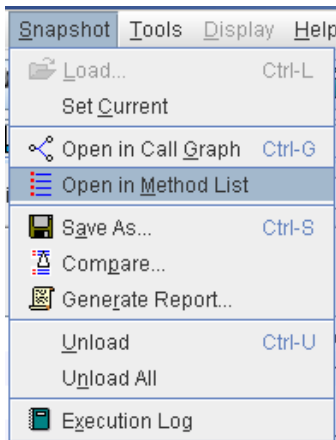
accederemos al icono , o bien al menú Snapshot, **Open in Call Graph**.



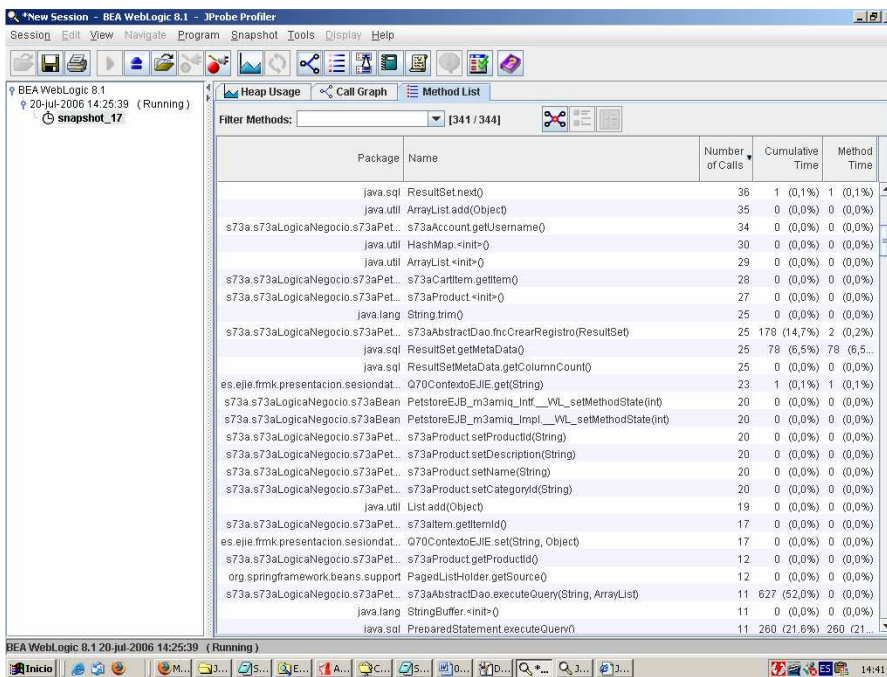
Una vez hemos accedido a esta opción, se puede ver la diagramación de llamadas a las diferentes clases existentes en los casos de uso y representando mediante gamas de colores los diferentes consumos de tiempo por cada proceso, pudiendo analizar de un modo visual el caso de uso y viendo dónde se consumen los recursos.



Para ver la lista de Métodos efectuados por el caso de uso ejecutado, una vez finalizado el Snapshot, accederemos al icono , o bien al menú Snapshot, **Open in Method List**.



Una vez hemos accedido a esta opción, se puede ver el listado de métodos llamados en los diferentes casos de uso, pudiendo analizar de un modo visual el caso de uso y viendo dónde se consumen los recursos. El listado se puede ordenar por los diferentes elementos existentes en su cabecera para poder efectuar un análisis rápido de lo acontecido en las pruebas.



Se puede acceder al **Method Detail** de cada evento, haciendo un doble clic sobre el método que se desee. Se mostrará entonces tanto los hijos o Childrens de método, como los padres o Parents.

Desde la siguiente ilustración se puede observar el listado de childrens asociados al método seleccionado:

Current Method: s73aAbstractDao.executeQuery(String, ArrayList)

Children Parents

Total Contribution to Current Method
Calls: 350 Time: 626 ms

Name	Calls From Current	Contri
PreparedStatement.executeQuery()	11	260
s73aAbstractDao.fncCrearRegistro(Re...	25	178
Q70Conecтор JDBC.getConnection(Stri...	11	171
s73aAbstractDao.fncPonerParametro(...	11	6
s73aLog.trace(String, Q70TraceLevel, ...	22	5
Connection.prepareStatement(String)	11	4
ResultSet.next()	36	1
PreparedStatement.close()	11	1
ResultSet.close()	11	0
s73aLog.isTraceActive()	22	0
Object.toString()	11	0
Class.getName()	22	0
ArrayList.add(Object)	25	0

Desde la siguiente ilustración se puede observar el listado de parents asociados al método seleccionado:

Current Method: s73aAbstractDao.executeQuery(String, ArrayList)

Children Parents

Total Contribution to Current Method
Number of Calls: 11 Cumulative Time: 627 ms
Method Time: 0 ms

Name	Calls to Current	Cumulative Time Contributed	Method Tim Contribute
s73aAccountDaoImpl.getAccount(String)	2	515 (82,1%)	0 (33,9%)
s73aCategoryDaoImpl.getCategory(String)	1	21 (3,4%)	0 (7,7%)
s73aProductDaoImpl.getProductListByCategory(String)	3	50 (8,0%)	0 (27,3%)
s73aProductDaoImpl.getProduct(String)	1	5 (0,8%)	0 (8,2%)
s73aItemDaoImpl.isItemInStock(String)	1	7 (1,1%)	0 (5,3%)
s73aItemDaoImpl.getItemListByProduct(String)	1	17 (2,7%)	0 (7,3%)
s73aItemDaoImpl.getItem(String)	2	12 (1,9%)	0 (10,3%)

5 JProbe Memory Debugger


El objetivo de JProbe Memory Debugger es analizar la cantidad de memoria que se utiliza por parte de nuestra aplicación en la pila de la JVM. Se expondrá una lista de los objetos en memoria, identificando los objetos, para poder observar dónde se encuentran los cuellos de botella de nuestros programas Java. Para ello, el programa se encargará de analizar los tiempos de ejecución de los métodos, de identificar las clases de los mismos, contabilizar los objetos que se van creando, e identificar métodos ineficientes y toda la cadena de llamadas entre ellos a lo largo de nuestra navegación por una aplicación J2EE.

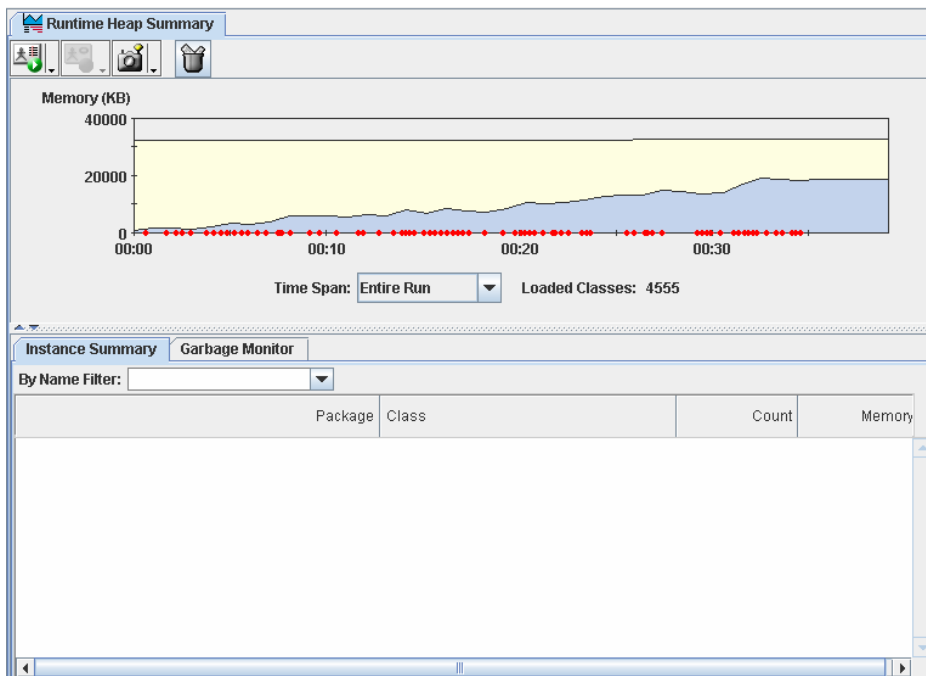
De esta forma, los programadores de aplicaciones J2EE podrán chequear el rendimiento global de una aplicación. Mediante este programa, se permite el análisis e identificación de cuellos de botella en la ejecución, identificando los métodos y clases implicados para su corrección por parte del equipo de desarrollo.

5.1 Ejecución de los casos de uso.


Una vez conectados al servidor, procederemos a ejecutar el control del caso de uso, pulsando la opción


Run

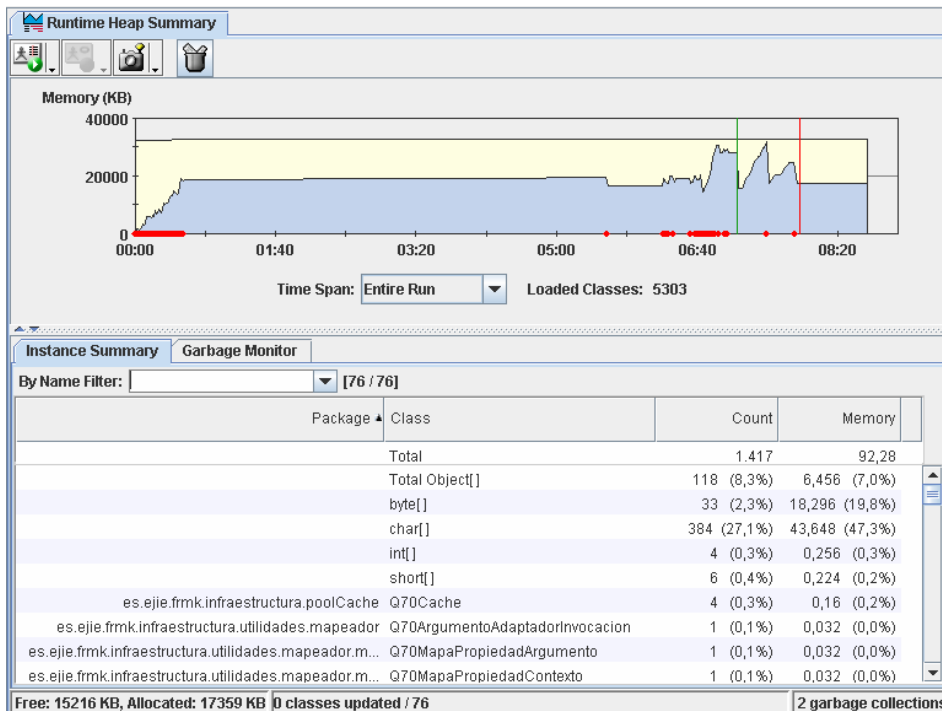
de , que inicialmente nos visualizará los consumos de memoria en el momento de la conexión al servidor de aplicaciones.



Una vez arrancado el servidor, procederemos a ejecutar nuestro aplicativo y a ejecutar los casos de uso que se pretenden analizar. Para ello, indicaremos a la herramienta que vamos a comenzar nuestro caso de uso.

Esto se efectúa mediante el acceso al botón  ubicado en la parte superior izquierda de la pantalla de resultados.

Una vez finalizamos los mismos procederemos a terminar el snapshot pulsando la opción de 

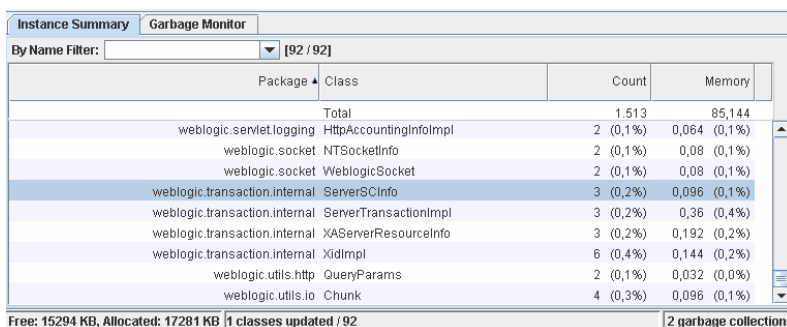


Como se puede observar en la anterior ilustración, se ven reflejados los diferentes cambios en la memoria mediante el acceso del RUNTIME HEAP SUMMARY.

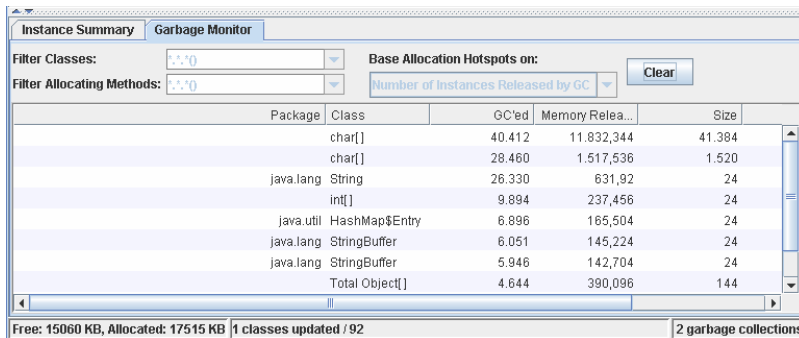
5.2 Análisis de los casos de uso.

Una vez hemos definido nuestros diferentes casos de uso, se procederá a visualizar cada uno de los diferentes INSTANCE SUMMARY y cada uno de los diferentes GARBAGE MONITOR que hayan dado como resultado el caso de uso analizado.

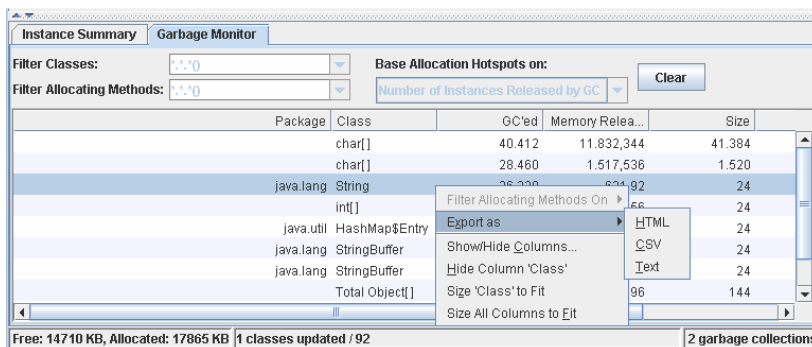
Por defecto, la pantalla de análisis siempre muestra el INSTANCE SUMMARY:



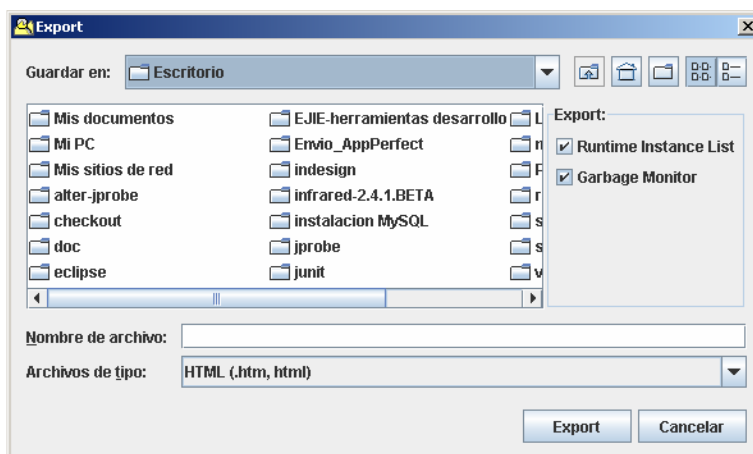
Para acceder a visualizar los GARBAGE MONITOR, accederemos a la pestaña que está ubicada a la derecha de INSTANCE SUMMARY:



Los resultados obtenidos en ambos apartados, son exportables a formatos de archivo estándar, como CSV, HTML y TXT.

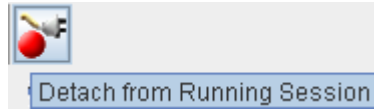


Para ello, haremos clic derecho sobre la lista de resultados, accederemos al menú **Export**, a continuación al submenú del tipo de archivo de salida y seleccionaremos la ubicación final, y a su vez qué es lo que se desea exportar, si la lista completa del runtime y/o el garbage:

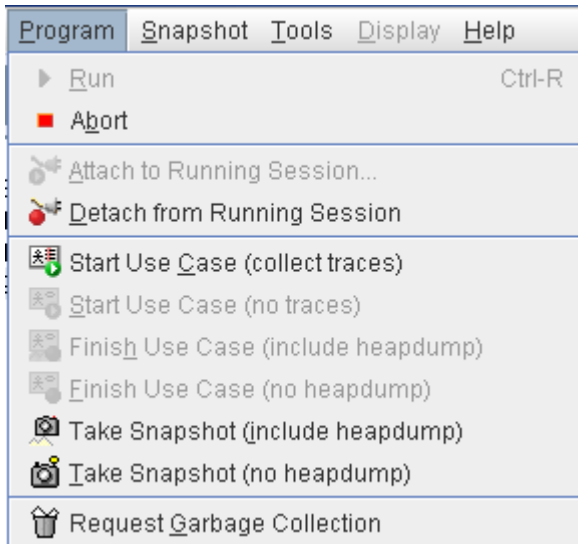


5.3 Resto de operativas asociadas a un caso de uso.

Para poder seguir trabajando con la herramienta, deberemos desconectarnos del servidor. Esta opción es









posible accediendo al botón **Detach from Running Session**, o bien a través del menú **Program**, opción **Detach From Running Session**.




Una vez se ha producido la desconexión al servidor, podremos observar que se han habilitado nuevas

opciones para poder efectuar el análisis del caso de uso más detallado:



-  Permite el acceso a las instancias de ejecución.
-  Permite el acceso a las clases de ejecución.
-  Permite el acceso al garbage monitor.
-  Permite la comparación entre dos snapshots.
-  Muestra el LOG de ejecución.
-  Permite generar un informe de salida.

El acceso al apartado de Instancias , nos muestra un listado detallado con la ejecución de las diferentes instancias.

Package	Class	Count	Memory
Total		1,416	92,256
java.lang	String	393 (27,8%)	9,432 (10,2%)
	char []	384 (27,1%)	43,648 (47,3%)
	object []	118 (8,3%)	6,456 (7,0%)
java.util	HashMap\$Entry	94 (6,6%)	2,256 (2,4%)
java.util	HashMap	56 (4,0%)	2,24 (2,4%)
java.util	HashMap\$Entry	44 (3,1%)	1,056 (1,1%)
	byte []	33 (2,3%)	18,296 (19,8%)
java.util	HashMap\$EntrySet	28 (2,0%)	0,448 (0,5%)
java.util	ArrayList	20 (1,4%)	0,48 (0,5%)
java.util	HashSet	19 (1,3%)	0,304 (0,3%)
java.lang	Object	18 (1,3%)	0,144 (0,2%)
java.lang.ref	Finalizer	17 (1,2%)	0,544 (0,6%)
weblogic.servlet.internal	AttributeWrapper	12 (0,8%)	0,288 (0,3%)
weblogic.jdbc.wrapper	PoolConnection_oracle_jdbc_driver_OracleConnection	12 (0,8%)	0,672 (0,7%)
s73a.s73aLogicaNegocio.s73aPetstore.s7...	s73aProduct	11 (0,8%)	0,264 (0,3%)
java.lang	Long	11 (0,8%)	0,176 (0,2%)
java.io	ExpiringCache\$Entry	8 (0,6%)	0,192 (0,2%)
java.lang.reflect	Constructor	8 (0,6%)	0,384 (0,4%)
java.util	Vector	8 (0,6%)	0,192 (0,2%)
	short []	6 (0,4%)	0,224 (0,2%)

Si queremos ver el detalle completo de una de ellas, bastará con seleccionarla y hacer un doble clic con el botón izquierdo del ratón sobre la instancia seleccionada:

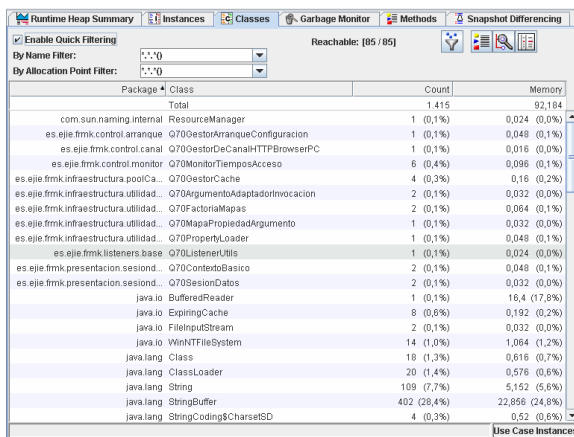
Package	Instance ID	Size	Creation	Referrers	References
s73a.s73aLogicaN...	s73aProduct 141478	24	07:32:058	0	0
s73a.s73aLogicaN...	s73aProduct 105393	24	07:25:943	0	0
s73a.s73aLogicaN...	s73aProduct 76397	24	07:21:142	0	0
s73a.s73aLogicaN...	s73aProduct 68786	24	07:20:370	0	0
s73a.s73aLogicaN...	s73aProduct 68780	24	07:20:369	0	0

Method	Source
s73aItemDaolmpl.getItem(java.lang.String)	s73aItemDaolmpl.java: 133
s73aPetStoreImpl.getItem(java.lang.String)	s73aPetStoreImpl.java: 266
s73aPetStoreEJBBean.getItem(java.lang.String)	s73aPetStoreEJBBean.java: 158
PetstoreEJB_m3amiq_ELOImpl.getItem(java.lang.String)	PetstoreEJB_m3amiq_ELOImpl.java: 479
NativeMethodAccessorImpl.invoke0(java.lang.reflect.Method, java.lang.Object, java.la...	NativeMethodAccessorImpl.java
NativeMethodAccessorImpl.invoke(java.lang.Object, java.lang.Object[])	NativeMethodAccessorImpl.java: 39
DelegatingMethodAccessorImpl.invoke(java.lang.Object, java.lang.Object[])	DelegatingMethodAccessorImpl.java: 25
Method.invoke(java.lang.Object, java.lang.Object[])	Method.java: 324
Q70InvocadorMetodos.invoke(java.lang.Object, java.lang.String, java.lang.Object[], ja...	Q70InvocadorMetodos.java: 76
Q70AdaptadorInvocacion.processFunctionInvocation(java.lang.String, java.lang.Strin...	Q70AdaptadorInvocacion.java: 287
Q70AdaptadorInvocacion.invokeFunction(java.lang.String, java.lang.String, java.lang...	Q70AdaptadorInvocacion.java: 184
Q70OperacionInterna.processAdaptadorInvocacion(java.lang.String, java.lang.String...	Q70OperacionInterna.java: 215
Q70OperacionInterna.processOI(es.ejje.frmk.presentation.sesiondatos.Q70Context...	Q70OperacionInterna.java: 97
Q70ActionOI.executeOI(java.lang.String, es.ejje.frmk.presentation.sesiondatos.Q70...	Q70ActionOI.java: 141

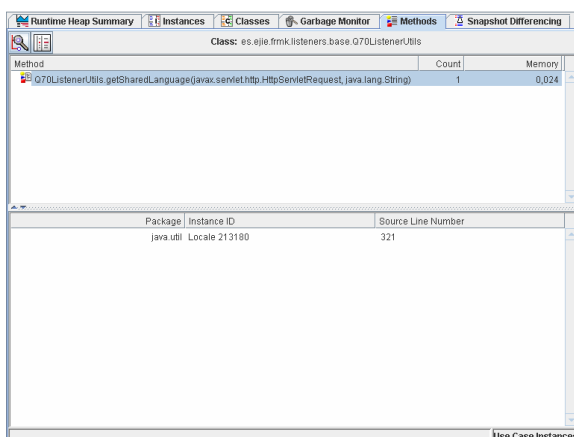
Si se dispone del código fuente, y éste se le indica bien inicialmente o a posteriori, el sistema nos lo mostraría. En caso de no habérselo indicado anteriormente, al hacer doble clic nos pediría la ruta de ubicación del código fuente para una clase determinada:



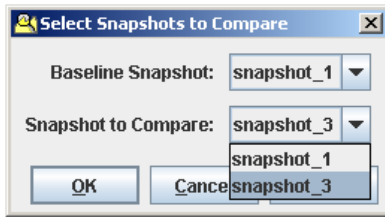
Si lo que se desea es visualizar las clases que han sido ejecutadas en el caso de uso seleccionado, accederemos sobre dicha opción :



Si queremos ver el detalle de los métodos de dicha clase, bastará con seleccionarla y hacer un doble clic con el botón izquierdo del ratón sobre la clase seleccionada:



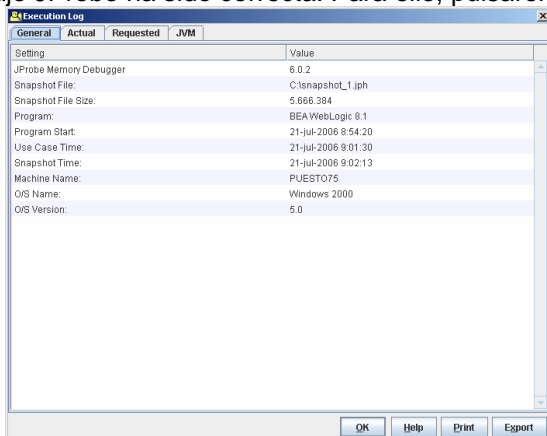
Para ver la diferencia de ejecución por ejemplo de un mismo caso de uso variando tan solo la línea de tiempo, usaremos la opción . Para la sesión activa, dispondremos de los diferentes snapshot generados, los cuales nos permitirá efectuar la comparación:



Una vez hemos seleccionado los snapshot a comparar, nos muestra la pantalla de resultados de dicha comparación:

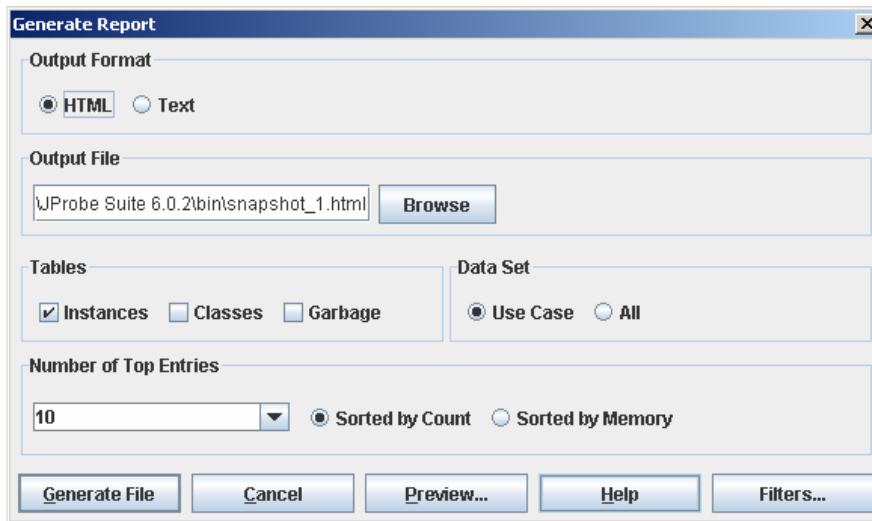
Package	Class	Count	Memory
Overall Difference			
		95	-7,168
weblogic.util.io	Chunk	2 (100,0%)	0,048 (100,0%)
weblogic.transaction.internal	XidImpl	2 (50,0%)	0,048 (50,0%)
weblogic.transaction.internal	XAServerResourceInfo	1 (50,0%)	0,064 (50,0%)
weblogic.transaction.internal	ServerTransactionImpl	1 (50,0%)	0,12 (50,0%)
weblogic.transaction.internal	ServerSCInfo	1 (50,0%)	0,032 (50,0%)
weblogic.socket	WeblogicSocket	2 (+)	0,08 (+)
weblogic.socket	NTSocketInfo	2 (+)	0,08 (+)
weblogic.servlet.internal	ServletOutputStreamImpl	2 (+)	0,112 (+)
weblogic.servlet.internal	PostInputStream	1 (+)	0,056 (+)
weblogic.servlet.internal	MuxableSocketHTTP	2 (+)	0,176 (+)
weblogic.servlet.internal	ChunkOutputWrapper	2 (+)	0,048 (+)
weblogic.servlet.internal	AttributeWrapper	-1 (-0,3%)	-0,024 (-0,3%)
weblogic.jms.backend	BackEnd\$3	1 (+)	0,016 (+)
weblogic.jms.backend	BETimerNode	1 (100,0%)	0,04 (100,0%)
weblogic.jdbc.wrapper	PoolConnection_oracle_jdbc_dr...	8 (66,7%)	0,448 (66,7%)
weblogic.jdbc.wrapper	JTSAResourceImpl	2 (66,7%)	0,032 (66,7%)
weblogic.jdbc.wrapper	JTSAConnection_oracle_jdbc_dri...	2 (66,7%)	0,16 (66,7%)
sun.reflect	NativeMethodAccessorImpl	1 (+)	0,024 (+)
sun.reflect	NativeConstructorAccessorImpl	2 (100,0%)	0,048 (100,0%)
sun.reflect	GeneratedMethodAccessoR?	1 (+)	0,008 (+)

Siempre es posible el consultar el LOG de ejecución, para poder verificar que la ejecución del caso de uso bajo JProbe ha sido correcta. Para ello, pulsaremos sobre la opción habilitada



Podemos ver el los general, el actual, el de las peticiones y el de la JVM. Este LOG es imprimible y exportable en formato HTML.

Por último, se puede generar un informe o reporte de la actividad ejecutada. Para ello, pulsaremos sobre la opción activada



La salida del informe puede ser en formato HTML o texto plano, se deberá seleccionar tanto el nombre del archivo como la ubicación de este. El tipo de tabla y el tipo de dato que se quiere mostrar en el informe también puede ser indicado y el número de entradas y como ordenarlas. Si el informe se previsualiza, queda con un aspecto similar al que se adjunta a continuación:

JProbe(R) MemoryDebugger Report of BEA WebLogic 8.1

```

JProbe Memory Debugger      7.0
Snapshot File:              C:\snapshot_1.jph
Snapshot File Size:         5.666.384
Program:                    BEA WebLogic 8.1
Program Start:              21-jul-2006 8:54:20
Use Case Time:              21-jul-2006 9:01:30
Snapshot Time:              21-jul-2006 9:02:13
Machine Name:               PUESTO75
O/S Name:                   Windows 2000
O/S Version:                 5.0
Number of Top Entries       10, Sorted by Memory
Data Set                    Use Case
Use Entire Trace            No
By Name Filters              *.*()
By Allocation Filters        *.*()
    
```

Instances View

Package	Class	Count	Memory
	char []	384	43,648

		(27,1%)	(47,3%)
	byte []	33 (2,3%)	18,296 (19,8%)
java.lang	String	393 (27,8%)	9,432 (10,2%)
	object []	118 (8,3%)	6,456 (7,0%)
java.util	Hashtable\$Entry	94 (6,6%)	2,256 (2,4%)
java.util	HashMap	56 (4,0%)	2,24 (2,4%)
java.util	HashMap\$Entry	44 (3,1%)	1,056 (1,1%)
weblogic.jdbc.wrapper	PoolConnection_oracle_jdbc_driver_OracleConnection	12 (0,8%)	0,672 (0,7%)
java.lang.ref	Finalizer	17 (1,2%)	0,544 (0,6%)
weblogic.servlet.internal	ServletRequestImpl	2 (0,1%)	0,496 (0,5%)

Classes View

Package	Class	Count	Memory
java.lang	StringBuffer	402 (28,4%)	22,856 (24,8%)
java.io	BufferedReader	1 (0,1%)	16,4 (17,8%)
weblogic.utils.io	Chunk	4 (0,3%)	8,24 (8,9%)
java.nio	HeapByteBuffer	1 (0,1%)	8,208 (8,9%)
java.lang	String	109 (7,7%)	5,152 (5,6%)
java.util	HashMap	135 (9,5%)	4,464 (4,8%)
org.apache.xml.utils	SuballocatedIntVector	110 (7,8%)	3,576 (3,9%)
java.util	Hashtable	100 (7,1%)	3,312 (3,6%)
weblogic.servlet.internal	RequestParser	21 (1,5%)	1,376 (1,5%)
oracle.jdbc.dbaccess	DBConversion	56 (4,0%)	1,344 (1,5%)

Garbage Monitor

Package	Class	GC'ed	Memory Released	Size	Alive	Allocated At
	char []	34143	9932344	290	8	java.lang.StringBuffer.expandCapacity(int)
	char []	23947	1277256	53	186	java.lang.StringBuffer.(int)

	char []	1627	1055080	648	0	sun.nio.cs.StreamEncoder.write(java.lang.String, int, int)
	int []	224	978432	4368	0	org.apache.xml.utils.SuballocatedIntVector.(int)
	object []	56	918400	16400	0	org.apache.xml.utils.ObjectVector.(int)
	int []	56	918400	16400	0	org.apache.xml.utils.IntVector.(int)
	object []	28	917952	32784	0	org.apache.xpath.VariableStack.()
	object []	28	917952	32784	0	org.apache.xpath.VariableStack.reset()
	byte []	68	558144	8208	0	java.nio.HeapByteBuffer.(int, int)
java.lang	String	22179	532296	24	194	java.lang.StringBuffer.toString()

6 Procedimiento de implantación

6.1 Configuración fichero JPL

Desde el Servicio de Asistencia Técnica se debe adjuntar el fichero *XXX-ejje.jpl*, donde XXX será el código de su proyecto a testear.

Nota: sólo se podrá arrancar Weblogic para su utilización con un programa determinado de Jprobe (análisis de rendimiento o memoria) , por lo tanto, para probar ambos, se necesitará de dos ficheros jpl y dos procesos de pruebas diferenciados.

Veamos un ejemplo de su contenido para su correcta ejecución:

```
#Mon Aug 07 11:59:41 CEST 2006
# Version Settings
#-----
jpl_version=2.0
jprobe_version= 7.0.0

# Program Settings
#-----
-jp_is_application=true
-jp_program_name=BEA WebLogic 8.1

# JVM Settings
#-----

# Analysis Settings
#-----
-jp_function=performance
-jp_track_objects=false
-jp_measurement=elapsed
-jp_collect_data=t29a.*.*():method

# Port console Settings
#-----
```

```

#(Reemplazar ##### por el número de puerto deseado).
#-jp_console_port=#####

# Messages
#-----
-jp_messages=system:default,threshold_assert:default

```

En la línea **-jp_function** se indicará qué programa se va a utilizar de JProbe:

performance: JProbe Profiler
heapdump: JProbe Memory Debugger

En la línea **-jp_collect_data=t29a.*():method** se podrá incluir un filtro con el código del proyecto en lugar de t29a, y de esa forma, se filtrarán sólo las clases de nuestra aplicación en la monitorización del programa. Si se desearan incluir más clases o método del programa, éstas se separarán por el símbolo “,”.

Por ejemplo, en el proyecto **t29a** se desea monitorizar solo las clases asociadas a **blueprints.address.ejb.AddressLocalHome** y **blueprints.address.ejb.AddressLocal**, la sintaxis del **-jp_collect_data** sería:

-jp_collect_data=t29a.blueprints.address.ejb.AddressLocalHome():method,blueprints.address.ejb.AddressLocal():method

En principio, es lo único que se debería modificar.

Si no se especifica el puerto en el apartado **-jp_console_port**, por defecto al arrancar el servidor de Aplicaciones con la opción de JProbe Console utilizará el puerto 51291. Si hubiese instancias previas arrancadas y el puerto no se identifica, se utilizaría el inmediatamente superior a este, siempre que no estuviese ocupado.

Las opciones recomendadas en la ejecución del **jplauncher** para EJIE son, por lo tanto, las siguientes:

- **-jp_function=[performance|heapdump]** : Elige el tipo de análisis a ejecutar bajo la herramienta JPROBE. Por defecto, pondremos siempre la opción de PERFORMANCE para arrancar JPROBE PROFILER y HEAPDUMP para JPROBE MEMORY DEBBUGER.
- **-jp_program_name="BEA WebLogic 8.1"** : Se le indica a JPROBE que la plataforma de testeo va a ser BEA WEBLOGIC 8.1
- **-jp_track_objects= [false|true]** : Contador de objetos para las asignaciones de los métodos. Por defecto estará a false.
- **-jp_measurement= [elapsed|cpu]** : cómo se medirá el tiempo. elapsed = incluye I/O y el programa se detiene brevemente; CPU = sigue el tiempo para el código. Inicialmente se realizará bajo el modo de ELAPSED.
- **-jp_collect_data=filter[,filter,...]** : Los diferentes tipos de filtros que JPROBE analizará. Si esta opción no se indica, por defecto se analizarán todos los objetos en ejecución, si no los indicados en el mismo. Se pueden añadir diferentes filtros, todos ellos separados por comas, y el tipo de análisis.
- **jp_messages=<category>:[none|default|<file>][,...]** : Control de los mensajes que se van a dar en el testeo.
- **-jp_export_jpl=<filename>** : Nombre del archivo JPL que se va a generar, incluyendo la carpeta donde se generará.
- **-jp_console_port=[interface:]port** : Puerto con el que se establecerá la conexión con JPROBE. Si no se incluye esta opción, se ejecutará con el puerto por defecto.

Nota: Cabe destacar que **Quest JProbe Suite 7.0 no funciona bajo BEA WEBLOGIC 5.1.**

6.2 Comprobación disponibilidad de licencias de cliente y servidor

Actualmente se dispone de una licencia para el motor de Jprobe y otra para la consola cliente de Jprobe. Ambas están instaladas en un servidor de licencias. Se puede consultar si están libres u ocupadas desde la consola Web del servidor de licencias, a través de: <http://hostname:8133/licensing/>.

Desde esta página se puede ver el apartado Available Licence(s) donde se muestran tanto las licencias disponibles como ocupadas, para la consola cliente y para el motor de Jprobe.

El primer dígito muestra el número de licencias libres, y el segundo las totales disponibles.