Departamento de Telemática

PROYECTO FIN DE CARRERA

MÓDULO EMPRESARIAL PARA JAVA PATH FINDER

Autor: Manuel Jesús Martín Gutiérrez

Tutor: Pablo Basanta Val

Leganés, Marzo de 2011

Título: MÓDULO EMPRESARIAL PARA JAVA PATH FINDER
Autor: Manuel Jesús Martín Gutiérrez
Director: Pablo Basanta Val

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 11 de Marzo de 2011 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

En primer lugar, quisiera agradecer a mi tutor Pablo sus consejos y orientación a lo largo de todo el proyecto, han sido clave para que éste llegase a su fin. Además quisiera agradecer toda la paciencia que ha tenido conmigo.

También quisiera agradecer a mi familia y a Ana por haberme apoyado y animado tanto durante todo este tiempo, no saben lo mucho que me han ayudado.

Finalmente, quisiera agradecer a los amigos que me han acompañado durante estos años de universidad porque no hubiera sido lo mismo sin ellos.

Resumen

Este proyecto consiste en hacer accesible vía web una herramienta de validación formal de aplicaciones Java, como es Java Path Finder, desarrollada por la NASA. Para este objetivo se analizan en profundidad la herramienta JPF, la tecnología Java EE para aplicaciones empresariales y el servidor de aplicaciones de código libre Glassfish de Sun. Este proyecto comprende además el diseño e implementación de la aplicación, utilizando las tecnologías analizadas, así como pruebas de rendimiento para conocer los límites de dicho sistema en un curso real.

Palabras clave: Java Path Finder, EJB, Java EE, JMS, System Integration Tool, Glassfish.

Abstract

This project deals with getting web access to a formal validation tool for Java applications called Java Path Finder, developed by NASA. For that aim, different technologies are analyzed in depth: the JPF tool, the Java EE technology for enterprise applications and the Glassfish application server developed by Sun which is open source. This project includes, in addition, the design and implementation of the entire application using the analyzed technologies, as well as performance tests to find its limits on real course.

Keywords: Java Path Finder, EJB, Java EE, JMS, System Integration Tool, Glassfish.

Índice de contenidos

Índice de figuras

Índice de tablas

CAPITULO 1. Introducción y objetivos

1.1 Introducción

La validación formal ([2],[3],[4],[5],[6]) nace de la necesidad de analizar el funcionamiento de sistemas y equipos muy complejos que utilizan aplicaciones concurrentes, las cuales, no basta con saber si respetan el lenguaje en el que han sido desarrolladas. Los equipos que utilizan dichas aplicaciones, en ocasiones, controlan muchos dispositivos demasiado importantes como para permitir un error, por ejemplo, los sistemas de un avión. Muchos de estos equipos utilizan aplicaciones concurrentes Java, por lo que el proyecto se centra en la validación de éste lenguaje.

Partiendo de la idea anterior, este proyecto se ha enfocado hacia un entorno docente, en el que los usuarios puedan validar formalmente sus prácticas o trabajos Java. La plataforma recibe prácticas y devuelve por correo el resultado de la validación.

Existen ya herramientas que permiten validar aplicaciones Java. Una de ellas es JPF ([1]), la cual, está en constante desarrollo y ofrece unas características muy útiles para el objetivo del proyecto. Sin embargo, esta herramienta consume mucha memoria y CPU del equipo en que está instalada, por lo que no permite trabajar paralelamente a la validación. Además, su instalación y configuración podrían resultar complicadas para usuarios sin demasiado conocimiento sobre la herramienta.

Por todo esto, se llega a la conclusión, de que hacer accesible la herramienta vía web, permitiría a los usuarios desentenderse de la instalación y configuración interna de la herramienta además de permitir trabajar de forma paralela, sin saturar el equipo.

Para este acceso web a la herramienta, se utilizará la tecnología Java EE ([7]) para aplicaciones empresariales, por ser portable entre plataformas y escalable. Además, se montará sobre un servidor Glassfish ([20]) que permite control y configuración de aplicaciones de esta tecnología, de forma sencilla e intuitiva.

1.2 Objetivos del Proyecto

El objetivo del proyecto es la creación de una aplicación empresarial Java EE, que utilizando la tecnología de la herramienta para la validación formal de aplicaciones Java llamada Java Path Finder, haga accesible dicha funcionalidad vía web.

De esto se deducen los siguientes objetivos parciales:

1- **Análisis de la herramienta para la validación formal Java Path Finder (JPF)**

Esto es, realizar estudio de las características de JPF, sus opciones de análisis de aplicaciones y diferentes configuraciones con el objetivo de adecuarla a las necesidades del proyecto.

2- **Análisis de la tecnología Java EE para la creación de aplicaciones empresariales**

Esto es, estudiar las diferentes posibilidades que ofrece la plataforma Java EE para la creación de componentes empresariales, analizando tipos de Enterprise Java Beans (EJBs), *servlets* y diferentes servidores de aplicaciones que puedan ofrecer una mejor integración de la tecnología JPF. Todo esto, con el objetivo de diseñar la estructura de nuestra aplicación web.

3- **Integración de la herramienta JPF en la tecnología Java EE**

Esto es, diseño de la aplicación conociendo las características de las dos tecnologías (JPF y Java EE), y teniendo en cuenta que nuestro objetivo es que la aplicación sea viable para el acceso de múltiples clientes.

4- **Implementación de la integración JPF – Java EE**

Esto es, implementación de las clases e interfaces necesarias así como el montaje y configuración del entorno en que se ejecutará; por ejemplo, servidor de aplicaciones y sistema de archivos.

5- **Análisis de la viabilidad de la integración en aplicación mediante el análisis de rendimiento, utilizando para ello un curso real**

Esto es, realización de pruebas de rendimiento con tiempos de respuesta, de ejecución, consumo de memoria y número de usuarios concurrentes en la aplicación, para finalmente comprobar la viabilidad de la integración de JPF como aplicación Java EE.

1.3 Estructura de la Memoria

Para el desarrollo de los objetivos parciales la memoria tiene esta estructura:

- Bloque I: Introducción
    - Capítulo 1. Introducción y objetivos.
- Bloque II: Estado del arte

  Análisis de las diferentes tecnologías involucradas en la integración de Java Path Finder y Java EE.

    - Capítulo 2. JPF: Java Path Finder.
    - Capítulo 3. Java EE: Java Enterprise Edition.
    - Capítulo 4. Servidores de Aplicaciones para Java EE.
- Bloque III: Diseño, implementación y validación empírica

  Diseño de la aplicación teniendo en cuenta la información obtenida sobre las tecnologías previas. Implementación del diseño y pruebas de rendimiento para comprobar la viabilidad del proyecto.

    - Capítulo 5. Diseño del módulo empresarial.
    - Capítulo 6. Aspectos de implementación del módulo empresarial.
    - Capítulo 7. Validación empírica.
- Bloque IV: Conclusiones y futuras líneas de trabajo

  Resultado y conclusiones sobre la consecución del proyecto para indicar el grado de satisfacción de los objetivos.

    - Capítulo 8. Conclusiones y líneas futuras de trabajo.
- Bloque V: Apéndices
- Bloque VI: Glosario
- Bloque VII: Referencias e hiperenlaces

CAPITULO 2. JPF: Java Path Finder

Para poder utilizar la herramienta JPF, es necesario tener un conocimiento más profundo de su funcionamiento y de qué características ofrece, así como las diferentes configuraciones que puedan beneficiar a la integración con Java EE. Por ello, en este capítulo se analiza la herramienta, haciendo hincapié en los aspectos que pueden ser útiles y beneficiosos o que puedan suponer un obstáculo para nuestro objetivo.

2.1 Introducción a JPF

JPF es un software de validación formal ([2],[3],[4],[5],[6]) para código Java. Básicamente es una máquina virtual de Java (JVM), que ejecuta un programa no sólo una vez, sino teóricamente en todos sus posibles caminos, buscando defectos o errores de ejecución como *deadlocks* (abrazos mortales) o excepciones no manejadas a lo largo de todos los caminos potenciales, además de ciertas propiedades que se quieran observar, añadiendo ciertas clases como entrada de la aplicación. Si JPF encuentra un fallo, indica la ejecución que ha llevado hasta el mismo. En esto se diferencia de otros validadores, pues éstos no indican los pasos que se han dado hasta llegar al fallo. Se puede ver un esquema básico de funcionamiento en la Figura 1.

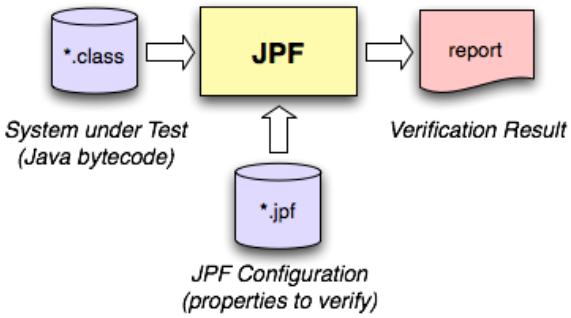

Figura 1: Esquema básico de la herramienta JPF (tomado de [1])

Está implementado en Java, por lo que no se ejecutará tan rápido como Java normal. Es una máquina virtual ejecutándose sobre otra.

Aunque hay semánticas de ejecución de códigos Java ya definidos, JPF tiene su propia semántica para acceder a las ejecuciones de aplicaciones. El conjunto de instrucciones de la máquina virtual está representado por un conjunto de clases que pueden ser reemplazadas o extendidas.

El conjunto de instrucciones por defecto hace uso de la capacidad de JPF llamada opciones de ejecución (*execution choices*). JPF identifica puntos en las aplicaciones desde donde la ejecución podrá proceder de manera diferente y después, sistemáticamente, analiza todos ellos. Las opciones típicas son las diferentes secuencias programadas o valores aleatorios, pero JPF permite introducir nuestro propio tipo de opciones como entradas de usuario o eventos de la máquina de estados.

El número de caminos de una aplicación puede crecer exponencialmente y normalmente lo hará hasta llegar a ser inmanejable, lo que se llama problema de explosión de estados. Para enfrentar este problema, se utiliza el método de coincidencia de estados (*State Matching*). Cada vez que se encuentra un punto de elección comprueba si ya ha encontrado un estado similar, en cuyo caso, abandona dicho camino para volver a un estado anterior inexplorado y continúa desde ahí. Es decir, JPF puede restablecer estados del programa.

Lo que JPF no puede hacer es ejecutar código nativo. Esto no es porque no se pueda conseguir, sino porque realmente no tendría mucho sentido; por ejemplo, las llamadas al sistema para escribir un archivo no pueden ser fácilmente revertidas. Sin embargo, hay solución para esto y es configurable, son las clases *native peers* (pares nativos) y *model* (modelo).

Las clases *native peers*, contienen métodos que son ejecutados en vez de los métodos nativos reales. Este código es ejecutado por la máquina virtual de Java, no por la de JPF, lo cual, puede, además, acelerar el proceso. Las clases *model* son simplemente sustitutas de las clases estándar como `java.lang.Thread`, que ofrece alternativas para los métodos nativos que son mucho más manejables por JPF.

En resumen, JPF se puede considerar como un marco general para técnicas de verificación de ejecución de código Java, ofreciendo un rico conjunto de mecanismos de configuración y abstracción, además de ser muy extensible. Hay fallos en las aplicaciones que sólo JPF puede encontrar, por lo que se define como una herramienta muy valiosa para aplicaciones con una misión crítica donde el error no es una opción.

2.2 Diseño de JPF

JPF se diseñó en base a dos principales abstracciones:

- La JVM (Máquina Virtual Java)
- El objeto `Search` (Búsqueda)

2.2.1 La máquina virtual

La JVM es el generador de estados específico de Java. Ejecutando instrucciones Java, la JVM genera representaciones de estados que pueden ser:

- Comprobado para igualdad (si el estado ha sido visitado antes)
- Accedido para obtener datos (estados de hilos y valores de datos)
- Almacenado
- Restaurado

Las principales parametrizaciones de la JVM, son clases que implementan el manejo de los estados (*matching*, *storing*, *backtracking*). La mayoría del esquema de ejecución se delega a la clase `SystemState`, la cual, por su parte utiliza la clase `SchedulerFactory` (un objeto factoría para `ThreadChoiceGenerators`) para generar secuencias programadas de interés.

Hay tres grandes métodos de la maquina virtual en el contexto de colaboración máquina virtual – búsqueda (VM – `Search`). La interacción de estos métodos con el paquete de búsqueda está representada en la Figura 2:

- *forward* – genera el siguiente estado, reporta si el estado generado tiene algún sucesor y si es así, lo almacena en una pila de retorno para una restauración eficiente.
- *backtrack* – restablece el último estado de la pila de retorno.
- *restoreState* – restablece un estado arbitrario (que no necesariamente estuviera en la pila de retorno).

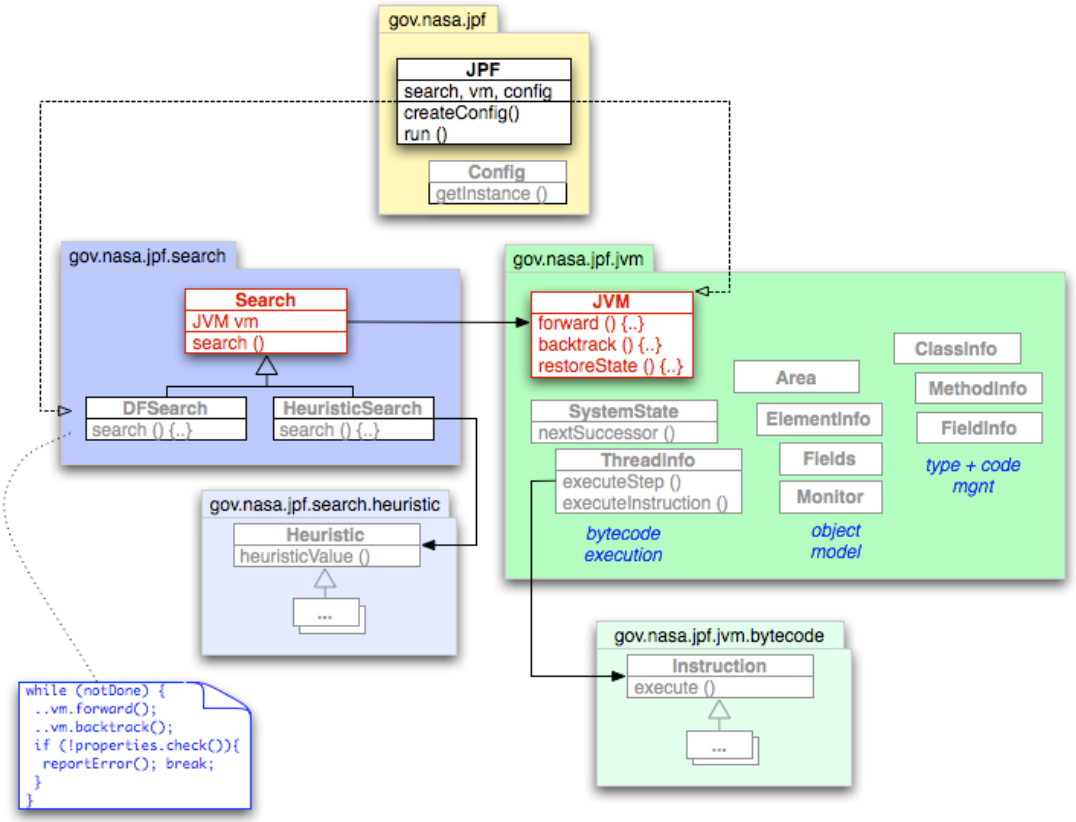

Figura 2. Interacción de paquetes de JPF (tomado de [1])

2.2.2 Estrategia de búsqueda

El objeto búsqueda es el responsable de seleccionar el estado desde el cual la JVM debería proceder, o indicando a la JVM que debe generar el siguiente estado (*forward*), o indicándole que debe regresar a un estado generado anteriormente. Los objetos `Search`, se podrían definir como controladores de la máquina virtual.

Dichos objetos también configuran y evalúan objetos *property*. Las principales implementaciones de `Search` incluyen una simple búsqueda *depth-first* (`DFSearch`) y una búsqueda basada en cola de prioridad, que puede ser parametrizada para hacer diferentes tipos de búsquedas basadas en la selección del estado más interesante del conjunto de todos los sucesores de un estado dado (`HeuristicSearch`). Una implementación simple de `Search`

provee principalmente un único método de búsqueda, el cual, incluirá el bucle principal que itera a través del espacio de estados relevantes hasta que ha sido totalmente explorado o la búsqueda encuentra un fallo.

2.3 Estructura de paquetes

El núcleo de JPF está dividido en los siguientes paquetes:

- **`gov.nasa.jpf`:**

La principal responsabilidad de este paquete es la configuración e instalación de los objetos del núcleo de JPF, como son `Search` y JVM. La configuración en sí misma está delegada a la clase `Config`, la cual contiene varios métodos para crear objetos o leer valores de una jerarquía de archivos de propiedades y opciones de línea de comandos. Más allá de la configuración, el objeto JPF posee una pequeña funcionalidad propia. Es un constructor, que se utiliza principalmente para arrancar JPF desde una aplicación Java, sin tener que ahondar en su configuración.

- **`gov.nasa.jpf.jvm`:**

Este paquete constituye el cuerpo principal del código del núcleo, incluyendo las diferentes construcciones que implementan el generador de estados Java. Conceptualmente, la clase más importante es JVM, pero, de nuevo, ésta delega la mayoría del trabajo a un conjunto de clases de segundo nivel, las cuales, unidas conforman la principal funcionalidad de JPF. Estas clases pueden ser divididas en tres categorías:

1- Manejo de Clases: Comienza con `ClassInfo` y contiene la mayor parte de la información invariante del estado de ejecución sobre campos y métodos.

2- Modelo de objetos: Todos los objetos dato en JPF se almacenan como *array* de enteros encapsulados por objetos `Field` (campo). El estado fijo específico de la ejecución de los objetos es capturado en instancias `Monitor`. Las instancias de `Field` y `Monitor` unidas conforman los objetos; los cuales se almacenarán como `ElementInfo`. El conjunto, que será una instancia `Area`, es simplemente un *array* dinámico de objetos `ElementInfo`. Los índices de dicho *array* son utilizadas como referencias a los objetos.

3- Ejecución de *bytecode*: Esto principalmente es una colaboración entre `SystemState` y `ThreadInfo` con algo de delegación hacia objetos que implementan *Partial Order Reduction* (POR, Sección 2.5). Esto comienza con el objeto JVM llamando a `SystemState.nextSuccessor()`, que se traduce en una llamada `ThreadInfo.executeStep()`, el cual, en un momento dado llamará a `ThreadInfo.executeInstruction()` para llevar a cabo la ejecución de *bytecode*. La ejecución actual es de nuevo delegada a instancias `Instruction`, específicas de *bytecode*, que por defecto residen en el paquete `gov.nasa.jpf.jvm.bytecode`.

- **`gov.nasa.jpf.search`:**

Este paquete es relativamente pequeño y principalmente contiene la clase `Search`, que es una clase abstracta de políticas de búsqueda. El método principal, que encapsula la política, es `Search.search()`, el cual, es el controlador de la JVM (llamando a `forward()`, `backtrack()` and `restore()`). Este paquete también contiene la política de búsqueda *depth-first* más simple: `DFSearch`. Se pueden encontrar políticas más interesantes en el sub-paquete `gov.nasa.jpf.search.heuristic`, el cual utiliza la clase `HeuristicSearch` en conjunción con objetos `Heuristic` para priorizar una cola de estados potenciales de sucesión.

2.4 Choice Generators

El chequeo de modelo de software ([3]) se basa en hacer las elecciones correctas para llegar a los estados interesantes, dentro de las limitaciones del sistema, del entorno de la herramienta y la ejecución. Ésto hace referencia al mecanismo utilizado por JPF para explorar sistemáticamente el espacio de estados, *Choice Generators*.

Choice Generators puede ser abordado desde una perspectiva de aplicación o desde una perspectiva de implementación JPF. Se harán ambas.

2.4.1 Motivación

Mientras la mayoría de las elecciones durante la aplicación JPF son relativas a la programación o planificación de hilos, el ejemplo que justifica obviamente la aproximación por implementación, pertenece a la rama de adquisición de datos no deterministas. El soporte para la adquisición de datos “aleatoria”, utilizando la interfaz `gov.nasa.jpf.jvm.Verify`, ha estado en JPF desde hace mucho tiempo.

Ésto funcionó bien para pequeños conjuntos de valores de elecciones (como `{true,false}` para *boolean*); pero, el mecanismo para la enumeración de todas las elecciones de un intervalo de tipo específico, ya es cuestionable para intervalos grandes (`Verify.getInt(0,10000)`) y falla completamente si el tipo de datos no permite, en ningún caso, conjuntos de elecciones finitas, como tipos de coma flotante.

Para abordar este caso, hay que olvidar el mundo ideal del chequeo de modelos, que considera todas las opciones posibles, y hacer uso de lo que se sabe del mundo real. Se deben utilizar heurísticos para hacer el conjunto de elecciones finito y manejable. Sin embargo, los heurísticos son específicos de la aplicación y del dominio, y sería una mala idea codificarlos como controladores (*drivers*) de *test*, de los que se le pasan a JPF para analizar. Ésto lleva a un número de requerimientos para el mecanismo de elección de JPF:

1. Los mecanismos de elección deben ser desacoplados. Por ejemplo, las elecciones de hilos deben ser independientes de las elecciones de datos.

2. El conjunto de elecciones y la enumeración debería estar encapsulada en objetos dedicados y de tipo específico. La VM sólo debería conocer los tipos más básicos y, por otro lado, utilizar una interfaz genérica para obtener elecciones.

3. La selección de clases que representan heurísticos de dominio específico y parametrización de instancias `ChoiceGenerator` debería ser posible en tiempo de ejecución, por ejemplo, a través de mecanismos de configuración de JPF

(propiedades).

El diagrama de la Figura 3, muestra ésto con un ejemplo que utiliza un valor de velocidad elegido aleatoriamente de tipo `Double`. Como ejemplo de heurístico, se utiliza un modelo de umbral. Por ejemplo, se desea saber cómo reacciona el sistema por debajo, dentro y por encima de un cierto valor específico de la aplicación (umbral) y se reduce un conjunto infinito de elecciones a sólo tres. Por supuesto, interesante es bastante subjetivo, y probablemente se quiera jugar con los valores de forma eficiente (umbral y uso de la heurística), sin tener que reconstruir la aplicación cada vez que se ejecute JPF.

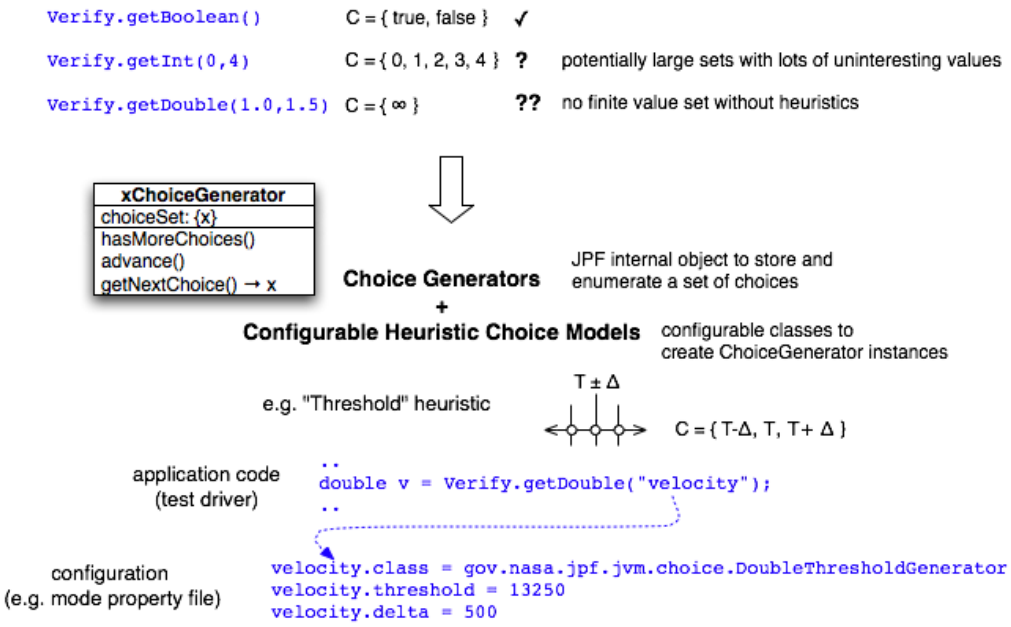

Figura 3. Uso del API de Verify (tomado de [1])

El código de ejemplo, no menciona la clase `ChoiceGenerator` utilizada (`DoubleThresholdGenerator`) sólo especifica un nombre simbólico (*velocity*) que utiliza JPF para buscar un nombre de clase asociado desde sus datos de configuración, inicializado a través de archivos de propiedades o por línea de comando. Pero no acaba aquí, la mayoría de heurísticos necesitan parametrización (umbral, *delta*...), y se lleva a cabo esto pasando los datos de configuración de JPF al constructor de `ChoiceGenerator` (por ejemplo, la propiedad *velocity*. *threshold*). Cada instancia de `ChoiceGenerator` conoce su nombre simbólico (*velocity*) y puede utilizar dicho nombre para buscar cualquier parámetro que necesite.

2.4.2 La perspectiva JPF

Disponer de dicho mecanismo es bueno para evitar la modificación del controlador de *test*. Pero sería mucho mejor utilizar consistentemente el mismo mecanismo no sólo para las elecciones de obtención de datos, sino también para elecciones de programación (o planificación). El mecanismo *Choice Generators* de JPF hace justamente esto, pero para entenderlo desde una perspectiva de implementación, se debe dar un paso atrás y mirar terminología JPF:

- ***State*** (estado) es una instantánea del estado de la ejecución actual de la aplicación (en su mayoría estados de hilos y *heaps*), más el historial de ejecución (*path*) que conduce a este estado. Cada estado tiene un número de identificación único. Dentro de JPF, State se encapsula en la instancia de `SystemState`, aunque hay algo de historial de ejecución, que guarda el objeto JVM. Esto incluye tres componentes:
  - ***KernelState*** – La instantánea de la aplicación (hilos, *heaps*).
  - ***Trail*** – La última transición (historial de ejecución).
  - **`ChoiceGenerator` actual y siguiente** – Los objetos que encapsula la enumeración de elecciones, que producen diferentes transiciones, pero no necesariamente nuevos estados.
- ***Transition*** (transición) es una secuencia de instrucciones que pasa de un estado a otro. No hay cambio de contexto dentro de una transición, todo está en el mismo hilo. Pueden existir múltiples transiciones que salen de un estado, pero no necesariamente a un nuevo estado.
- ***Choice*** (elección) es lo que empieza una nueva transición. Esto puede ser un hilo diferente o diferentes valores aleatorios en los datos.

En otras palabras, la posible existencia de *Choices* (elecciones u opciones) es lo que termina la última *Transition* (transición), y la selección de un valor de *Choice* (opción) se opone a la siguiente *Transition*. La primera condición corresponde a la creación de un nuevo `ChoiceGenerator` y dejar que `SystemState` sepa sobre él. La segunda condición consiste en pedir el nuevo valor de *Choice* a este `ChoiceGenerator`, ya sea internamente dentro de la JVM, en una instrucción o en un método nativo.

En la Figura 4 se representa gráficamente el mecanismo *Choice Generator*, cuyos términos se han descrito en los párrafos anteriores.

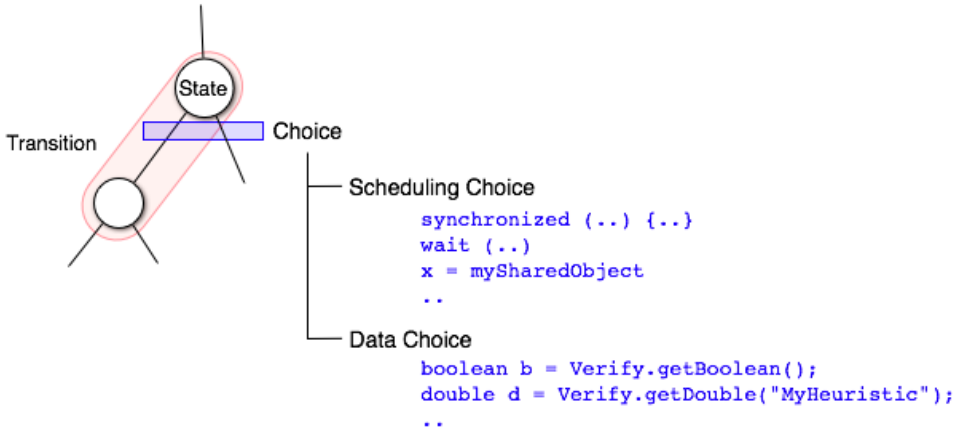

Figura 4. Mecanismo *Choice Generator* (tomado de [1])

Los detalles de funcionamiento de *Choice Generators* se pueden encontrar en la Sección A.1 del Apéndice.

2.5 On-The-Fly Partial Order Reduction

El número de las diferentes combinaciones planificadas es el principal factor para el tamaño del espacio de estados de los programas concurrentes. Afortunadamente, para la mayoría de los objetivos prácticos no es necesario explorar todas las posibles intercalaciones de instrucciones para todos los hilos. El número de estados inducidos planificados puede ser reducido significativamente agrupando todas las secuencias de instrucciones en un hilo que no puede tener efectos fuera de sí mismo, concentrándolos en una sola transición.

JPF emplea un on-the-fly POR que no confía en instrumentación de usuario o en análisis estático. JPF automáticamente determina, en tiempo de ejecución, qué instrucciones tienen que ser tratadas como fronteras de transición entre estados. Si POR está habilitado (configurado mediante las propiedades *vm.por*), una petición avanzada a la VM ejecuta todas las instrucciones en el hilo actual, hasta que se da una de las siguientes condiciones:

1. La siguiente instrucción es relevante para la planificación.
2. La siguiente instrucción produce un resultado no determinístico.

La detección de ambas condiciones se delega al objeto instrucción (`Instruction.execute()`), transmitiendo información sobre el actual estado de ejecución de la VM y del contexto de hilos. Si la instrucción es un divisor de transición, crea un `ChoiceGenerator` y se planifica a sí misma para la re-ejecución.

Cada tipo de instrucción de código corresponde a una subclase concreta de `gov.nasa.jpf.Instruction` que determina la relevancia de la programación basada en los siguientes factores:

- **Tipo de Instrucción** – Debido a la naturaleza basada en pila (*stack*) de la JVM, solo el 10% de las instrucciones de código Java son relevantes en la programación; por ejemplo, pueden tener efectos en los límites de los hilos (o procesos). Las instrucciones interesantes incluyen sincronización directa (*monitorenter*, *monitorexit*, *invokeX* en métodos sincronizados), acceso a campos (*putX*, *getX*), acceso a elementos de *array* (*Xaload*, *Xastore*) e invocación de llamadas de ciertos `Threads` (`start()`,`sleep()`,`yield()`,`join()`) y métodos de `Object` (`wait()`, `notify()`).
- **Alcanzabilidad de objetos** – Además de instrucciones de sincronización directa, el acceso a campos es el tipo de interacción principal entre hilos. Sin embargo, no todas las instrucciones *putX/getX* tienen que ser consideradas. Sólo las que se refieren a objetos que son alcanzables por dos hilos (como mínimo) pueden causar condiciones de carrera en los datos. Aunque el análisis de alcanzabilidad es una operación costosa, la VM ya realiza una tarea similar durante la recolección de basura, la cual se extiende para soportar POR.
- **Información de `Thread` y `Lock`** – Incluso si el tipo de instrucción y la alcanzabilidad de objetos sugiere relevancia en la programación (*scheduling*), no hay necesidad de romper la transición actual en caso de que no haya otro hilo ejecutable. Además, la adquisición y la liberación del bloqueo (*monitorenter*, *monitorexit*) no tienen que ser consideradas como límites de transición si tiene que

suceder de forma recursiva. Sólo la primera y última operación de bloqueo puede dar lugar a la reprogramación.

El esquema de funcionamiento de POR está representado en la Figura 5, indicando cómo se tienen en cuenta los factores descritos en los párrafos anteriores para la programación de la ejecución de instrucciones.

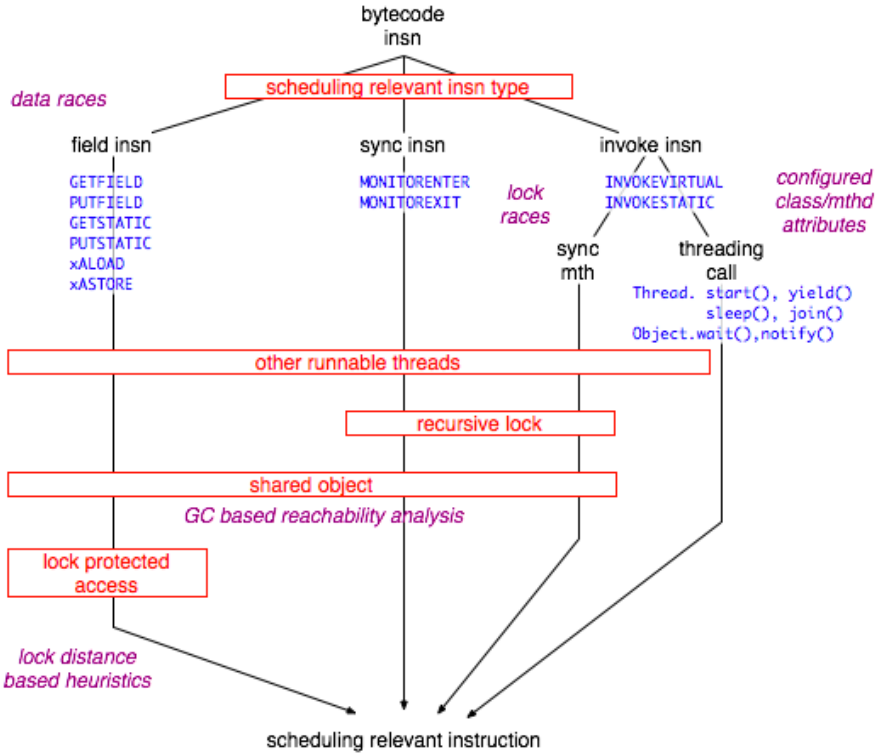

Figura 5. Funcionamiento POR (tomado de [1])

Mientras JPF utiliza estas informaciones para deducir la relevancia en la programación, existen tres mecanismos para controlar explícitamente límites de transiciones (por ejemplo, intercalado de hilos):

- **`Attributor`** – Una clase configurable concreta de este tipo es utilizada por JPF durante la carga de clases para determinar objetos, métodos y campos de atributos de la clase seleccionada y de las configuraciones de clases. Los atributos más importantes con respecto a POR son la atomicidad de métodos y los niveles de relevancia en la programación:

  (a) nunca relevante.

  (b) siempre relevante.

  (c) sólo relevante en el contexto de otros ejecutables.

  (d) sólo relevante en el nivel más alto de bloqueo.

  El `Attributor` por defecto todo el código `java.*` atómicamente; lo que puede ser demasiado agresivo (puede causar excepciones `BlockedAtomicExceptions`).

- **`VMListeners`** – Un escuchador puede pedir explícitamente una reprogramación llamando a `ThreadInfo.yield()` en respuesta de una notificación de ejecución de instrucción.

- **`Verify`** – La clase `Verify` sirve como un API para comunicación entre la aplicación de *test* y JPF, y contiene funciones `beginAtomic()`, `endAtomic()` para controlar el intercalado de hilos.

Los detalles de funcionamiento de POR se pueden encontrar en la Sección A.2 del Apéndice.

2.6 Sistema de atributos

Mientras JPF almacena valores para operandos, variables locales y campos de forma muy similar a una VM normal, también contiene un mecanismo de extensión de almacenamiento que permite asociar objetos arbitrarios con dichos valores. Estos objetos se pueden utilizar en pares nativos (*native peers*) o escuchadores (*Listeners*) para añadir información de estados (almacenado/restaurado) que automáticamente sigue el flujo de datos.

JPF provee un API para modificar y acceder a estos atributos, el cual está localizado en `gov.nasa.jpf.jvm.Fields` (para atributos de campos) y `gov.nasa.jpf.jvm.StackFrame` (para variables locales y operandos). Una vez establecidos, la JVM copia los atributos cada vez que lee o escribe los campos asociados o ranuras de `StackFrame`. Ésto se puede observar en el esquema de la Figura 6.

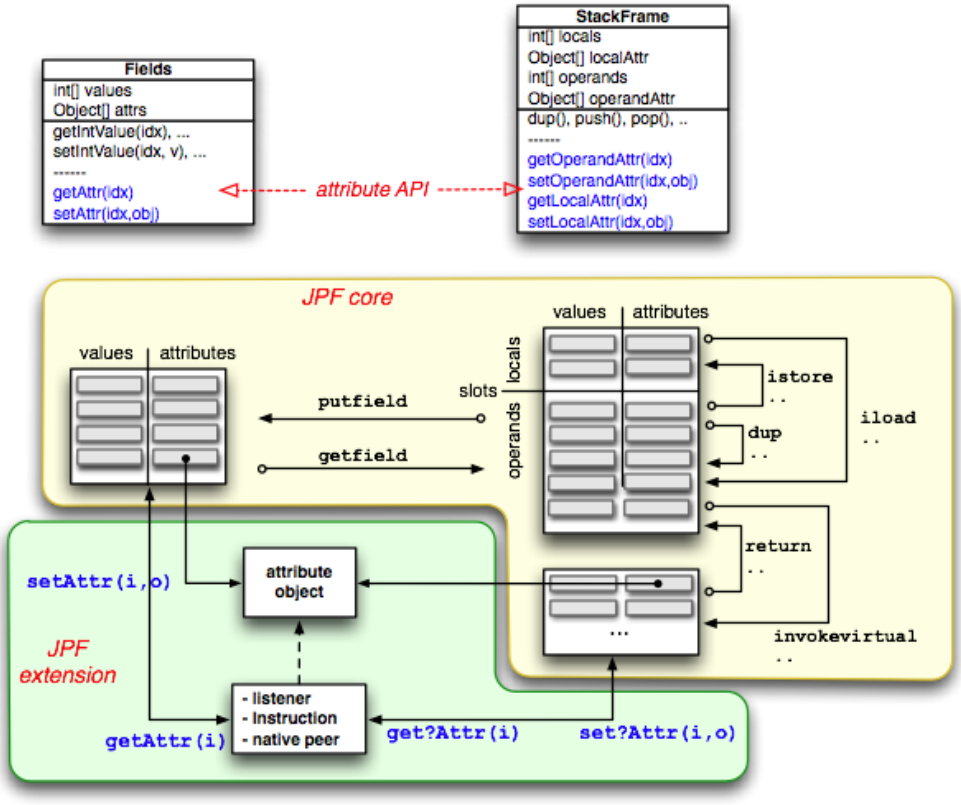

Figura 6. Sistema de atributos JPF (tomado de [1])

2.6.1 Utilización

Dichos atributos se pueden utilizar para representar valores simbólicos o límites de errores numéricos. Hay que tener en cuenta que los atributos imponen un coste de ejecución adicional. Por ello, no se tratan valores concretos y normales como tipos de atributos (se utilizan tipos `int`, por ejemplo). El valor de ésto es que los atributos coexistan con valores normales y concretos para permitir cosas como ejecuciones mixtas, simbólicas y concretas.

2.7 Listeners

Los escuchadores (*Listeners*) son quizás el mecanismo de extensión más importante de JPF. Proveen una forma de observar e interaccionar con una ejecución de JPF con nuestras propias clases. Se configuran dinámicamente en tiempo de ejecución y no requieren modificación del núcleo de JPF. Se ejecutan al mismo nivel que JPF por lo que no hay casi límite en lo que se puede hacer con ellos.

El principio general es simple: JPF provee una implementación que notifica a instancias de observadores registradas sobre ciertos eventos en el nivel de búsqueda (`Search`) y la JVM. Estas notificaciones cubren un amplio espectro de operaciones de JPF, desde eventos a bajo nivel, como *instructionExecuted*, hasta eventos de alto nivel, como *searchFinished*. Cada notificación es parametrizada con la correspondiente fuente (instancia `Search` o JVM), lo cual puede ser utilizado por el escuchador notificado para obtener más información sobre el evento o el estado interno de JPF. El funcionamiento de los *Listeners* se puede ver esquematizado en la Figura 7.

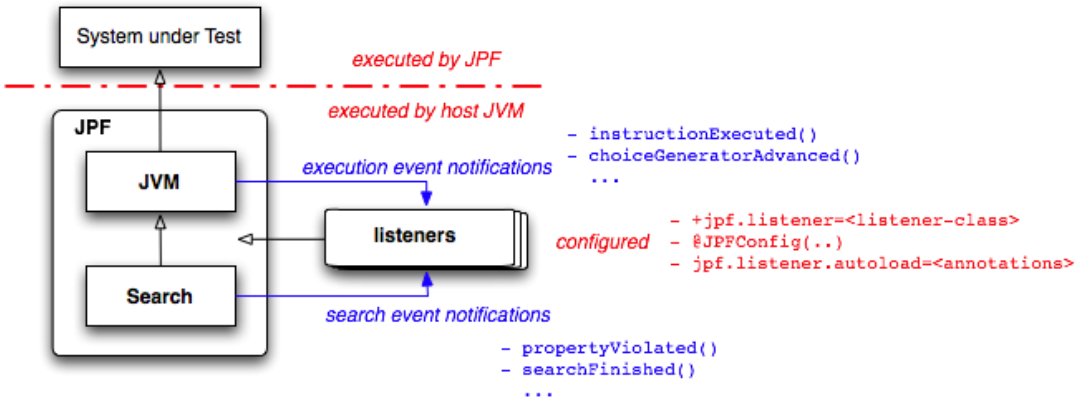

Figura 7. Esquema de funcionamiento de *Listeners* (tomado de [1])

2.7.1 Tipos de Listeners

Hay dos tipos básicos de interfaces escuchadoras dependiendo de las correspondientes fuentes de eventos: `SearchListeners` y `VMListeners`. Estas interfaces son bastante grandes y los escuchadores a menudo necesitan implementar ambas, por lo que también se proveen clases adaptadoras. Son interfaces que contienen todos los métodos requeridos vacios. Los escuchadores solo deberán implementar los métodos que le interesen de dichas interfaces.

Las clases adaptadoras se utilizan para la mayoría de implementaciones de escuchadores. Dichas clases también soportan dos mecanismos más de interfaz/extensión que a menudo se utilizan junto con `Search/VMListeners`: `Property`, para definir características del programa, y `PublisherExtension`, para producir salida dentro del sistema de reporte de JPF. Los tipos y su interacción se pueden ver en la Figura 8 y son los siguientes:

- **`ListenerAdapter`**: es la interfaz del adaptador dedicado a `SearchListener`, `VMListener` (Ver la Sección A.3 del Apéndice) and `PublisherExtension`. Es lo más utilizado para obtener información en tiempo de ejecución.

- **`PropertyListenerAdapter`**: es la interfaz adaptadora que se utiliza en caso de que el escuchador implemente una propiedad (`Property`) del programa. (Ejemplo de implementación: `PreciseRaceDetector`; en la Sección A.3.2 del Apéndice).

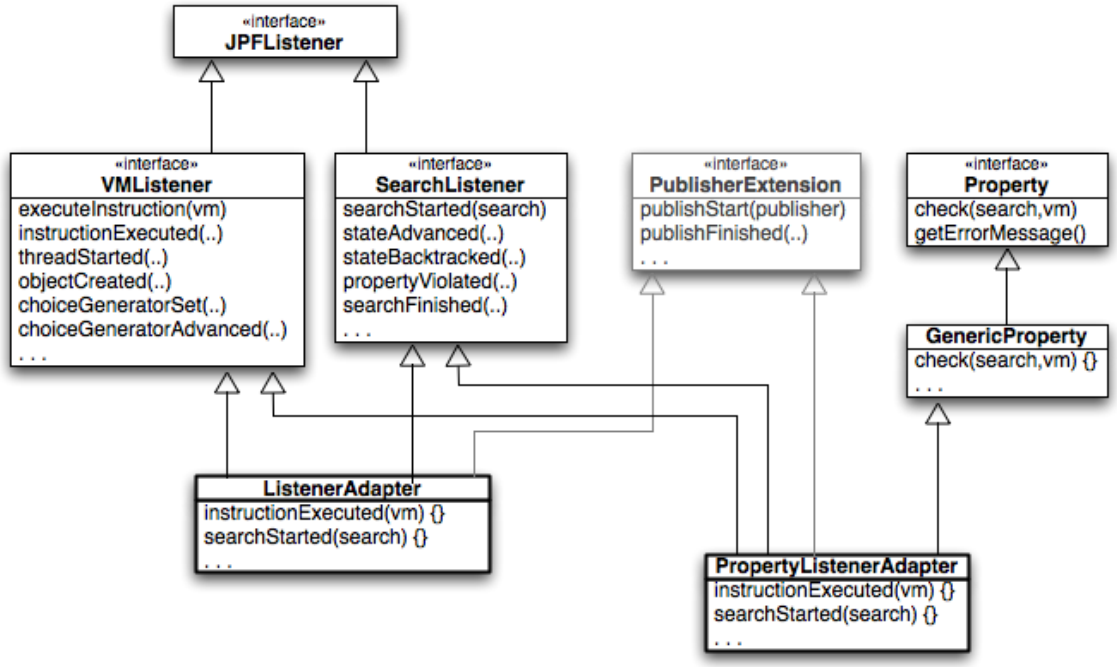

Figura 8. Tipos de *Listeners* e interacción (tomado de [1])

Elegir el tipo correcto de escuchador es muy importante, ya que JPF registra automáticamente escuchadores y propiedades basados en este tipo. Se puede implementar directamente la interfaz simple de escuchador, pero entonces habría que hacer el registro propio.

Normalmente las notificaciones por sí mismas no son suficientes y el escuchador necesita obtener más información de JPF. Para esto se proveen los objetos `Search` y VM como argumentos de la notificación y el escuchador tendrá que utilizarlos para preguntar o interaccionar con JPF. Por lo tanto, es importante implementar el escuchador dentro del paquete correcto. (Detalles de tipos y configuración en la Sección A.3 del Apéndice).

2.8 Model Java Interface (MJI)

Incluso siendo sólo una aplicación Java (sólo clases Java), JPF puede ser visto como una máquina virtual Java (JVM) en sí misma. La consecuencia es que los archivos *.class*, a veces, son procesados de una forma diferente en una JVM que ejecuta JPF:

1- Como clases Java normales manejadas y ejecutadas por la JVM del *host* (clases de la librería estándar de Java y clases de implementación de JPF).

2- Como clases modeladas, manejadas y procesadas (verificadas) por JPF.

La búsqueda de clases en ambas capas se basa en la variable/parámetro de la línea de comandos CLASSPATH, pero esto no debería ofuscar el hecho de que se deba distinguir claramente entre estos dos modelos. En particular, JPF (capa *Model*) tiene su propio modelo de clases y objetos, lo cual es completamente diferente e incompatible con los modelos de clases y objetos (ocultos) del host JVM subyacente que ejecuta JPF.

Cada JVM estándar soporta lo que se denomina *Java Native Interface* (JNI), que se utiliza para delegar ejecución del nivel Java (código controlado por la JVM) hacia la capa nativa (código máquina). Esto se utiliza normalmente para ciertas funcionalidades de la interfaz hacia la plataforma del sistema operativo (OS) o la arquitectura.

Curiosamente, existe una necesidad análoga de reducir el nivel de ejecución en JPF desde código controlado por JPF a código controlado por la JVM. De acuerdo con esta analogía, la interfaz específica de JPF se llama *Model Java Interface* (MJI). Las diferentes capas e interfaces y su disposición se pueden ver en la Figura 9.

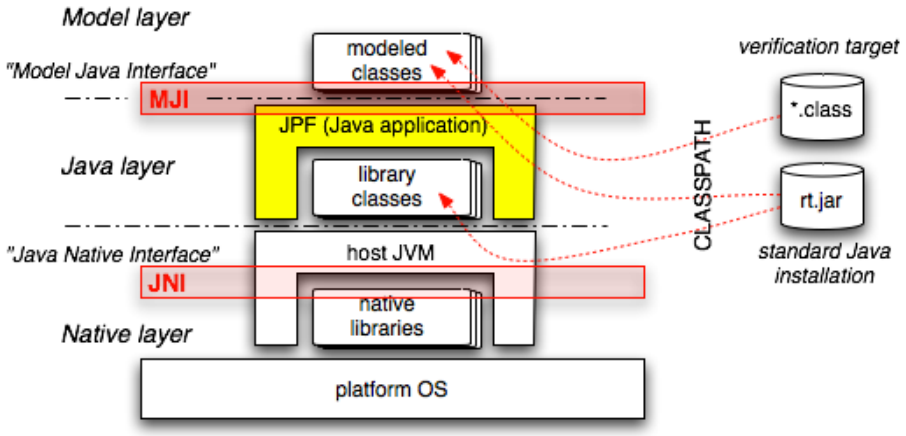

Figura 9. Capas e interfaces JPF (tomado de [1])

A pesar de que MJI ofrece un amplio rango de aplicaciones, hay tres usos principales para la delegación de la ejecución de código al host JVM:

1- **Intercepción de métodos nativos** – sin un mecanismo de reducción de abstracción, JPF sería forzado a ignorar completamente métodos nativos, por ejemplo, fallaría en aplicaciones basadas en los efectos secundarios de dichos métodos; lo cual puede ser inaceptable (muchos métodos nativos pueden ser ignorados si se restringe el conjunto de objetivos de verificación). Por ello, se incluye este mecanismo de intercepción de invocaciones de métodos, el cual permite delegar por medio de

llamadas de reflexión Java a las clases dedicadas. Hay dos tipos de clases involucradas, que residen en diferentes capas: ***Model Class*** y ***NativePeer Class***. (Ver la Sección A.4 del Apéndice)

2- **Interfaz de funcionalidad del nivel de sistema JPF** – algunas funciones del nivel del sistema de clases de la librería estándar (`java.lang.Class`, `java.lang.Thread`) tienen que ser interceptadas, incluso si no son nativas, porque afectan a la clase interna de JPF, modelo de objetos e hilos (carga de clases, creación/lanzamiento de hilos). Cabe señalar que MJI se puede también utilizar para extender la funcionalidad de JPF sin cambiar su implementación.

3- **Reducción del espacio de estados** – delegando la ejecución de código hacia el *host* JVM sin seguimiento de estados, se pueden dividir partes grandes del espacio de estados, siempre que se sepa que los efectos secundarios correspondientes del método no son relevantes para la comprobación de propiedades.

Además de estos usos estándar, existen aplicaciones más exóticas como la obtención de información sobre la exploración del espacio de estados de JPF y ponerlo a disposición tanto para JPF como para el objeto de la verificación. (Detalles sobre herramientas y ejemplos en la Sección A.4 del Apéndice).

2.9 Bytecode Factories

Normalmente una VM define la semántica de su lenguaje de programación. En el caso de Java, el conjunto de instrucciones correspondiente representa una máquina multi-hilo, donde los valores se guardan en el conjunto (*heap*) o dentro de ranuras locales – y/o operandos- dentro del marco de pila.

JPF es diferente. La JVM y sus constructores asociados, como `ThreadInfo`, `ClassInfo` o `ElementInfo`, ofrecen todos los medios necesarios para implementar un intérprete Java normal, pero JPF delega el uso de estos medios a las instrucciones.

Cada *bytecode* que es ejecutado por JPF se representa mediante su correspondiente objeto `Instruction` que normalmente se instancia durante el tiempo de carga de clase. Las clases `Instruction` del modo de ejecución estándar se pueden encontrar en el paquete `gov.nasa.jpf.jvm.bytecode`.

Cuando se va a ejecutar un *bytecode*, la JVM simplemente llama al método `execute()` de esta instancia de `Instruction`. Pase lo que pase dentro de estos métodos, define la semántica de ejecución.

La explicación de este fenómeno es que JPF utiliza una factoría configurable para elegir e instanciar las clases `Instruction`. Ofreciendo una `InstructionFactory` propia concreta, junto con un conjunto de clases `Instruction` relacionadas, se puede cambiar la semántica de ejecución Java.

2.9.1 Utilidad

Una razón para extender la semántica normal es crear chequeos adicionales. Esto está hecho en la extensión *[[ext:número|número]]* JPF, que sobrescribe clases numéricas de

bytecode con versiones que comprueban *over-/underflow* entre otras cosas (ver detalles de implementación y configuración en la Sección A.5 de los Apéndices).

2.10 Conclusiones

JPF es una herramienta de validación formal muy completa que permite el análisis de aplicaciones de manera controlada, lo cual es muy útil para el fin del proyecto. Después de analizar toda la funcionalidad de la herramienta, se sabe que JPF podrá comprobar y manejar:

- Secuencias programadas.
- Variaciones en los datos de entrada.
- Eventos del entorno.
- Elecciones de control de flujo.

Como aspecto negativo sobre JPF se puede prever que siendo básicamente una máquina virtual ejecutándose sobre otra, consumirá mucha memoria física y CPU de la máquina donde se ejecute.

Como aspecto positivo, la herramienta permite centrarse en características específicas de las aplicaciones a validar utilizando *Listeners*. Además, una opción muy beneficiosa que ofrece es la llamada *On-the-fly Partial Order Reduction* (POR), la cual si está activada, permite analizar qué instrucciones son relevantes y cuales no, lo que supone una mayor eficiencia y un menor número de estados a analizar. Dicha opción deberá ser activada para la integración.

CAPITULO 3. Java EE: Java Enterprise Edition

En este capítulo se analiza la tecnología Java EE con la que se integrará la herramienta JPF para que pueda ser accedida via web. Se profundizará en las características que puedan ser interesantes y permitan el acceso de un gran número de usuarios.

3.1 Introducción a Java EE

Java EE ([7]) es una plataforma de programación que permite desarrollar y ejecutar software de aplicaciones en Java con una arquitectura distribuida de *n* niveles, basándose en componentes software modulares ejecutados sobre un servidor de aplicaciones. Su estructura está representada en la Figura 10:

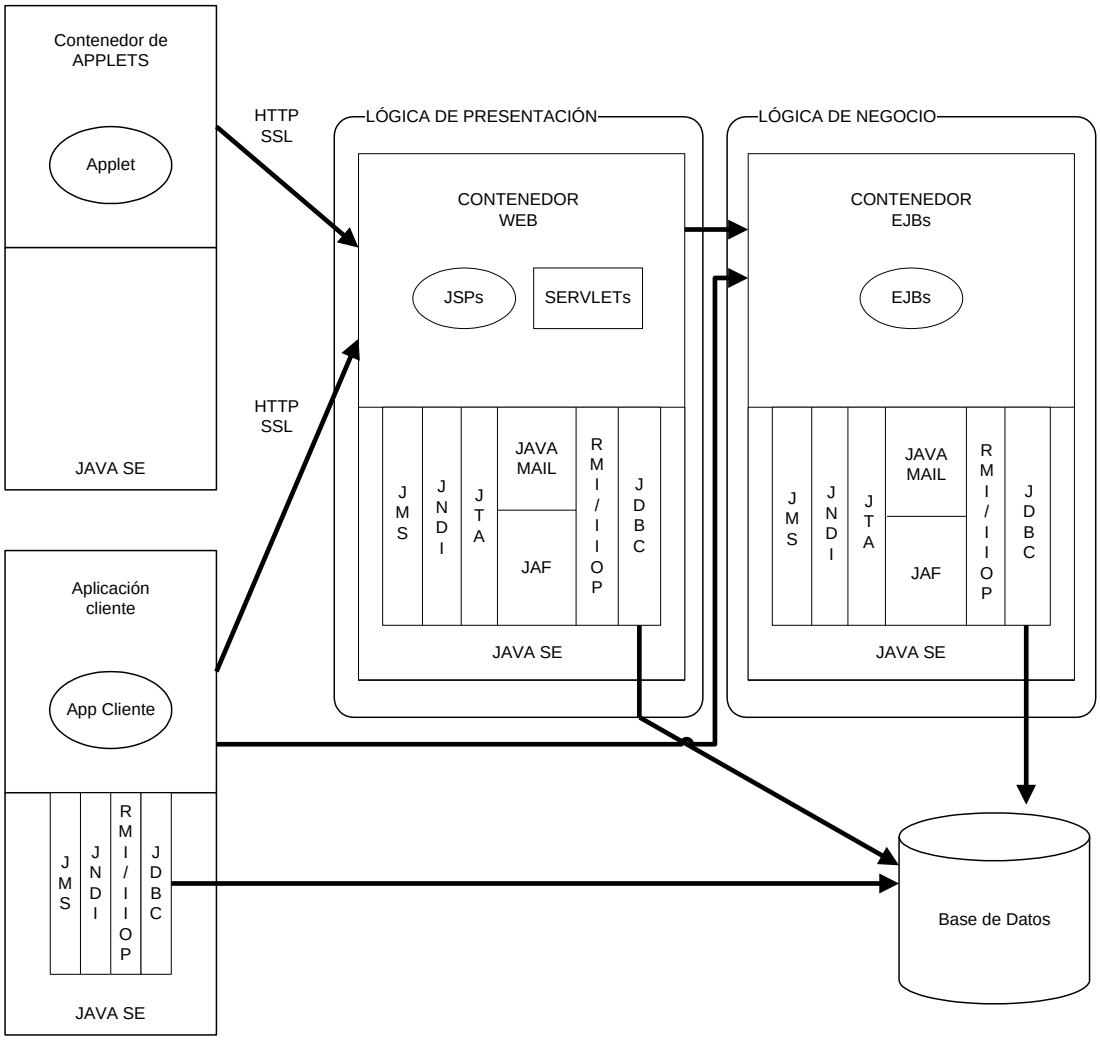

Figura 10. Estructura de una aplicación Java EE

Incluye varias especificaciones de API, como JDBC, RMI, e-mail, JMS, Servicios web y XML y define cómo coordinarlos. Además también incluye algunas especificaciones únicas

para componentes. Éstas incluyen Enterprise JavaBeans, *servlets*, portlets, Java Server Pages y varias tecnologías de servicios web.

Todo esto permitirá crear una aplicación empresarial portable entre plataformas, escalable e integrable con las tecnologías anteriormente mencionadas, lo cual hace que ésta sea una opción muy interesante para el proyecto. Otros beneficios añadidos son que el servidor de aplicaciones puede manejar transacciones, seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, de forma que para el desarrollador la tarea se reduce al diseño de la lógica de negocio.

3.1.1 APIs

En esta sección se describen brevemente algunos de los diferentes paquetes y APIs que contiene la tecnología Java EE. Son los siguientes:

- **APIs Generales:** Son APIs que extienden la funcionalidad de las APIs base de Java SE.

- **EJBs (`javax.ejb`):** La API *Enterprise JavaBeans* (EJBs) define un conjunto de APIs que un conjunto de objetos distribuidos soportará para suministrar persistencia, RPCs (llamadas remotas RMI o RMI-IIOP), control de concurrencia, transacciones y control de acceso para objetos distribuidos.

- **JNDI (`javax.naming`):** Los paquetes `javax.naming`, `javax.naming.directory`, `javax.naming.event`, `javax.naming.ldap` y `javax.naming.spi` definen la API de *Java Naming and Directory Interface* o JNDI. Permite a los clientes descubrir y buscar objetos y nombres a través de un nombre y será independiente de la aplicación subyacente.

- **JDBC (`java.sql`):** Los paquetes `java.sql` y `javax.sql` definen el API de JDBC. JDBC es un API que maneja la conectividad Java con base de datos. Define métodos para peticiones y actualizaciones de datos de la base de datos.

- **JTA (`java.transaction`):** Estos paquetes definen la *Java Transaction API* (JTA). JTA establece una serie de interfaces Java entre el manejador de transacciones y las partes involucradas en el sistema de transacciones distribuidas: el servidor de aplicaciones, el manejador de recursos y las aplicaciones transaccionales.

- **JAXP (`javax.xml`):** Estos paquetes definen el API de *Java API for XML Processing* (JAXP). Es un API que se encarga de la manipulación y tratamiento de archivos XML.

- **JMS (`javax.jms`):** Estos paquetes definen el API de *Java Messaging Service* (JMS). JMS es un estándar de mensajería de Java EE que permite crear, enviar, recibir y leer mensajes. Provee un servicio fiable y flexible para un intercambio asíncrono de información de negocio crítica.

- **JPA (`javax.persistence`):** Este paquete provee las clases e interfaces para gestionar la interacción entre los proveedores de persistencia, las clases administradas y los clientes del API de persistencia Java (JPA). JPA busca unificar

la manera en que funcionan las utilidades que proveen un mapeo objeto-relacional. Su utilidad es no perder las ventajas de orientación a objetos al interactuar con la base de datos y permitir utilizar objetos regulares (POJOs – *Plain Old Java Objects*).

3.2 EJBs: Enterprise JavaBeans

La tecnología EJB ([8],[9]) es la arquitectura de componentes del lado del servidor de la plataforma Java EE. Permite un desarrollo rápido y simplificado de aplicaciones distribuidas, transaccionales, seguras y portables para la tecnología Java. Forma parte de la lógica de negocio de una aplicación empresarial Java.

3.2.1 Versiones de los EJBs

La especificación EJB ha ido evolucionando a la par que lo hacía la propia especificación J2EE. Las diferentes versiones que han existido hasta la fecha son:

- **EJB 1.0:** la especificación original.
- **EJB 1.1:** la primera incluida dentro de J2EE.
- **EJB 2.0:** incluida en J2EE 1.3 añadía las interfaces Locales y los Message-Driven beans.
- **EJB 2.1:** incluida en la última revisión de J2EE, la 1.4.
- **EJB 3.0:** Ahora con Cluster y está incluida en JEE 5.1 (La última estable).
- **EJB 3.1:** incluida en JEE 6 en diciembre de 2009.

3.2.2 Java EE 5 y EJB 3.0: Versiones para el proyecto

EJBs 3.0 es un API que forma parte del estándar de construcción de aplicaciones empresariales Java EE 5 ([11]). Existen tres tipos diferentes de EJBs:

- **EJB de Entidad (*Entity Bean*):**

Encapsulan los objetos del lado del servidor que almacena los datos. Presentan la característica fundamental de persistencia. En EJB 3.0 los Entity Beans pasan a pertenecer al API JPA y cambia la forma de implementación.

JPA es el API de persistencia desarrollada para Java EE e incluida en el estándar EJB 3. Su objetivo es no perder las ventajas de la orientación a objetos al interactuar con una base de datos, y permitir el uso de POJOS. Consta de:

- *Java Persistence API*
- *Query Language*
- *Object relational mapping metadata*

Ciclo de vida: El control de un *Entity Bean* reside en el API *EntityManager*. Cuando se produce un evento en dicho *Bean*, *EntityManager* llama a los métodos *lifecycle-callbacks* si la clase los ha implementado. Éstos son: *PrePersist, PostPersist, PreRemove, PostRemove, PreUpdate, PostUpdate, PostLoad.*

- **EJB de sesión (*Session Bean*):**

Gestionan el flujo de información del servidor. Representan procesos ejecutados en respuesta a una solicitud del cliente. Puede haber de dos tipos:

- Con estado (*stateful*): Los beans de sesión con estado son objetos distribuidos que poseen un estado, éste estado no es persistente, pero el acceso al bean se limita a un solo cliente. Su ciclo de vida se puede ver en la Figura 11.

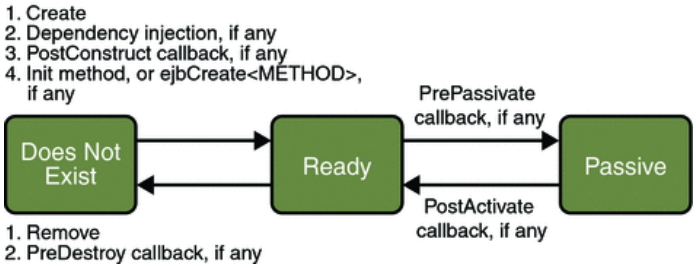

Figura 11. Ciclo de vida de un Session Bean con estado (tomado de [11])

- Sin estado (*stateless*): Son objetos distribuidos que carecen de estado asociado, permitiendo por tanto que se los acceda concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método. Su ciclo de vida se puede ver en la Figura 12.

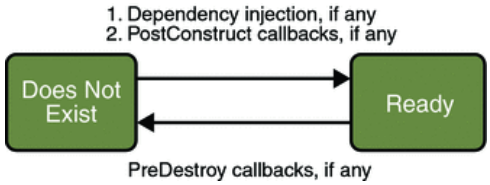

Figura 12. Ciclo de vida de un Session Bean sin estado (tomado de [11])

- **EJB de mensajes (*Message-driven Bean*)**

Son los únicos beans con funcionamiento asíncrono. Utilizan el *Java Messaging System* (JMS), se suscriben a un tema (*topic*) o a una cola (*queue*) y se activan al recibir un mensaje dirigido a dicho tema o cola. No requieren de instanciación por parte del cliente. Su ciclo de vida se puede ver en la Figura 13.

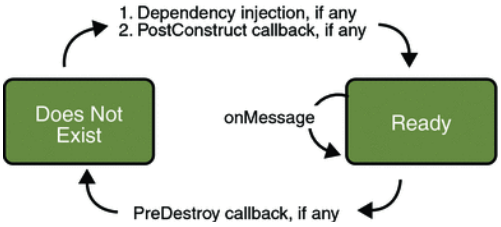

Figura 13. Ciclo de vida de un Message Bean (tomado de [11])

3.2.2.1 Interfaces Local y Remota de EJBs 3.0

Interfaz Remota: Los *Beans* ofrecen métodos concretos que pueden ser llamados por un cliente, aunque pueden existir métodos que no sean públicos. Para ello, a cada *Bean* le corresponde una interfaz ya sea Local o Remota, como mínimo un *Bean* deberá implementar una de ellas. La interfaz Remota es para aquellos *Beans* que puedan ser llamados desde una máquina virtual de Java distinta a donde se encuentra el servidor de aplicaciones. Incluye la anotación `@Remote`.

Interfaz Local: Normalmente se utilizan para ofrecer servicios a otros *Session Beans* que se encuentran en el mismo servidor de aplicaciones, es decir, en la misma máquina virtual, donde pueden tratarse de métodos que aparezcan en un *Bean* cuya interfaz es remota. Ya no es necesario incluir métodos del ciclo de vida en ésta interfaz, como se podía hacer en la versión 2.1. Incluye la anotación `@Local`.

3.2.2.2 Cambios introducidos por la versión EJB 3.0

El modelo de programación propuesto por la versión 2.1 conllevaba una serie de inconvenientes que limitaron mucho el uso de esta especificación. El objetivo de los EJBs 3.0 es simplificar el desarrollo de aplicaciones Java y estandarizar el API de persistencia para la plataforma Java. Las principales características son:

1. ***Metadata Annotation* (Anotaciones Java):** EJB 3.0 utiliza anotaciones para simplificar el desarrollo de componentes. Ahora no hay necesidad de escribir un descriptor de despliegue, pero todavía lo soporta. Las anotaciones podrán ser sobrescritas por dicho descriptor.

2. **Encapsulamiento de dependencias del entorno:** Ahora se utilizan las anotaciones y el mecanismo de inyección de dependencias para encapsular las dependencias del entorno y el acceso a JNDI.

3. **Especificación de EJBs más simplificada:** Ahora ya no hay necesidad de escribir interfaces locales y de componente ni de implementar la interfaz `javax.ejb.EnterpriseBean` para la clase EJB. La clase EJB es ahora una clase pura de Java, también conocida como POJO, y la interfaz es conocida como POJI, una simple interfaz Java. Por esto se podrá desarrollar una aplicación empresarial mucho más rápido.

4. **Inyección de dependencia:** El API de búsqueda y la utilización del entorno y las referencias a recursos de los EJBs se ha simplificado utilizando anotaciones.

5. **Simplificación de la persistencia:** La persistencia de los objetos entidad, ahora es más simple a través de la introducción del API de persistencia de Java (JPA). Se introduce un nuevo API llamado `EntityManager`, el cual se utiliza para crear, encontrar, eliminar y actualizar entidades. Ahora estos objetos soportarán herencia y polimorfismo. Para establecer dichas relaciones en EJB 3.0 se utilizarán anotaciones.

6. **Interfaces de *Callback*:** Elimina la necesidad de implementar métodos de *callback* de ciclo de vida innecesarios.

7. **Timer Service:** Se suele utilizar cuando se desea realizar un proceso controlado temporalmente, ya sea en una fecha determinada o cada cierto tiempo. Se suele utilizar en *Stateless Session Beans* y *Message Driven Beans* (MDBs).

8. **Interceptors y Entity Listeners:** Un interceptor es una clase cuyos métodos se ejecutan cuando se llama al método de otra clase totalmente diferente. Se podrá configurar para *Session Beans* y MDBs, pero no para los *Entities*.

9. **web Services:** Cualquier servicio que pueda llamarse mediante los protocolos utilizados en el *World Wide web* por un cliente situado remotamente. Toda la comunicación está basada en XML (petición y respuesta). Estos XMLs se crean bajo el estándar SOAP (Simple *Object Access Protocol*). SOAP define reglas de serialización, empaquetado de datos y generación de mensajes. En EJB 3.0 se pueden utilizar anotaciones para su implementación.

3.3 Servlets y Java Server Pages

Estas tecnologías del lado del servidor se incluyen en el estándar Java EE. Forman parte de la lógica de presentación de una aplicación empresarial Java.

3.3.1 Servlets

Un *servlet* es un objeto que se ejecuta en un servidor o en un contenedor Java EE, especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML. Otra opción que permite generar contenido dinámico son los JSPs (*Java Server Pages*).

Un *servlet* implementa la interfaz `javax.servlet.Servlet` o hereda de alguna de las clases convenientes para un protocolo específico, por ejemplo, `javax.servlet.HttpServlet`. Al implementar esta interfaz, el *servlet* es capaz de interpretar los objetos del tipo `HttpServletRequest` y `HttpServletResponse`, quienes contienen la información de la página que invocó el *servlet*. Las invocaciones que admite son de tipo GET, POST o PUT entre otras y cada una de ellas tendrá un método específico en el *servlet* que las controlará.

Entre el servidor de aplicaciones y el *servlet* existe un contrato que determina cómo han de interactuar.

3.3.1.1 Ciclo de vida

El ciclo de vida de un *servlet* se divide en los siguientes puntos:

1. El cliente solicita una petición al servidor vía URL

2. El servidor recibe la petición:

    - Si es la primera, se utiliza el motor de *servlets* para cargarlo y se llama al método `init()`.

    - Si ya está iniciado, cualquier petición se convierte en un nuevo hilo. Un *servlet* puede manejar múltiples peticiones de clientes.

3. Se llama al método `service()` para procesar la petición devolviendo el resultado al cliente.

4. Cuando se apaga el motor de un *servlet* se llama al método `destroy()` que lo destruye y libera los recursos.

3.3.2 Java Server Pages (JSPs)

JSPs es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Las JSPs permiten la utilización de código Java mediante *scripts*. Además, es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de bibliotecas de etiquetas (*TagLibs* o *Tag Libraries*) externas e incluso personalizadas. Se puede considerar una alternativa a los *servlets* o se pueden utilizar de forma complementaria.

Los JSPs son en realidad *servlets*: un JSP se compila a un programa en Java la primera vez que se invoca y del programa en Java se crea una clase que se empieza a ejecutar en el servidor como un *servlet*. La principal diferencia entre los *servlets* y los JSPs es el enfoque de la programación: un JSP es una página web con etiquetas especiales y código Java incrustado, mientras que un *servlet* es un programa Java puro que recibe peticiones y genera a partir de ellas una página web.

3.4 Conclusiones

Habiendo analizado la tecnología Java EE se ve que ofrece características muy útiles para nuestro objetivo, como por ejemplo la tecnología JMS (*Java Messaging Service*) la cual permitiría controlar una cola de peticiones e ir atendiendo dichas peticiones de la forma que mas convenga mediante los EJBs de mensajes o MDBs (*Message Driven Beans*).

Además ofrece mecanismos para manejar las peticiones inicialmente, es decir, cuando se accede al servidor, como son los *servlets* y las JSPs. Éstos permiten crear una interfaz con la que los usuarios podrán interactuar, ya que generan código HTML.

CAPITULO 4. Servidores de Aplicaciones

Para poder ofrecer la herramienta JPF de validación via web, debe estar integrada con la tecnología Java EE y estar montado todo en un servidor de aplicaciones, para que la aplicación final Java EE funcione. En este capítulo se analizan los servidores de aplicaciones y en especial el servidor Glassfish de Sun ([20]), por ser de código libre y por su sencillez a la hora de su instalación y configuración.

4.1 Definición

El concepto de servidor de aplicaciones está relacionado con el concepto de sistema distribuido. Un sistema distribuido, en oposición a un sistema monolítico, permite mejorar tres aspectos fundamentales en una aplicación:

- Alta disponibilidad.
- Escalabilidad.
- Mantenimiento.

El estándar Java EE permite el desarrollo de aplicaciones empresariales de forma sencilla y eficiente. Una aplicación desarrollada con dicha tecnología podrá ser desplegada en cualquier servidor de aplicaciones que cumpla con el estándar. Un servidor de aplicaciones por tanto es una implementación de la especificación Java EE que proporcionará servicios que soportan la ejecución y disponibilidad de las aplicaciones desplegadas. Es el corazón de un gran sistema distribuido y proporciona una estructura en tres capas que permite estructurar nuestro sistema de forma más eficiente tal y como se puede ver en la Figura 14.

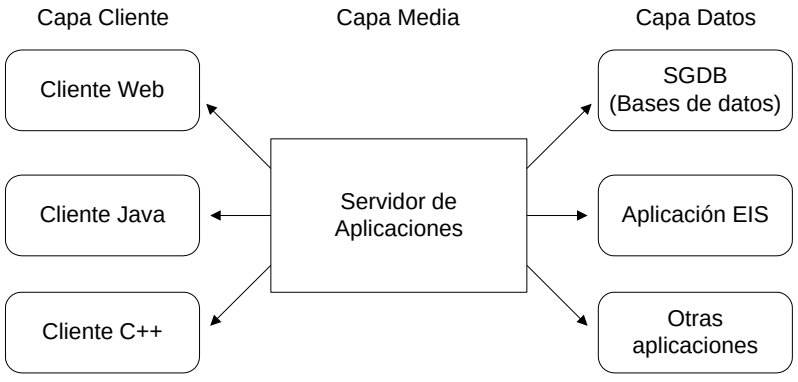

Figura 14. Capas de una aplicación empresarial

Los servidores de aplicaciones funcionan como contenedores de componentes de las aplicaciones JEE. Estos componentes son *servlets*, JSPs, y EJBs y permiten implementar diferentes capas de la aplicación, como la interfaz de usuario, la lógica de negocio, la gestión de sesiones de usuario o el acceso a bases de datos remotas.

Típicamente incluyen también middleware (o software de conectividad) que les permite intercomunicarse con diferentes servicios. Brindan además soporte a una gran variedad de estándares, como HTML, XML, IIOP, JDBC y SSL, que permiten su funcionamiento en ambientes web y la conexión a una gran variedad de fuentes de datos. En la Figura 15, mediante un diagrama de aplicación empresarial se indica lo que corresponde a la parte de servidor de aplicaciones.

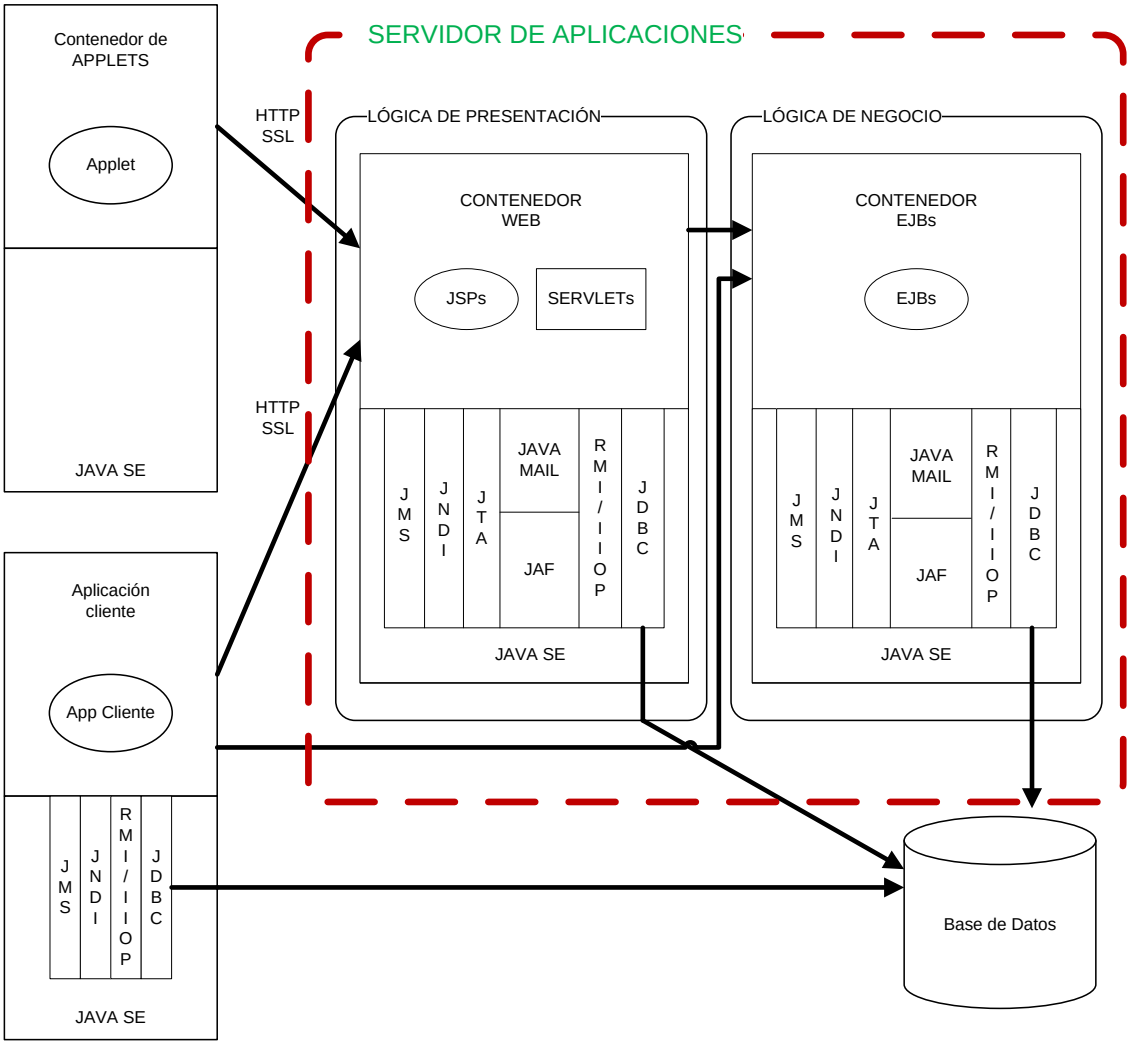

Figura 15. Partes del servidor de aplicaciones

4.2 Diferentes servidores de aplicaciones:

Existen diversas implementaciones de servidor de aplicaciones con diferentes características:

- *BEA WebLogic* ([16])
- *JBoss* ([17])
- *IBM WebSphere* ([18])
- *Sun Glassfish* ([20])
- *Borland AppServer* ([19])

De éstos servidores los más populares en la actualidad, por ser libres, son JBoss y Glassfish de Sun. En el proyecto se utilizará Glassfish entre otras cosas por su sencillez en la instalación, configuración y por ser de código abierto.

4.3 Servidor Glassfish de Sun

Glassfish de Sun es un servidor de aplicaciones sencillo y rápido basado en la plataforma Java EE para el desarrollo de aplicaciones empresariales. En la Tabla 1 se pueden ver sus características y funcionalidades:

Características	Beneficios
Compatible con Java EE 5 y 6	Implementa las últimas versiones de Java EE y está continuamente actualizándose.
Mejora la productividad del desarrollador	Con APIs Java EE simplificadas y anotaciones, se reduce el tamaño del código a desarrollar, por lo que se mejora en la productividad.
Arquitectura orientada a servicios (SOA)	Soporta JAX-WS 2.0, JAXB 2.0, y Open ESB, proveyendo una arquitectura abierta y extensa para la colaboración entre tecnologías de integración y servicios web.
Herramientas de desarrollo	Puede ser integrado con IDEs como NetBeans y Eclipse para un desarrollo más cómodo y eficiente.
Interfaz web sencilla	El servidor posee una interfaz web muy intuitiva que permite configurar completamente el funcionamiento del servidor de forma sencilla.
Instalación sencilla	Su instalación no conlleva mucha dificultad y existen multitud de guías para este fin.
Documentación abundante	Hay mucha documentación sobre uso, administración y desarrollo, lo cual es muy útil.
Código libre	Se puede obtener gratuitamente.

Tabla 1. Características del servidor de aplicaciones Glassfish

4.4 Conclusiones

El servidor Glassfish de Sun ofrece muchos beneficios para el proyecto, ya que permite su integración con IDEs como Eclipse, lo cual favorecerá un desarrollo más eficiente de nuestra aplicación empresarial. Su instalación es sencilla, y para su configuración existen infinidad de manuales y tutoriales. Además, siendo de código libre, supone un ahorro.

CAPITULO 5. Diseño del módulo empresarial

Una vez se han analizado las tecnologías a utilizar en el proyecto, se diseñará la aplicación indicando sus casos de uso, los componentes que la conformarán y el flujo de interacción.

El objetivo del proyecto es, básicamente, enviar prácticas a la aplicación para que sean validadas y devolver el resultado mediante correo. Por lo tanto, el diseño de esta aplicación web comprenderá la validación de prácticas vía página web, vía cliente de consola, el tratamiento de las prácticas enviadas (extracción y compilación), la posibilidad de gestión de archivos de validación y la notificación del resultado.

5.1 Casos de uso

El proyecto tiene básicamente tres casos de uso: validación accediendo desde una página web, validación accediendo desde el cliente de consola y gestión de archivo de validación, como se puede ver en la Figura 16.

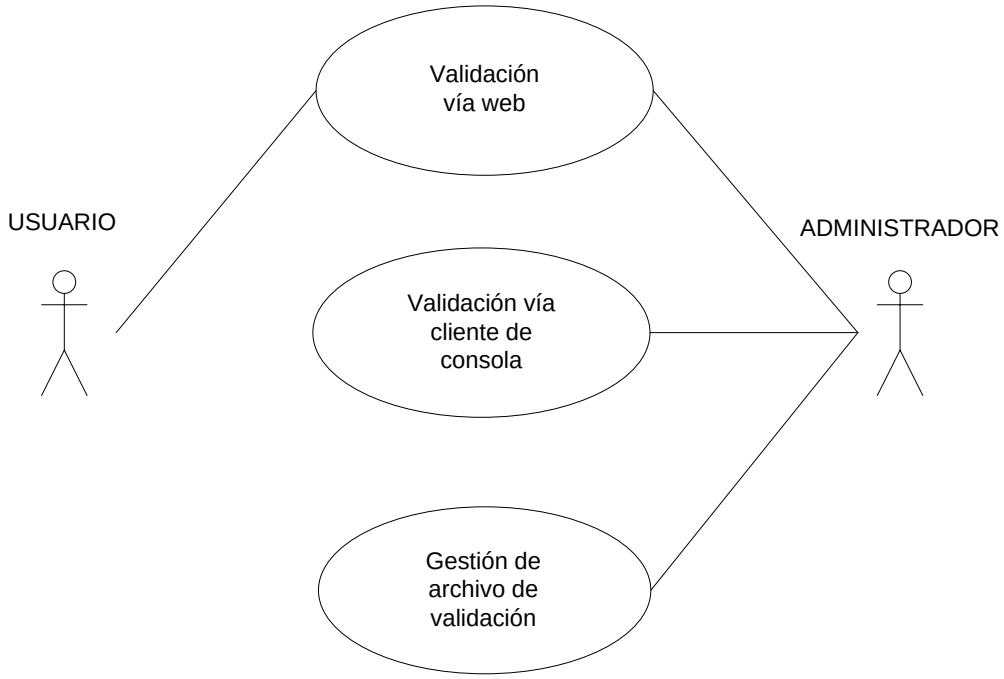

Figura 16. Casos de uso

5.1.1 Caso de uso: Validación vía web

En este caso de uso, los actores que intervienen son tanto el administrador como el usuario, ejerciendo en realidad el mismo papel, sin diferencias, como se observa en la Figura 17.

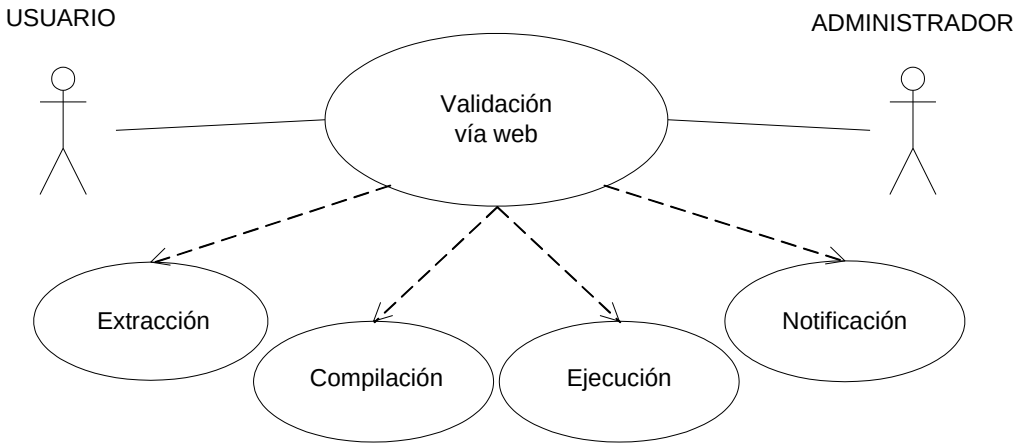

Figura 17. Caso de uso: Validación vía web

Previo a la consecución del caso de uso, el administrador podrá enviar al usuario la dirección web con la configuración necesaria ya codificada en la misma, de forma que se establezcan unos parámetros que el usuario no pueda modificar.

Para este envio se utilizará cualquier servidor de correo o incluso podrá ser colgada en otra página web para dar acceso a un grupo grande de personas, pero todo esto ajeno al sistema que modela este proyecto. La dirección enviada podrá tener o no configuración prefijada, si no es así, el usuario podrá modificar a su antojo la configuración en la interfaz web a la que apunta dicha dirección.

En este caso de uso intervienen todos los módulos del sistema con el objetivo final de validar la aplicación, y enviar el resultado de nuevo al usuario o el administrador del sistema.

Como se ve en el diagrama de interacción de la Figura 18, el usuario/administrador podrá enviar solo un paquete de archivos (correspondiente a una aplicación) por cada acceso a la web.

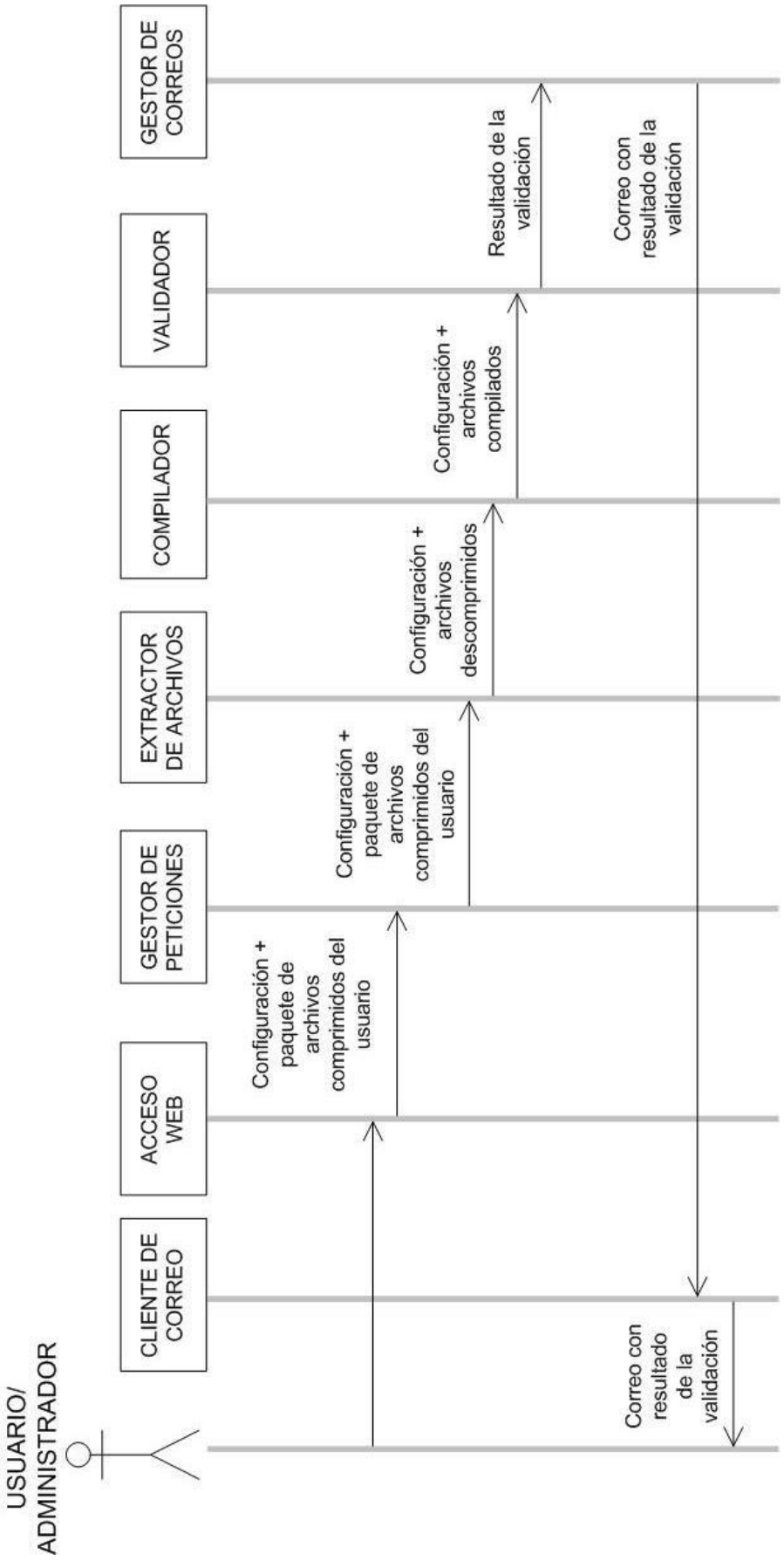

Figura 18. Diagrama de interacción de validación vía web

5.1.2 Caso de uso: Validación vía cliente de consola

Este caso de uso es similar al anterior como se puede ver en la Figura 19:

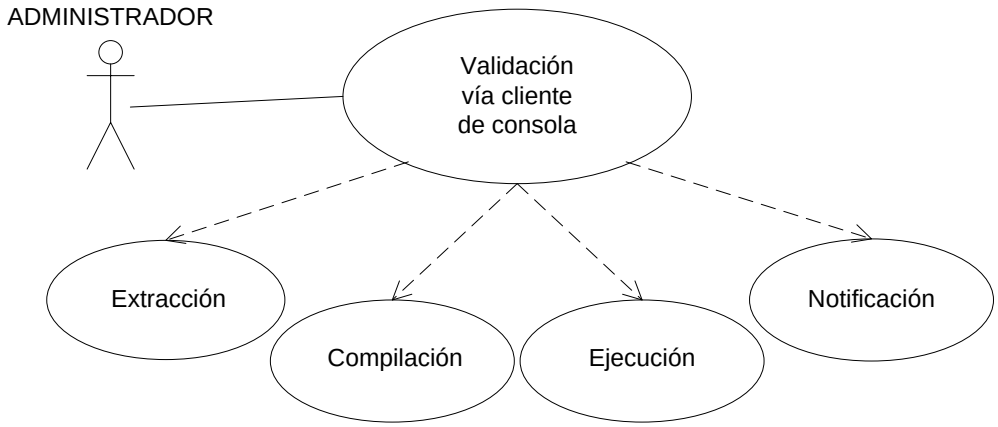

Figura 19. Caso de uso validación vía cliente de consola

Los módulos que interactúan en este caso de uso son los mismos que en el acceso web, la diferencia está en la interfaz y las opciones adicionales que ofrece.

En este caso el actor es el administrador únicamente, pues el cliente de consola ofrece opciones que el actor de tipo usuario no utiliza o no debería utilizar. La opción adicional que ofrece el cliente de consola, aplicable a este caso, es la posibilidad de seleccionar un directorio lleno de paquetes (diferentes aplicaciones) y poder enviarlas a la vez en el mismo acceso. Los módulos tratarían cada paquete como petición independiente y finalmente se devolvería un correo por cada aplicación enviada. En la Figura 20 se muestra su diagrama de interacción correspondiente.

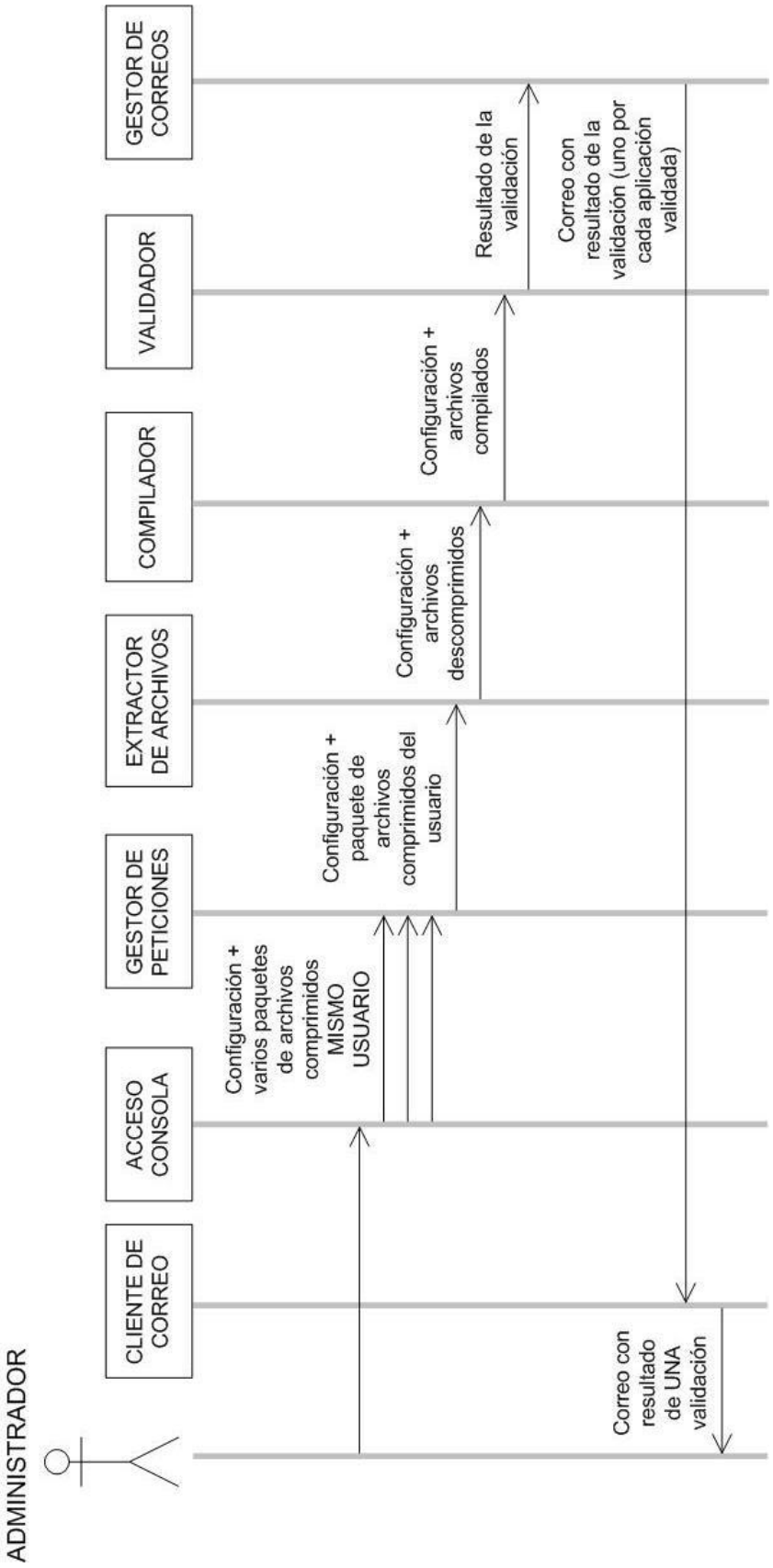

Figura 20. Diagrama de interacción de validación vía cliente de consola

5.1.3 Caso de uso: Gestión de archivos de validación

Este caso de uso, representado en la Figura 21, únicamente concierne al administrador y permite al administrador poder subir sus propios archivos de validación para que, después al enviar la dirección web configurada al usuario, pueda incluirla.

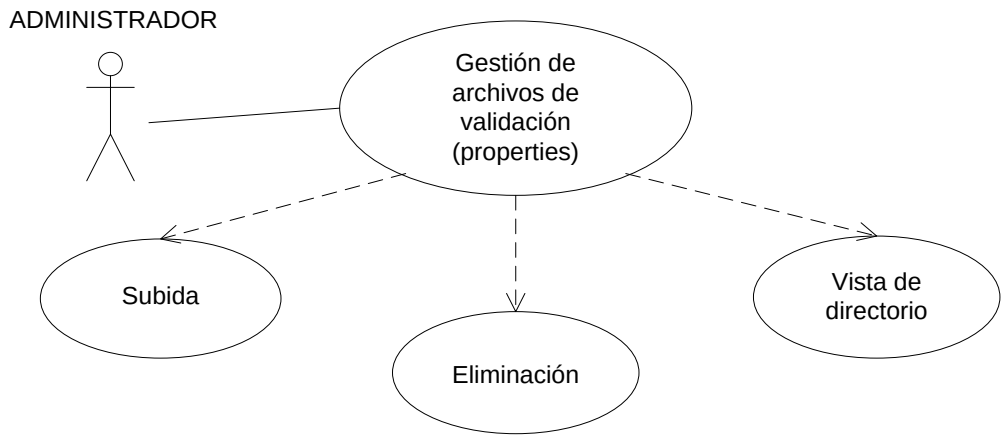

Figura 21. Caso de uso: Gestión de archivos de validación

Este caso de uso incluye tres acciones:

- Subida de archivo.
- Eliminación de archivo.
- Vista del directorio.

5.1.3.1 Subida de un archivo

En esta acción sólo interaccionan dos módulos del sistema, el gestor de peticiones, el cual se encarga de gestionar la transacción y de indicar el directorio donde se almacenará; y el extractor de archivos, ya que el archivo esperado será un archivo comprimido. Para su posterior utilización, la herramienta JPF tendrá registrada la dirección del directorio, y solo sería necesario indicar en la configuración de la dirección web que el administrador envía al usuario. En la Figura 22 se puede ver el diagrama de interacción de la subida de un archivo.

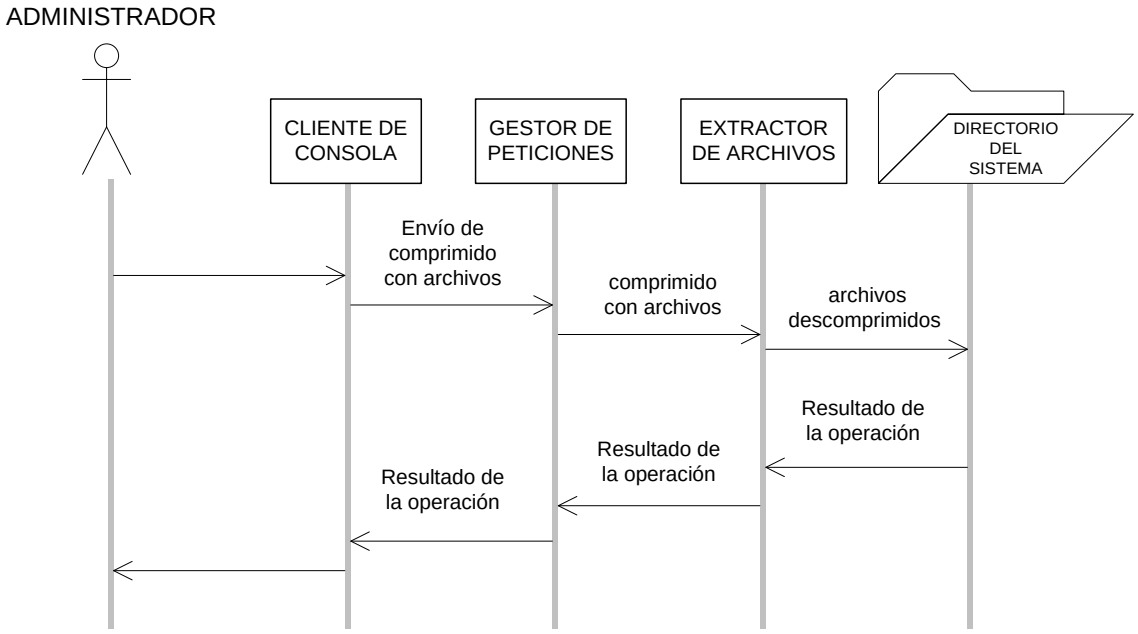

Figura 22. Diagrama de interacción de la subida de un archivo

5.1.3.2 Eliminado de archivo

También se ofrecerá la posibilidad de eliminar archivos que ya no sean necesarios para que no aumente innecesariamente el tamaño del directorio. El diagrama de interacción de la eliminación de un archivo se puede ver en la Figura 23.

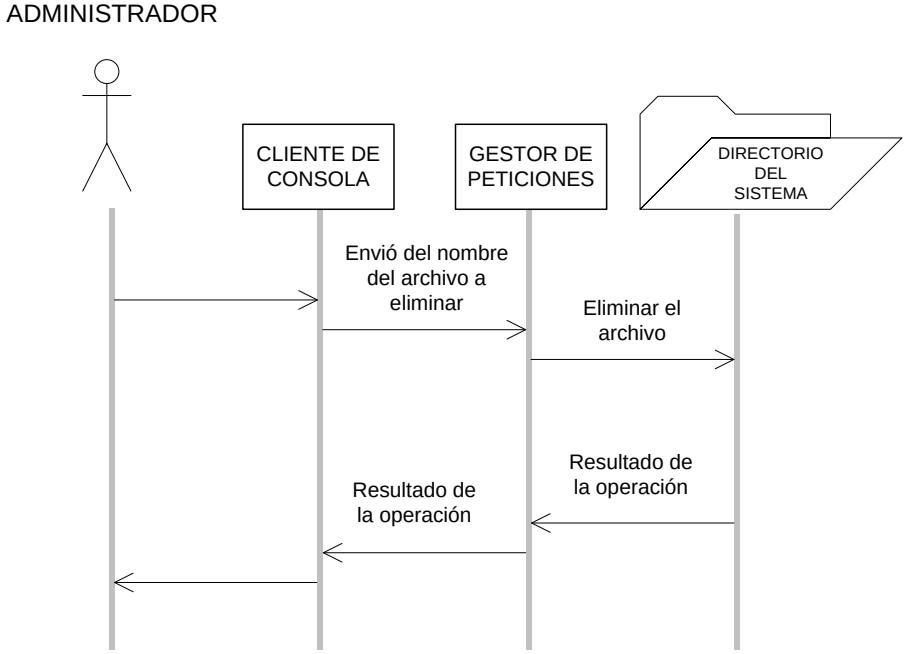

Figura 23. Diagrama de interacción de la eliminación de un archivo

5.1.3.3 Vista del directorio de archivos

Heredado de las dos acciones anteriores, se permitirá al administrador ver el contenido del directorio para que tenga un mejor control de los archivos subidos y de los archivos que se pueden eliminar. El diagrama de interacción de la vista del directorio se puede ver en la Figura 24.

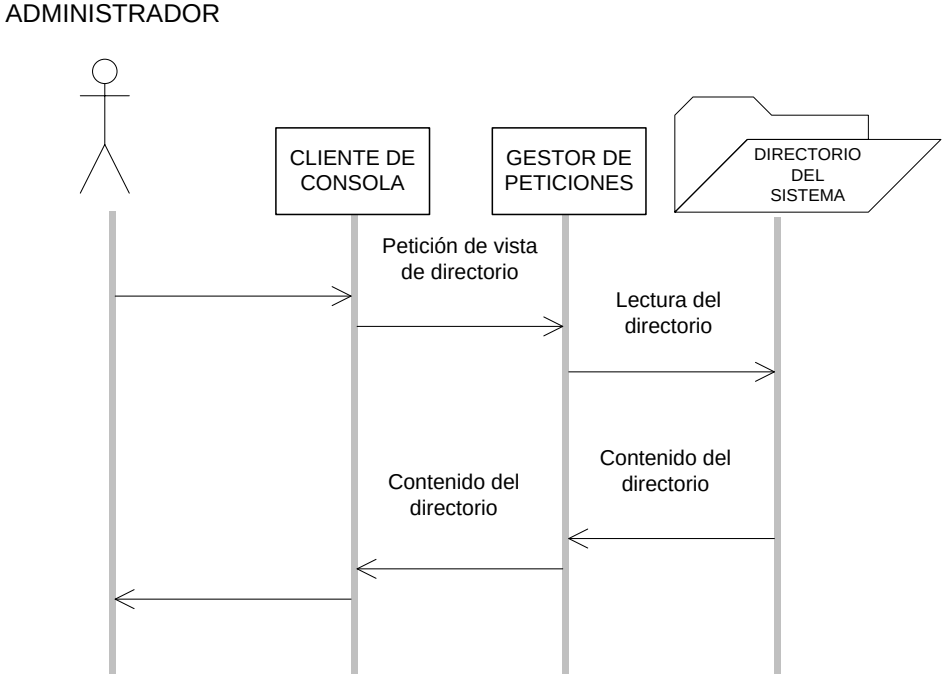

Figura 24. Diagrama de interacción de la vista del directorio

CAPITULO 6. Aspectos de implementación

Una vez diseñada la aplicación, se muestran ciertos aspectos de la aplicación ya implementada, para dar una idea de cómo está estructurada y los diferentes componentes que la forman.

6.1 Diagrama de componentes de la aplicación

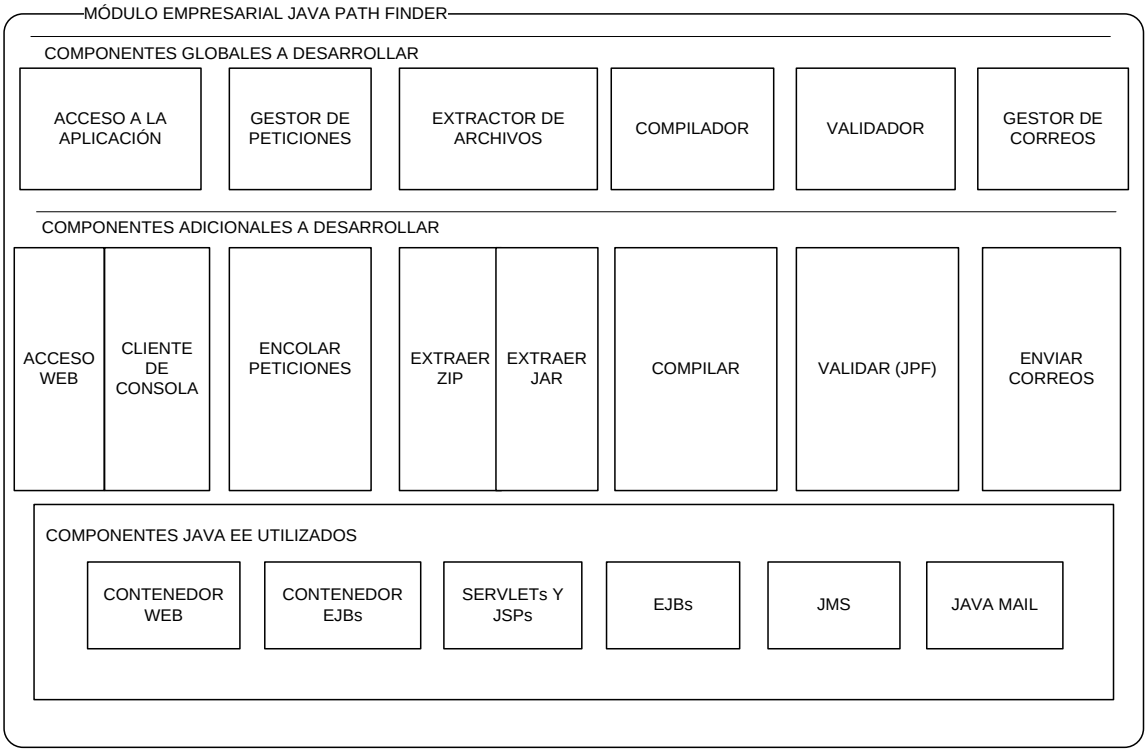

Figura 25. Diagrama de componentes de la aplicación

En el diagrama de componentes de la aplicación, que se puede ver en la Figura 25, se muestran todos los módulos que se implementan, sus componentes y la tecnología utilizada para ello. En las siguientes páginas se especificarán detalles de la implementación de estos módulos.

6.2 Acceso a la aplicación

Para el acceso a la aplicación se ha optado por dos tipos de interfaces: cliente web y cliente de consola.

6.2.1 Cliente web

El cliente web puede ser utilizado por todo tipo de usuarios (tanto usuarios normales como administrador), e incluye campos con todo lo necesario para la configuración de la posterior

validación. Está compuesto por una página JSP, que incluye la presentación con código HTML, y un *servlet* que gestiona las llamadas a la página y los datos que se envían desde ella. Este *servlet* forma parte de la implementación del módulo de gestión de peticiones que se describirá más adelante. En la Figura 26 se puede ver el cliente web completo.

Figura 26. Vista del cliente web completo

Los modos que ofrece esta interfaz son:

- Modo Normal: Se utiliza la configuración básica de la herramienta JPF, sin ningún tipo de modificadores, tal y como esté configurado por defecto en el servidor.

- Modo Usuario: Se utiliza la configuración básica de la herramienta JPF, pero con un archivo de chequeo adicional que el usuario haya incluido en el mismo comprimido enviado. La aplicación lo tomará automáticamente y lo tendrá en cuenta en la validación.

- Modo Experto: Se utiliza una configuración totalmente seleccionada por el usuario. Para esto, se debe rellenar el formulario que aparece justo debajo de la selección de modo. Dicho formulario se puede ver en la Figura 27.

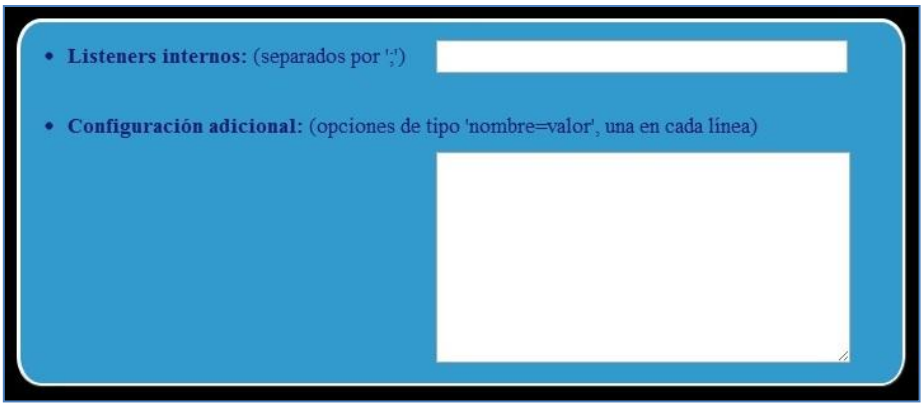

Figura 27. Panel modo experto de la web

Se ha implementado además la opción de incrustar en la URL (*www-urlencoded*) ciertos campos de forma que el navegador solo muestre los campos de envío de archivo. Así, el administrador podrá controlar con qué configuraciones se debe validar los archivos de los usuarios. Esta vista se muestra en la Figura 28.

Figura 28. Vista inicial del acceso web

6.2.2 Cliente de Consola

El cliente de consola es la forma más adecuada de utilizar la herramienta para el administrador, ya que incluye opciones como el envío consecutivo de varios paquetes de aplicaciones con una sola llamada, y la subida de *properties* que podrán ser utilizadas posteriormente en la validación. Además posee la misma funcionalidad que el cliente web.

Los dos tipos de funcionamiento del cliente son:

- **Validación:** Permite la validación de una o varias aplicaciones, y para su utilización se necesita pasar unos argumentos como se describe a continuación:

    1. **ID:** Es un identificador.

2. **Dirección de correo:** Para que una vez validado, se envíe el resultado.
3. **IP del servidor:** Para conectar con el sistema donde esté alojada la aplicación.
4. **Cadena con los datos de configuración (urlencoded):** `modo=xxx&propInternas=xxx&choicegen=xxx&umbral=xxx&max=xxx&min=xxx&confAdicional=xxx`
5. **Directorio/Archivo comprimido:** Se puede indicar un directorio con varias aplicaciones o un archivo comprimido con una aplicación para validar.

- **Gestión de archivos de propiedades (*properties*):** Permitirá subir y eliminar *properties*, además de ver el directorio con todos los archivos subidos. Para este funcionamiento se le pasan estos argumentos:

    1. **ID:** Es un identificador.
    2. **IP del servidor:** Para conectar con el sistema donde esté alojada la aplicación.
    3. **Opción -propfiles:** Para indicar que el modo de funcionamiento es de gestión de properties.
    4. **Archivo comprimido, opción de vista de directorio (-v) u opción de eliminado de archivo:** Se puede indicar un directorio con varias aplicaciones o un archivo comprimido con una aplicación para validar.

En caso de opción de eliminación de archivo:

5. **Nombre del archivo a eliminar.**

En la Sección B.6 del Apéndice se detalla la implementación de este módulo.

6.3 Gestor de peticiones

El módulo de gestión de componentes está formado por varios componentes que cumplen la función de manejar tanto los datos y archivos recibidos como la entrega de éstos datos y archivos a los módulos de compilación y validación.

6.3.1 Servlet de recepción

El *servlet* de recepción maneja los datos recibidos tanto por el cliente de consola como del cliente web.

En el caso de la validación, los archivos que recibe son enviados al módulo de extracción de archivos y una vez hecho esto, con los datos de configuración y la ubicación de los archivos extraídos se forma un mensaje con la tecnología JMS que se enviará al componente de gestión de accesos o de encolado.

6.3.2 Gestor de accesos

Como la validación es un procedimiento que consume bastantes recursos, se necesita controlar el número de aplicaciones que validan a la vez sin perder las peticiones que se reciban en el caso de que el servidor llegue al límite de validaciones simultáneas. Éste componente se dedica a gestionar las peticiones que le llegan y almacenarlas hasta que puedan ser compiladas y validadas.

La plataforma Java EE permite mantener una cola utilizando un tipo específico de EJB, el EJB de mensajes. Este EJB utiliza la tecnología JMS.

Nuestro componente, implementado como un EJB de mensajes, obtiene el mensaje enviado por el *servlet* de recepción vía JMS y lanza una llamada a la compilación y a la validación cuando están disponibles. En la Figura 29 se muestra un esquema básico de funcionamiento de la cola de mensajes.

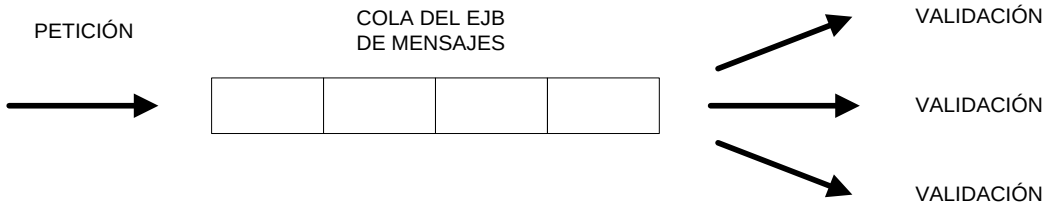

Figura 29. Esquema de funcionamiento de la cola de mensajes JMS

En la Sección B.1 del Apéndice se detalla la implementación de la conexión JMS.

La parte de la gestión del número de aplicaciones simultáneas que se pueden validar se realiza mediante la configuración del módulo de la validación que se describe más adelante.

6.4 Extractor de archivos

Este módulo recibe del *servlet* de recepción los archivos comprimidos y los descomprime en el directorio indicado. Puede descomprimir tanto archivos *.zip* como con *.jar*.

Están implementadas como dos clases Java normales a las que accede el *servlet* de recepción mediante los métodos que se muestran en la Figura 30 y la Figura 31.

En la Sección B.2 del Apéndice se detalla la implementación de este módulo.

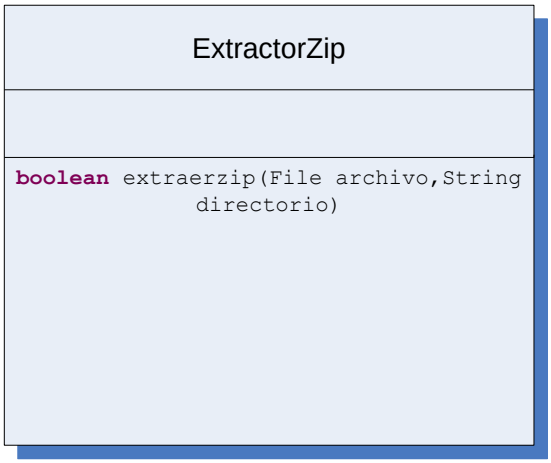

Figura 30. Clase `ExtractorZip`

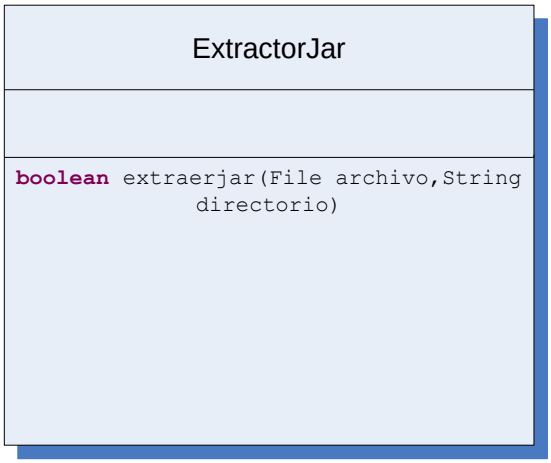

Figura 31. Clase `ExtractorJar`

6.5 Compilador

La aplicación JPF necesita tener el código fuente accesible para indicar las líneas de código donde se da el error, por lo que los archivos que se envíen deben ser *.java*. Sin embargo, para analizarlos utiliza únicamente *.class*, por lo tanto es necesario tener un componente que compile la aplicación. Este módulo además gestiona la jerarquía de carpetas en las que se ubicarán los archivos compilados.

Este módulo está implementado como una clase Java normal a la que se accede con los métodos mostrados en la Figura 32.

En la Sección B.3 del Apéndice se detalla la implementación de este módulo.

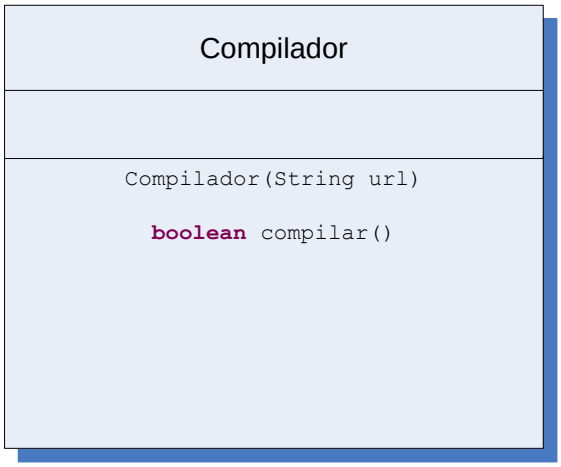

Figura 32. Clase `Compilador`

6.6 Validador

Este módulo se encargará de la validación de la aplicación enviada utilizando para ello la herramienta JPF. Este componente es el núcleo de toda la aplicación.

Como se vio en la sección de la gestión de peticiones, se debe limitar el número de aplicaciones que se validan simultáneamente. Para ello, se implementa el módulo utilizando la tecnología Java EE y, más concretamente, un EJB de sesión sin estado. Ésto permite tener un número de instancias idénticas disponibles que recibirán las peticiones del gestor de conexiones. Ese número de instancias será el mismo número de aplicaciones que se podrán validar simultáneamente y será posible configurarlo en el servidor de aplicaciones.

El módulo de validación se compone de dos clases:

1. **Clase `Ejecutor.java`**: Es una clase Java normal que contiene el código de la llamada a la herramienta JPF con la configuración indicada. Además, utiliza el módulo de gestión de correos para enviar el resultado de la validación. Su interfaz se muestra en la Figura 33.

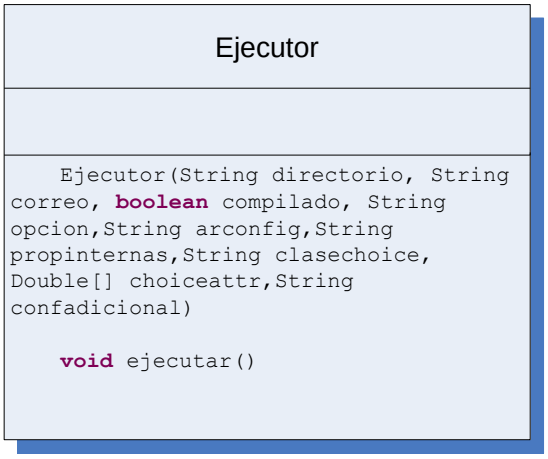

Figura 33. Clase `Ejecutor`

2. **Clase Validador**`.java`: Es la clase que implementa el EJB de sesión sin estado. Realiza la llamada al compilador y a la clase `Ejecutor.java`. Su interfaz se muestra en la Figura 34.

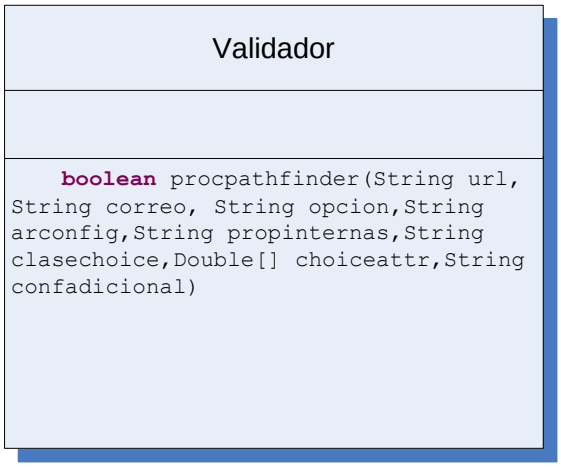

Figura 34. Clase `Validador`

Una vez que se libera una instancia del EJB de sesión sin estado (`Validador.java`), el EJB de mensajes del gestor de peticiones toma de la cola la siguiente petición y se lo envía, utilizando para ello su interfaz de acceso.

En la Sección B.4 del Apéndice se detalla la implementación de este módulo.

6.7 Gestor de correos

Para informar al usuario de la aplicación sobre el proceso de validación se utiliza el envío de correos electrónicos, tanto para indicar la finalización del proceso con el resultado de JPF, como para notificar en ciertas partes del proceso indicando el fallo o el funcionamiento correcto.

Este módulo se compone de una clase, llamada `Cartero.java`, que únicamente contiene el código que facilita el envío de correos por parte de otros módulos. Utiliza la librería `javax.mail` de Java y la interfaz que ofrece se muestra en la Figura 35.

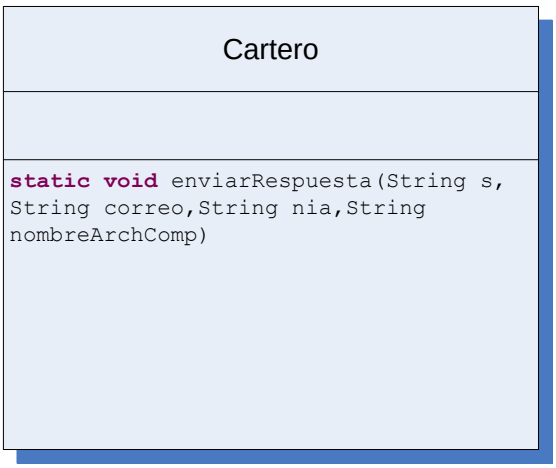

Figura 35. Clase `Cartero`

En la Sección B.5 del Apéndice se detalla la implementación de este módulo.

6.8 Conclusiones

Como resultado de los diferentes componentes desarrollados, descritos en las secciones anteriores, se tiene un módulo que cumple con los objetivos del proyecto. Dicho módulo provee un acceso web y un acceso vía consola, ambos configurables, que admiten el envío de prácticas. Una vez recibidas las prácticas, se extraen, se compilan y se validan, utilizando como núcleo JPF, para finalmente devolver por correo el resultado.

CAPITULO 7. Validación empírica

En este capítulo se analizará si la aplicación empresarial desarrollada, que integra JPF con la tecnología Java EE, es viable. Para ello se dispondrán unos escenarios de pruebas y se tomarán datos útiles para conocer los límites del software desarrollado.

7.1 Estado actual de la implementación

La aplicación está implementada y montada sobre un servidor Glassfish. Las peticiones, ya sean desde el cliente de consola o desde la página web, las recibe un *servlet*, el cual envía las peticiones a una cola JMS. Un MDB va atendiendo las peticiones de la cola y enviándolas a un Session Bean sin estado, el cual se encarga de hacer llamadas a clases controlan la extracción, compilación, ejecución con JPF y del posterior envío del correo con el resultado. El número de aplicaciones simultáneas que atiende el MDB inicialmente se ha configurado a cinco, pero será necesario hacer pruebas para ver cuál es el número de validaciones simultáneas más adecuado.

7.2 Escenarios de prueba

Para comprobar la viabilidad de nuestro proyecto, será necesario estresar la aplicación para comprobar tiempos de resolución, número de validaciones simultáneas posibles y carga de CPU.

Para todo ello se han dispuesto tres escenarios de pruebas:

1. Validación de una sola aplicación.
2. Validación de un número alto de peticiones.
3. Validación de peticiones simultáneas en el servidor.

Para los tres escenarios se tomará en cuenta únicamente el tiempo de validación de la herramienta JPF, ya que es la que introduce el mayor retardo. Por lo tanto el tiempo en el resto de procesos que lleva a cabo el servidor, como el tiempo de retardo de la petición en la red, se tomará como despreciable. Además, sólo se tendrán en cuenta las aplicaciones correctas ya que si no pasan la validación se detecta de forma inmediata.

Todas las pruebas de la aplicación se han realizado con un Intel Core i7 a 1,60 GHz y 4 GB de memoria RAM.

7.2.1 Validación de una sola aplicación

El tiempo de validación de una aplicación dependerá del número de estados a analizar. En el caso de que la aplicación genere pocos estados, la validación podría terminar en muy poco tiempo. JPF establece un límite de memoria asignada, ya que la aplicación podría proporcionar un número muy grande de estados o incluso infinitos, lo cual provocaría que la exploración tardara demasiado o que no parase nunca. Si se llega a este límite (lo más normal es que se llegue), JPF pararía la ejecución y si no ha encontrado ningún fallo, la considera validada

indicando que no contiene errores, e indicando también que el motivo de la parada de la búsqueda es que se ha llegado al límite (*Search constraint: Search.min_free= XXXXXX*).

El límite de memoria asignado por defecto a JPF es de 1024 MB. Este valor puede ser modificado en caso de que nuestro equipo no lo soporte (valores de 512 o 256 MB) o si se tiene un equipo de mayor capacidad (2048 MB). En las pruebas de este proyecto se ha mantenido el valor por defecto.

7.2.1.1 Tipos

El tiempo que tarda en llegar al límite dependerá de si se utiliza el API de `java.util.concurrent` o no (además de las características del equipo).

7.2.1.1.1 Aplicación que utiliza el API de `java.util.concurrent`

Si lo utiliza dicho API, la aplicación llega al límite analizando un número menor de estados, pues JPF incluye un complemento para dichas aplicaciones, que trata las fugas de memoria.

La aplicación para comprobar esto será una aplicación con dos variables compartidas a la que se le irá cambiando el número de hilos para ver las diferencias de tiempo. El código de la aplicación se encuentra en la Sección C.1 del Apéndice. Esta aplicación ha sido utilizada en prácticas de la asignatura *Sistemas concurrentes*[1].

Esta aplicación se compone de 4 tipos de hilos y 2 variables compartidas:

- **main**: Es el hilo encargado de inicializar el sistema. Hay 1 hilo de este tipo. El funcionamiento de este hilo es tal y como sigue: primero se crea un objeto de la clase `Santa`, al ser un hilo se lanza su ejecución; posteriormente se crean hilos de tipo `Reno` e hilos de tipo `Duende` en distintos bucles y se lanza su ejecución.

- **Santa**: Hay 1 solo hilo de este tipo. Sus principales funciones son repartir regalos, arreglar problemas y dormir, en ese orden de preferencias. En un principio el hilo está dormido a la espera de que aparezcan los renos o se hayan producido al menos 3 fallos en la cadena de producción. En el primer caso, se simula el reparto de los regalos mediante una espera de 100 ms. y se notifica a los renos su permiso para irse de vacaciones. En el segundo caso, se corrigen todos los fallos que hayan surgido hasta el momento y se notifica a los respectivos duendes para que prosigan su producción. Si se producen ambos casos, primero se reparten primero los regalos y no se permite a los duendes despertar a `Santa`, pues ya está despierto.

- **Reno**: Sus principales funciones son irse de vacaciones y ayudar a Santa a repartir los regalos. Cuando vuelven todos, el último notifica a Santa y pasan todos a esperar el permiso para irse de vacaciones. Reciben este permiso cuando Santa acaba de repartir los regalos. Al llegar cada reno incrementa una variable compartida por todos, y la decrementa cuando se va. El número de hilos `Reno` se variará para la prueba.

[1] http://www.it.uc3m.es/tsc/

- **Duende**: Su principal función es producir juguetes, aunque en este caso se producen errores temporalmente por lo que deben notificar a Santa para que les corrija. Cada hilo empieza simulando el tiempo que tarda en producirse un fallo (2 s.), incrementa una variable compartida (contadorDUENDES) y comprueba que al menos hay 3 fallos y que Santa está dormido para despertarlo. Se encarga Santa de decrementar la variable compartida. Cada duende espera el permiso de Santa para seguir su proceso de producción. El número de hilos `Duende` se variará para la prueba.

Las variables compartidas son:

- **contadorRENOS**: Esta variable es de tipo entero, compartida por los renos (hilos reno1, reno2, reno3). Todos ellos la incrementan cuando llegan de vacaciones a repartir los regalos y la decrementan cuando terminan y se van de nuevo.

- **contadorDUENDES**: Esta variable es de tipo entero, compartido por los duendes (hilos duende1, duende2, duende3) y el hilo Santa. Los duendes la incrementan cuando les surge algún error en la producción y santa la decrementa cuando corrige el error y lo notifica al correspondiente duende.

El resultado obtenido, variando el número de hilos, se muestra en la Tabla 2.

Nº de Hilos (2 variables compartidas)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
5	0:03:02	16404
10	0:01:13	9919
15	0:01:01	8320
20	0:00:53	6739
25	0:00:51	5650
30	0:00:53	5655
35	0:00:53	5658
40	0:00:54	5666
45	0:00:53	5671
50	0:00:54	5674

Tabla 2. Prueba Nº de hilos/Tiempo/Nº estados (con `java.util.concurrent`)

Como se puede observar en la Tabla 2, al ir aumentando el número de hilos, se aumenta la velocidad con que se consume memoria y por lo tanto se reduce el tiempo de validación y el número de estados que son analizados.

En la Tabla 3 se ve el resultado obtenido al aumentar el número de variables compartidas.

Nº de Variables (con 5 Hilos)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
2	0:03:02	16404
4	0:07:07	17161
6	0:11:01	20481
8	0:16:19	23765
10	0:23:27	27011
12	0:28:46	30225

Tabla 3. Prueba Nº de variables/Tiempo/Nº estados (con `java.util.concurrent`; 1 semáforo para todas las variables añadidas)

Se ve que se incrementa el tiempo en llegar al límite porque el aumento de variables compartidas hace que se reduzca la velocidad con que se consume la memoria para un número de hilos constante. Al reducir la velocidad de consumo de memoria se permite el análisis de un mayor número de estados.

Algo que hace que se aumente más el tiempo es el número de semáforos (`java.util.Semaphore`) por acceso a variable. En la Tabla 3 se ha utilizado un semáforo para las variables añadidas, es decir, en el caso de 4 variables, se han añadido 2 variables, y para su acceso se ha utilizado un semáforo.

En la Tabla 4 se muestra el resultado obtenido utilizando dos semáforos. Se puede ver el efecto que tiene en el tiempo de alcance del límite.

Nº de Variables (con 5 Hilos)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
2	0:03:02	16404
4	0:04:24	15317
6	0:04:57	16563
8	0:05:17	17701
10	0:05:47	18748
12	0:05:57	19707

Tabla 4. Prueba Nº de variables/Tiempo/Nº estados (con `java.util.concurrent` y 2 semáforos para todas las variables añadidas)

Se puede observar que simplemente añadiendo un semáforo más para las variables añadidas, el tiempo en llegar al límite se reduce bastante, pero sigue teniendo tendencia creciente. Si se prueba a utilizar un semáforo por cada variable añadida el resultado que se obtiene es el que se muestra en la Tabla 5.

Nº de Variables (con 5 Hilos)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
2	0:03:02	16404
4	0:04:20	14874
6	0:04:44	16333
8	0:05:28	17514
10	0:05:29	18415
12	0:05:37	19697

Tabla 5. Prueba Nº de variables/Tiempo/Nº estados de (con `java.util.concurrent` y 1 semáforo por variable añadida)

Con un semáforo por variable también hay tendencia creciente en el tiempo, con valores similares a los obtenidos en la Tabla 5. Lo mismo sucede con el número de estados.

7.2.1.1.2 Aplicación que no utiliza el API de `java.util.concurrent`

Se analiza ahora el caso de que la aplicación no utilice el api de `java.util.concurrent`. La aplicación utilizada para éste propósito modela la gestión de una cuenta bancaria. Su código se puede ver en la Sección C.2 del Apéndice.

Esta aplicación se compone de 3 tipos de hilos y 1 variable compartida:

- **main**: Es el hilo encargado de inicializar el sistema. Hay 1 hilo de este tipo. El funcionamiento de este hilo es tal y como sigue: primero se crea un objeto de la clase `CuentaCorriente` que es la que contiene la variable compartida y cuya instancia se utilizará de cerrojo por el resto de hilos. Posteriormente se crean hilos de tipo `GastadorH` y `AhorradorH` y se lanza su ejecución.

- **AhorradorH**: Sus única función es aumentar la variable compartida (saldo) en 1000 unidades. Se variará el número de hilos de este tipo para la prueba.

- **GastadorH**: Su función es decrementar la variable compartida a la mitad en caso de que su valor sea mayor que 2. El número de hilos `GastadorH` se variará para la prueba.

Las variables compartidas son:

- **saldo**: Esta variable es de tipo entero, compartida por los hilos `GastadorH` y `AhorradorH`. Se incrementará el número de variables compartidas de este tipo para las pruebas.

El resultado obtenido variando el número de hilos (nº hilos de `AhorradorH` + nº hilos de `GastadorH`) se muestra en la Tabla 6.

Nº de Hilos (2 variables compartidas)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
5	0:04:24	377947
10	0:04:14	343946
15	0:04:09	312730
20	0:04:02	286772
25	0:04:01	263731
30	0:03:57	244718
35	0:03:58	227501
40	0:03:48	213006
45	0:03:47	199747
50	0:03:47	188809

Tabla 6. Prueba -Nº de hilos/Tiempo/Nº estados- de aplicaciones (sin `java.util.concurrent`)

Como se puede observar, el tiempo para alcanzar el límite al ir aumentando el número de hilos desciende. Lo que hace que disminuya el número de estados analizados. Esto supone que el aumento del número de hilos hace que se consuma memoria más rápido, como sucedía en caso de utilizar el API de `java.util.concurrent`, y que la memoria que se puede dedicar al análisis del número de estados disminuye, y como consecuencia, disminuye el número de estados analizados.

Si se aumenta el número de variables compartidas, el resultado obtenido es el mostrado en la Tabla 7.

Nº de Variables (con 5 Hilos)	Tiempo en llegar al límite (1024MB) (h:mm:ss)	Nº de estados
2	0:04:16	377947
4	0:04:12	364919
6	0:04:10	352086

8	0:04:10	340123
10	0:04:08	328947
12	0:04:07	319051

Tabla 7. Prueba Nº de variables/Tiempo/Nº estados (sin `java.util.concurrent`)

Se observa que el tiempo en llegar al límite se mantiene para esta aplicación, pero que al aumentar el número de variables compartidas, disminuye la memoria utilizada para el análisis de estados de forma más suave que en el caso del aumento del número de hilos.

7.2.1.2 Comparación

Para comparar los distintos tipos primero se verá la diferencia de tiempos de validación según el número de hilos, utilizando para ello el gráfico mostrada en la Figura 36.

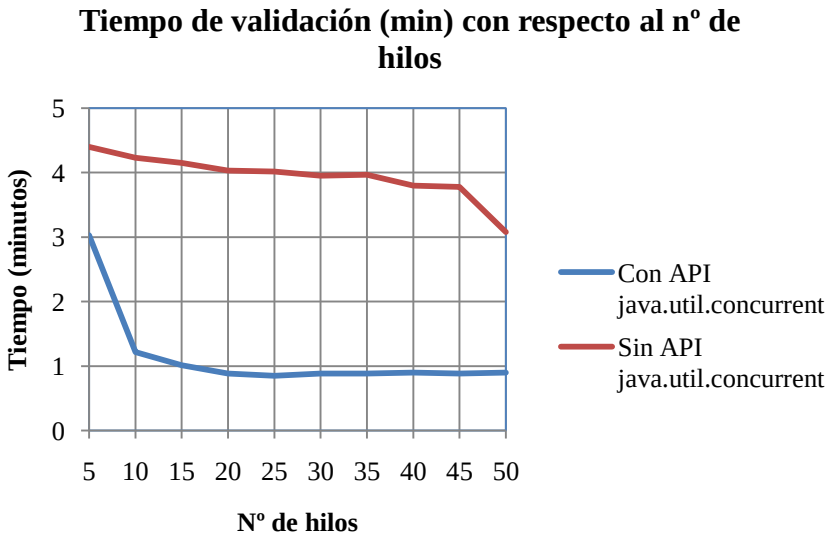

Figura 36. Gráfico Tiempo de validación/Nº de hilos

En el gráfico de la Figura 36, se observa que utilizando el API de concurrencia, a mayor número de hilos, menor tiempo en llegar al límite de memoria, pero no es una disminución muy rápida (excepto en el paso de 45 a 50 hilos). En el caso en que se no se utiliza el API de concurrencia, sucede que el tiempo en llegar al límite desciende muy rápidamente conforme se aumenta el número de hilos. Esto sucede aproximadamente hasta la prueba con 25 hilos, a partir de ese momento, se mantiene más o menos constante. Además los tiempos de la que utiliza API de concurrencia son mucho menores aún utilizando el mismo número de hilos y variables compartidas (dos variables).

Con respecto al número de estados analizados, con el API de concurrencia se analiza una cantidad significativamente menor de estados, como se puede ver en el gráfico mostrada en la Figura 37. Esto es debido que se consume memoria más rápidamente.

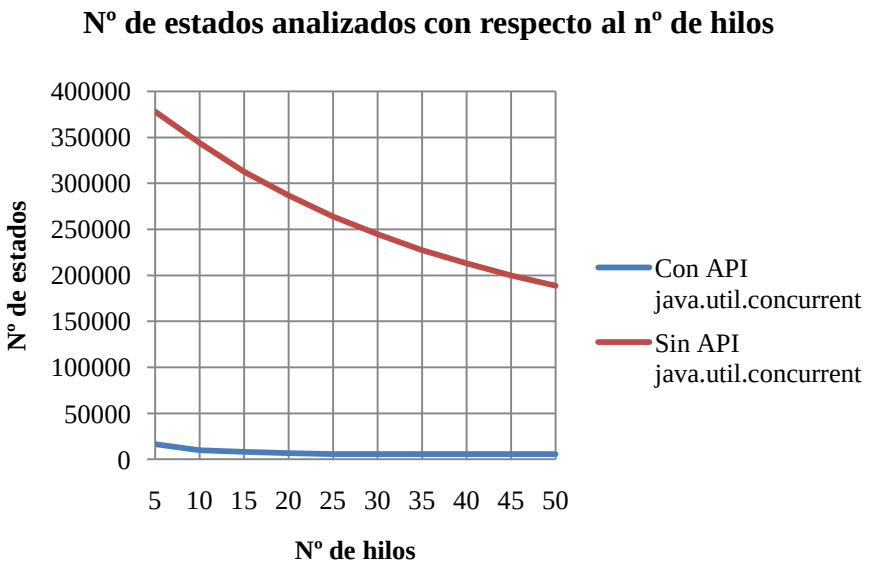

Figura 37. Gráfico Nº de estados/Nº de hilos

En la Figura 38 se muestra un gráfico en la que se puede ver el tiempo para alcanzar el límite de tres aplicaciones que utilizan el API de concurrencia, con distintas configuraciones en número de semáforos Java, y el tiempo de una aplicación que no utiliza dicho API.

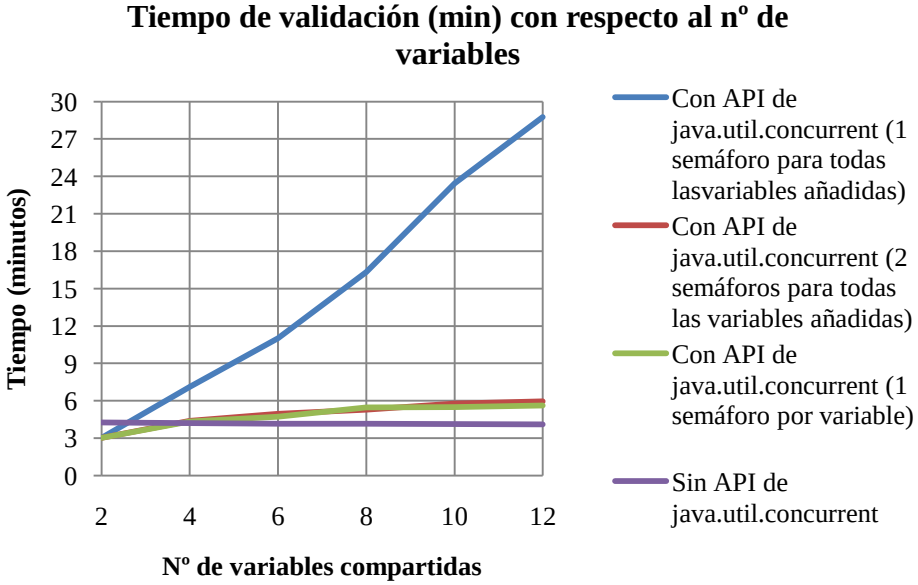

Figura 38. Gráfico Tiempo de validación/Nº de variables

Primero se puede observar que entre las tres aplicaciones que utilizan el API hay una que va aumentando el tiempo que necesita para alcanzar el límite, con respecto al número de variables compartidas utilizadas, de una forma muy notable. Este caso es en el que se utiliza un único semáforo para todas las variables añadidas. Dicho incremento de tiempo no es muy aceptable, ya que podría incrementar demasiado el tiempo de su estancia en el servidor, influyendo negativamente en la cola de peticiones utilizada en el proyecto.

Utilizando un semáforo más ya se consigue reducir el crecimiento del tiempo con respecto al número de hilos. Si se aumenta el número de semáforos hasta obtener un semáforo por variable, no se nota demasiada diferencia en el incremento de tiempo con respecto al caso que utiliza dos semáforos.

Aparte de esto, suponiendo que la configuración de las aplicaciones con API de concurrencia se realiza de forma que se reduce al máximo el crecimiento del tiempo, la principal diferencia con las aplicaciones que no utilizan dicho API es que, éstas últimas, con el aumento del número de variables decrecen en el tiempo, aunque de forma muy suave.

Con los mismos datos que el gráfico de la Figura 38 pero analizando los estados en vez del tiempo, resulta el gráfico mostrado en la Figura 39.

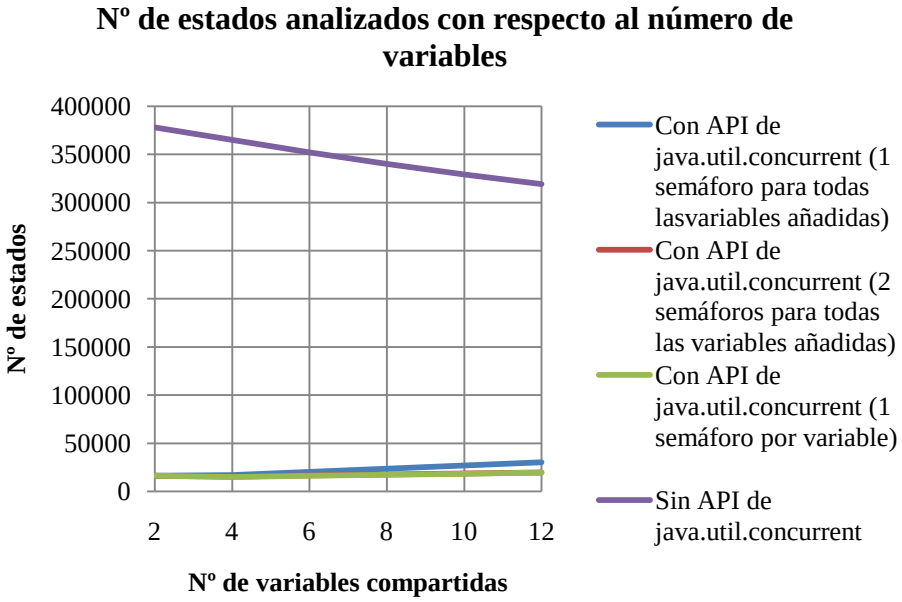

Figura 39. Gráfico Nº de estados/Nº de variables

En esta gráfica se puede observar que con respecto al número de variables compartidas de las aplicaciones, se analizan muchos más estados en las que no utilizan el API de concurrencia. El otro tipo de aplicaciones, sin tener en cuenta el número de semáforos con respecto a las variables añadidas, mantienen aproximadamente el mismo número de estados por número de variables compartidas.

En conclusión para la eficiencia de la aplicación empresarial del proyecto, el mejor caso sería validar aplicaciones sin API de concurrencia, pues se analizan muchos más estados y el incremento en el tiempo no es demasiado importante (teniendo en cuenta que 4 minutos no es demasiado para la aplicación JPF).

7.2.2 Validación simultánea de aplicaciones

Al validar varias aplicaciones a la vez se puede dilatar el tiempo de cada validación independiente si se encuentran en el servidor, es decir, si se realizan simultáneamente. Las razones de esto es que los recursos del sistema se deben dividir entre las diferentes validaciones.

En cualquier caso, esta diferencia variará dependiendo del número de aplicaciones simultáneas que se permitan. Para probar la validación simultánea se utilizarán aplicaciones que consuman memoria rápidamente, de forma que la CPU llegue a su grado mayor de funcionamiento y que las aplicaciones, durante un pequeño intervalo de tiempo, consuman el máximo de memoria permitido a la vez.

Para este escenario se utilizará la aplicación utilizada en la sección 7.2.1.1.2 que modela la cuenta de un banco con la configuración de 5 hilos y 2 variables compartidas. El resultado de esto se muestra en la Tabla 8.

Nº de aplicaciones simultáneas	% Memoria máximo	% CPU máximo	Media de incremento de tiempo (+h:mm:ss)
2	76%	28%	+0:00:54
3	99%	40%	+0:01:53
4	99%	51%	+0:05:08
5	99%	65%	+0:09:34
6	99%	75%	+0:20:18

Tabla 8. Tabla de consumo de recursos con respecto al nº de aplicaciones simultáneas validando

Para seleccionar la mejor opción, se realiza un cálculo suponiendo un envío de 30 peticiones. Con esto se puede calcular el tiempo teórico de la validación de esas 30 peticiones teniendo en cuenta el número de aplicaciones simultáneas a validar. El resultado de esta prueba se muestra en la Tabla 9.

Nº de aplicaciones simultáneas	Tiempo total teórico de validación de 30 peticiones (h:mm:ss). Resultado de: *(Nº de peticiones / Nº de aplicaciones simultáneas) * (Tiempo de validación de la aplicación + Media del incremento de tiempo)*
2	1:17:30
3	1:00:09
4	1:10:30
5	1:23:00
6	2:02:50

Tabla 9. Tiempo teórico de validación de treinta peticiones con respecto al nº de aplicaciones simultáneas

Se eligen tres validaciones simultáneas ya que, según el cálculo teórico, sería la opción que reduce el tiempo de validación del total de las peticiones.

7.2.3 Validación de un número alto de peticiones

Una vez configurado el servidor para admitir únicamente tres aplicaciones simultáneas, se procede a hacer una prueba enviando peticiones a la vez para ver cuánto tiempo tarda. El resultado obtenido se muestra en la Tabla 10.

Nº de peticiones	Tiempo (h:mm:ss)
5	0:14:00
10	0:26:00

15	0:33:00
20	0:44:00
25	0:56:00
30	1:09:00

Tabla 10. Tiempo de validación total de peticiones

Se observa que el tiempo en validar 30 peticiones se aproxima al calculado. En la Figura 40 se puede observar gráficamente el resultado:

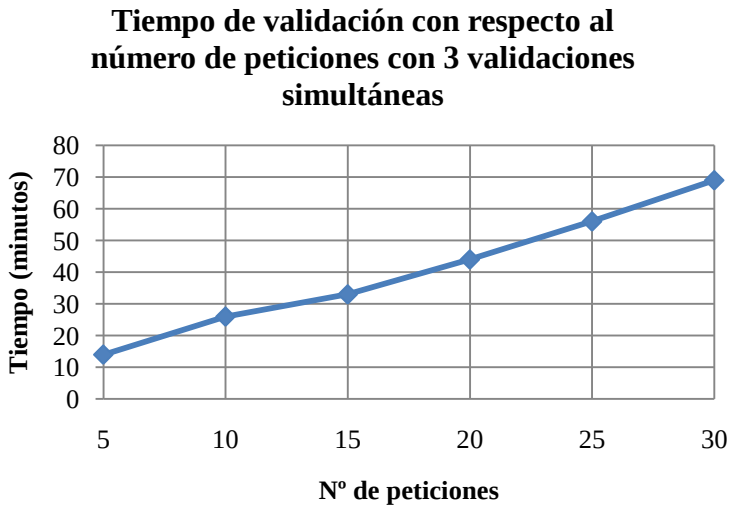

Figura 40. Gráfico Tiempo de validación/Nº de peticiones con 3 validaciones simultáneas

7.2.4 Validación de entrega de prácticas real

Finalmente, se probará una entrega de prácticas real, con 25 prácticas de la asignatura de *Sistemas concurrentes*. Como las entregas son en parejas, el número de alumnos es de 50. La aplicacion es la misma utilizada en la prueba de aplicación con api de `java.util.concurrent` de la sección 7.2.1.1.1 , pero realizadas por alumnos de la universidad. En la Tabla 11 se puede observar el tiempo de resolución, cuántas validan y cuantas no.

Entrega	Validación	Motivo	Tiempo	Calificación
1	No Ok	Deadlock	0:00:01	8
2	No Ok	Condición de carrera	0:00:01	7
3	No Ok	Deadlock	0:00:01	9,5
4	No Ok	Deadlock	0:00:01	8
5	Ok	-	0:01:07	8,5
6	No Ok	Deadlock	0:00:01	6
7	No Ok	Condición de carrera	0:00:01	7,5
8	Ok	-	0:01:40	8,5
9	Ok	-	0:04:57	9
10	No Ok	Condición de carrera	0:01:36	6,5
11	No Ok	Condición de carrera	0:00:03	7
12	No Ok	Condición de carrera	0:00:01	7,5
13	No Ok	Deadlock	0:00:09	8
14	Ok	-	0:00:48	9

15	No Ok	Deadlock	0:00:01	10
16	No Ok	Deadlock	0:00:01	7
17	Ok	-	0:06:15	7
18	No Ok	Deadlock	0:00:06	7
19	No Ok	Deadlock	0:00:01	9
20	No Ok	Deadlock	0:19:07	8
21	Ok	-	0:16:31	7
22	Ok	-	0:01:55	9
23	No Ok	Condición de carrera	0:00:02	5
24	No Ok	Deadlock	0:00:02	8
25	Ok	-	0:32:31	7,5

Tabla 11. Validación de una entrega real de prácticas

El tiempo total para validar todas las prácticas es de: 59 minutos. Con respecto a la relación del resultado de la validación con la nota obtenida, hay que tener en cuenta que en la corrección de las prácticas se valoraba, junto con la práctica, una memoria de la misma. Además, los *deadlocks* son difíciles de detectar y ciertos fallos que generan las condiciones de carrera pueden ser valorados de diferente manera, dependiendo de si son graves o no.

7.3 Conclusiones

De las pruebas realizadas se obtienen las siguientes conclusiones:

- Dependiendo del tipo de aplicación a validar, el tiempo variará dependiendo de la velocidad con la que consuma memoria ya que siempre finalizará al llegar al límite de memoria.
- El número de aplicaciones a validar simultáneamente en el servidor para que haya eficiencia es de tres.
- Es posible aceptar un número grande de peticiones, pero teniendo en cuenta que puede llevar bastante tiempo

El proyecto, por lo tanto, queda configurado para atender tres peticiones simultáneas, dejando el resto en espera en la cola JMS. El tiempo en resolver la validación de un número grande de peticiones es aceptable, teniendo en cuenta la duración de la validación de una petición independiente.

CAPITULO 8. Conclusiones y líneas futuras de trabajo

El objetivo del proyecto era la creación de un módulo empresarial Java EE para hacer accesible via web la herramienta de validación formal JPF. Para ello se comenzó con el estudio de las diferentes tecnologías implicadas: JPF, Java EE y el servidor Glassfish.

Del estudio de la herramienta JPF se han destacado dos funcionalidades muy interesantes. Una de esas funcionalidades son los *Listeners* que permiten centrarse en características específicas de las aplicaciones a validar. La otra funcionalidad destacada es la llamada *On-the-fly Partial Order Reduction* (POR), la cual permite analizar qué instrucciones son relevantes y cuales no a la hora de validar, lo que supone mayor eficiencia y menor número de estados a analizar.

Lo más útil para este proyecto de la tecnología Java EE es la tecnología JMS (*Java Messaging Service*), la cual permite controlar una cola de peticiones e ir atendiendo dichas peticiones de la forma que mas convenga mediante los EJBs de mensajes o MDBs (*Message Driven Beans*). También son muy útiles los mecanismos para manejar las peticiones inicialmente, es decir, cuando se accede al servidor, como son los *servlets* y las JSPs.

Del análisis del servidor de aplicaciones Glassfish se vió que era un servidor muy completo en funcionalidades, sencillo de configurar y de código libre. Por lo tanto, es un servidor en el que se puede desplegar el módulo empresarial.

Después del estudio de las diferentes tecnologías implicadas, se prosiguió con su completa implementación y con un análisis de rendimiento posterior, analizando tiempos de validación, número de estados, carga de CPU y consumo de memoria. Finalmente y teniendo en cuenta los resultados de dichas pruebas, se ha llegado a la conclusión de que el proyecto es viable y que se adapta completamente al entorno docente para el que ha sido ideado.

Como líneas futuras de trabajo, se podrán desarrollar ideas ya introducidas en este proyecto pero que se salen del marco en el que se ha encuadrado.

A nivel de la herramienta JPF, podría desarrollarse la utilización de *Choice Generators*, generadores de opciones de búsqueda propios de cada usuario, haciendo que de alguna manera ésta funcionalidad pudiera utilizarse vía web, dando un control más fino sobre el análisis o validación de las aplicaciones.

A más alto nivel, el siguiente paso de este proyecto sería aprovechar una de las características que ofrecen los servidores Glassfish, como es el balanceo de carga. Esta funcionalidad permitiría un número más alto de peticiones y un tiempo de resolución menor para el conjunto de estas peticiones, ya que se distribuirían en diferentes servidores.

APÉNDICE A. JPF: Aspectos técnico

En esta sección de los apéndices se profundiza más en ciertos aspectos interesantes de la herramienta JPF, y se organiza de esta manera:

- *Choice Generators* (Sección A.1)
- *Partial Order Reduction* (POR) (Sección A.2)
- *Listeners* (Sección A.3)
- *Model Java Interface* (MJI) (Sección A.4)
- *Bytecode Factories* (Sección A.5)
- Configuración básica (Sección A.6)
- Ejecución de JPF (Sección A.7)
- Salida de JPF (Sección A.8)
- API de Verify (Sección A.9)
- Extensiones (Sección A.10)
- Instalación de la herramienta (Sección A.11)

A.1 Choice Generators: Funcionamiento

Choice Generator es un mecanismo muy interesante de JPF por lo que en esta sección se va a detallar su funcionamiento.

A.1.1 Detalles de funcionamiento

En esta sección se asume que se está en una *Transition* que ejecuta una instrucción *get_field* (hay que recordar que JPF ejecuta código Java) y que el objeto correspondiente al que pertenece este campo está compartido entre hilos. Por razones de simplicidad, se asume además que no hay sincronización al acceder al objeto (o se ha configurado *vm.sync_detection* para que esté desactivado). Se asume también que hay otros hilos ejecutables en este momento. Entonces se tiene una opción (*Choice*). El resultado de la ejecución podría depender del orden en que se han programado los hilos, y por lo tanto acceder a este campo. Podría haber una condición de carrera en los datos. Para tener más claridad sobre el escenario planteado, en la Figura 41 se puede ver el funcionamiento de las transiciones entre estados

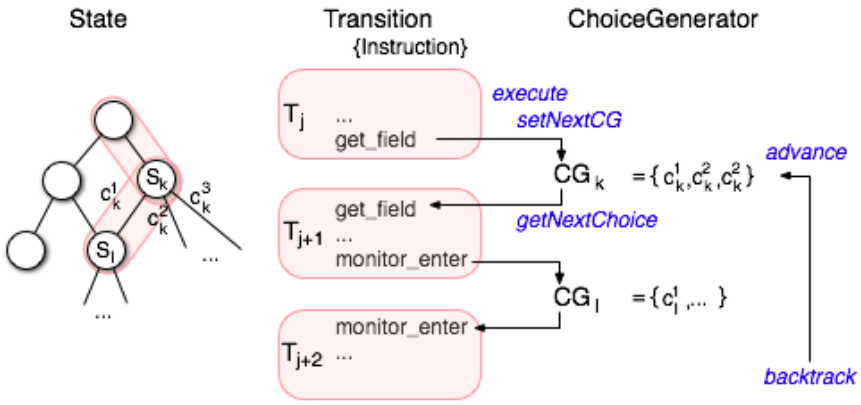

Figura 41. Transiciones *Choice Generator* (tomado de [1])

Como consecuencia, cuando JPF ejecuta esta instrucción *get_field*, el método `gov.nasa.jpf.jvm.bytecode.GET_FIELD.execute()` hace tres cosas:

1. Crea un nuevo `ChoiceGenerator` (`ThreadChoiceGenerator` en este caso) que tiene todos los hilos ejecutables en este momento como opciones posibles.
2. Registra este `ChoiceGenerator` llamando a `SystemState.setNextChoiceGenerator()`.
3. Se auto-programa para la re-ejecución (simplemente se devuelve a sí mismo como la nueva instrucción a ejecutar dentro del hilo que se está ejecutando actualmente).

En este punto, JPF termina esta *Transition* (que es básicamente un bucle dentro de `ThreadInfo.executeStep()`), almacena una instantánea de el *State* actual y después lanza la nueva *Transition* (se ignorará `Search` y la posible vuelta atrás por el momento). El `ChoiceGenerator` creado y registrado al final de la *Transition* anterior se convierte en el nuevo y actual `ChoiceGenerator`. Todo *State* tiene exactamente un objeto `ChoiceGenerator` asociado a él, y toda transición tiene exactamente una opción del `ChoiceGenerator` que la lanza. Toda *Transition* termina en una instrucción que produce el siguiente `ChoiceGenerator`.

La nueva *Transition* es lanzada por *SystemState* configurando el anteriormente registrado `ChoiceGenerator` como el actual y llamando a su método `ChoiceGenerator.advance()` para situarlo en su siguiente opción. Después `SystemState` comprueba si el `ChoiceGenerator` actual es un `SchedulingPoint` (un `ThreadChoiceGenerator` que tiene como objeto ser utilizado para propósitos de programación o planificación) y, si lo es, obtiene el siguiente hilo para ejecutar desde él. Luego comienza la siguiente *Transition* llamando `ThreadInfo.executeStep()` en él.

`ThreadInfo.executeStep()` básicamente se mantiene en un bucle hasta que un `Instruction.execute()` se devuelve a sí misma. Cuando un `ThreadInfo.executeStep()` posterior re-ejecuta ésta instrucción, la instrucción indica que es la primera instrucción en una nueva *Transition*, y por lo tanto no tiene que crear un `ChoiceGenerator` sino que procede con sus operaciones normales.

Si no hay siguiente instrucción, `Search` determina que el estado se ha visto antes y la VM vuelve atrás. `SystemState` vuelve al antiguo estado, y comprueba opciones que no hayan sido

exploradas todavía de su `ChoiceGenerator` asociado llamando a `ChoiceGenerator.hasMoreChoices()`. Si hay más opciones, sitúa el `ChoiceGenerator` en la siguiente llamando a `ChoiceGenerator.advance()`. Si todas las opciones han sido procesadas, el sistema vuelve atrás otra vez (hasta que su primer `ChoiceGenerator` acabe y terminaría la búsqueda). En la Figura 42 se muestra un esquema detallado con este funcionamiento.

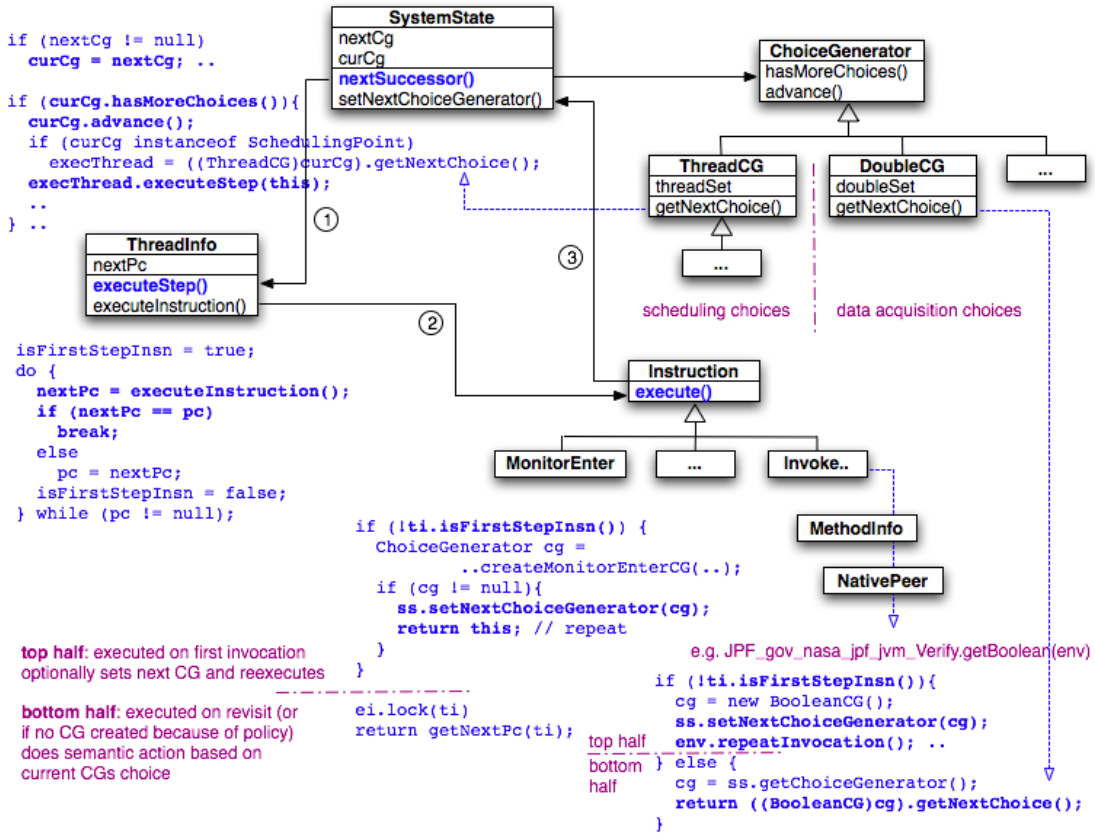

Figura 42. Funcionamiento detallado de *Choice Generator* (tomado de [1])

Los métodos que crean `ChoiceGenerator` tienen una estructura particular que divide su cuerpo en dos partes:

- ***Top half* (Mitad superior)** – (potencialmente) crea y registra un nuevo `ChoiceGenerator`.
- ***Bottom half* (Mitad inferior)** – El que hace el trabajo real que podría depender de la adquisición de un nuevo valor.

La única diferencia entre opciones de programación (o planificación) y opciones de adquisición de datos es que las primeras se gestionan internamente por la JVM (específicamente utilizado por `SystemState` para determinar el siguiente hilo a ejecutar) y la adquisición de datos es gestionada por la mitad inferior (*Bottom half*) de un `Instruction.execute()`, un método nativo, o un método de *callback* de un *Listener* (en cuyo caso debe obtener el `ChoiceGenerator` actual de `SystemState`, y después llamar explícitamente a `ChoiceGenerator.getNextChoice()` para obtener el valor de la opción).

Como detalle de implementación, la creación de `SchedulingPoints` se delega a un objeto `SchedulerFactory`, que encapsula una política de programación proveyendo un conjunto consistente de `ThreadChoiceGenerators` para el numero fijado de instrucciones, que son relevantes para la programación (*monitor_enter*, llamadas a métodos sincronizadas o `Object.wait()`). Los clientes de esta `SchedulerFactory`, por lo tanto, deben ser conscientes de que el objeto de esta directiva podría no devolver un nuevo `ChoiceGenerator`, en cuyo caso el cliente directamente procede con la mitad inferior de la ejecución, y no rompe la actual *Transition*.

Las clases estándar e interfaces para el mecanismo *Choice Generator* se pueden encontrar en el paquete `gov.nasa.jpf.jvm` e incluyen:

- *ChoiceGenerator*
- *BooleanChoiceGenerator*
- *IntChoiceGenerator*
- *DoublechoiceGenerator*
- *ThreadChoiceGenerator*
- *SchedulingPoint*
- *SchedulerFactory*
- *DefaultSchedulerFactory*

Implementaciones concretas se pueden encontrar en el paquete `gov.nasa.jpf.jvm.choice` que incluye clases como:

- *IntIntervalGenerator*
- *IntChoiceFromSet*
- *DoubleChoiceFromSet*
- *DoubleThresholdGenerator*
- *SchedulingChoiceFromSet*

A.2 POR: Detalles de funcionamiento

POR es una característica muy interesante y útil para el proyecto por lo que se analiza en profundidad en esta sección del apéndice.

El esfuerzo principal para el soporte de POR de JPF va enfocado a extender su marca y barrido de colector. La alcanzabilidad de POR es un subconjunto de la alcanzabilidad del colector, por lo tanto el mecanismo *piggybacks* en la fase transversal de marca de objetos. Es complicado por el hecho de que ciertos enlaces de referencia existen solo en la capa de implementación (oculta) de la VM. Por ejemplo, todo hilo tiene una referencia a su `ThreadGroup`, y éste a su vez tiene referencias a todos los hilos incluidos, por lo tanto – desde la perspectiva de un colector de basura – todos los hilos de un grupo son mutuamente alcanzables.

Si la aplicación bajo análisis no utiliza reflexión Java y consultas de tiempo de ejecución como la enumeración de hilos, la alcanzabilidad de POR debería seguir las reglas de la accesibilidad tan estrictamente como sea posible.

POR de JPF no soporta todavía modificadores de acceso protegidos y privados, incluye un mecanismo para especificar que ciertos campos no deberían ser usados para promover la alcanzabilidad de POR. Este atributo se configura mediante `Attributor` configurado en el tiempo de carga de clases.

Con este mecanismo, calcular la alcanzabilidad de POR se convierte en un acercamiento hacia adelante que se divide en dos fases. La fase 1 marca de forma no recursiva todos los objetos de la raíz (la mayoría campos estáticos, y pilas de hilos), guardando el identificador del hilo de referencia. En el caso de que un objeto sea alcanzable desde un campo estático, o por dos hilos, su estatus se cambia a compartido (*shared*). La fase 2 atraviesa recursivamente todos los objetos, propagando a cualquiera el estado compartido o el identificador del hilo de referencia a través de todos los campos de referencia que no estén marcados como cortafuegos de alcanzabilidad. En la Figura 43 se muestra gráficamente las fases descritas.

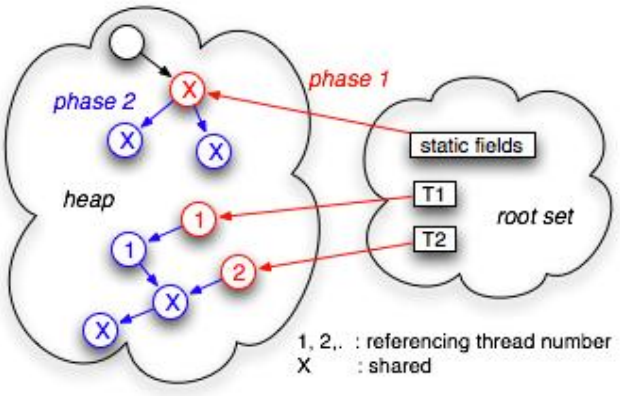

Figura 43. Fases de POR (tomado de [1])

Otra vez, si el transversal da con un objeto que ya está marcado como referenciado por otro hilo, cambia el estado del objeto a compartido, y desde ahí propaga dicho estado en vez del identificador del hilo.

Para mayor reducción de cambios de contexto irrelevantes, JPF puede comprobar la protección de bloqueo para determinar si el acceso a un campo es relevante para la programación (o planificación). Si la propiedad *vm.por.sync_detection* se establece a *true*, JPF busca candidatos potenciales de bloqueo cuando analiza las instrucciones *GET_x/SET_x*. La política de detección de candidatos de bloqueo es configurable con la propiedad *vm.por.fieldlockinfo.class*; la propiedad por defecto `gov.nasa.jpf.jvmStatisticFieldLockInfo` sólo aplaza la decisión guardando el número de bloqueos que, cuando se ejecuta la instrucción del campo, calcula la intersección del conjunto en posteriores accesos.

Si el conjunto no está vacío para un número configurable de accesos a campos, el campo se marca como protegido de bloqueo, y por tanto, no tratado como límite de transición. Si por el contrario el conjunto debía estar vacio, se mostrara una advertencia como la que aparece en la Figura 44.

```
    Warning: unprotected field access of: Event@70.count in thread: "Thread-1"
oldclassic.java:107
      sync-detection assumed to be protected by: Event@70
      found to be protected by: {}
      >>> re-run with 'vm.por.sync_detection=false' or exclude field from
checks <<<
```

Figura 44. Advertencia de campo desprotegido

Y el campo de acceso será otra vez tratado como un divisor de transición.

El conjunto de campos a ser analizados puede ser especificado mediante las propiedades *vm.por.include_fields* y *vm.por.exclude_fields*. Por defecto, no son analizados los campos de las clases de la librería del sistema (muy útil para evitar cambios de contexto para objetos globales como `System.out`).

Incluso con todas estas optimizaciones, algunos divisores de transiciones no deseados es probable que permanezcan. Esto es debido en su mayoría a dos limitaciones:

1. JPF solo considera alcanzabilidad y no accesibilidad.
2. Las condiciones escritas una vez y son leídas de forma múltiple, no pueden ser detectadas a priori por campos que no son *final*, o que no están dentro de objetos inmutables (`java.lang.String` por ejemplo).

Especialmente el último caso podría ser sujeto a nuevas mejoras.

A.3 Listeners: Tipos y configuración

En esta sección se muestran códigos de los tipos de *Listeners*.

A.3.1 SearchListener

Las instancias de `SearchListener` se utilizan para monitorizar el proceso de búsqueda del espacio de estados. Proveen de métodos de notificación para la mayoría de las acciones de `Search`.

```
public interface SearchListener extends JPFListener {

  /* Se obtiene esto después de entrar en el bucle de búsqueda, pero antes
de el primer paso */
  void searchStarted (Search search);

  /* Se obtiene el estado siguiente */
  void stateAdvanced (Search search);

  /* El estado está completamente explorada */
  void stateProcessed (Search search);

  /* El estado fue marcha atrás un paso */
  void stateBacktracked (Search search);

  /* Alguien almacenó el estado */
  void stateStored (Search search);

  /* Un estado generado previamente fue restaurado */
  void stateRestored (Search search);

  /* JPF encontró una violación de propiedades */
  void propertyViolated (Search search);

  /* Hubo un evento de restricción en la búsqueda, se vuelve atrás. Podría
haber sido convertido en una propiedad, pero normalmente es un atributo de la
búsqueda, no la aplicación*/
  void searchConstraintHit (Search search);
```

```
    /* Terminó, con o sin error previo */
    void searchFinished (Search search);
}
```

Figura 45. Código de la interfaz `SearchListener`

Para la primera búsqueda estándar en profundidad (`gov.nasa.jpf.search.DFSearch`), las implementaciones de escuchadores pueden tomar el modelo de notificación mostrado en la Figura 46.

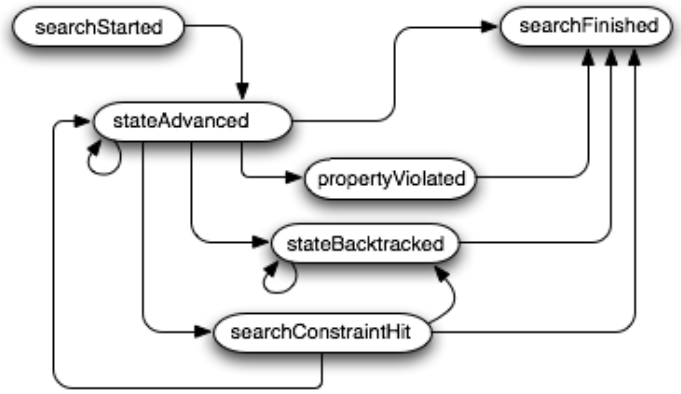

Figura 46. Modelo de notificación de *Listeners* (tomado de [1])

A.3.2 VMListener

Se utilizan para seguir el procesado detallado de la VM; por ejemplo, monitorizar instrucciones específicas de ciertos entornos de ejecución (como instrucciones IF de Java para análisis de cobertura, o instrucciones PUTFIELD o GETFIELD para condiciones de carrera potenciales). El código de la interfaz `VMListener` se puede ver en la Figura 47.

```
    /**Interfaz para registrar llamadas de la VM El rol del observador en la
misma se llama patrón*/
    public interface VMListener extends JPFListener {

  //--- Operaciones básicas con bytecode

    /* JVM va a ejecutar una instrucción */
    void executeInstruction (JVM vm);

    /* VM ha ejecutado la instrucción siguiente(puede ser utilizado para
analizar ramas, monitorizar PUTFIELD / GETFIELD einstrucciones INVOKExx /
RETURN) */
    void instructionExecuted (JVM vm);

  //--- Operaciones con los hilos

    /* nuevo hilo entró en el método run() */
    void threadStarted (JVM vm);

    /* el hilo espera obtener un bloqueo*/
    void threadBlocked (JVM vm);

    /* el hilo está esperando a una señalización */
    void threadWaiting (JVM vm);

    /* el hilo se ha notificado */
    void threadNotified (JVM vm);
```

```
        /* el hilo se ha interrumpido */
        void threadInterrupted (JVM vm);

        /* el hilo salió del método run() */
        void threadTerminated (JVM vm);

        /* nuevo hilo fue programado por la VM */
        void threadScheduled (JVM vm);

              //--- Manejo de la clase

        /* fue cargada una nueva clase */
        void classLoaded (JVM vm);

              //--- Operaciones del objeto

        /* un nuevo objeto se ha creado */
        void objectCreated (JVM vm);

        /* el objeto fue recogido por el recolector de basura (después de una
finalización potencial) */
        void objectReleased (JVM vm);

        /* notifica si un objeto bloqueado fue tomado(esto incluye una entrega
automática durante un wait()) */
        void objectLocked (JVM vm);

        /* notifica si un objeto bloqueado fue liberado(esto incluye re-
adquisición automática después de un notify())*/
        void objectUnlocked (JVM vm);

        /* notifica si se ha ejecutado un wait() */
        void objectWait (JVM vm);

        /* indica si un objeto notifica a un hilo a la espera */
        void objectNotify (JVM vm);

        /* indica si un objeto notifica a todos los hilos a la espera */
        void objectNotifyAll (JVM vm);

        /* ciclo comenzado del recolector de basura */
        void gcBegin (JVM vm);

        /* ciclo terminado del recolector de basura */
        void gcEnd (JVM vm);

        /* se lanzó excepción */
        void exceptionThrown (JVM vm);

              //--- Operaciones ChoiceGenerator

        /* nuevo ChoiceGenerator registrado */
        void choiceGeneratorSet (JVM vm);

        /* nueva elección del ChoiceGenerator actual */
        void choiceGeneratorAdvanced (JVM vm);

        /* el ChoiceGenerator actual procesó todas las elecciones */
        void choiceGeneratorProcessed (JVM vm);

    }
```

Figura 47. Código de la interfaz `VMListener`

El código de la Figura 48 es una versión abreviada de `PreciseRaceDetector`. La idea básica es que cada vez que encuentra un punto programado (un nuevo

`ThreadChoiceGenerator`), que está relacionado con el acceso a un campo de un objeto compartido, se comprueba si cualquiera de los otros hilos en ejecución está accediendo en ese momento al mismo campo del mismo objeto. Si al menos una operación es un *putfield* (insertar campo), se tendrá una condición de carrera potencial.

El ejemplo muestra tres aspectos que son típicos:

- Los escuchadores a menudo utilizan un pequeño número de métodos de notificación.
- A menudo no requieren una gran cantidad de código.
- A veces hay que profundizar en los constructores internos de JPF, para extraer instancias como las de `ThreadInfo`, `FieldInfo` y `ChoiceGenerator`.

```
public class PreciseRaceDetector extends PropertyListenerAdapter {

       FieldInfo raceField;
       ...

       //--- Sección Property
       public boolean check(Search search, JVM vm) {
              return (raceField == null);
       }

       //--- Sección VMListener
       public void choiceGeneratorSet(JVM vm) {
              ChoiceGenerator<?> cg = vm.getLastChoiceGenerator();

              if (cg instanceof ThreadChoiceFromSet) {
                     ThreadInfo[] threads =
                     ((ThreadChoiceFromSet)cg).getAllThreadChoices();
                     ElementInfo[] eiCandidates = new
                                          ElementInfo[threads.length];
                     FieldInfo[] fiCandidates = new
                                          FieldInfo[threads.length];

                     for (int i=0; i<threads.length; i++) {
                            ThreadInfo ti = threads[i];
                            Instruction insn = ti.getPC();

                            if (insn instanceof FieldInstruction) {
                                   // Ok, es un get/putfield
                                   FieldInstruction finsn =
                                                 (FieldInstruction)insn;
                                   FieldInfo fi = finsn.getFieldInfo();

                            if(StringSetMatcher.isMatch(fi.getFullName(),
                                                 includes, excludes)){
                                   ElementInfo ei =
                                                 finsn.peekElementInfo(ti);

                                   /* Se comprueba si se ha visto en
                                          otro hilo antes */
                                   int idx=-1;
                                   for (int j=0; j<i; j++) {
                                          if ((ei == eiCandidates[j]) &&
                                                 (fi == fiCandidates[j])) {
                                                 idx = j;
                                                 break;
                                          }
                                   }

                                   if (idx >= 0){
```

```
                                        /* Si, se tiene acceso multiple al
                                        mismo objeto/campo */
                                               Instruction otherInsn =
                                                      threads[idx].getPC();
                                               if(isPutInsn(otherInsn) ||
                                                             isPutInsn(insn)){
                                                      raceField =
                                        ((FieldInstruction)insn).getFieldInfo();
                                               ..
                                                      return;
                                               }
                                        } else {
                                               eiCandidates[i] = ei;
                                               fiCandidates[i] = fi;
                                        }
                                 }
                          }
                   }
            }

      public void executeInstruction (JVM jvm) {
            if (raceField != null) { //Se ha terminado, se procede a reportar

                  ThreadInfo ti = jvm.getLastThreadInfo();
                  ti.breakTransition();
            }
      }
}
```

Figura 48. Código de la clase `PreciseRaceDetector`

A.3.3 Instanciación

La instanciación explícita de un escuchador se puede hacer en cualquier caso. Si se especificó como una propiedad JPF, su clase necesitará tener o un constructor por defecto o un constructor que tome un único argumento de la clase `gov.nasa.jpf.Config`. El objeto `Config`, que es pasado a este constructor por JPF es el mismo que se utilizó para la configuración del mismo JPF. Este es el método preferido si el escuchador tiene que ser parametrizado. En el caso del ejemplo anterior, esto puede ser utilizado para filtrar campos relevantes con expresiones regulares. Un ejemplo de ésto, se puede ver en la Figura 49.

```
      public class PreciseRaceDetector extends PropertyListenerAdapter {
            ...
            StringSetMatcher includes = null;
            StringSetMatcher excludes = null;

            public PreciseRaceDetector (Config conf) {
                  includes =
StringSetMatcher.getNonEmpty(conf.getStringArray("race.include"));
                  excludes =
StringSetMatcher.getNonEmpty(conf.getStringArray("race.exclude"));
             }
            ...
            public void choiceGeneratorSet(JVM vm) {
                  ...
                  FieldInfo fi =..
                  if (StringSetMatcher.isMatch(fi.getFullName(), includes,
excludes)){
                  ...
            }...
```

Figura 49. Código de ejemplo de instanciación de un escuchador

A.3.4 Configuración

Las configuraciones de los escuchadores se pueden hacer de formas diferentes: Desde línea de comandos o mediante un archivo *.jpf*, a través de la *Shell* (consola) de JPF, o desde el sistema objeto de *test* utilizando anotaciones Java.

Los escuchadores deberán estar en el CLASSPATH.

1- **Linea de comandos:** La propiedad *jpf.listener* puede ser utilizada para especificar una lista, separada por dos puntos, de nombres de clases de escuchadores:

```
bin/jpf ... +jpf.listener=x.y.MyFirstListener:x.z.MySecondListener
```

2- **Archivo de propiedades *.jpf*:** Si se tienen varios escuchadores y/o un número de otras opciones JPF, es más conveniente añadir la propiedad *jpf.listener* a un archivo *.jpf*:

```
target = jpftest.Racer

jpf.listener=gov.nasa.jpf.tools.PreciseRaceDetector
```

3- **Anotaciones autocargadas:** Considera que el sistema objeto de *test* está marcado con anotaciones Java que representan propiedades. Por ejemplo, se puede utilizar `@NonNull` para expresar que un método no puede devolver un valor *null*:

```
import gov.nasa.jpf.NonNull;
...
@NonNull X computeX (..) {
   //.. algun cálculo complejo
}
...
```

Se puede utilizar el archivo *.jpf* (o la línea de comandos) para indicar a JPF que se debe cargar automáticamente y registrar los escuchadores correspondientes, si encuentra la anotación anteriormente indicada durante la carga de la clase:

```
...
jpf.listener.autoload = gov.nasa.jpf.NonNull,...
jpf.listener.gov.nasa.jpf.NonNull = gov.nasa.jpf.tools.NonNullChecker
...
```

4- **Anotaciones JPFConfig:** También se puede explícitamente dirigir a JPF para que cargue el escuchador desde dentro de tu aplicación utilizando la anotación `@JPFConfig`:

```
import gov.nasa.jpf.JPFConfig;
...
@JPFConfig ({"jpf.listener+=.tools.SharedChecker", ...})
public class TestNonShared implements Runnable {
...
}
```

Sin embargo, esto no está recomendado desde fuera de los *tests* JPF, porque la aplicación funcionaría, pero no compilaría.

5- **API Verify:** Un método menos utilizado es configurar los escuchadores para que usen el API `gov.nasa.jpf.jvm.Verify` desde dentro de su aplicación. Con esto se puede controlar el momento exacto en el que se carga el escuchador. De esta forma, el ejemplo de anterior quedaría así:

```
import gov.nasa.jpf.jvm.Verify;
...
public class TestNonShared implements Runnable {
...
   public static void main (String[] args){
   ...
Verify.setProperties("jpf.listener+=.tools.SharedChecker",...);
   ...
   }
}
```

Este método sólo se debe utilizar en casos especiales (modelos escritos explícitamente para verificación JPF) ya que no funcionaría fuera de JPF.

6- **API JPF embebido:** Si JPF es arrancado explícitamente desde dentro de otra aplicación, los escuchadores se podrán instanciar y configurar de esta manera:

```
MyListener listener=new MyListener(..);
...
Config config = JPF.createConfig( args);
JPF jpf = new JPF( config);
jpf.addListener(listener);
jpf.run();
...
```

A.4 MJI: Detalles de la funcionalidad

A.4.1 Componentes MJI

La funcionalidad básica de MJI consiste en un mecanismo para interceptar invocaciones de métodos y delegarlos por medio de llamadas de reflexión Java a las clases dedicadas. Hay dos tipos de clases involucradas que residen en diferentes capas:

- ***Model Class***: ésta es la clase ejecutada por JPF que podría ser totalmente desconocida para la JVM *host*.
- ***NativePeer Class***: esta es la clase que contiene las implementaciones de los métodos para interceptar y ejecutar en la JVM *host*.

Como parte de la implementación JPF, MJI automáticamente se ocupa de determinar qué invocaciones a métodos deben ser interceptadas, buscando los correspondientes métodos pares nativos. En la Figura 50 se muestra gráficamente la intercepción de métodos:

Esto no sería útil sin poder acceder al modelo de objetos JPF desde los métodos de `NativePeer`. En vez de requerir que todas las implementaciones de `NativePeer` residan en un paquete interno de JPF, existe una clase interfaz llamada `MJIEnv`, que puede ser utilizada para volver a JPF de forma controlada.

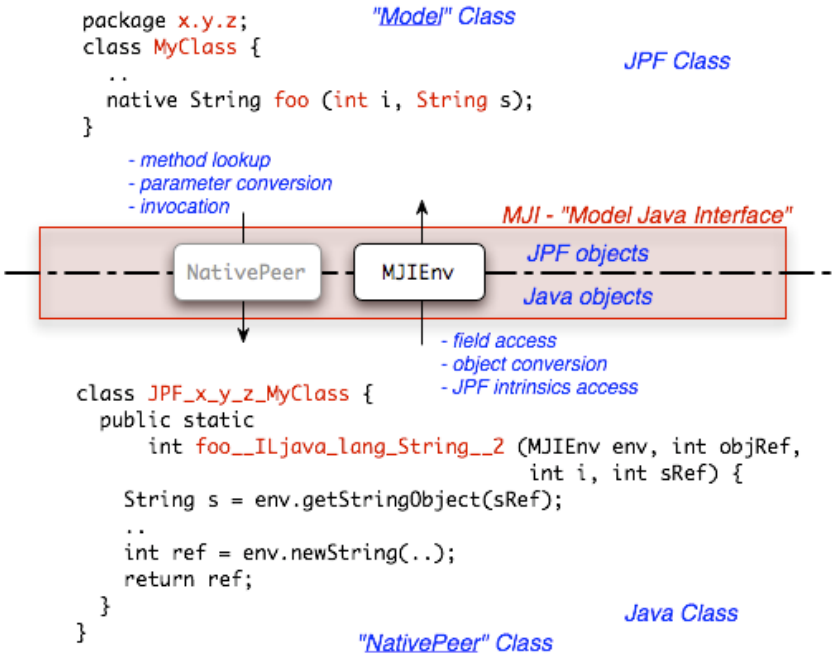

Figura 50. Intercepción de métodos (MJI) (tomado de [1])

Las implementaciones `NativePeer` que residen en el paquete `gov.nasa.jpf.jvm` (mismo paquete de `MJIEnv`) pueden básicamente acceder a todas las características internas de JPF. Fuera de este paquete, el API disponible en `MJIEnv` está principalmente restringido a acceder al objeto JPF (obteniendo y guardando valores). En la Figura 51 se muestra gráficamente la forma en que se invocan los pares nativos.

Antes de que un método `NativePeer` pueda ser utilizado, JPF tiene que establecer la correspondencia entre la clase `Model` y la `NativePeer`. Esto tiene lugar en tiempo de carga de la clase `Model`. MJI utiliza un esquema de nombres para buscar pares nativos, utilizando el nombre del paquete y el nombre de la clase `Model` para deducir el nombre de la clase `NativePeer`. Esto se puede ver gráficamente en la Figura 52.

Como el paquete de la clase `Model` está codificado en el nombre de la clase `NativePeer`, el paquete de `NativePeer` puede ser elegido libremente. Análogamente a JNI, los nombres de los métodos `NativePeer` incluyen la firma del método `Model` codificando sus parámetros. Si no hay ambigüedad potencial, la codificación de la firma no es obligatoria.

Todos los métodos nativos en `NativePeer` deben ser `public static`. No hay correspondencia entre objetos de JPF y de la JVM. En cambio, MJI automáticamente añade dos parámetros: `MJIEnv` y `objRef` (o `classRef` en caso de que sea un método estático). El objeto `MJIEnv` puede ser utilizado para regresar a JPF, `objRef` es un control para el objeto `this` correspondiente de JPF (o para el objeto `java.lang.Class` en caso de que sea un método estático).

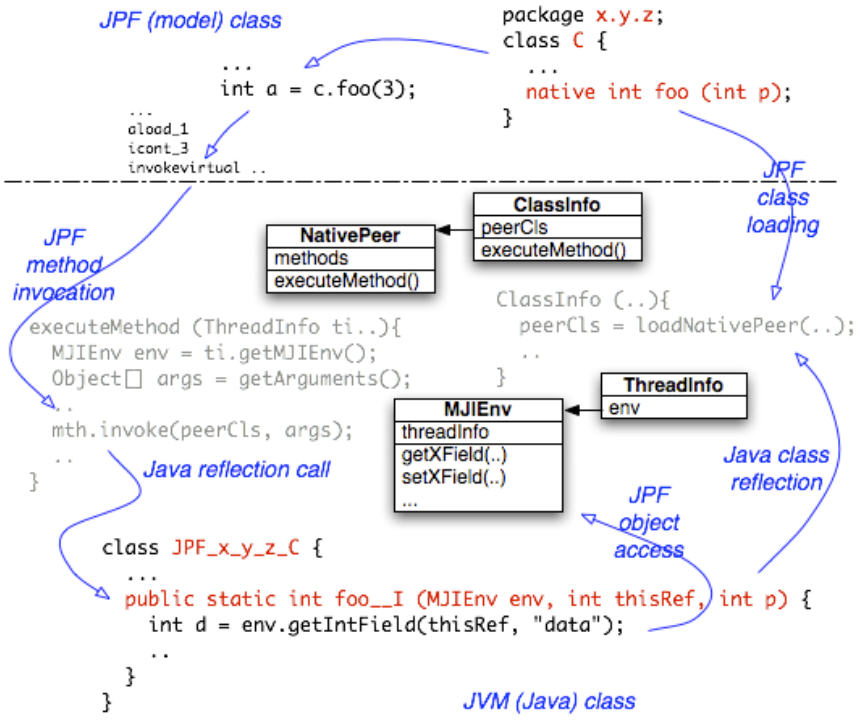

Figura 51. Invocación de pares nativos (tomado de[1])

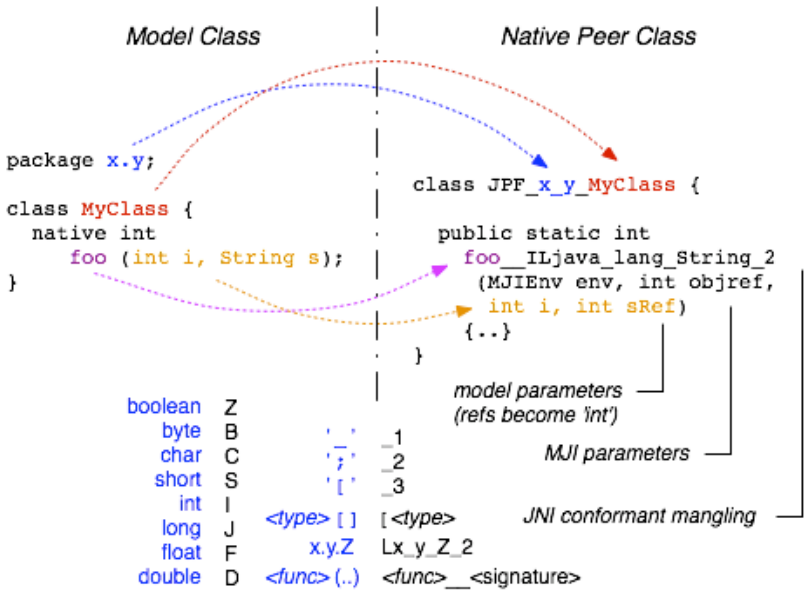

Figura 52. Correspondencia entre la clase `Model` y la clase `NativePeer` (tomado de [1])

Más allá de la analogía con JNI, MJI puede ser utilizado para interceptar:

- Métodos no nativos.
- Inicializaciones de clases.
- Constructores.

Es importante decir que esa correspondencia de tipo no incluye referencias. Todas las referencias (tipos objeto) en el lado de JPF se transforman en controles (valores *int*) en el lado JVM. Lo pasado en el parámetro `MJIEnv` debe ser utilizado para convertir/analizar el objeto JPF. Como por defecto MJI utiliza el mecanismo de llamada reflexiva estándar de Java, hay una penalización significativamente en velocidad, lo cual, otra vez es análogo a JNI.

Incluso si no está directamente relacionado con MJI, se debe mencionar que algunas clases `Model` específicas de JPF no pueden ser cargadas vía CLASSPATH, porque como contienen código JPF no son compatibles con la host JVM. Dichas clases se deben mantener en directorios/*jars* separados, que son especificados con la opción de línea de comandos JPF *-jpf-bootclasspath* o *-jpf-classpath*. Esto es el caso en su mayoría para las clases del sistema. Por otro lado, las clases `Model` no tienen por qué ser específicas de JPF. Es perfectamente correcto proveer un `NativePeer` para una clase Java estándar, si algunos métodos de esa clase deben ser interceptados. Las clases `NativePeer` pueden contener cualquier número de métodos y campos no nativos, pero éstos no deben ser `public static` para evitar problemas de búsqueda.

A.4.2 Herramientas

Para facilitar el tedioso proceso de modificar los nombres de los métodos manualmente, MJI incluye una herramienta para, automáticamente, crear esqueletos de clases `NativePeer` de una clase `Model` dada. Se llama *GenPeer*. El proceso de traducción utiliza reflexión Java (*Java Reflection*). Existe un número de opciones de línea de comandos que pueden ser mostrados llamando a *GenPeer* sin argumentos. Esta herramienta escribe directamente a *stdout*. En la Figura 53 se muestra el funcionamiento de dicha herramienta.

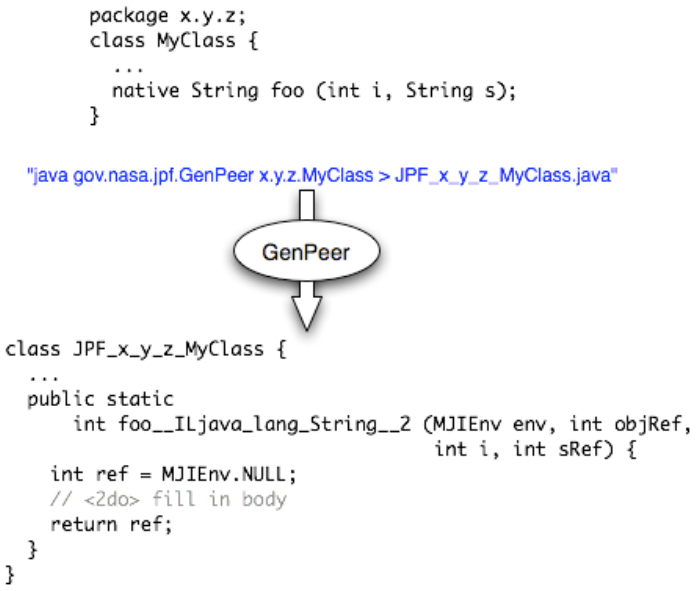

Figura 53. Funcionamiento de GenPeer (tomado de [1])

A.4.3 Ejemplos

El ejemplo siguiente es un extracto de un *test* de regresión JPF que muestra como se interceptan diferentes tipos de métodos.

- **Clase `Model`:**

Esto se ejecuta en JPF (debe estar en *vm.classpath* de JPF). El código se puede ver en la Figura 54.

```
    public class TestNativePeer {
      static int sdata;

      static {}
      int idata;

      TestNativePeer (int data) {}

      public void testClInit () {
            if (sdata != 42) {
                  throw new RuntimeException("native 'clinit' failed");
            }
      }

      public void testInit () {
            TestNativePeer t = new TestNativePeer(42);
            if (t.idata != 42) {
                  throw new RuntimeException("native 'init' failed");
            }
      }

      native int nativeInstanceMethod(double d,char c,boolean b, int i);

      public void testNativeInstanceMethod () {
            int res = nativeInstanceMethod(2.0, '?', true, 40);
            if (res != 42) {
                  throw new RuntimeException("native instance method
failed");
            }
      }

      native long nativeStaticMethod (long l, String s);

      public void testNativeStaticMethod () {
            long res =  nativeStaticMethod(40, "Blah");
            if (res != 42) {
                  throw new RuntimeException("native instance method
failed");

            }
      }

      native void nativeException ();

      public void testNativeException () {
            try {
                  nativeException();
            } catch (UnsupportedOperationException ux) {
                  String details = ux.getMessage();

                  if ("caught me".equals(details)) {
                        return;
                  } else {
                        throw new RuntimeException("wrong native exception
details: " + details);
                  }
            } catch (Throwable t) {
                  throw new RuntimeException("wrong native exception type: "
+
                  t.getClass());
            }
            throw new RuntimeException("no native exception thrown");
```

```
    }
  }
```

Figura 54. Clase `Model`

- **Clase `NativePeer`:**

Esto lo ejecuta la JVM *host* (debe incluirse en el CLASSPATH). El código se puede ver en la Figura 55.

```
  public class JPF_gov_nasa_jpf_jvm_TestNativePeer {

    public static void $clinit (MJIEnv env, int rcls) {
          env.setStaticIntField(rcls, "sdata", 42);
    }

    public static void $init__I (MJIEnv env, int robj, int i) {
          env.setIntField(robj, "idata", i);
    }

    public static int nativeInstanceMethod__DCZI__I (MJIEnv env, int robj,
double d, char c, boolean b, int i) {
          if ((d == 2.0) && (c  == '?') && b) {
                return i + 2;
          }
          return 0;
    }

    public static long nativeStaticMethod (MJIEnv env, int rcls, long l, int
stringRef) {
          String s = env.getStringObject(stringRef);
          if ("Blah".equals(s)) {
                return l + 2;
          }
          return 0;
    }

    public static void nativeException____V(MJIEnv env, int robj) {
    env.throwException("java.lang.UnsupportedOperationException", "caught
me");

    }
  }
```

Figura 55. Clase `NativePeer`

A.5 Bytecode Factories: Implementación y configuración

A.5.1 Implementación

Como hay un gran número de *bytecodes* Java, sería tedioso tener que implementar todas las clases `Instruction` solo para sobrescribir un par de ellas. Se puede reducir el esfuerzo de tres formas:

1- *`GenericInstructionFactory`*:

Utilizando `GenericInstructionFactory` como clase base para nuestra `InstructionFactory`. Esto solo requiere especificar un paquete alternativo donde residan las

clases alternativas, junto con el conjunto de *bytecodes* que deben ser sobrescritas. El código resultante puede ser bastante corto, como se puede ver :

```
public class NumericInstructionFactory extends GenericInstructionFactory {

  static final String[] BC_NAMES = {
        "DCMPG", "DCMPL",  "DADD", "DSUB", "DMUL", "DDIV",
        "FCMPG", "FCMPL",  "FADD", "FSUB", "FMUL", "FDIV",
                    "IADD", "ISUB", "IMUL", "IDIV","IINC",
                    "LADD", "LSUB", "LMUL", "LDIV"
};

  protected static final String BC_PREFIX =
"gov.nasa.jpf.numeric.bytecode.";

  protected static final String[] DEFAULT_EXCLUDES = {"java.*","javax.*"};

  public  NumericInstructionFactory (Config conf){
        super(conf, BC_PREFIX, BC_NAMES, null, DEFAULT_EXCLUDES);

        NumericUtils.init(conf);
  }
}
```

2- Super Delegación:

Se puede derivar las clases que sobrescriben *bytecode* de las que hay en `gov.nasa.jpf.jvm.bytecode`. Si solo se quiere añadir algunas comprobaciones antes o después de que se lleve a cabo la operación normal, se pueden utilizar las clases `Instruction` estándar, y llamar a `super.execute(...)` desde dentro de las clases derivadas.

3- Atributos:

Mientras las semánticas de ejecución se van haciendo más complejas, probablemente se necesite información adicional asociadas a variables. JPF ofrece un control automático *[[atributos|atributo Sistema]]* para este propósito. Se puede adjuntar objetos a operandos y campos locales, y JPF se ocupará de la propagación de estos atributos siempre que manipulen *stackframes* u objetos *heap*.

A.5.2 Configuración

Solo requerirá una propiedad JPF, que podrá ser indicada vía línea de comandos o mediante el archivo de propiedades *.jpf*:

```
vm.insn_factory.class = gov.nasa.jpf.numeric.NumericInstructionFactory
```

A.6 Configuración básica de la herramienta JPF

Después de todo esto, se sabe que JPF podrá comprobar y manejar:

- Secuencias programadas.
- Variaciones en los datos de entrada.
- Eventos del entorno.

- Elecciones de control de flujo.

Por lo tanto lo que queda por conocer es cómo configurarlo para un funcionamiento básico con el que comenzar a utilizar la herramienta.

Para configurar la herramienta JPF se vale de un diccionario central que se inicializa mediante un conjunto jerárquico de archivos de propiedades Java (`java.util.properties`) que tienen como objetivo cuatro capas diferentes del sistema:

- *System*: valores obligatorios de los constructores básicos del núcleo JPF (*jpf-core*).
- *Site*: para componentes JPF opcionales instalados.
- *Project*: configuraciones para cada componente JPF instalado.
- *Application*: las propiedades de la clase y programa que JPF debe comprobar.

La inicialización se realiza en orden de prioridad, lo que significa que se podrán sobrescribir propiedades desde posteriores etapas de configuración. Esta inicialización se muestra gráficamente en la Figura 56.

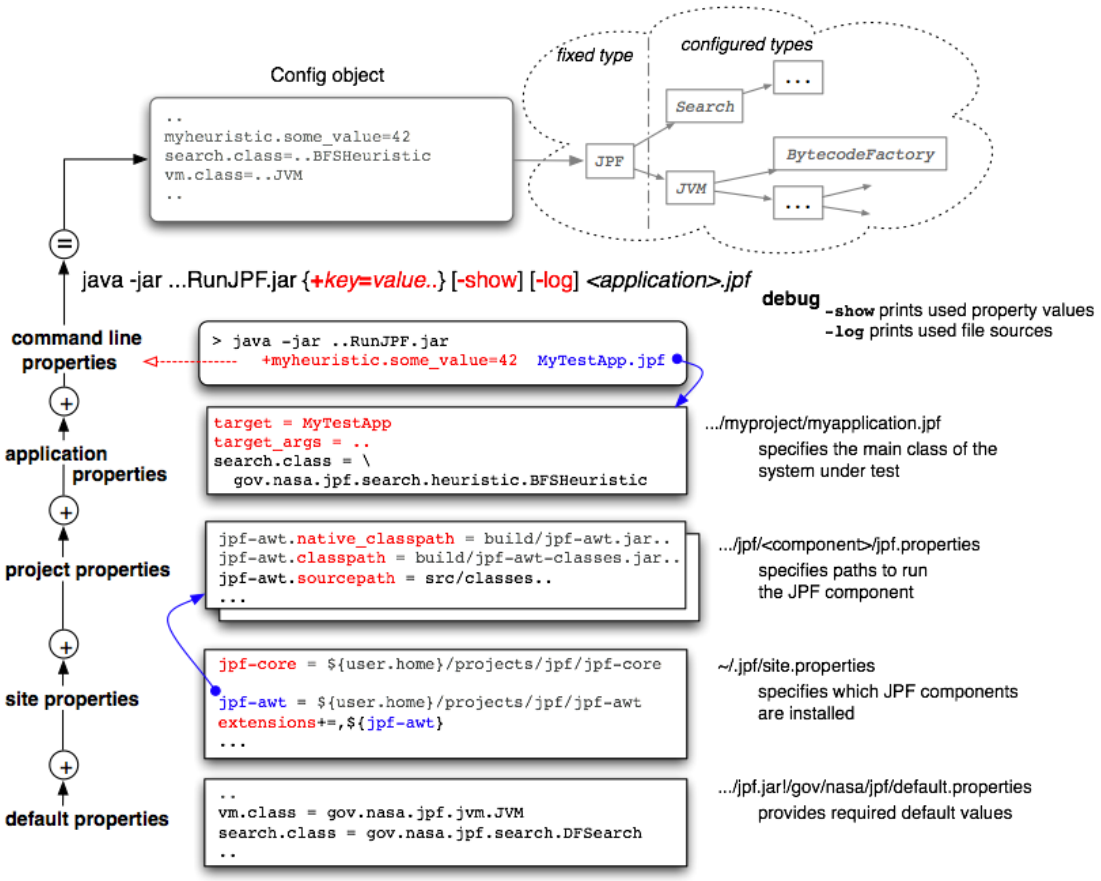

Figura 56. Inicialización de la configuración en JPF (tomado de [1])

Tipos de propiedades:

- **Propiedades por defecto:**

Este es el primer paso que da un conjunto de valores por defecto para sus componentes básicos. Como esto depende bastante de la versión en uso, se supondrá un archivo *jpf.jar* donde se encuentren todas las clases del núcleo JPF. El archivo correspondiente *default.properties* se busca como un recurso desde la clase `gov.nasa.jpf.JPF`. No es necesario especificar nada simplemente hay que asegurarse tener el archivo *jpf.jar* en el CLASSPATH. Realmente ni siquiera se necesita esto si se lanza JPF mediante el *script* que se incluye, o explícitamente a través de *java –jar RunJPF.jar*, los cuales buscan la localización de *jpf.jar* desde el archivo *site.properties.* No se debe modificar el archivo *default.properties* si no se es desarrollador de la parte del núcleo de JPF.

- **Propiedades *Site*:**

El archivo *site.properties* es específico de la máquina y no parte de ningún componente JPF, lo que significa que se creará como parte del proceso de instalación. Contiene dos tipos de información:

1. Localización del núcleo JPF (*jpf-core*).
2. Extensiones de JPF instaladas.

Cada extensión se lista como un par directorio/nombre, y después añadido a la lista de extensiones separadas por comas. El orden en el que se definen las extensiones importa, ya que determinará el orden en el que cada uno de estos componentes será inicializado, lo cual, básicamente mapea hacia una lista ordenada de entradas de CLASSPATH.

El archivo debe estar almacenado en *${user.home}/.jpf/ site.properties*, con *${user.home}* como valor de la propiedad del sistema Java estándar *user.home*.

- **Propiedades de proyecto:**

Cada componente JPF contiene un archivo *jpf.properties* en su directorio raíz, tanto si es *jpf-core* o cualquier otra extensión. Este archivo define las tres direcciones que se necesitan configurar para que el componente funcione adecuadamente:

1. ***native_classpath*:** El *classpath* de la VM *host*.
2. ***classpath*:** El *classpath* que JPF utiliza para ejecutar la aplicación objeto de *test*.
3. ***sourcepath*:** entradas que JPF utiliza para localizar el código fuente en caso de necesitar crear trazas del programa.

Un archivo *jpf.properties* se debe almacenar en el directorio raíz de un proyecto de componente JPF.

Los archivos *jpf.properties* son ejecutados en orden de definición en el archivo *site.properties*, con una salvedad: si se lanza JPF desde un árbol de directorios que contenga un archivo *jpf.properties*, éste tendrá siempre preferencia.

Tanto *site.properties* como *jpf.properties* pueden definir otros pares variable/valor, pero hay que tener en cuenta que el sistema se comportará de forma distinta dependiendo de donde se lance JPF. Además estos archivos deben ser consistentes en cuanto a nombres de componentes, etc. Esto también debe ser así para el archivo *build.xml* de *Ant* ([25]).

- **Propiedades de Aplicación:**

Para lanzar JPF se necesita indicar la clase principal que debe empezar a ejecutar (*target*). Además, también se pueden definir una lista de argumentos (*target_args*) y un número cualquiera de propiedades JPF que definen como se desea que sea comprobada la aplicación.

- **Propiedades de línea de comandos:**

Finalmente, se puede sobrescribir o extender cualquier configuración previa incluyendo pares del tipo +<*variable*>=<*valor*> como opción de línea de comandos.

A.6.1 Sintaxis de propiedades especiales

***variable=...${x}..:*** Reemplaza *${x}* con el valor que esté en ese instante en la variable x. También funciona recursivamente, como por ejemplo: *classpath = mypath;${classpath}*. *Ant* soporta la expansión normal de los valores, pero la expansión recursiva no funciona correctamente, por lo que habrá que usar una de las siguientes extensiones:

***variable+=valor*:** añade *valor* a lo que esté almacenado en *variable* en ese momento. No puede haber espacio en blanco entre *variable* y "+=". Esto sólo funcionará en JPF.

***+variable=valor*:** añade *valor* delante de lo que esté almacenado en ese momento en *variable*. Si se desea utilizar esto desde línea de comandos, habrá que utilizar "++" en vez de "+"; ya que con "+" solo vale en el archivo de propiedades.

Si se omite la parte "=." en una opción de la línea de comandos, se asigna el valor *true* por defecto a la variable correspondiente.

***${config_path}*:** está automáticamente configurado con la dirección del directorio del archivo de propiedades que se está parseando. Puede ser útil para indicar direcciones relativas.

***${config}*:** está configurado con la dirección del archivo que se está parseando.

***@requires=<variable>*:** se puede utilizar como la forma corta de cargar un archive de propiedades. Este es un mecanismo simple para prevenir la carga de un archivo *jpf.properties* si es necesario sobrescribir opciones de otro componente. Esto no lanzará una excepción si la variable requerida no se encuentra, esto solo asegura cargar el archivo de propiedades que contiene el *@requires*.

***@include=<archivo-propiedades>*:** carga recursivamente el archive de propiedades referenciado. Esto es útil para propiedades específicas de extensiones JPF que no se pueden incluir en el archivo *jpf.properties* de la extensión porque haría funcionar mal otros proyectos. Se deben poner dichas opciones en un archivo de propiedades distinto dentro del directorio raíz de la extensión y referenciar la dirección con *${config_path}/..* o *${project}/..*

A.7 Ejecución de JPF

Hay cinco formas generales para ejecutar JPF:

1- Desde línea de comandos.

2- Desde un IDE (Eclipse...) sin utilizar *plugins*.

3- Desde un IDE utilizando *plugins*.

4- Desde una clase de *test JUnit*.

5- Explícitamente desde un programa Java cualquiera.

Sólo se verá la opción de línea de comandos, de *test JUnit* y desde un programa cualquiera.

A.7.1 Línea de comandos

Hay diferentes formas dependiendo de varios grados de su infraestructura de ejecución. La manera más simple es utilizar el *script bin/jpf* de la distribución *jpf-core.* Desde el directorio de la aplicación objeto de *test* se podrá ejecutar de esta manera:

```
> <jpf-core-dir>/bin/jpf +classpath=. <application-main-class>
```

O preferiblemente,

```
> <jpf-core-dir>/bin/jpf <application-property-file>.jpf
```

Si se desea evitar *scripts* específicos de plataforma, sólo hay que utilizar esto:

```
> java -jar <jpf-core-dir>/build/RunJPF.jar +classpath=. <application-
main-class>
```

Esto hace uso de un pequeño *.jar* de arranque que es parte de la distribución de *jpf-core*, éste sólo incluye clases que son requeridas para lanzar el proceso de arranque. Dichas clases procesan automáticamente los diferentes archivos de configuración de JPF. Si la aplicación objeto de *test* es extensa, se recomienda añadir a la línea de comandos la opción *-Xmx1024m*, lo cual evita quedarse sin memoria.

Por último, se puede también directamente lanzar JPF y asignarle un *classpath* específico. Sería algo como esto:

```
> java -classpath <jpf-core-dir>/build/jpf.jar:<jpf-core-dir>/lib/bcel.jar
gov.nasa.jpf.JPF +classpath=. <application-main-class>
```

Los argumentos procesados por JPF se pueden dividir en 3 grupos diferentes:

1- Opciones de línea de comandos JPF:

Estas opciones deben aparecer al principio de la línea de comandos, y todas empiezan con guión (‘-’). El conjunto de las opciones actualmente soportadas son:

- ***-help***: muestra información de uso de la herramienta.
- ***-log***: muestra los pasos de configuración.
- ***-show***: muestra el diccionario de configuraciones después de la ejecución, es decir, el conjunto de opciones y propiedades con sus valores asignados durante dicha ejecución.

2- Propiedades JPF (*properties*):

Este es el segundo conjunto de opciones, todas empiezan con un signo más (*“+”*) y cumplen el modelo +<*clave*>=<*valor*>, como por ejemplo:

```
.. +cg.enumerate_random=true
```

Todas las propiedades de los diferentes archivos de configuración podrán ser sobrescritas desde la línea de comandos, por lo que no hay límite en el número de opciones. Si se desea extender algún valor de éstas propiedades se podrá realizar con una de estas diferentes notaciones:

- **+<*clave*>+=<*valor*>**: añade <valor> al final de los valores previamente asignados.
- **++<*clave*>=<*valor*>**: que coloca <valor> al principio de los valores previamente asignados.
- **+<*clave*>=*...${<clave>}...***: ofrece un control explícito sobre posiciones de la extensión.

Si la parte =<*valor*> se omite, se establece el valor true por defecto. Si se desea establecer una propiedad con valor *null*, sólo se debe omitir la parte <*value*> de esta forma: +<*clave*>=.

3- Especificación de objeto de test:

Para especificar qué aplicación debe analizar JPF hay 2 formas:

- Indicando explícitamente el nombre de clase y los argumentos:

```
> jpf ...  x.y.MyApplication arg1 arg2 ..
```

- Mediante el archivo de propiedades (*.*jpf*):

```
> jpf ... MyApplication.jpf
```

Se recomienda utilizar la segunda forma, pues permite almacenar todas las opciones en un archivo de texto que se puede tener junto con el código fuente de la aplicación objeto de *test*. Además, también permite lanzar JPF desde *NetBeans* o *Eclipse* únicamente seleccionando el archivo *.jpf* (Esto es para las extensiones principalmente). El archivo de propiedades requerirá una entrada *target* de esta forma:

```
# Archivo de propiedades de JPF para verificar x.y.MyApplication
```

```
target = x.y.MyApplication
target_args = arg1,arg2
```

A.7.2 Ejecutando JPF desde tests JUnit

JPF viene con una infraestructura de *test* basada en *JUnit* que se utiliza para su propio *test* de regresión. Este mecanismo puede ser utilizado también para crear nuestros propios controladores de *test* para ser ejecutados por *JUnit*. La estructura de la fuente de los *tests* es simple:

```
import gov.nasa.jpf.util.test.JPFTestSuite;
import org.junit.Test;

public class MyTest extends JPFTestSuite {

public static void main(String[] args) throws InvocationTargetException {
        runTestsOfThisClass(args);
  }

  @Test
  public void testSomeFunction() {
        if (verifyNoPropertyViolation()) {
              someFuntction(); ..   // Esta sección la verifica JPF
        }
  }
  //...más métodos @Test
```

También se puede ejecutar dicho *test* con la tarea de *Ant* estándar <*junit*>. Así:

```
<property file="${user.home}/.jpf/site.properties"/>
  <property file="${jpf-core}/jpf.properties"/>
  ...
        <junit printsummary="on" showoutput="off" haltonfailure="yes"
              fork="yes" forkmode="perTest" maxmemory="1024m">
        ...
        <classpath>
        ...
              <pathelement location="${jpf-core}/build/jpf.jar"/>
        </classpath>

        <batchtest todir="build/tests">
              <fileset dir="build/tests">
              ...
                    <include name="**/*Test.class"/>
              </fileset>
        </batchtest>
        </junit>
  ...
```

Sólo es necesario que el archivo *jpf.jar* esté en el *classpath* de la VM de la máquina cuando se compila y ejecuta el *test*, ya que `gov.nasa.jpf.util.test.JPFTestSuite` utilizará la configuración normal de JPF para establecer las opciones *native_classpath*, *classpath* y *sourcepath* en tiempo de ejecución.

Si se desea control explícito sobre el *classpath* de la VM de la máquina (opción *native_classpath* de JPF), se puede utilizar como clase base `gov.nasa.util.test.TestJPF` (la cual no utiliza el *classloader* de JPF), pero en este caso se necesitará añadir todos los archivos jar requeridos por todos los componentes JPF que se necesiten para el *test* (*jpf-*

<proyecto>/build/jpf-<proyecto>.jar y *jpf-<proyecto>/lib/*jar*, para todos los proyectos necesarios).

Si no se tiene control sobre el archivo *build.xml* por el tipo específico del proyecto del IDE, hay que añadir *jpf.jar* como un archivo jar externo a la configuración del proyecto del IDE.

Además de añadir *jpf.jar* a *build.xml* o a la configuración del proyecto del IDE, se puede querer añadir un archivo *jpf.properties* al directorio raíz del proyecto para configurar cosas como dónde debe encontrar JPF las clases y códigos fuente a analizar. Un ejemplo genérico podría ser éste:

```
# Ejemplo de archive de propiedades para establecer las variables de
entorno específicas del proyecto.

# No se require classpath native si esto no es un proyecto JPF en sí
mismo.

# Donde JPF encuentra los archives .class a ejecutar.
classpath=build/classes;build/test/classes

# Donde están las fuentes, en caso de querer que JPF cree una traza.
sourcepath=src;test

#Otras opciones JPF como 'autoloaders' etc.
listener.autoload+=,javax.annotation.Nonnull
listener.javax.annotation.Nonnull=.aprop.listener.NonnullChecker
...
```

A.7.3 Ejecutar JPF explícitamente desde un programa Java

El patrón correspondiente será tal que así:

```
Config conf = JPF.createConfig(args);
//... modifica la configuración de acuerdo a las necesidades...

try {
  JPF jpf = new JPF(conf);
  jpf.run();
  if (jpf.foundErrors()){
  ...
  }
} catch (..) {..}
```

A.8 Salida de JPF

Hay tres formas en las que JPF puede producir salida, cada una de ellas para un propósito diferente:

- **Resultado o salida de la aplicación:** Qué está haciendo la aplicación.
- **Registro JPF (*logging*):** Qué está haciendo JPF.
- **Sistema de reporte de JPF (*reporting*):** Resultado de la ejecución JPF.

A.8.1 Salida de la aplicación

Es la forma más simple de salida, la cual normalmente consiste en llamadas `System.out.println(...)` embebidas en el código de la aplicación. Es posible que estas llamadas se ejecuten varias veces por la propia aplicación JPF al analizar la aplicación.

```
public class MyApplication ..{
  ...
  boolean cond = Verify.getBoolean();
  System.out.println("and the cond is: " + cond);
  ...
}
```

Esto producirá como salida:

```
...
and the cond is: true
...
and the cond is: false
...
```

La segunda ejecución de la llamada viene precedida de una operación de retorno de JPF. Pero el retorno podría no ser visible. Como puede ser confuso encontrar la misma salida dos veces, hay dos opciones de configuración para controlar el comportamiento de la salida:

- **vm.tree_output={true|false} :** indica si se mostrará en la consola cada vez que se ejecuta una llamada `System.out.println(...)`.
- **vm.path_output={true|false}:** No mostrará inmediatamente la salida en la consola, sino que lo almacenará el camino de los procesos subsecuentes hasta que JPF termine. Esto mostrará la misma salida que una ejecución en una JVM normal.

A.8.2 Registro

Este tipo de salida es más interesante, ya que indica qué hace JPF internamente. Soporta varios niveles de detalle y está basado en la infraestructura estándar de `java.util.logging`. Esto permitiría utilizar `LogHandlers` y `Formatters` estándar adaptados, pero también existen dichas clases específicas de JPF, para permitir la configuración de registro a través del sistema estándar de configuración, mediante archivos de propiedades.

Se podrá configurar, tanto el destino de la salida de registro, como el nivel de registro (*log.level – severe,warning,info,fine,finer,finest*). Sólo se necesita declarar una instancia de registro estática con un identificador apropiado al inicio de la clase, y utilizar el API para crear la salida:

```
...
import java.util.logging.Level;
import java.util.logging.Logger;

package x.y.z;

class MyClass .. {
  static Logger log = JPF.getLogger("x.y.z");
  ...
  log.severe("there was an error");
```

```
  ...
  log.warning("there was a problem");
  ...
  log.info("something FYI");
  ...
  if (log.isLoggable(Level.FINE)){
        ...
        log.fine ("this is some detailed info about: " + something);
        ...
  }...
}
```

Si se desea mostrar el registro en una consola diferente que incluso esté funcionando en una máquina remota, se podrá utilizar en dicha máquina remota la clase `gov.nasa.jpf.tools.LogConsole` de esta forma:

```
$ java gov.nasa.jpf.tools.LogConsole <puerto>
```

Después en la máquina en la que se realiza el *test* se ejecutará de esta forma:

```
$ jpf +log.output=<maquina>:<puerto> ... MyTestApp
```

La máquina por defecto es *localhost*, y el puerto es el *20000*.

A.8.3 Reportes

El sistema de reportes se utiliza para mostrar el resultado de una ejecución JPF, indicando violación de propiedades, trazas, estadísticas y mucho más. Esto podría involucrar diferentes formatos de salida (XML o texto) y objetivos (consola, IDE). Esto es un mecanismo extensible para adaptarse a nuevas herramientas y propiedades. Este mecanismo también se configura mediante archivo de propiedades.

El concepto básico es que esto dependerá de un conjunto predefinido de fases de salida, cada uno de ellos con una lista ordenada de temas configurada. Las fases de salida soportadas actualmente son:

- *start*
- *transition*
- *property_violation*
- *finished*

No hay tema de *transition* estándar definido todavía. Los temas estándar de *property_violation* incluyen:

- *error*
- *trace*
- *snapshot*
- *output*
- *statstics*

Por último, las listas de temas *finished* que normalmente resumen la ejecución de JPF:

- *result*

- *statstics*

El API de Reportes se compone de tres componentes principales:

- *Reporter* (el reportador).
- Objetos *Publisher*.
- Objetos *PublisherExtension*

En la Figura 57 se detalla éste API.

`Reporter` es el recolector de datos. También controla y notifica a `Publisher` cuando se ha llegado a cierta fase de salida. Los objetos `Publisher` son productores de salida con formato específico, el más prominente es `ConsolePublisher` (para la salida normal en consolas). Los objetos `PublisherExtension` pueden ser registrados por `Publisher` en tiempo de arranque, por ejemplo desde *Listeners* que implementen propiedades o modos de análisis.

La configuración es sencilla e implica el manejo de un conjunto de propiedades JPF de la categoría *report*. La primera indica la clase `Reporter`, la cual no se debe modificar a no ser que se quiera implementar diferentes modos de recolección de datos:

```
report.class=gov.nasa.jpf.report.Reporter
```

La siguiente indica una lista de instancias `Publisher` a utilizar, utilizando nombres simbólicos:

```
report.publisher=console:xml
```

Cada uno de estos nombres simbólicos debe tener un nombre de clase correspondiente definido así:

```
report.console.class=gov.nasa.jpf.report.ConsolePublisher
```

Finalmente, se debe especificar por cada nombre simbólico y fase de salida qué temas deberían ser procesados y en qué orden. Por ejemplo:

```
report.console.property_violation=error:trace:snapshot
```

El orden de estos temas importa, y ofrece un control completo sobre el formato de reporte. Los valores por defecto están en el archivo *default.properties*.

Las clases `Publisher` pueden tener sus propias propiedades adicionales. Por ejemplo, la implementación de `ConsolePublisher` puede ser configurada con respecto a la información que se incluye en las trazas (*bytecodes* y nombres de métodos) y redirigir la salida (archivo o *socket*). Para redirigir la salida archivo se utiliza la siguiente propiedad:

```
# Guardar el reporte en un archivo
report.console.file=My_JPF_report
```

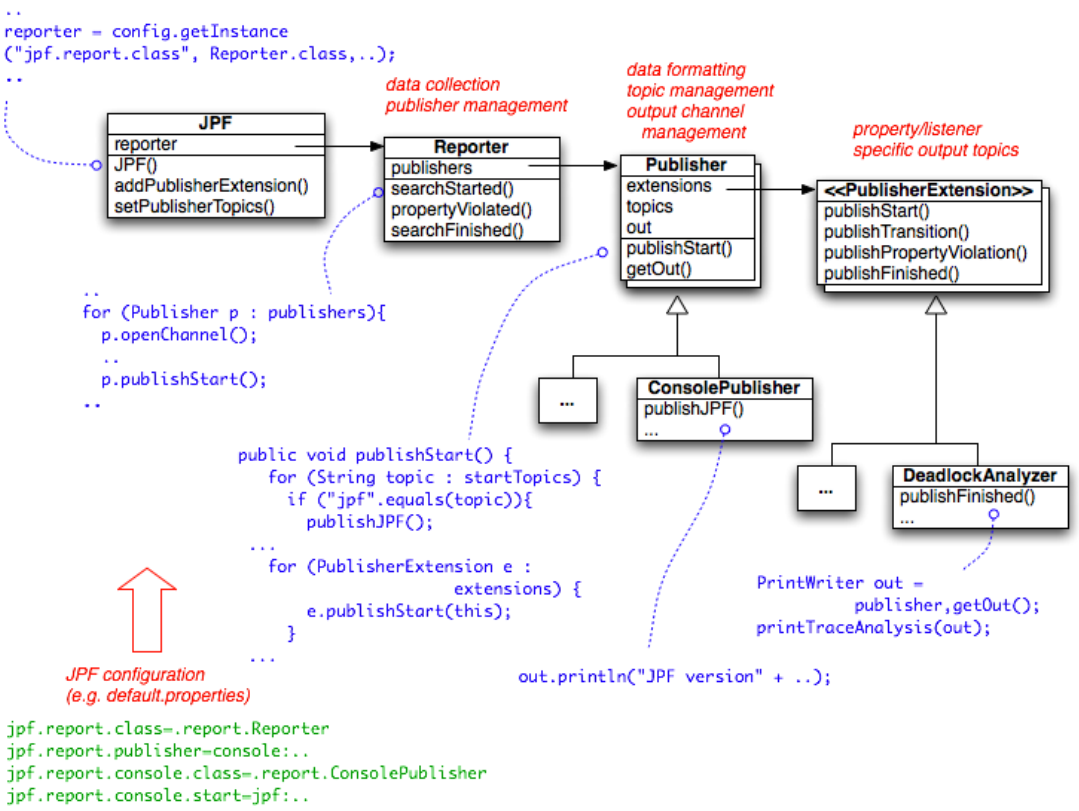

Figura 57. API de reportes JPF (tomado de [1])

Todo lo que tiene que ver con clases e interfaces del núcleo se encuentra en el paquete `gov.nasa.jpf.report`. La manera más común de extender el sistema es utilizar una implementación de `PublisherExtension` propia, que implica realizar dos pasos:

1- Implementar los métodos de fase y formato específicos requeridos.

2- Registrar la extensión para la clase `Publisher` específica.

Como ejemplo, la clase `DeadlockAnalyzer`:

```
public class DeadlockAnalyzer extends ListenerAdapter {
  ...
  public DeadlockAnalyzer (Config config, JPF jpf){
        jpf.addPublisherExtension(ConsolePublisher.class, this);  //(1)
        ...
  }
  ...
  public void publishPropertyViolation (Publisher publisher) { //(2)
        PrintWriter pw = publisher.getOut();
        publisher.publishTopicStart("thread ops"
                                    +publisher.getLastErrorId());
  }
}
```

A.9 El API de Verify

Aunque el propósito inicial de JPF es analizar aplicaciones que no dependen de JPF, se puede utilizar para analizar programas que hayan sido explícitamente creados para ejecutarse bajo JPF. Hay dos maneras de hacer esto:

- **Anotaciones JPF** (`@gov.nasa.jpf.annotation.JPFConfig` y `@gov.nasa.jpf.annotation.FilterField`).
- **API de `gov.nasa.jpf.jvm.Verify`.**

El primer método no modifica o añade código específico a la aplicación, solo provee de directivas a JPF en forma de anotaciones Java (@...). El segundo sólo se debe utilizar en *drivers* de *test* específicos de JPF.

A.9.1 Anotaciones JPF

Actualmente se soportan dos tipos de anotaciones: **`@JPFConfig`** y **`@FilterField`**.

- **`@JPFConfig`**: permite establecer opciones o propiedades JPF para clases y métodos de la aplicación, lo cual, es útil para añadir escuchadores específicos. Hay que tener cuidado con los efectos de las propiedades porque no todos los comportamientos pueden ser cambiados en tiempo de ejecución y normalmente no se podrán revertir características con este método.

```
import gov.nasa.jpf.annotation.JPFConfig
...
@JPFConfig({"listener+=,gov.nasa.jpf.aprop.listener.SharedChecker
", ..})
public class MyClass {...}
```

- **`@FilterField:`** se utiliza para marcar ciertos campos como no relevantes, para la correspondencia de estados. Esto será útil cuando hay que añadir información de depuración, como contadores, que de otra forma aumentarían el espacio de estados:

```
import gov.nasa.jpf.annotation.FilterField;
...
public class MyClass {
        ...
        @FilterField int counter;
        ...
}
```

Ésto no cambia la ejecución del programa de ninguna forma y no afecta al retorno a estados anteriores de JPF. Sólo indica a JPF que ignore los campos marcados cuando genere los estados del programa.

A.9.2 API Verify

Idealmente JPF se utiliza para verificar aplicaciones Java arbitrarias, pero a menudo, estas aplicaciones son modelos de Java de otros sistemas. En este caso, puede ser útil llamar al API

de JPF desde dentro de la aplicación, para así obtener información de JPF o directamente de su ejecución.

El API de JPF está centralizado en la clase `gov.nasa.jpf.jvm.Verify` que incluye métodos de las siguientes categorías principales:

1. **Non-deterministic data choice generators:**

Es casi la categoría principal del API, la cual se puede ajustar para escribir *drivers* de *test* que sean "Inspectores de Modelo" (*Model Checkers*) de forma consciente. La idea es obtener valores de datos de entrada no determinísticos de JPF, de forma que pueda analizar sistemáticamente todas las "elecciones" o "opciones" (*choices*) relevantes.

En la forma más simple, puede ser utilizado así:

```
// código del driver de test
import gov.nasa.jpf.jvm.Verify;
...
boolean cond = Verify.getBoolean();
/* El código siguiente se ejecutará para ambas condiciones
 (true y false)*/
...
```

Pero los generadores de opciones no determinísticos no se limitan a valores que puedan ser completamente enumerados basándose en su tipo (como *Booleans* o *Integers*). JPF también soporta generadores de opciones basados en heurísticos configurados, donde los valores elegidos dependen de la aplicación; y pueden ser especificados en archivos de propiedades, tal y como se muestra en la Figura 58.

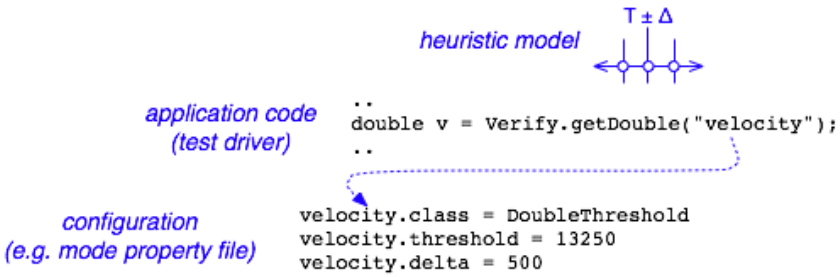

Figura 58. Ejemplo de configuración del API de Verify (tomado de [1])

2. **Search pruning (poda de búsqueda):**

Es útil para propiedades muy específicas de la aplicación, en las que es obvio que algunos valores no son de interés con respecto a la propiedad.

```
//...Calcula algunos datos...
Verify.ignoreIf(data > someValue);
//...Hace algo con los datos...
```

Si la instrucción es *true*, JPF no continua ejecutando el camino actual; vuelve atrás, al anterior puno de elección no determinista.

3. **State annotation (Anotación de estado):**

Basado en ciertas combinaciones de valores, una aplicación podría dar a JPF sugerencias sobre la relevancia del estado de un programa que puede ser posteriormente utilizado por implementaciones de `Search` y/o `Heuristics`.

```
//...Calcula algunos datos...
Verify.interesting( data < someValue );
//...Hace algo con los datos...
```

Ésto no para la ejecución de JPF, pero almacenará un atributo interesante para el estado en que se encuentra. Su versión más general se utiliza para adjuntar cadenas (*Strings*) arbitrarias a estados.

```
//...Calcula algunos datos...
if (data < someValue) {
  Verify.setAnnotation("critical data value");
  ...
  //...Hace algo con los datos...
```

Esta categoría es casi la menos importante porque *Search- y VMListeners* son mecanismos mejores para almacenar, no solo cadenas de texto, sino objetos arbitrarios como notaciones de estado.

4. Verification log output (Registro de verificación de salida):

Esta es la categoría más simple y se utiliza para diferenciar la salida normal de un programa (que es ejecutado y analizado por JPF) y de salidas que son estrictamente relevantes para la verificación. Por ejemplo, no deberían aparecer cuando se ejecute el programa fuera de JPF. No sorprende que contenga un gran número de métodos `print(...)`.

5. Explicit Atomicity Control (Control de atomicidad explícito):

Esta categoría puede ser utilizada para controlar el número de hilos intercalados que JPF debe analizar. Aunque esto es problemático en términos de defectos potenciales de pérdida es, en ocasiones, la única forma de contraer el espacio de estados para que JPF pueda verificar una aplicación dada.

```
Verify.beginAtomic();
...
// Todo el código de aquí es ejecutado por JPF en una transición
...
Verify.endAtomic();
```

El control de atomicidad directo fue utilizado, principalmente, antes de que el automático *On the-fly Partial Order Reduction* (POR) fuera implementado; y sólo permanece relevante para aplicaciones que son problemáticas con respecto a POR. Ésto incluye específicamente acceso frecuente a campos alcanzables, pero no visibles en programas concurrentes.

En general, el control de atomicidad explícita será reemplazado por futuras extensiones de POR y podrían aparecer en próximas actualizaciones.

6. Otros:

Métodos de `Verify` más exóticos permiten obtener información durante la ejecución de JPF, lo cual es persistente y puede ser, más tarde, pedido por el código empotrado de JPF (Programas que ejecutan JPF). Esto utiliza un mecanismo de MJI (*Model Java Interface*) donde

la clase par nativa (`JPF_gov_nasa_jpf_jvm_Verify`) es utilizada para configurar algunos datos durante la ejecución de JPF. Dichos datos podrán ser recuperados más tarde por el código de la clase modelo (`gov.nasa.jpf.jvm.Verify`) que es ejecutada fuera de JPF. Esto se utiliza actualmente para implementar contadores que en su momento serán utilizados para verificar JPF.

Se debería resaltar que mientras la mayoría de los APIs Verify tienen implementaciones alternativas que permiten ejecución fuera de JPF, las aplicaciones que las utilizan no se construyen o compilan fuera del entorno de JPF. Su uso, por lo tanto, está recomendado para *drivers* específicos de *test* JPF.

A.10 Extensiones de JPF

JPF está dividido en diferentes módulos de ejecución, siendo tratado cada uno como un proyecto en sí mismo. El componente *jpf-core* es obligatorio, ya que contiene la máquina virtual y mecanismos utilizados por el resto de los proyectos; el resto de módulos son opcionales. Cada componente posee su propio repositorio. Seguidamente, se muestra una lista con los más importantes:

Núcleo del proyecto JPF:

- ***jpf-core***

Plugins o extensiones:

- ***jpf-actor***: Herramienta para *testeo* sistemático de programas actores
- ***jpf-aprop***: Propiedades basadas en anotaciones Java y sus correspondientes verificadores.
- ***jpf-awt***: Librería específica de implementación para `java.awt` y `javax.swing`.
- ***jpf-awt-shell*** : Consola especializada para la verificación del modelo de `java.awt` y `javax.swing`.
- ***jpf-concurrent***: Librería optimizada para JPF de `java.util.concurrent`.
- ***jpf-delayed***: Pospone valores de opciones no deterministas hasta que son utilizadas.
- ***jpf-guided-test***: Entorno de trabajo para guiar la búsqueda utilizando heurísticos y análisis estático.
- ***jpf-mango***: Especificación y generación de artefactos de prueba.
- ***jpf-numeric***: Una alternativa de instrucción configurado para la inspección de programas numéricos.
- ***jpf-racefinder*** : Un detector preciso de condiciones de carrera en un modelo de memoria Java relajada.
- ***jpf-rtembed*** : Verificación de programas Java para plataformas de tiempo real y embebidas.
- ***jpf-statechart***: Modelado de estados mediante UML.
- ***net-iocache***: Extensión de la caché E/S para manejar comunicaciones de red.

En el proyecto sólo se han analizado tres, porque pueden aportar algo de funcionalidad interesante: ***jpf-awt***, ***jpf-awt-shell*** *y* ***jpf-concurrent***.

A.10.1 *jpf-awt / jpf-awt-shell*

La extensión *jpf-awt* contiene las librerías modeladas de `java.awt` y `javax.swing`, las cuales, abstraen todo lo relacionado con el aspecto gráfico pero preservan el control de flujo.

Para la simulación de los datos de usuario, se utiliza un *script* que permite opciones no deterministas. Este *script* debe contener instrucciones como éstas:

```
// Pulsado de botón
$acquire_status.doClick()
// Entrada de texto
$Sequence:input.setText("turn")
// Distintas combinaciones
ANY {$<FORCED|QUEUED|SINGLE_STEP>.doClick()}
ANY {$<FORCED|QUEUED|SINGLE_STEP>.doClick()}
ANY {$<FORCED|QUEUED|SINGLE_STEP>.doClick()}
// Selección de listas
ANY {$Robots:list.setSelectedIndex([0-3])}
// Pulsado de botón de envío
$Send.doClick()
```

La estructura de cada instrucción es esta: `$Nombre_elemento.acción()`

La estructura para hacer diferentes combinaciones (`ANY{}`) es bastante intuitiva. No hay demasiada información sobre más tipos de instrucciones de este *script*.

Para introducir en una validación de JPF este *script* se utilizará la opción *+awt.script=Archivo_script*. Además, para poder utilizar esta opción, hay que añadir dos propiedades a la validación, que son las siguientes::

- *+listener=gov.nasa.jpf.listener.ChoiceTracker*
- *+choice.class=gov.nasa.jpf.awt.UIActionGenerator*

La extensión *jpf-awt-shell* principalmente añade a la consola de JPF dos cosas:

- Visor de *scripts* de datos de usuario.
- Trazas sobre *scripts* de usuario.

A.10.2 *jpf-concurrent*

Esta extensión contiene una versión optimizada de la librería `java.util.concurrent`. Permite un manejo más eficiente de todo lo que tiene que ver con ésta librería, y añade un *Listener* que permite solucionar problemas de fugas de memoria.

Este *Listener* se incluiría en una validación JPF añadiéndolo de la siguiente forma:

```
+listener=gov.nasa.jpf.concurrent.ObjectRemovalListener
```

Además, debe estar incluida la siguiente propiedad:

```
+cg.threads.break_start=true
```

A.11 Instalación de JPF

A.11.1 Requerimientos del sistema

Previo a la instalación de JPF hay que tener instalado en el sistema una versión actual del SDK de Java (JDK 6 al menos). Además, es aconsejable que el sistema tenga al menos 2 GB de memoria RAM, ya que JPF es una aplicación que puede consumir muchos recursos del sistema.

Para la descarga de JPF y de sus extensiones será necesario una aplicación llamada Mercurial, la cual puede ser obtenida de http://mercurial.selenic.com/wiki/.

No hay necesidad de instalar *Ant*, *JUnit* u otras librerías, ya que las versiones compatibles de dichos componentes están incluidas dentro de los paquetes descargados.

A.11.2 Descarga e instalación del núcleo

El código fuente de JPF se encuentra en repositorios de Mercurial, en el directorio http://babelfish.arc.nasa.gov/hg/jpf . Será necesario clonar los sub-repositorios.

Para descargar el sub-repositorio del núcleo de JPF (*jpf-core*), que contiene la funcionalidad base de la herramienta, se utiliza el programa Mercurial desde línea de comandos de esta forma:

- Ir a la carpeta donde se quiere instalar JPF:

```
> cd ~\Pathfinder
```

- Dentro de esa carpeta ejecutar la siguiente línea de comandos:

```
> hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
```

Para terminar de instalar el núcleo, será necesario compilar el sub-repositorio obtenido. Para ello, dentro de la carpeta generada *../jpf-core/* se debe lanzar el siguiente comando

```
> ..\bin\ant
```

Este comando utiliza la aplicación *Ant*, que incluye el paquete descargado para compilarlo. De esta forma aparecerán los archivos *.class* y *.jar* en una carpeta llamada *../jpf-core/build*. Además, dentro de la carpeta *../jpf-core/bin* aparecerán los archivos ejecutables de la aplicación JPF.

Si se desea ejecutar mediante línea de comandos la aplicación desde cualquier ubicación, hay que incluir en la variable de entorno del sistema PATH el directorio *../jpf-core/bin*.

Para comprobar que se ha instalado la herramienta correctamente, si se ejecuta el comando *jpf*, en la pantalla aparecerá algo como lo que se muestra en la Figura 59:

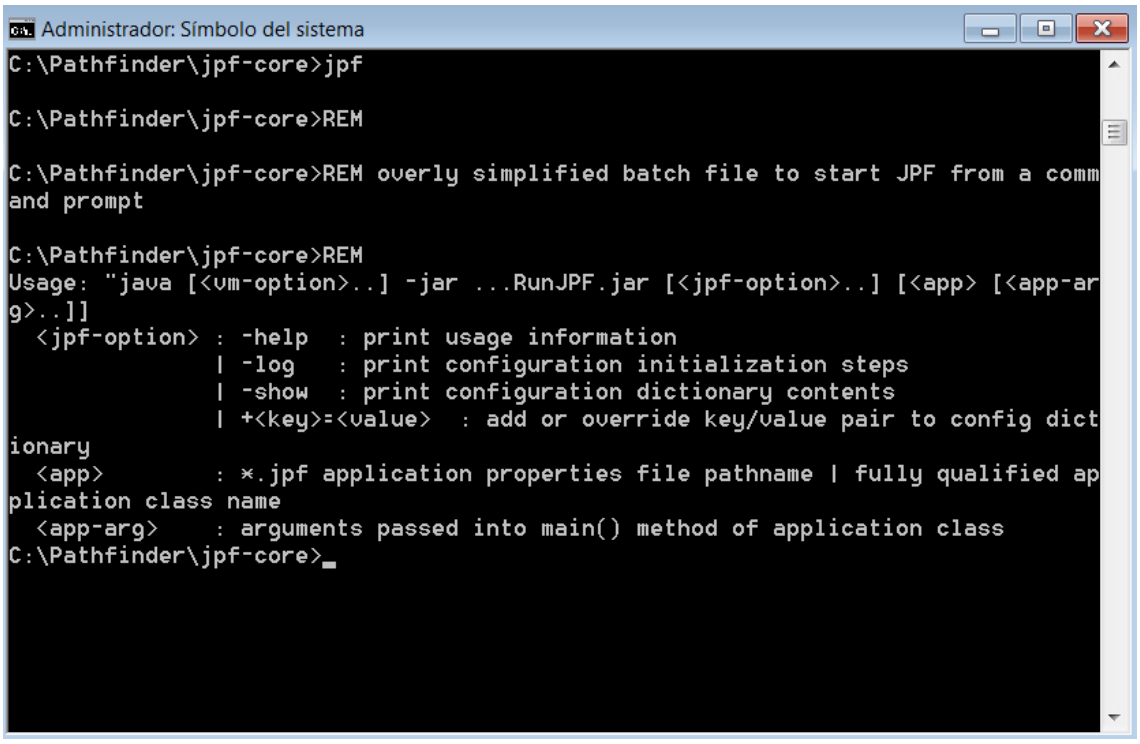

Figura 59. JPF funcionando en la consola del sistema

A.11.3 Instalación de extensiones

Para instalar las extensiones de JPF que se consideren necesarias, deben ser descargadas utilizando el programa Mercurial, de la forma descrita en la descarga del núcleo. En este caso, la dirección utilizada dependerá de la extensión que se decida instalar, por ejemplo:

- **jpf-awt**: > hg clone https:// bitbucket.org/pcmehlitz/jpf-awt
- **jpf-awt-shell**: > hg clone https:// bitbucket.org/pcmehlitz/jpf-awt-shell
- **jpf-concurrent**: > hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-concurrent

Éstas deben ser descargadas en el mismo directorio que el núcleo *jpf-core*.

Para terminar de instalar las extensiones, será necesario compilar los sub-repositorios obtenidos. Para ello, dentro de la carpeta generada *../jpf-nombre_Extensión/*, se debe lanzar el siguiente comando:

```
> ..\bin\ant
```

Una vez compilado, hay que configurar para que el núcleo utilice dichas extensiones.

Primero, se genera el archivo *site.properties* en el directorio raíz en el que se descargaron todos los sub-repositorios con este contenido:

```
# Directorio donde se encuentran todos los sub-repositorios.
jpf.home = C:/Pathfinder
# Ubicación del núcleo (jpf-core)
jpf-core = ${jpf.home}/jpf-core
```

Y además, hay que añadir las ubicaciones de las extensiones:

```
# concurrent extension
jpf-concurrent = ${jpf.home}/jpf-concurrent
extensions+=,${jpf-concurrent}
# awt extension
jpf-awt = ${jpf.home}/jpf-awt
extensions+=,${jpf-awt}
# awt-shell extension
jpf-awt-shell = ${jpf.home}/jpf-awt-shell
extensions+=,${jpf-awt-shell}
```

Una vez hecho esto, solo queda comprobar si la aplicación ha incluido correctamente las extensiones. Para esto, se procede a ejecutar el comando *jpf –show +site=../site.properties*, el cual, debe dar como salida toda la configuración de la herramienta incluyendo lo que se muestra en la Figura 60:

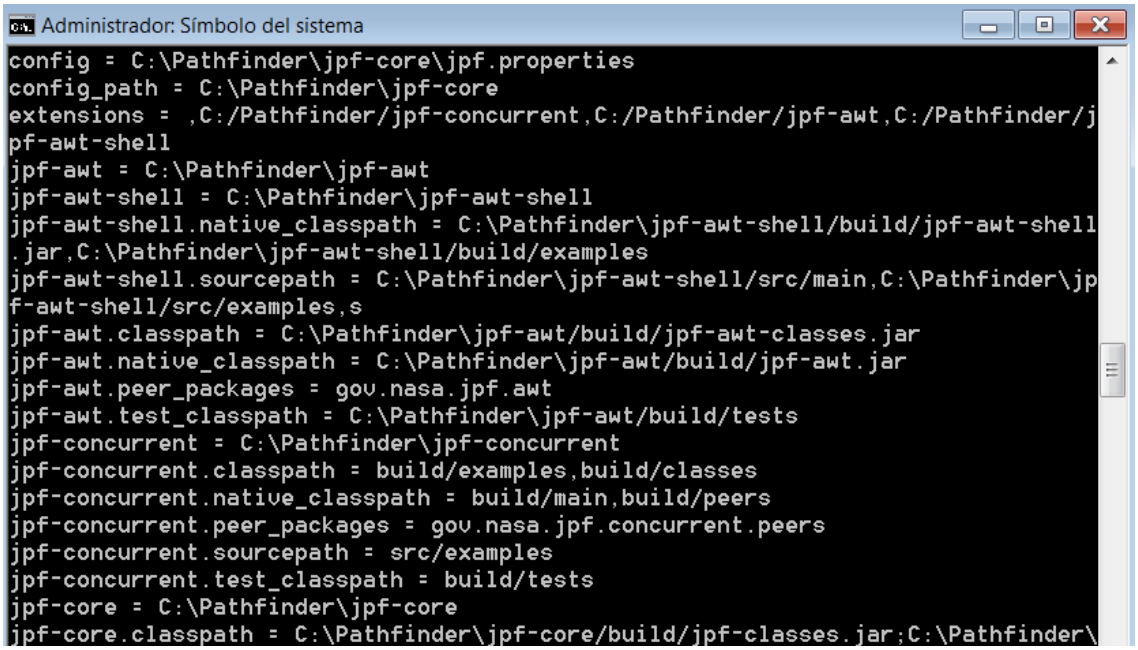

Figura 60. Configuración de JPF con extensiones (desde consola)

Para utilizar todas las extensiones configuradas, la herramienta JPF se debe lanzar siempre con la propiedad *+site=Archivo_site*.

APÉNDICE B. Aspectos específicos de la implementación

Esta sección del apéndice contiene la forma de implementación de ciertos módulos o funcionalidades del proyecto interesantes. Estos son:

- Conexión JMS (Sección B.1)
- Extracción de archivos (Sección B.2)
- Compilación (Sección B.3)
- Ejecución (Sección B.4)
- Envío de correos (Sección B.5)
- Cliente de consola (Sección B.6)
- Organización de los archivos de la aplicación (Sección B.7)

B.1 Conexión JMS

B.1.1 Introducción a JMS

El API JMS o *Java Message Service* ([21]) es un estándar de mensajería que permite la aplicación basada en componentes en la plataforma *Java Enterprise Edition* (Java EE). Permite crear, enviar, recibir y leer mensajes.

JMS provee un servicio fiable y flexible para un intercambio asíncrono de información de negocio crítica. El API contiene los siguientes rasgos:

- Message-Driven Beans: permiten la recepción asíncrona de mensajes.
- Envíos y recepciones de mensajes JMS, que pueden participar en las transacciones de JTA (Java Transaction API).
- Interfaces de Java EE Connector Architecture que permiten integrar implementaciones JMS de otros proveedores.

La inclusión del API de JMS mejora la plataforma Java EE simplificando el desarrollo de las aplicaciones. Permite interacciones entre componentes de dicha plataforma y sistemas capaces de enviar mensajes, de forma débilmente acoplada, fiable y asíncrona.

La arquitectura del contenedor de los Enterprise JavaBeans (EJBs) de la plataforma J2EE mejora el API de JMS de dos formas:

- Permitiendo el consumo concurrente de mensajes.

- Proveyendo soporte para transacciones distribuidas como actualización de bases de datos, procesamiento de mensajes y conexiones con sistemas EIS utilizando la Java EE *Connector Architecture*, pudiendo participar todos en el mismo contexto.

El funcionamiento que se desea obtener de esta tecnología es éste: Una aplicación cliente envía mensajes a la cola (queue), el proveedor de JMS (en este caso el Servidor Java EE) entrega los mensajes a las instancias de Message-Driven Beans (MDB), las cuales procesarán los mensajes.

B.1.2 Implementación

El módulo de JMS utilizado en el proyecto se compone de dos partes:

- Cliente: En el *servlet* de recepción de peticiones se usa para enviar las peticiones a la cola de JMS.
- Message-Driven Bean: Que se encarga de gestionar los mensajes de la cola.

B.1.2.1 Creación de cliente

Para la creación de clientes se necesitará utilizar las siguientes clases:

- *javax.naming.InitialContext*
- *javax.jms.QueueConnectionFactory*
- *javax.jms.Queue, javax.jms.QueueConnection*
- *javax.jms.QueueSession*
- *javax.jms.QueueSender*

Primero se crearán atributos en la clase cliente. Estos atributos son de las clases anteriormente indicadas:

```
InitialContext jndiContext = null;
QueueConnectionFactory queueConnectionFactory = null;
Queue queue = null;
QueueConnection conexion = null;
QueueSession sesion = null;
QueueSender mensajero = null;
```

Después se inicializarán. El atributo `jndiContext` de la clase `InitialContext` se utilizará para obtener el contexto inicial del directorio de nombres (JNDI) y se inicializa llamando al constructor por defecto:

```
jndiContext = new InitialContext();
```

Seguidamente se deberán obtener `QueueConnectionFactory` y `Queue`. `QueueConnectionFactory` modela las conexiones con la cola (`Queue`) de mensajes y se obtiene utilizando `jndiContext` de esta forma:

```
queueConnectionFactory = (QueueConnectionFactory)
jndiContext.lookup("jms/MyQueueConnectionFactory");
```

Se utiliza el método `lookup()`, el cual busca en el directorio de nombres el recurso, que en este caso se llama *jms/MyQueueConnectionFactory*, y obtiene la instancia. No es un nombre al azar, ya que debe haberse definido en el servidor de aplicaciones y registrado con el nombre indicado para que funcione. Para la cola de mensajes (`Queue`) se utilizará el mismo método:

```
queue = (Queue) jndiContext.lookup("jms/MyQueue");
```

Tanto la inicialización de `InitialContext` como las de `QueueConnectionFactory` y `Queue` pueden lanzar una excepción `NamingException`.

Para crear la conexión se utiliza la instancia de `QueueConnectionFactory` y con ella se llama a su método `createQueueConnection()`. Este método genera una instancia de la clase `QueueConnection` que se puede almacenar en la variable `conexion` inicializada con valor *null*.

```
conexion = queueConnectionFactory.createQueueConnection();
```

Se puede iniciar una sesión con la cola a través de esta conexión utilizando el método `createQueueSession(boolean transacted, int acknowledgeMode)` de la clase `QueueConnection`. El primer parámetro indica si el envío es con confirmación de recepción (*false*) o no (*true*). El segundo, indica el tipo de confirmación y será ignorado en caso de que el primer parámetro tenga valor *true*. Tipos de confirmación:

- **`Session.AUTO_ACKNOWLEDGE`**: asentimiento automático, cuando se recibe bien un mensaje de un cliente.
- **`Session.CLIENT_ACKNOWLEDGE`**: asentimiento de cliente, el cual llama al método de asentimiento del mensaje. Confirmando el mensaje del que se ha llamado el método de asentimiento, se confirman todos los mensajes que la sesión ha consumido.
- **`Session.DUPS_OK_ACKNOWLEDGE`**: asentimiento que permite duplicados. Indica a la sesión que confirme vagamente la entrega de mensajes. Los consumidores deben soportar mensajes duplicados.

La instancia obtenida se almacena en el atributo de la clase `QueueSession` (*sesion*) de la siguiente manera (utilizando, en este caso, una sesión con confirmación automática, que suele ser lo más común):

```
sesion = conexion.createQueueSession(false,
                                     Session.AUTO_ACKNOWLEDGE);
```

Para terminar de inicializar los elementos necesarios se obtiene una instancia de la clase `QueueSender` (variable `mensajero`). Dicha instancia se obtiene mediante el método de la clase `QueueSession`, `createSender` (`Queue queue`). Se le pasa por parámetro la instancia de la clase `Queue` (variable `queue`):

```
mensajero = sesion.createSender(queue);
```

La instanciación de `QueueConnectionFactory`, `QueueSession` y `QueueSender` puede lanzar una excepción del tipo `JMSException` que deberá ser capturada adecuadamente.

Finalmente, para terminar el código concerniente al cliente resta la creación y envío de un mensaje. Los mensajes son modelados por la interfaz `Message` (también del paquete `javax.jms`)

y contiene los métodos necesarios para construir, añadir propiedades o llamar a confirmación. Algunos métodos de esta interfaz que pueden ser de utilidad:

- **`void acknowledge()`:** confirma el mensaje con el que se llama al método y todos los anteriores (relacionado con el tipo de confirmación en modo cliente).
- **`void clearBody()`:** Borra el cuerpo del mensaje.
- **`get/setJMSDeliveryMode()`:** Obtiene o configura el modo de entrega (si es *set*, se le pasa un tipo *int* indicando el modo).
- **`get/setJMSDestination()`:** Obtiene o configura el destino (si es *set*, se le pasa una instancia de tipo *Destination*).
- **`get/setJMSTimestamp()`:** Obtiene o configura la marca de tiempo del mensaje (si es *set*, se indica con un tipo *long*).

Estos métodos pueden usarse tanto en envío, como en recepción.

Para una aplicación sencilla, la clase que puede modelar nuestro mensaje es una sub-interfaz llamada `TextMessage`. Los métodos que añade a de los que debe implementar por herencia son: `void setText(java.lang.String)` y `java.lang.String getText()`. En código:

```
TextMessage message = null;
message = sesion.createTextMessage();
message.setText(".............");
```

Se utiliza el método `createTextMessage()`, de la clase `QueueSession`, para generar la instancia de `TextMessage`. Una vez que se ha generado el mensaje con el contenido deseado, se procede al envío utilizando la instancia de `QueueSender` (variable `mensajero`). Para ello se utiliza el método `send(Message message)` de la clase `QueueSender`:

```
mensajero.send(message);
```

B.1.2.2 Creación de un Message-Driven Bean

Para modelar un Message-Driven Bean (EJB 3.0) se pueden utilizar diferentes IDEs. En este caso se utiliza Eclipse como referencia.

Pulsando con el botón derecho del ratón sobre la carpeta *ejbModule* de un proyecto EJB, se realiza lo que se indica en la Figura 61.

Una vez creado se completa la anotación `@MessageDriven` que aparece justo antes de la declaración de la clase, de esta manera:

```
@MessageDriven (activationConfig = { @ActivationConfigProperty(
                propertyName = "destinationType",
                       propertyValue = "javax.jms.Queue")},

                       mappedName = "jms/MyQueue",
                       messageListenerInterface = MessageListener.class)
```

Mediante ésta anotación de los EJB 3.0, se indica que cola de mensajes debe escuchar el MDB.

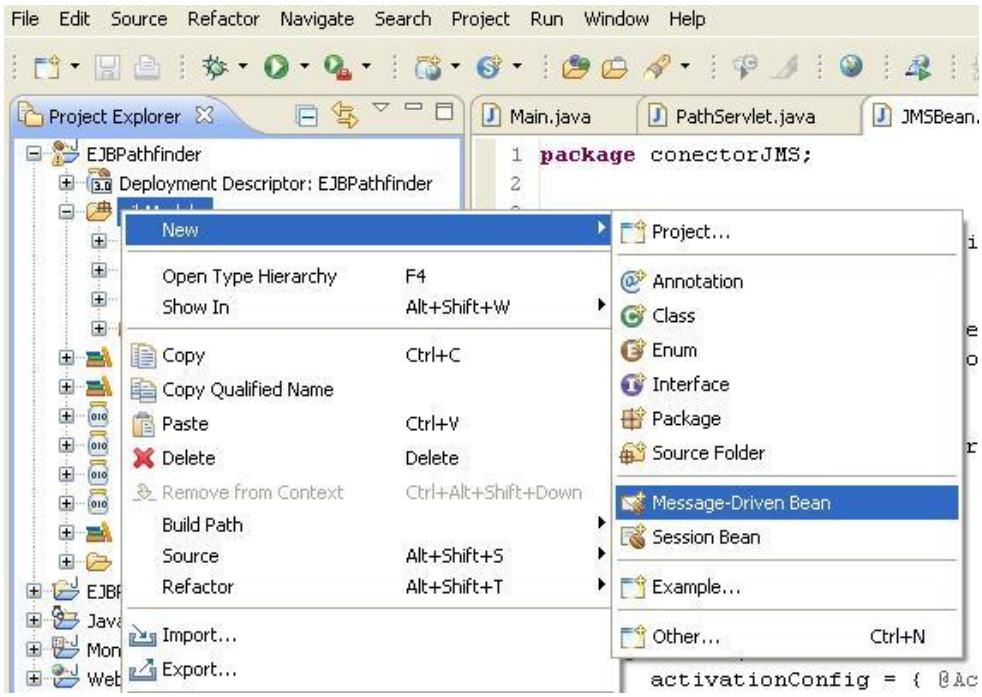

Figura 61. Creación de un MDB con Eclipse

El parámetro **`activationConfig`** indica la configuración del MDB en su entorno de operación. Esto incluye información sobre modos de asentimiento, destinos esperados o tipos de puntos finales (*endpoints*), entre otros. La configuración de éste parámetro se especifica usando un *array* de anotación del tipo `@javax.ejb.ActivationConfigProperty`, el cual especifica las propiedades mediante nombre (`propertyName`) y valor (`propertyValue`). En este caso, se indica que la propiedad a configurar es el tipo de destino y el valor de dicho destino es un objeto de la clase `javax.jms.Queue`.

El parámetro **`mappedName`** se utiliza para indicar el nombre JNDI del destino de mensajes del MDB. El tipo de datos que se introduce es un *String*, que en este caso toma el valor *jms/MyQueue*. Éste debe coincidir con el nombre de la cola obtenido del directorio de nombres en el cliente.

El parámetro **`messageListenerInterface`** indica la interfaz escuchadora de mensajes que utiliza MDB. Se debe utilizar si el MDB no implementa explícitamente la interfaz mensaje-escuchador, o si implementa más de una interfaz del tipo `java.io.Serializable`, `java.io.Externalizable` o cualquiera de las interfaces del paquete `javax.ejb`. El tipo de datos es `Object.class`.

Otros atributos de ésta anotación, no utilizados en este caso, son:

- ***name***: Especifica el nombre del MDB. Por defecto es el nombre no cualificado de la clase (*unqualified name*).
- ***description***: Especifica una descripción del MDB.

Ninguno de los cinco atributos indicados es obligatorio.

El MDB debe implementar las interfaces `MessageListener` y `MessageDrivenBean`:

```
public class JMSBean implements MessageListener, MessageDrivenBean {
```

Una vez hecho esto, se debe implementar el método de la interfaz `MessageListener` que manejará la recepción de los mensajes. Se le llamará automáticamente (cuando la aplicación esté desplegada en el servidor) al recibir en la cola un mensaje. Dentro de éste método se pueden utilizar los métodos de la clase `Message` y `TextMessage` para obtener la información necesaria.

```
public void onMessage(Message inMessage)
```

Un ejemplo del contenido del método `onMessage` es el que sigue:

```
TextMessage msg = null;
String datos = null;

try{
  //Si lo recibido es un mensaje de texto...
  if(inMessage instanceof TextMessage){
         //Se almacena
         msg = (TextMessage) inMessage;
         //Se pasa a String
         datos = msg.getText();
         System.out.println("MESSAGE BEAN: Mensaje recibido:
                              "+msg.getText());
  }else{
         System.out.println("Mensaje de tipo erroneo:
         "+inMessage.getClass().getName());
  }
}catch(JMSException e){
  e.printStackTrace();
}catch(Throwable te){
  te.printStackTrace();
}
```

Otros métodos que pueden aparecer al implementar `MessageDrivenBean` son:

```
@Override
public void ejbRemove()throws EJBException {}

@Override
public void setMessageDrivenContext(MessageDrivenContext arg0)throws
                                                        EJBException {}
```

Se pueden dejar vacíos, ya que pertenecen al ciclo de vida y contexto del MDB. De esta forma se deja que el contenedor de EJBs del servidor lo maneje a su antojo.

B.1.2.3 Configuración de los recursos JMS en el servidor Glassfish

Primero se debe entrar en la consola de administración del servidor Glassfish (versión utilizada 2.1.1). Normalmente se puede acceder introduciendo en cualquier navegador web la dirección http://localhost:4848 (se utiliza *localhost* si el servidor está instalado en la máquina local, si no se utiliza la dirección *ip*).

Una vez introducidos usuario y contraseña (se necesitan permisos de administrador), en la lista de la izquierda de la pantalla se selecciona *JMS Resources*. Está ubicado dentro de la pestaña *Resources*, como se muestra en la Figura 62.

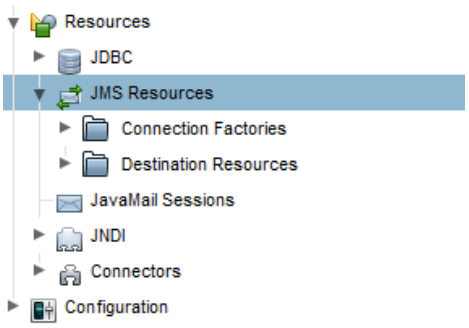

Figura 62. JMS Resources en Glassfish

Para configurar una instancia del tipo `QueueConnectionFactory` se selecciona la carpeta *Connection Factories*. Aparece una pantalla a la derecha de esta lista, que se puede ver en la Figura 63, en la que se pulsa el botón *new*.

Figura 63. Pantalla *JMS Connection Factories*

Una vez hecho esto, aparece otra pantalla con un formulario que en este caso se rellena como se indica en la Figura 64.

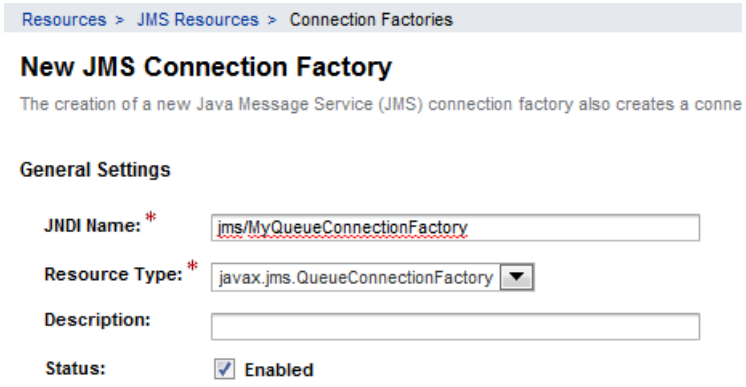

Figura 64. Formulario de creación de *QueueConnectionFactory*

En la Figura 64 se puede ver que se indica el nombre JNDI y el tipo de recurso. En este caso es *jms/MyQueueConnectionFactory* y `javax.jms.QueueConnectionFactory` respectivamente, ya que es el nombre del tipo `QueueConnectionFactory` que se utiliza en el ejemplo de la creación del cliente JMS y del MDB. Debe ser el mismo para que funcione.

Después se indica si las transacciones JMS serán locales o remotas de la forma que se muestra en la Figura 65 (en este caso locales).

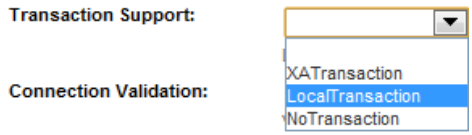

Figura 65. Tipo de transacciones JMS

Seguidamente se guardan los datos introducidos mediante un botón *ok* que se encuentra a la derecha de lo mostrado en la Figura 59.

Para configurar una instancia de `Queue` se pulsa la pestaña *Destination Resources*, debajo de *JMS Resources*. Aparece una pantalla a la derecha, mostrada en la Figura 66, en la cual, se pulsa el botón *new*.

Figura 66. Pantalla *Destination Resources*

Ésto muestra un formulario, mostrado en la Figura 67, en el que se puede introducir un nombre JNDI, en nuestro caso *jms/MyQueue*, un nombre de destino físico, en el que se introduce el nombre cualquier nombre, en nuestro caso *JMSBean* y el tipo de recurso, que será `javax.jms.Queue`. El nombre *JMSBean* hay que recordarlo pues se utiliza en otro formulario de configuración.

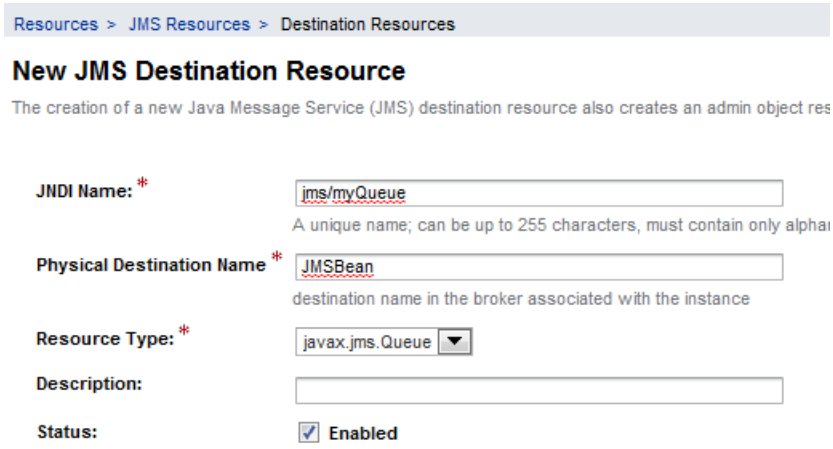

Figura 67. Formulario de creación de *Queue*

Seguidamente se guardan los datos introducidos mediante un botón *ok* que se encuentra a la derecha de lo mostrado en la Figura 62.

Se procede a pulsar en una pestaña llamada *Java Message Service*, dentro de la pestaña *Configuration*, tal y como se muestra en la Figura 68.

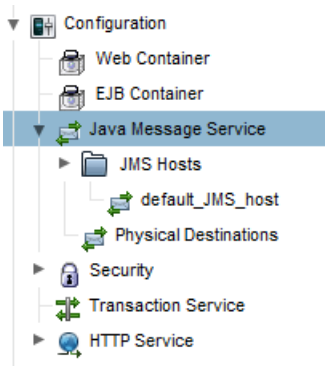

Figura 68. Pestaña de configuración de JMS

Aparece una pantalla con opciones configurables, en las que en este caso, hay que introducir los datos que se ven en la Figura 69.

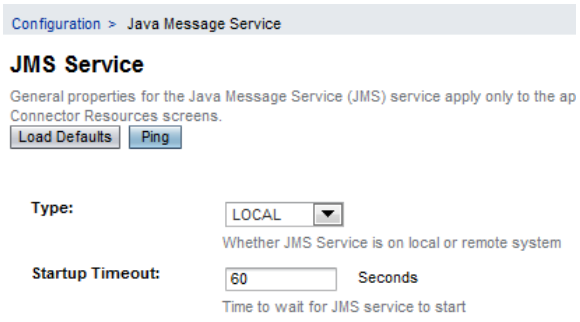

Figura 69. Configuración del servicio JMS del servidor

Después de guardar los datos anteriores, se pulsa en la pestaña *Physical Destinations*, aparece lo mostrado en la Figura 70 y se pulsa el botón new de la pantalla que aparece en el lado derecho.

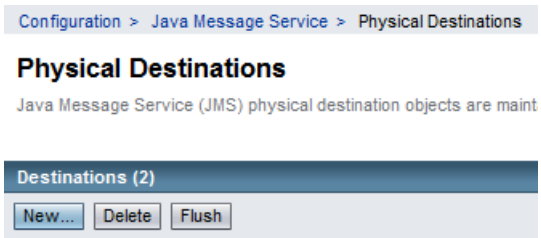

Figura 70. Pantalla de *Physical Destinations*

Se muestra un formulario que hay que rellenar de la forma que se indica en la Figura 71.

Figura 71. Formulario de *Physical Destination*

En este formulario es en el que hay que introducir el nombre del destino físico utilizado al crear la cola *JMSBean*. En el tipo de destino físico se indica `javax.jms.Queue`.

Para finalizar con la configuración se accede a la pestaña *EJB Container*, como se muestra en la Figura 72.

Figura 72. Pestaña *EJB Container*

El lado derecho de la pantalla entonces muestra tres pestañas. Se selecciona *MDB settings* y se muestra el formulario de configuración de la Figura 73.

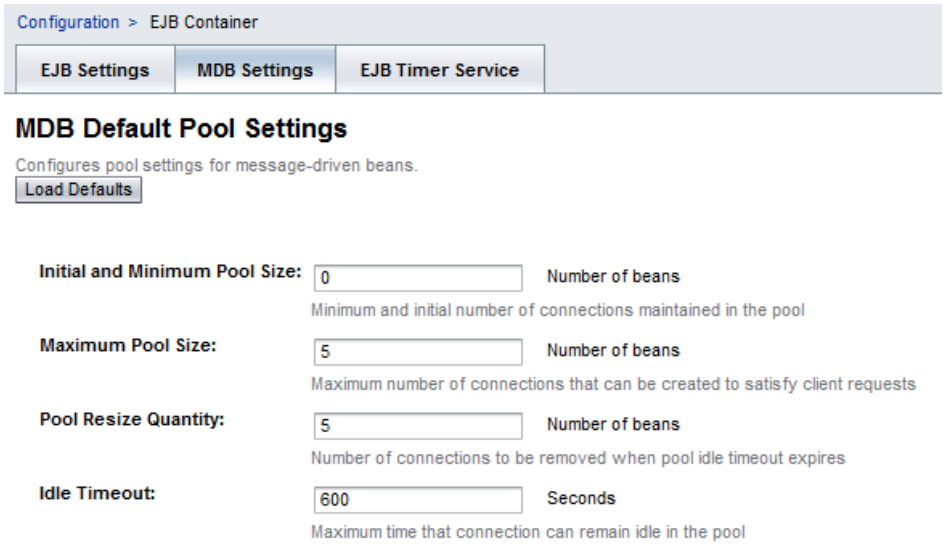

Figura 73. Opciones de MDB en Glassfish

En este formulario se permite configurar, entre otras propiedades, el número máximo de peticiones de la cola de mensajes que gestionará a la vez el MDB (opción *Maximum Pool Size*). Una vez configurado esto, ya debe funcionar la conexión JMS.

B.2 Extracción de archivos

Para la extracción de archivos comprimidos se han utilizado dos librerías de Java:

- `java.util.jar` ([15])
- `java.util.zip` ([14])

Primero se inicializan las instancias a utilizar en la extracción:

```
File archivo= new File ("Archivo.jar_o_Archivo.zip");
String directorio ="Directorio_de_extracción";
```

```
ZipFile azip = new ZipFile (archivo);
```

O si es un archivo *jar*:

```
JarFile ajar = new JarFile (archivo);
```

La instanciación tanto de `ZipFile` como de `JarFile` puede lanzar una excepción del tipo `IOException`. Se crean los flujos que permiten la lectura del archivo comprimido:

```
FileInputStream fis = null;
BufferedInputStream bis = null;
```

Para el caso de un archivo *zip* se incializaría de esta forma:

```
fis = new FileInputStream(azip.getName());
bis = new BufferedInputStream(fis);

ZipInputStream jis = new ZipInputStream(bis);
```

En el caso de un archive *jar*:

```
fis = new FileInputStream(ajar.getName());
bis = new BufferedInputStream(fis);

JarInputStream jis = new JarInputStream(bis);
```

Para ir obteniendo archivo a archivo de la instancia de `JarInputStream / ZipInputStream` se necesita un bucle que vaya leyendo de este flujo y almacenarlo en alguna variable. Para leer del flujo existen estos métodos:

- **Para *zip***: `ZipInputStream.getNextEntry()`, que devuelve una instancia del tipo `ZipEntry`.
- **Para *jar***: `JarInputStream.getNextJarEntry()`, que devuelve una instancia del tipo `JarEntry`.

Para escribir lo obtenido en un archivo en el directorio deseado, primero se debe crear un flujo del tipo `FileOutputStream` (variable *fos*), al que se le pasa un tipo *String* con la dirección completa del archivo de salida (utilizando la variable *directorio* inicializada al principio). Después se crea un *buffer* de escritura de datos de archivo. Éste debe ser del tipo `BufferedOutputStream`, al que se le debe pasar como parámetro el flujo mencionado y un entero que indique el tamaño del *buffer* (en este caso, 8192 es suficiente). Mediante un bucle se va leyendo los datos de la entrada, con un método de la clase `JarInputStream / ZipInputStream` , y escribiendo en el nuevo archivo con un método de `BufferedOutputStream`:

```
int count =0;
BufferedOutputStream dest = new BufferedOutputStream(fos, 8192);
while((count = jis.read(data, 0, 8192)) != -1 ){
  dest.write( data, 0, count );
}
```

La variable *data* es un *array* de tipo byte del mismo tamaño que el buffer de escritura de datos (8192).

El bucle completo es de esta forma:

```
int count = 0;
byte data[] = new byte[8192];
ZipEntry/JarEntry entry;

// Se van obteniendo las entradas del flujo
while((entry= jis.getNextEntry()/getMextJarEntry())!= null ){
  //Se genera el nombre del archive con su dirección completa
  //Aquí se incluye el directorio de extracción
  String destFN = directorio + File.separator + entry.getName();
  //Se genera un formato correcto de dirección del archivo
  if( destFN.indexOf("/") != -1 ){
        destFN = destFN.replace("/", File.separator);
  }

  //Si la entrada no es un directorio..
  if( !entry.isDirectory()){
        //Se guarda el archivo en la unidad
        //Primero se crea el flujo para la escritura
        FileOutputStream fos = null;
        try{
              fos = new FileOutputStream( destFN );

        }catch(FileNotFoundException fnf){
              /* En caso de que haya alguna carpeta intermedia que no se
               haya generado, se crea */
              int endIndex = destFN.lastIndexOf(File.separator);

              String aux = destFN.substring(0, endIndex);
              File auxf = new File (aux);
              auxf.mkdirs();
              // Y se crea el flujo
              fos = new FileOutputStream( destFN );
        }
        /* Se escribe el contenido del archivo el contenido
        del archivo */
        dest = new BufferedOutputStream( fos, 8192);
        while((count = jis.read(data, 0, 8192)) != -1 ){
              dest.write( data, 0, count );
        }
        //Se cierran los flujos de escritura de archivo
        dest.flush();
        dest.close();
        fos.flush();
        fos.close();
        // Si es un directorio, se crea
  }else{
        File f = new File(destFN);
        f.mkdirs();
  }
}
//Se cierran el resto de flujos
jis.closeEntry();
jis.close();
fis.close();
azip.close();
//Fin de extracción
```

Esto es todo lo necesario para extraer los archivos comprimidos que se utilizan en el proyecto.

B.3 Compilación de archivos

Para la compilación de archivos se utilizan dos librerías de java:

- java.util
- javax.tools

Primero, se inicializan las instancias necesarias para la compilación. Se necesita una instancia del tipo `JavaCompiler` ([13]), la cual manejará la compilación de los archivos. Para controlar los archivos a compilar se necesita una instancia del tipo `StandardJavaFileManager`. Finalmente, se utilizan *Strings* para almacenar tanto el directorio del código fuente, como el directorio donde se desean generar los archivos compilados:

```
public JavaCompiler compiler = null;

public StandardJavaFileManager fileManager = null;

public String directorioext = "Directorio_raiz_Código_fuente";

public String directorioclass = "Directorio_Clases";
```

Seguidamente se procede a instanciar tanto `JavaCompiler` como `StandardJavaFileManager` de esta forma:

```
compiler = ToolProvider.getSystemJavaCompiler();

fileManager = compiler.getStandardFileManager(null, null, null);
```

Se deben generar las carpetas donde se desea generar los archives compilados, utilizando para ello la variable `directorioclass` inicializada:

```
File directorio = new File(directorioclass);
directorio.mkdirs();
```

Para configurar correctamente la instancia del tipo `StandardJavaFileManager` se necesita indicar el tipo de archivos que va a controlar, ya que archivos Java pueden ser tanto archivos de código fuente, como archivos *.class*. Para esto, se necesita generar una instancia de tipo `Set<JavaFileObject.Kind>`, pues lo requiere la instancia de `StandardJavaFileManager`:

```
Set<JavaFileObject.Kind> theset = new HashSet<JavaFileObject.Kind>();
//Se añade el tipo .java
theset.add(JavaFileObject.Kind.SOURCE);
```

También se le indica a la instancia de `StandardJavaFileManager` donde se encuentran los archivos *.java* (puede lanzar `IOException`):

```
fileManager.setLocation(StandardLocation.SOURCE_PATH,
                            Arrays.asList(new File(directorioext)));
```

Una vez hecho esto, se crea una unidad de compilación que incluya los archivos configurados en la instancia de `StandardJavaFileManager`. Se utiliza el método `list()` de la misma clase, y se pasan como parámetros un `StandardLocation.SOURCE_PATH` , un tipo *String* con el nombre del paquete (en blanco en nuestro caso, pues no es necesario), la instancia de `Set<JavaFileObject.Kind>` (variable `theset`) y un tipo *boolean*, que indica si se desea que se compile, de forma recursiva, todos los subdirectorios de la dirección indicada (en este caso con valor *true*). Para la almacenar las unidades de compilación se utiliza un tipo `Iterable<? extends JavaFileObject>` (puede lanzar `IOException`):

```
Iterable<? extends JavaFileObject> compilationUnits1 =
        fileManager.list(StandardLocation.SOURCE_PATH,"",theset, true);
```

Se procede a crear una variable de tipo `Iterable<String>` con las siguientes opciones de compilación:

```
Iterable<String> opciones = Arrays.asList(new String[]
                            {"d",directorioclass,Xlint:unchecked"});
```

Si se desea se puede generar un archivo de registro en el que se obtenga el resultado de la compilación de esta forma (en nuestro caso un archivo llamado *compilationRes* en el directorio de las clases compiladas):

```
FileWriter ffwr = null;
File fr = new File(directorioclass+File.separator+"compilationRes");

try {
  fr.createNewFile();
  fr.setWritable(true);
  ffwr = new FileWriter(fr);
} catch (IOException e) {
  e.printStackTrace();
}
```

Finalmente, para obtener la tarea de compilación, se utiliza el método `getTask()` de `JavaCompiler` . Este método tiene como parámetros:

- **`Writer`**: Flujo de salida del resultado de compilación (en este caso `FileWriter`, variable `ffwr`).
- **`JavaFileManager`** (variable `fileManager`).
- **`DiagnosticListener<? super JavaFileObject>`** (*null* en este caso).
- **`Iterable<String>`**: Las opciones de compilación (variable `opciones`).
- **`Iterable<String>`** (*null* en este caso).
- **`Iterable<? extends JavaFileObject>`**: Unidades de compilación (variable `compilationUnits1`).

En la misma línea que se genera la tarea, se puede lanzar la ejecución de la compilación llamando al método `call()` de esta forma:

```
b = compiler.getTask(ffwr, fileManager, null, opciones, null,
                                                compilationUnits1).call();
```

Ya solo resta cerrar los flujos que se hayan mantenido abiertos y la compilación habrá terminado.

```
try {
  ffwr.close();
  fileManager.close();
} catch (IOException e) {
  e.printStackTrace();
}
```

B.4 Ejecución

Para poder ejecutar una aplicación recibida, con muchos archivos *.class* es necesario saber qué archivo contiene el método `main()`. Para ello se utiliza la clase `java.lang.ClassLoader`. La idea es cargar cada una de las clases generadas por el módulo de compilación e ir buscando dicho método.

```
ClassLoader nuevoLoader= null;
```

Para inicializar la instancia de la clase `java.lang.ClassLoader`, se utiliza un método estático de la clase `com.sun.enterprise.loader.ClassLoaderUtils` llamado `getClassLoader()`, el cual puede lanzar `IOException`. Los parámetros de este método son:

- **Tipo `File[]`:** Con los directorios donde se encuentran los archivos *.class* donde buscar.
- **Tipo `File[]`:** Con los directorios con archivos *jar*.
- **Tipo `ClassLoader`:** La instancia padre del `ClassLoader` utilizado.

En este caso se analiza un directorio cada vez por cuestiones de implementación del resto de los módulos. Cada iteración de directorio se puede implementar de esta forma:

```
File fi[] =new File[1];
fi[0]= directorio;

nuevoLoader = ClassLoaderUtils.getClassLoader(fi, null, null);
```

Una vez obtenida la instancia de `ClassLoader` se va obteniendo clase por clase del directorio utilizando el método `loadClass()` de `ClassLoader`, que tiene como parámetro el nombre de la clase a cargar:

```
Class<?> clase = null;
try{
  clase = nuevoLoader.loadClass("Nombre de la clase"));

} catch (ClassNotFoundException e1) {
  e1.printStackTrace();
}
```

Para buscar en la clase cargada se puede utilizar el método `getDeclaredMethods()` de la clase `java.lang.Class<?>`, el cual devuelve un *array* de la clase `java.lang.reflect.Method`. Utilizando un bucle se puede ir buscando el método `main()` de esta forma:

```
Method [] metodo = clase.getDeclaredMethods();
int cont = 0;
boolean haymain = false;

while(cont < metodo.length){
  String name = metodo[cont].getName();
  if(name.equals("main")){
        haymain = true;
        cont = metodo.length;
  }else{
        cont++;
  }
}
```

Para ir obteniendo los nombres de los métodos se utiliza el método `getName()` de la clase `Method`.

Una vez obtenida la clase con método `main()` se podrá utilizar en la ejecución JPF.

B.4.1 Implementación de la ejecución de JPF

En el caso de este proyecto, la ejecución de JPF se lanzará desde una clase Java, pero no de forma embebida, tal y como se indica en el API. La razón es que la aplicación JPF lanza una excepción *JPF out of memory* al implementarlo de forma embebida al lanzarlo desde un *Session Bean*.

La opción por la que se ha optado es lanzar la aplicación JPF de línea de comandos. Para lanzar cualquier aplicación de línea de comandos (en este caso del S.O. Windows), se utiliza la clase `java.lang.Runtime`.

Primero, para obtener la instancia de `Runtime`, se utiliza el método estático `getRunTime()` de la misma clase, el cual, devuelve el tipo `Runtime` en el que se encuentra ejecutando la aplicación.

Para lanzar un comando se utiliza el método `exec()` de la instancia de `Runtime`, al que se le pasa por parámetro un tipo *String* con el comando que se desea ejecutar. Este método devuelve una instancia del tipo `java.lang.Process`:

```
Process p = Runtime.getRuntime().exec(cmd);
```

Si se desea obtener la salida del proceso ejecutado, se crea una instancia de la clase `java.io.BufferedReader`. A su constructor se le pasa una instancia de `java.io.InputStreamReader`, generada previamente pasándole a su constructor la instancia `java.io.InputStream` obtenida del proceso mediante el método `getInputStream()` de la clase `Process`:

```
BufferedReader in = new BufferedReader( new
                                    InputStreamReader(p.getInputStream()));
```

Seguidamente, mediante un bucle, se va llamando al método `readLine()` de la instancia de `BufferedReader`, el cual, devuelve un tipo *String* con el que se puede hacer lo que se desee (al terminar de utilizar el flujo de lectura, habrá que cerrarlo con el método `close()`):

```
String line = null;
System.out.println("<exe> Proceso:");
while ((line = in.readLine()) != null) {
  System.out.println(line);
}
System.out.println("<exe> Fin del proceso");
in.close();
```

Para esperar a que acabe la ejecución se utiliza el método `waitFor()` de la instancia de `Process`, que devuelve un entero con el resultado de la ejecución (0 si termina bien, 1 si hay algún error):

```
int res = p.waitFor();
```

Finalmente, para asegurarse de que el proceso ha terminado y se han liberado los recursos, se puede destruir el proceso mediante el método `destroy()` de la instancia de `Process`:

```
p.destroy();
```

El código clave es éste:

```
try {
  Process p = Runtime.getRuntime().exec(cmd);
  BufferedReader in = new BufferedReader( new
                                    InputStreamReader(p.getInputStream()));
  String line = null;
  System.out.println("<exe> Proceso:");
  while ((line = in.readLine()) != null) {
        System.out.println(line);
  }
  System.out.println("<exe> Fin del proceso");
  in.close();

  res = p.waitFor();
  p.destroy();

} catch (IOException e) {
  e.printStackTrace();
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

La instancia de tipo *String* que contiene el comando que se desea ejecutar (y sus argumentos) debe tener éste formato para un S.O. Windows:

```
String cmd = "cmd /C comando argumento1 argumento2";
```

B.5 Envío de correos

En el proyecto se necesita enviar un correo al terminar la validación. Para este fin se utiliza la librería `javax.mail` ([12]). Para enviar un correo se necesita una cuenta de correo, y un servidor de correo que lo entregue.

Para solucionar esto, en este caso, se ha utilizado *gmail.com.* Se ha creado una cuenta de correo para nuestra aplicación, cuya dirección es *javapathfinder.uc3m@gmail.com*. De esta forma, teniendo una cuenta *gmail* el servidor permitirá enviar correos a cualquier dirección.

Para enviar un correo, lo primero es configurar el envío para que conecte con el servidor de *gmail*. Para esto se crea una instancia de la clase `java.util.Properties`:

```
java.util.Properties props = new java.util.Properties();
```

Para ir añadiendo propiedades se utilize el método `put()` de la clase `Properties`, al que se le pasan 2 parámetros:

- **`String propiedad`:** Indica el nombre de la propiedad a configurar.
- **`String valor`:** Indica el valor de la propiedad a configurar.

Para la correcta configuración de la conexión con *gmail*, éstas son todas las propiedades necesarias:

```
props.put("mail.smtp.host", "smtp.gmail.com");
props.put("mail.smtp.port", "587");

props.setProperty("mail.smtp.starttls.enable", "true");
props.setProperty("mail.smtp.user","javapathfinder.uc3m@gmail.com");
props.setProperty("mail.smtp.auth", "true");
```

Con estas propiedades se configura una sesión, es decir, se crea una instancia de la clase `javax.mail.Session` utilizando el método estático `getDefaultInstance()` de `Session`:

```
Session session = Session.getDefaultInstance(props, null);
```

Se le pasa como parámetros el objeto que contiene las propiedades y una instancia de la clase `Authenticator` que en este caso no es necesaria, por lo que se pasa un valor *null*.

Después se construye el mensaje. Se crea una instancia de la clase `javax.mail.Message` utilizando el constructor `MimeMessage()` al que se le pasa la instancia que contiene la sesión.

```
Message msg = new MimeMessage(session);
```

Seguidamente, se van incluyendo en el mensaje los campos necesarios para el envío de esta forma:

```
msg.setFrom(new InternetAddress("JPFPathfinder@it.uc3m.es"));

msg.setRecipient(Message.RecipientType.TO, new
                                    InternetAddress("MAIL_DE_DESTINO"));
msg.setSubject("ASUNTO");

msg.setText("CONTENIDO");
```

Para el envío de este mensaje se crea una instancia del tipo `javax.mail.Transport` mediante el método `getTransport()` de la clase `Session` que recibe como parámetro el tipo de protocolo de transporte, que en este caso es *smtp*.

```
Transport t = session.getTransport("smtp");
```

Utilizando los datos de acceso a nuestra cuenta de correo, se lanza una conexión con el método `connect()` de la clase `Transport`.

```
t.connect("javapathfinder.uc3m@gmail.com","jpf12345");
```

Finalmente se llama al método `sendMessage()` de la clase `Transport`, con parámetros:

- **`Message mensaje`**: la instancia de la clase `Message` generada.
- **`Address[] direcciones`***:* *array* con las direcciones de destino del mensaje. Se puede obtener mediante el método `getAllRecipients()` de la clase `Message`.

```
t.sendMessage(msg,msg.getAllRecipients());
```

Y se cierra la conexión:

```
t.close();
```

B.6 Aspectos de la implementación del cliente de consola

B.6.1 Conexión con el servidor

Para la conexión del cliente de consola con el servidor se utilizarán peticiones *http*. Estas peticiones se enviarán al mismo *servlet* de recepción que se utiliza en el acceso web. Para su implementación se utilizan las librerías `java.io` y las siguientes del paquete `org.apache.commons` ([22]):

- `org.apache.commons.httpclient`
- `org.apache.commons.httpclient.methods`
- `org.apache.commons.httpclient.methods.multipart`

Primero se crea una instancia de la clase `org.apache.commons.httpclient.HTTPClient`, para poder hacer el envío.

```
HttpClient client = new HttpClient();
```

Seguidamente, se crea una instancia de la clase `org.apache.commons.httpclient.methods.PostMethod` mediante su constructor, el cual recibe como parámetro la dirección web del *servlet* de recepción:

```
PostMethod mPost = new
        PostMethod("http://"+args[2]+":8080/WebPathfinder/PathServlet");
```

Para añadir parámetros a la petición, se utiliza un *array* de tipo `org.apache.commons.httpclient.methods.multipart.Part`. Los parámetros que se añaden a dicho *array* pueden ser de diferentes subtipos. Los que se han utilizado son:

- **`StringPart`***:* para tipos *String*.
- **`FilePart`***:* para tipos *File*.

Los constructores de estos subtipos tienen la misma estructura. Su primer parámetro es el nombre mediante el cual se podrá acceder a la instancia a enviar, y el segundo es la propia instancia a enviar:

```
Part[] parts={ new StringPart("NIA", args[0]),
               new StringPart("correo", args[1]),
               new StringPart("x", opcion),
               new StringPart("interna", propInternas),
               new StringPart("conf",confAdicional),
               new StringPart("consola","true"),
               new FilePart("archivo", f)};
```

Una vez establecidos los parámetros, se configura la petición utilizando el método `setRequestEntity`() de la clase `PostMethod`. A éste método, se le pasa una instancia de la clase `MultipartRequestEntity` inicializada con su constructor. Dicho constructor recibe como parámetros el *array* de tipo `Part` y una instancia de la clase `HTTPMethodParams`, obtenida mediante la llamada al método `getParams()` de la clase `PostMethod`. El código correspondiente es el que sigue:

```
mPost.setRequestEntity(new MultipartRequestEntity(parts,
                                                  mPost.getParams()));
```

Finalmente se procede al envío de la petición, mediante la llamada al método `executeMethod()` de la clase `HTTPClient`. Este método recibe como parámetro la instancia de `PostMethod` (variable `mPost`), configurada ya correctamente, y devuelve un entero indicando el resultado de la petición. Dicho método, además, puede lanzar las excepciones `IOException` y `HttpException`:

```
int status = client.executeMethod(mPost);
```

B.6.2 Obtención de respuesta

Para obtener el resultado de la petición se utiliza el método `getResponseBodyAsStream()` de la clase `PostMethod`, el cual, devuelve un flujo del tipo `java.io.InputStream`.

```
InputStream ist = mPost.getResponseBodyAsStream();
```

Con esta instancia se crea una instancia de la clase `java.io.InputStreamReader`, pues su constructor la recibe como parámetro.

```
InputStreamReader istr = new InputStreamReader(ist);
```

Seguidamente se genera una instancia de la clase `java.io.BufferedReader` pasándole al constructor la instancia de la clase `InputStreamReader` de esta forma:

```
BufferedReader br = new BufferedReader(istr);
```

Utilizando un bucle, se va leyendo del flujo la respuesta a nuestra petición mediante el método `readLine()` de la clase `BufferedReader`:

```
String resultado = "";
String aux= null;
while ((aux=br.readLine()) != null){
  resultado=resultado+""+aux;
}
```

Finalmente, es necesario cerrar todos los flujos que se han utilizado:

```
br.close();
istr.close();
ist.close();
```

B.7 Organización de Archivos de la aplicación

B.7.1 Sistema de archivos

Todos los archivos que utiliza la aplicación del proyecto están en una carpeta en la *C:\Pathfinder*. Dentro de esa carpeta hay otras seis carpetas y dos archivos como se puede ver en la Figura 74:

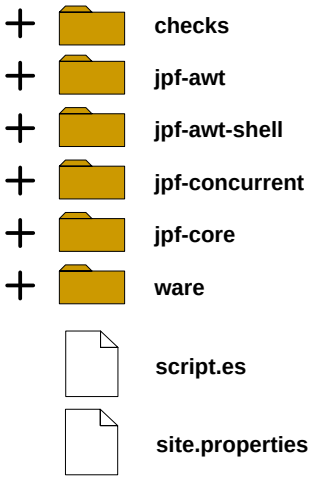

Figura 74. Directorios y archivos de la aplicación

Directorios:

- ***checks*:** Directorio en el que se almacenan los archivos que sube el administrador desde el cliente de consola.
- ***jpf-awt*:** Directorio de la extensión de JPF que modela los paquetes `java.awt` y `javax.swing`.
- ***jpf-awt-shell*:** Directorio de la extensión de JPF que modela los paquetes `java.awt` y `javax.swing`. Implementa la funcionalidad de la consola JPF para dichos paquetes.
- ***jpf-concurrent*:** Directorio de la extensión de JPF que modela el paquete `java.util.concurrent`.
- ***jpf-core*:** Directorio del nucleo de la aplicación JPF.
- ***ware*:** Directorio en el que se almacenan temporalmente los archivos a validar por JPF (tanto comprimidos como clases compiladas).

Archivos:

- ***script.es*:** *script* vacio que toma por defecto la aplicación, cuando valida aplicaciones con librerías `java.awt` o `javax.swing`. Sólo se utiliza en caso de que el usuario no haya enviado su propio *script* dentro del archivo comprimido.
- **site.properties:** Archivo de propiedades que controla las extensiones utilizadas por el núcleo JPF.

B.7.2 Estructura de archivos comprimidos

Los archivos comprimidos que se envíen a la aplicación deben tener el formato mostrado en la Figura 75, aunque no todos los archivos no son necesarios:

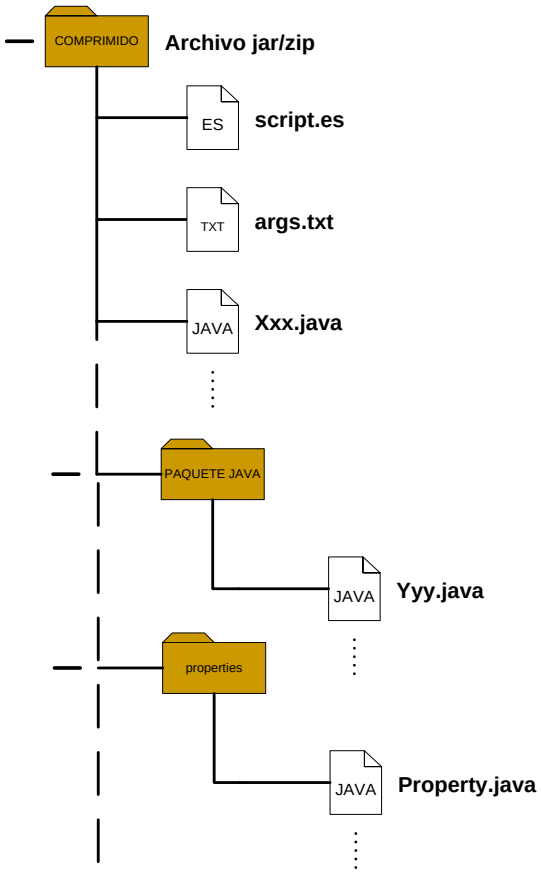

Figura 75. Estructura de archivos comprimidos

El archivo ***script.es*** no es obligatorio. Sólo se utiliza en caso de que la aplicación utilice las librerías `java.awt` y `javax.swing`. En esos casos, este *script* sirve para indicar una serie de instrucciones que simularían la entrada de un usuario, en la interfaz gráfica de la aplicación a analizar. Si no es necesaria ningún tipo de entrada, se omite el archivo.

El archivo ***args.txt*** contiene los argumentos que se le necesiten pasar a cualquier aplicación que los requiera al lanzar su ejecución. Tampoco es obligatorio ya que no todas las aplicaciones necesitan argumentos de entrada.

La carpeta **properties** contendrá los *Listeners* que se utilizarán en caso de que se incluya ya que no es obligatorio.

El resto de archivos serán **archivos *java* o paquetes con archivos *java***, en caso de que éstos pertenezcan a un paquete.

APÉNDICE C. Aplicaciones de prueba

En esta sección se muestra el código de las aplicaciones utilizadas en la validación empírica. Estas son:

- Aplicación con el API de `java.util.concurrent` (Sección C.1)
- Aplicación sin el API de `java.util.concurrent` (Sección C.2)

C.1 Aplicación con API `java.util.concurrent`

```
import java.util.concurrent.*;

/* Clase Santa que hereda de la clase Thread */
public class Santa extends Thread{

        // variable compartida por los hilos de la clase Reno
        private int contadorRENOS;
        // variable compartida por los hilos de la clase Duende
        private int contadorDUENDES;
        // booleano que comprueba si Santa está repartiendo los regalos
        private boolean ocupado;

        // semáforo que controla el acceso al contador de duendes
        private Semaphore accesoDUENDE = new Semaphore(20);
        // semáforo que controla el acceso al contador de renos
        private Semaphore accesoRENO = new Semaphore(1);
        // semáforo que espera que Santa haya solucionado los problemas
        private Semaphore problemas = new Semaphore(0);
        // semáforo que espera que Santa dé permita a los renos irse
         private Semaphore vacaciones = new Semaphore(0);
        // semáforo que despierta a Santa en los casos que debe trabajar
        private Semaphore santa = new Semaphore(0);

        // constructor de la clase Santa
  public Santa (){
        ocupado= false;
        contadorRENOS = 0;
        contadorDUENDES = 0;
  }

  // métodos que devuelven los semáforos creado
  public Semaphore getaccesoDUENDE(){return accesoDUENDE;}

  public Semaphore getaccesoRENO(){return accesoRENO;}

  public Semaphore getProblemas(){return problemas; }

  public Semaphore getVacaciones(){return vacaciones;}

  public Semaphore getSanta() {return santa;}

  // métodos que permiten trabajar con las variables
  public int getRENOS(){return contadorRENOS;}

  public int getDUENDES(){return contadorDUENDES;}

  public void setDUENDES (int duendes){contadorDUENDES = duendes;}

  public void setRENOS (int renos){contadorRENOS = renos;}
```

```
public boolean getOCUPADO(){return ocupado;}

// método que hace las principales funciones de Santa
public void run(){

	while ( true){

		try{
			System.out.println("Santa: Me voy a dormir");
			santa.acquire();
			System.out.println("\nSanta: Me han despertado");
			if(contadorRENOS == 9){
				ocupado = true;
				System.out.println("Santa: Estoy colocando el
							trineo para empezar a
							repartir los regalos");

				// Se añade el retardo correspondiente
				try{
					Thread.sleep(100);
				}catch( Exception e){e.printStackTrace();}

				System.out.println("Santa: Ya se han
							repartido todos los regalos");
				for(int k = 0; k < 9 ; k++){
					vacaciones.release();
				}
				ocupado = false;
			}

			if(contadorDUENDES >= 3){
				ocupado = true;
				System.out.println("Santa: Estoy solucionando
							los problemas de produccion");

				while(contadorDUENDES >=0){
					System.out.println("Santa: aun
							me quedan por solucionar"
					+(contadorDUENDES + 1) + " problemas");

					contadorDUENDES--;
					problemas.release();
					getaccesoDUENDE()).release();
				}
				ocupado = false;
			}
		}catch(Exception e){
			System.out.println("error en el sistema");
		}
	}
}

//main
public static void main(String s[]){

	try{
		System.out.println("# INICIALIZANDO EL PROGRAMA #");
		//Se crea el hilo Santa
		Santa santa = new Santa();
		//Se arranca la ejecución del hilo
		santa.start();

		//se crean los renos y se lanza su ejecución
		for(int i = 1 ; i <= 9 ; i++){
			Reno reno = new Reno (santa, "reno" + i);
			reno.start();
		}
```

```
                //Se crean los duendes y se lanza su ejecución
                for(int j = 1 ; j <= 20 ; j++){
                        Duende duende = new Duende(santa,"duende "+j);
                        duende.start();
                }
        }catch(Exception e){}

  }// fin main

}// fin clase Santa

/* Clase Reno que hereda de la clase Thread */
class Reno extends Thread{

  // atributos de la clase
  private Santa claus;
  private String nombre;

  // constructor
  public Reno(Santa claus, String nombre){
        this.claus = claus;
        this.nombre = nombre;
  }

  /*  método que hace las principales funciones de los renos:
        repartir los regalos y estar de vacaciones */
  public void run(){

        while(true){

                try{
                        Thread.sleep(12000);
                }catch(Exception e){e.printStackTrace();}

                try{
                        (claus.getaccesoRENO()).acquire();
                        claus.setRENOS(claus.getRENOS() +1);
                        System.out.println(nombre + ": He vuelto de
                                                                vacaciones");
                        if(claus.getRENOS() == 9){
                                (claus.getSanta()).release();
                        }
                        (claus.getaccesoRENO()).release();
                        (claus.getVacaciones()).acquire();
                        (claus.getaccesoRENO()).acquire();
                        claus.setRENOS(claus.getRENOS() -1);
                        System.out.println(nombre + ": Me voy de
                                                                vacaciones");
                        (claus.getaccesoRENO()).release();

                }catch(Exception e){e.printStackTrace();}
        }
  }
}//Fin de Reno

/* Clase Duende que hereda de la clase Thread */
class Duende extends Thread{

  // atributos de la clase
  private Santa claus;
  private String nombre;
  private int contador;

  // constructor de la clase
  public Duende(Santa claus,String nombre){
        this.nombre = nombre;
        this.claus = claus;
  }
```

```
  /*  método que hace las principales funciones de los duendes:
        controlar la producción de juguetes */
  public void run(){

        while(true){

              try{
                    Thread.sleep(2000);
              }catch(Exception e){e.printStackTrace();}

              try{
                    System.out.println(nombre + ": Problema en la
                                                cadena de produccion");

                    (claus.getaccesoDUENDE()).acquire();
                    contador = claus.getDUENDES() + 1;
                    claus.setDUENDES(contador);

                    if(claus.getDUENDES()== 3 &&! (claus.getOCUPADO())){
                          (claus.getSanta()).release();
                    }
                    (claus.getProblemas()).acquire();

                    System.out.println(nombre + ": Se ha corregido mi
                                                fallo en la produccion");

              }catch(Exception e){e.printStackTrace();}
        }
  }
}//Fin de Duende
```

Figura 76. Código de la aplicación de prueba con API de `java.util.concurrent`

C.2 Aplicación sin API `java.util.concurrent`

```
public class CuentaCorriente {

  String usu; // Nombre del usuario
  String clave; // Clave del usuario
  //Variable compartida
  protected int saldo; // Saldo disponible en la cuenta

  public CuentaCorriente(String nombre, String clav){
        usu = nombre;
        clave = clav;
  }

  public boolean ingresa(int cant){
        saldo = saldo + cant;
        return true;
  }

  public boolean reintegrar(int cant){
        if (cant<= saldo){
              saldo = saldo - cant;
              return true;
        }else
              return false;
  }

  public int saldoDisponible() {
        return saldo;
  }
}
```

```
/* Hilo Ahorrador */
public class AhorradorH extends Thread{

  private String nombre;
  private CuentaCorriente cuenta;//Cerrojo
  private int ingresos;

  public AhorradorH(String nombr, CuentaCorriente cuent){
        nombre = nombr;
        cuenta = cuent;
        ingresos = 0;
  }

  public void run(){

        try {
              while (true) {
                    synchronized(cuenta) {
                          cuenta.ingresa(1000);
                          System.out.println("Ahorrador(" + nombre +
                                "): Ahorro "+ 1000 +". Quedan: "
                                + cuenta.saldoDisponible());
                          cuenta.notifyAll();
                    }
                    ingresos += 1000;
                    sleep(5000);
              }
        }catch(Exception e) {}
  }
}

/* Hilo Gastador*/
public class GastadorH extends Thread{

  private String nombre;
  private CuentaCorriente cuenta;
  private int acumulado;

  public GastadorH(String nom, CuentaCorriente cuent){
        nombre = nom;
        cuenta = cuent;
        acumulado = 0;
  }

  public void run(){

        int extraccion;

        try {
              while (true) {
                    for (int i = 0; i<4; i++) {
                          int c = 0;
                          synchronized(cuenta) {
                                if(cuenta.saldoDisponible()< 2){
                                      cuenta.wait();
                                }
                                c = cuenta.saldoDisponible()/2;
                                extraccion = new
                                Integer(c).intValue();
                                cuenta.reintegrar(extraccion);
                                System.out.println ("Derrochador(" +
                                      nombre + ")Saco " + extraccion +
                                      ". Quedan: " +
                                            cuenta.saldoDisponible());
                                if(cuenta.saldoDisponible()< 2){
                                      cuenta.wait();
                                }
```

```
                        }
                        acumulado = acumulado + extraccion;
                        sleep(1000);
                    }
                    extraccion = cuenta.saldoDisponible()*2;

                    synchronized(cuenta) {
                        cuenta.reintegrar(extraccion);
                        System.out.println("No hay dinero
                            suficiente para sacar
                            "+extraccion+".");
                    }
                    sleep(1000);
                }
            }catch (Exception e) {}
    }
}

/*Clase que lanza los hilos*/
public class GestionCuenta {

    public static void main (String arg[]){

        CuentaCorriente cuenta = new CuentaCorriente("login","password");
        for( int i=0; i<4; i++){
            GastadorH hiloG = new
            GastadorH("Gastador_"+i,cuenta);
            hiloG.start();
        }

        for( int p=0; p<1; p++){
            AhorradorH hiloA = new AhorradorH("Ahorrador_"+p,cuenta);
            hiloA.start();
        }
    }
}
```

Figura 77. Código de la aplicación de prueba sin API de `java.util.concurrent`

APÉNDICE D. Instalación y configuración de herramientas

En esta sección de los apéndices se muestra cómo instalar y configurar las herramientas que se han utilizado durante la implementación del proyecto. Estas son:

- JDK 6 (Sección D.1)
- Servidor Glassfish (Sección D.2)
- IDE Eclipse (Sección D.3)

D.1 Instalación JDK 6

Para instalar el Java Development Kit (JDK) 6 de Java ([10]), únicamente se debe ir a la página web de descargas de Oracle:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Y así obtener un instalador válido para nuestra versión de sistema operativo.

Normalmente se instalará en *C:\Archivos de Programa\Java\jdk1.6.0_XX* donde XX será el número de actualización.

Para configurar Java en nuestro equipo se deben modificar ciertas variables de entorno. Para esto, se accede a *Equipo* (*Mi PC* en Windows XP) y con el botón derecho del ratón se hace *click*. En el menú que aparece se selecciona *propiedades* y aparecerá una ventana. En Windows Vista y 7, en el lateral izquierdo, aparecerá la opción *Configuración avanzada del sistema*. Al pulsar dicha opción, aparece una ventana con pestañas (en el caso de XP, aparecerá esta pantalla después de hacer *click* con el ratón). Se selecciona entonces la pestaña *Opciones avanzadas* y se pulsa un botón llamado *Variables de Entorno*.

En la pantalla de *Variables de Entorno*, aparecerán dos cuadros, el superior es para variables de usuario, y el inferior es para variables del sistema.

En las variables de usuario se crea una nueva, llamada JAVA_HOME, con valor "*C:\Archivos de Programa\Java\jdk1.6.0_XX"*.

En las variables del sistema se crea una que se llame CLASSPATH, con valor "*.;"*.(en caso de que exista, se pone dicho valor al principio). Esta variable indica dónde debe buscar los archivos *.class* la aplicación Java.

Finalmente, para poder utilizar Java desde línea de comandos, se incluye en la variable PATH el "*C:\Archivos de Programa\Java\jdk1.6.0_XX\bin"*. Si hay más valores, para separarlos de los otros se utiliza ";".

D.2 Instalación de Glassfish

Como la versión de Glassfish utilizada en el proyecto es la 2.1.1, primero hay que bajar el instalador de esa versión. La dirección web de donde descargarlo es ésta:

http://glassfish.java.net/public/downloadsindex.html#top

En dicha página se obtiene un archivo *glassfish-installer-v2.1.1-windows-ml.jar* ejecutable. Y se procede de la siguiente manera:

1. Se ejecuta esto en línea de comandos para extraer los archivos:

```
>java -Xmx256m -jar filename.jar
```

2. Se accede directorio donde se haya descargado:

```
>cd Glassfish
```

3. Y para la instalación se utiliza la herramienta *Ant* que trae dentro el propio Glassfish. Así:

```
>lib\ant\bin\ant -f setup.xml
```

Ahora, para arrancarlo, solo habrá que ir al directorio */bin*, dentro del directorio raíz de Glassfish (en la consola del sistema) y ejecutar este comando:

```
>asadmin start-domain domain1
```

Y para pararlo:

```
>asadmin stop-domain domain1
```

D.3 Instalación de Eclipse

Para instalar Eclipse ([24]), primero se debe bajar el instalador del programa de esta página:

http://www.eclipse.org/downloads/

En el caso del proyecto se utiliza la versión para programación de aplicaciones Java EE. El archivo que se descarga para Windows es un ejecutable sencillo de instalar.

D.3.1 Configurar Glassfish dentro de Eclipse

Para configurar un servidor Glassfish dentro de Eclipse, para poder probar aplicaciones, primero se selecciona la pestaña *Servers* de la pantalla inferior del programa. En el fondo de la pantalla de la pestaña se pulsa con el botón derecho del ratón y se hace click en *New* y seguidamente en *server*, como se ve en la Figura 78.

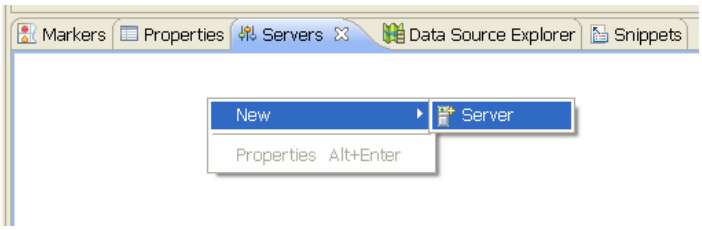

Figura 78. Pestaña *Servers* de Eclipse

Seguidamente, aparecerá una pantalla, mostrada en la Figura 79, en la que se podrá seleccionar el tipo de servidor a instalar. Si no aparece en la lista, se baja pulsando en el enlace situado en la parte superior derecha de la ventana.

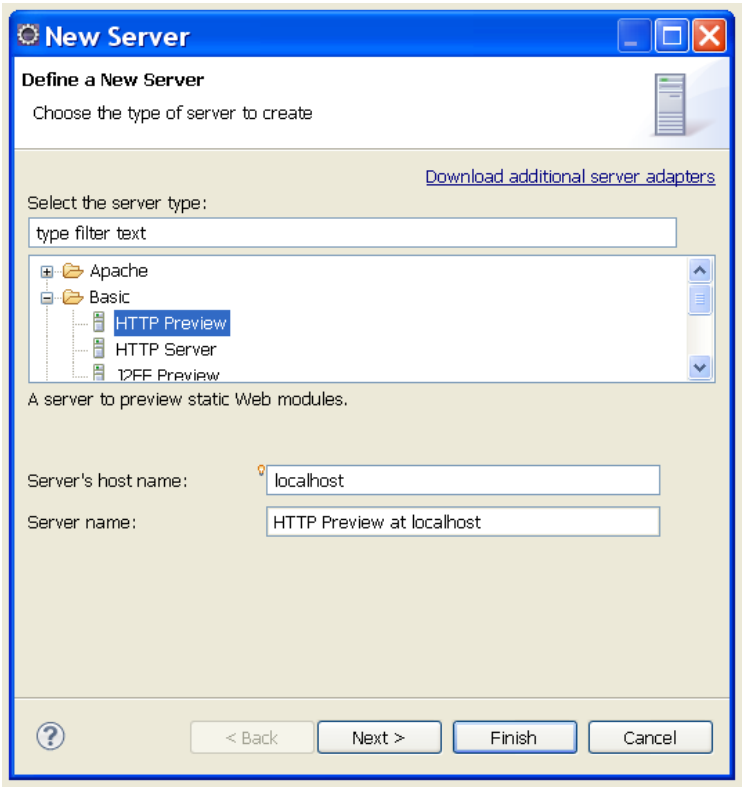

Figura 79. Ventana *New Server*

Eclipse comenzará a buscar herramientas para diferentes tipos de servidores que encuentre en internet, una vez aparezca *Oracle Glassfish Server Tools* se selecciona y se pulsa en el botón *next* de la ventana mostrada en la Figura 80.

El programa pedirá que se acepte algún contrato de licencias, mostrando una ventana como la de la Figura 81. Se acepta y se comenzará a descargar la herramienta. Tardará un poco, dependiendo de la conexión a Internet que se tenga.

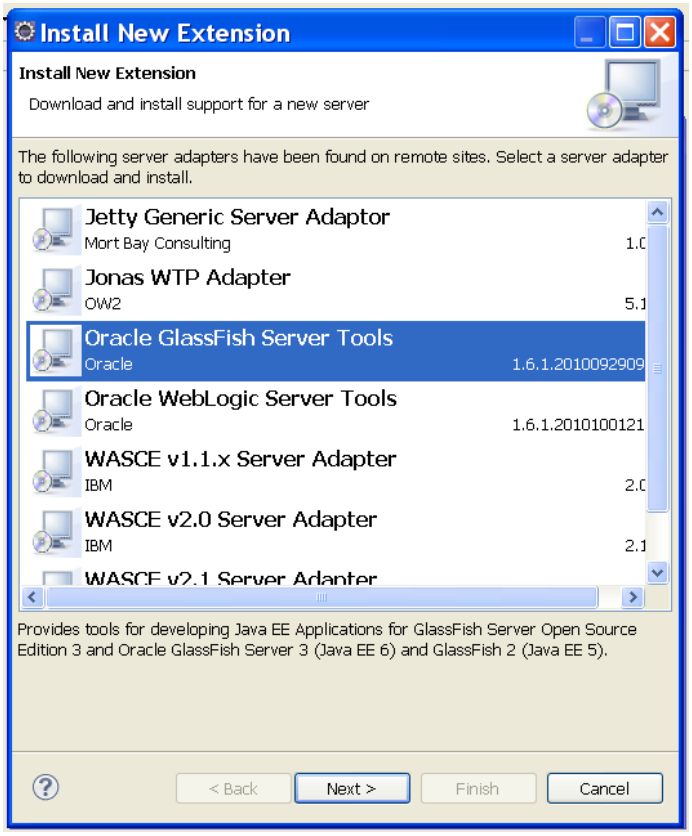

Figura 80. Ventana de descarga de herramientas de servidores

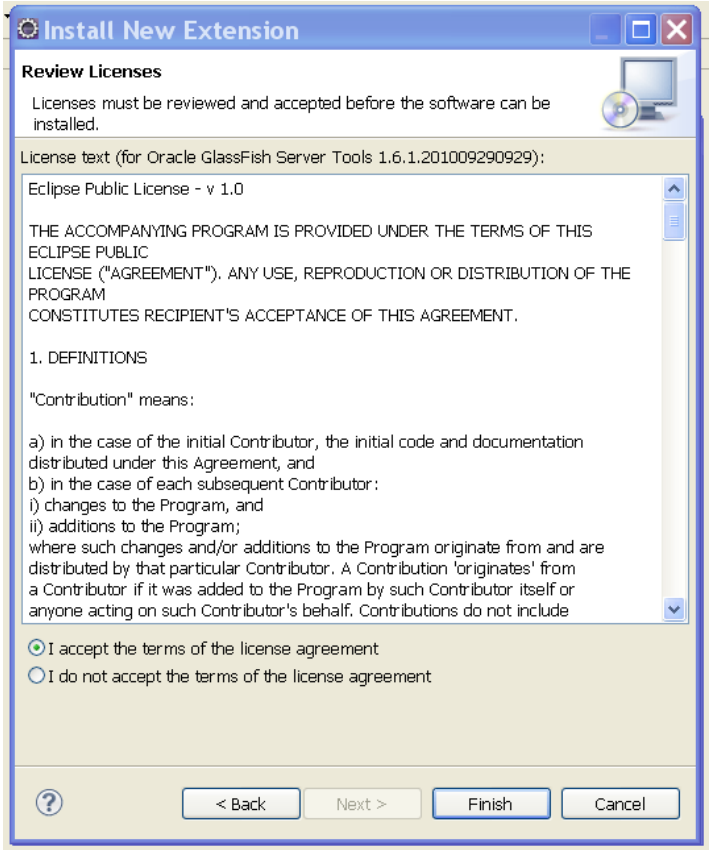

Figura 81. Licencia de uso Glassfish

Una vez instalado pedirá que se reinicie Eclipse. Ya reiniciado se vuelve a ir a la pestaña *Servers* y se crea uno nuevo como se indica al principio de la sección. Aparece la pantalla con la lista de servidores, como se muestra en la Figura 82, pero ya incluyendo Glassfish.

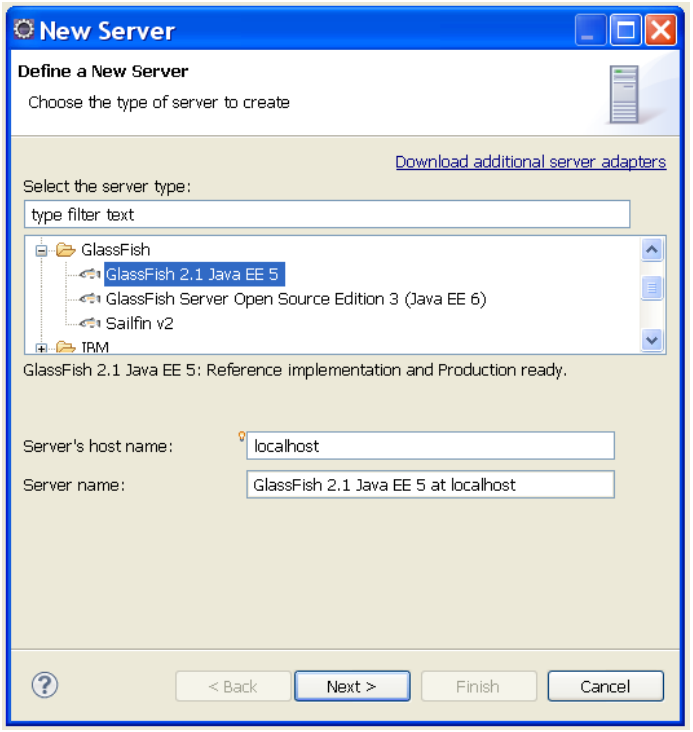

Figura 82. Ventana *New Server* con Glassfish

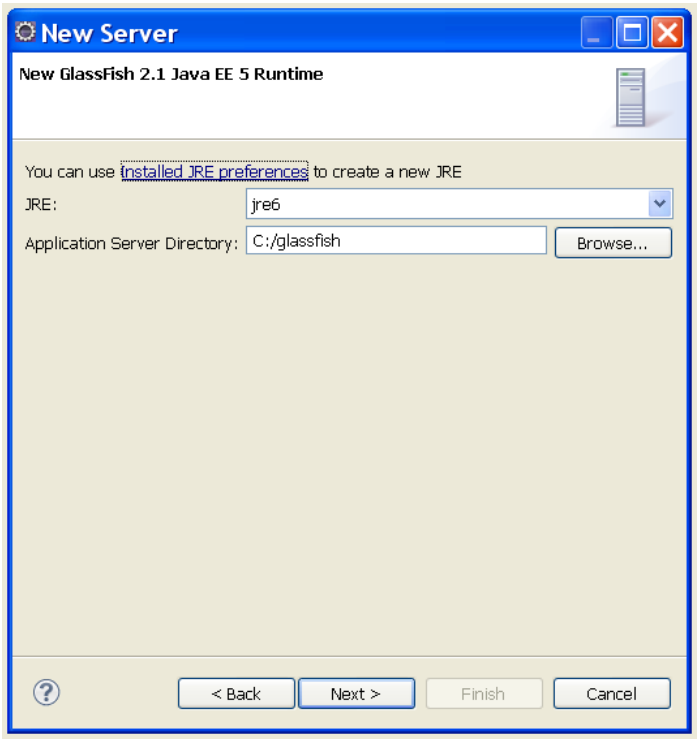

Figura 83. Configuración de JRE para Glassfish

Se selecciona el servidor deseado (versión 2.1 en este caso), se pulsa en el botón *Next*. En la ventana de la Figura 83 se configura la ubicación del entorno de ejecución Java a utilizar y el directorio raíz del servidor Glassfish.

En la ventana de la Figura 84 aparecerán los datos de acceso al servidor para poder controlarlo desde Eclipse.

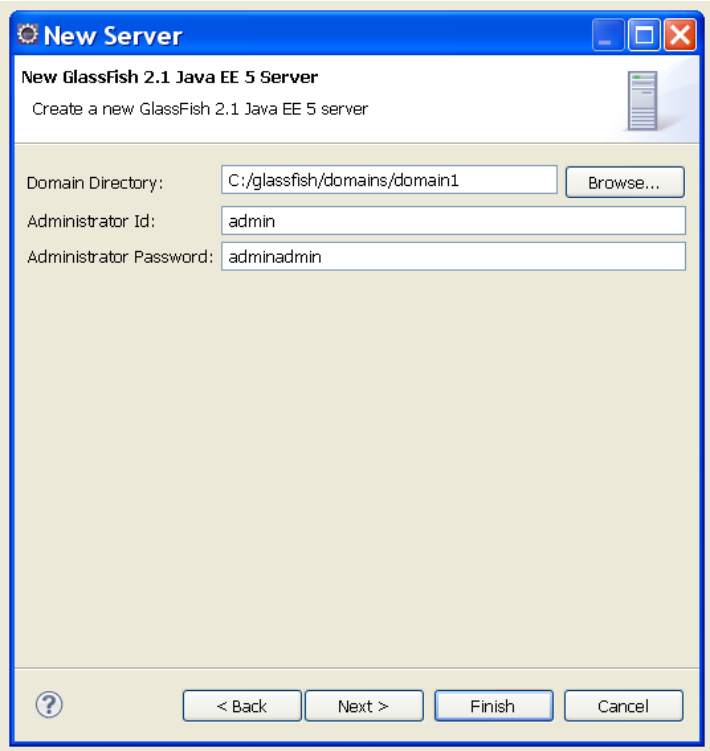

Figura 84. Datos de acceso a Glassfish

Finalmente, aparece otra ventana en la que se puede añadir algún proyecto Java EE para desplegarlo. Si no es el caso se pulsa finalizar y ya estaría todo configurado. El servidor, con todas sus opciones, aparecerá en la pestaña *Servers*, mostrada en la Figura 85.

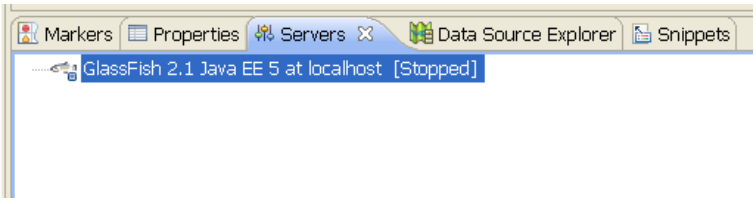

Figura 85. Servidor configurado

APÉNDICE E. Presupuesto

1.- Autor: Manuel Jesús Martín Gutiérrez

2.- Departamento: Ing. Telemática

3.- Descripción del Proyecto:

- Titulo: **Módulo Empresarial Java Path Finder**
- Duración (meses) **7**

Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

14280 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F.	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
					0,00	
Basanta Val, Pablo		Ingeniero Senior	0,24	4.289,54	1.029,49	
Martín Gutiérrez, Manuel Jesús		Ingeniero	4	2.694,39	10.777,56	
					0,00	
					0,00	
		Hombres mes 4,24		**Total**	11.807,05	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPO

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
PC con Win. 7 y Office	800,00	100	7	60	93,33
					0,00
					0,00
					0,00
					0,00
					0,00
				Total	93,33

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado

B = periodo de depreciación (60 meses)

C = coste del equipo (sin IVA)

D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
	Total	0,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,…

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	11.807
Amortización	93
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	2.380
Total	14.280

APÉNDICE F. Glosario

API: (Application Programming Interface) Es el conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

CPU: (Central Processing Unit) Es el componente del ordenador y otros dispositivos programables, que interpreta las instrucciones contenidas en los programas y procesa datos.

EJB: (Enterprise Java Beans) Una de las API que forman parte del estándar de construcción de aplicaciones empresariales Java EE de Sun Microsystems. Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor.

HTML: (Hyper Text Markup Language) Lenguaje de marcado predominante para la elaboración de páginas web. Es utilizado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes.

HTTP: (Hyper Text Transfer Protocolo) Protocolo usado en cada transacción de la web (WWW).

IDE: (Integrated Development Environment) Es una aplicación software que ofrece facilidades a programadores para el desarrollo de software. Normalmente se compone de: Editor de código, compilador o intérprete, herramienta de construcción automática y un depurador.

IP: (Internet Protocol) Protocolo no orientado a conexión utilizado tanto en origen como en destino para la comunicación de datos a través de una red de paquetes conmutados no fiable de mejor entrega posible sin garantías.

Java EE o JEE: (Java Platform, Enterprise Edition) Es una plataforma de programación—parte de la Plataforma Java—para desarrollar y ejecutar software de aplicaciones en Lenguaje de programación Java con arquitectura de n niveles distribuida, basándose ampliamente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones.

Java SE o JSE: (Java Platform, Standard Edition) Colección de APIs del lenguaje de programación Java, utiles para muchos programas de la plataforma Java.

Java: Lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 1990.

JAVASCRIPT: Lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. Se utiliza principalmente en su forma del lado cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. Existe también una forma de Javascript del lado del servidor (SSJS).

JPF: (Java Path Finder) Software de verificación o validación de programas ejecutables Java. Fue desarrollado por la Nasa y declarado código libre en 2005.

JVM: (Java Virtual Machine) Es una máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (*Java Bytecode*), el cual es generado por un compilador del lenguaje Java.

POJI: (Plain Old Java Interface) Interfaces Java de los POJO.

POJO: (Plain Old Java Objects) Clases simples Java que no dependen de ningún *framework* especial.

Sistemas distribuidos: Modelo para resolver problemas de computación masiva utilizando un gran número de computadoras en una infraestructura de telecomunicaciones distribuida.

Validación formal: Método para asegurar que un programa actúe de la forma prevista sean cuales sean sus valores de entrada. Es un método de verificación funcional que combina aspectos de control tradicionales y métodos de simulación basados en la verificación lógica con simulación simbólica y técnicas de análisis estático.

WWW: (World Wide Web) Sistema de documentos de hipertexto y/o hipermedios enlazados y accesibles a través de Internet.

APÉNDICE G. Referencias e hiperenlaces

[1] Nasa; **"Java Path Finder, JPF;** 2009; http://babelfish.arc.nasa.gov/trac/jpf

[2] Veritable.com; **"Formal Validation"**; www.veritable.com/Computer/Formal_Validation/formal_validation.html

[3] Klaus Havelund and Jens U. Skakkeb. 1999. Applying Model Checking in Java Verification. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink (Eds.). Springer-Verlag, London, UK, 216-231.

[4] Tihomir Gvero, Milos Gligoric, Steven Lauterburg, Marcelo d'Amorim, Darko Marinov, and Sarfraz Khurshid. 2008. State extensions for java pathfinder. In *Proceedings of the 30th international conference on Software engineering* (ICSE '08). ACM, New York, NY, USA, 863-866. DOI=10.1145/1368088.1368224 http://doi.acm.org/10.1145/1368088.1368224

[5] O. Tkachuk, M. B Dwyer. 2010. Environment generation for validating event-driven software using model checking, *Software, IET* , vol.4, no.3, pp.194-209, June 2010 doi: 10.1049/iet-sen.2009.0017

[6] Wikipedia; **"Formal Verification"**; en.wikipedia.org/wiki/Formal_verification

[7] Oracle; **"Java EE Documentation"**; http://www.oracle.com/technetwork/java/javaee/documentation/index.html

[8] Oracle; **"EJBs Specifications";** http://www.oracle.com/technetwork/java/docs-135218.html

[9] Wikipedia; **"Enterprise JavaBeans"**; http://es.wikipedia.org/wiki/Enterprise_JavaBeans

[10] Oracle; **"Java SE Documentation"**; http://www.oracle.com/technetwork/java/javase/documentation/index.html

[11] Oracle; **"Java EE 5 Tutorial"**; http://download.oracle.com/javaee/5/tutorial/doc/bnbmt.html

[12] Sun Microsystems; **"JavaMail API"**; http://java.sun.com/products/javamail/javadocs/javax/mail/package-summary.html

[13] Sun Microsystems; **"JavaCompiler API"**; http://download.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html

[14] Sun Microsystems; **"java.util.zip API"**; http://download.oracle.com/javase/6/docs/api/java/util/zip/package-summary.html

[15] Sun Microsystems; **"java.util.jar API"**; http://download.oracle.com/javase/6/docs/api/java/util/jar/package-summary.html

[16] Oracle; **"BEA Weblogic server"**; http://www.oracle.com/technetwork/middleware/Weblogic/overview/index.html

[17] JBoss; **"JBoss application server"**; http://www.jboss.org/jbossas

[18] IBM; **"IBM WebSphere"**; http://www.ibm.com/software/Webservers/appserv/was/

[19] Borland; **"Borland AppServer"**; http://www.borland.com/us/products/appserver/index.html

[20] Oracle; **"Glassfish server"**; http://glassfish.java.net/

[21] Sun Microsystems; **"JMS Specifications"**; http://java.sun.com/products/jms/jms1_0_2-spec.pdf

[22] Apache Software Foundation; **“Jakarta Commons HTTPClient Tutorial”**; http://hc.apache.org/httpclient-3.x/tutorial.html

[23] Stanford; **“Oreilly MultipartParser API”**; http://www.stanford.edu/group/coursework/docsTech/oreilly/com.oreilly.servlet.multipart.MultipartParser.html

[24] Eclipse Foundation; **“Eclipse IDE”**; http://www.eclipse.org/

[25] Apache Software Foundation; **“Apache Ant User Manual”**; http://ant.apache.org/manual/index.html