Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Departamento de Ciencias de la Computación

INGENIERÍA INFORMÁTICA

CURSO ACADÉMICO 2009/2010

Optsicom Remote Experiment System
Plugin para eclipse

Proyecto Fin de Carrera

– Autor –

Carlos Llorente López

–Tutor–

Francisco Gortázar Bellas

18 de junio de 2010

Copyright

Esta memoria forma parte del Proyecto de Fin de Carrera de la titulación de Ingeniería Informática en la Escuela Técnica Superior de Ingeniería Informática de la Universidad Rey Juan Carlos de Madrid, realizado por Carlos Llorente López en Junio de 2010. Copyright 2010, CLL. Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la licencia Reconocimiento-Compartir bajo la misma licencia 3.0 España", proporcionada por CREATIVE COMMONS.

Agradecimientos

En primer lugar me gustaría dar las gracias a Patxi y Mica por todo el tiempo que me han dedicado y la ayuda que me han prestado durante la realización de este proyecto. A Patxi por todo lo que me ha enseñado sobre el desarrollo de plugins y Eclipse y a Mica por todo lo que me ha enseñado sobre Java. He aprendido muchas cosas con vosotros dos. Gracias.

También quiero darle las gracias a mi familia por apoyarme durante este tiempo y animarme a seguir adelante incluso cuando las cosas se complicaban y parecía que no había manera de hacerlas continuar.

Por último y sobre todo me gustaría dar las gracias a dos amigos muy importantes para mi, vosotros sabéis quienes sois así que no hace falta que os nombre, gracias por todos esos buenos ratos y todo el apoyo y confianza que me habéis prestado.

Gracias a todos.

Resumen

Actualmente Eclipse es uno de los entornos de desarrollo (IDE) más utilizados por los programadores para desarrollar aplicaciones. Aunque se pueden desarrollar aplicaciones en cualquier lenguaje, Eclipse esta pensado para el desarrollo de aplicaciones Java.

Eclipse se puede extender mediante lo que se conocen como *plugins*. Los plugins son funcionalidades extras que se le proporcionan al IDE para facilitar la tarea del programador. Estos plugins al igual que el IDE están desarrollados en Java.

La idea de este proyecto ha sido la de implementar un plugin para Eclipse que permitiera solucionar los diversos problemas con los que se encontraban los profesores de la universidad a la hora de desarrollar aplicaciones. De esta manera surgió la idea de crear un plugin mediante el cual los profesores puedan ejecutar y depurar remotamente *aplicaciones Java versionadas.*

Hablamos de *aplicaciones Java versionadas* ya que en el momento en el que se ejecuta o depura un programa se genera un fichero comprimido *.zip* con todos los ficheros fuentes del programa y sus dependencias. De esta forma se puede recuperar siempre el código fuente correspondiente a una determinada ejecución del programa.

Índice general

1. Introducción **13**

2. Objetivos **15**

2.1. Objetivo principal 15

2.2. Objetivos secundarios 15

2.2.1. Ejecución de aplicaciones remotas desde Eclipse 15

2.2.2. Monitorización de las ejecuciones 16

2.2.3. Depuración remota 16

2.2.4. Versionado 16

3. Metodologías y Tecnologías **17**

3.1. Metodologías 17

3.1.1. Metodología en espiral 17

3.2. Tecnologías 20

3.2.1. Java 20

3.2.2. RMI 26
3.2.3. Eclipse 28
3.2.4. Java Development Tools (JDT) 32
3.2.5. Standard Widget Toolkit(SWT) 32
3.2.6. Plug-in Development Environment (PDE) 32
3.2.7. Subversion 33

4. Manual de usuario 35

4.1. Ejecución o depuración de una aplicación en local 35
4.2. Ejecución o depuración de una aplicación en remoto 38
4.3. Recuperar el estado de una ejecución 40

5. Manual de instalación 43

5.1. Requisitos 43
5.2. Procedimiento 44

6. Descripción Informática 49

6.1. Especificación de requisitos 49
6.1.1. Requisitos funcionales 49
6.1.2. Requisitos no funcionales 52
6.2. Diseño 52
6.2.1. Casos de uso 52

6.2.2. Diagrama de clases 53
6.2.3. Diagramas de secuencia 59
6.3. Implementación 62
6.3.1. Extensiones (Extensions) 63
6.3.2. Generación de ficheros .zip 66
6.3.3. Socket Secure Layer (SSL) y RMI 66
6.3.4. Desarrollo de la *feature* y del *update_site* 67
6.4. Pruebas 67
6.4.1. Generación correcta de los ficheros .zip 68
6.4.2. Ejecución y depuración de una aplicación en local 68
6.4.3. Ejecución y depuración de una aplicación en remoto 68
6.4.4. Visualización del estado de una ejecución en remoto 69

7. Conclusiones 71
7.1. Logros alcanzados 71
7.2. Trabajos futuros 72

Bibliografía 73

Índice de figuras

3.1. Esquema del modelo en espiral 18

3.2. Funcionamiento de la tecnología RMI. 29

3.3. Arquitectura de la plataforma Eclipse. 31

4.1. Depuración en local de una aplicación Java versionada. 37

4.2. Perspectiva de depuración de Eclipse. 37

4.3. Menú de ejecución remota de una aplicación java versionada. 38

4.4. Panel o ventana principal del acceso remoto. 39

4.5. Perspectiva de depuración de Eclipse y salida por consola. 40

4.6. Vista Optsicom View. 41

4.7. Funcionalidad de la vista Optsicom View. 42

5.1. Selección del workspace de eclipse 44

5.2. Install new Software... 45

5.3. Ventana de instalación 1. 46

5.4. Ventana de instalación 2. 47

6.1. Casos de uso 52

6.2. Paquetes principales de la aplicación 53

6.3. Clases de los paquetes *optsicom.res.server.impl* y *optsicom.res.server* 54

6.4. Clases del paquete *optsicom.res.client* 56

6.5. Clases del paquete *optsicom.res.client.local* 56

6.6. Clases del paquete *optsicom.res.client.remote* 57

6.7. Ejecución local 59

6.8. Ejecución remota 60

6.9. Desconexión de una ejecución remota 61

6.10. Pestaña Extensions 62

6.11. Pestaña Overview 63

Capítulo 1

Introducción

Actualmente en el ámbito universitario se trabaja realizando experimentos, los cuales pueden ser bastante complejos y muy pesados de ejecutar. Por ello los programadores deben ejecutar sus aplicaciones en otras máquinas más potentes que las disponibles en sus despachos. Así pues los profesores e investigadores se encuentran en la situación de tener que transportar el código fuente de sus programas a otras máquinas mas potentes cada vez que desean ejecutarlo.

En un principio cada vez que se deseaba ejecutar un programa se debía copiar el código fuente en un dispositivo de almacenamiento portátil y transportarlo hasta otra máquina más potente en la que se llevaba a cabo la ejecución. Esto suponía una gran pérdida de tiempo ya que la máquina en cuestión no tenía por qué estar precisamente cerca.

Más adelante y gracias a las aplicaciones que permiten el acceso remoto a las máquinas a través de protocolos como *SSH* y *FTP* o a la utilización de herramientas de control de versiones como *SVN* o *CVS* el programador ya no tenía que preocuparse en transportar el código fuente y simplemente lo podía copiar desde una máquina a otra de manera remota sin tener que utilizar un dispositivo de almacenamiento portátil para el transporte y sin necesidad de moverse del sitio.

Aun así esto seguía siendo un inconveniente ya que se requería de conocimientos en administración de sistemas por parte del desarrollador para configurar las aplicaciones citadas anteriormente (*SVN*, *CVS*, *SSH*, *FTP*) y poder utilizarlas correctamente.

Además si por alguna razón se producía un error en la ejecución, en primer lugar, no se tenía constancia del mismo hasta que se establecía de nuevo una conexión con la máquina remota, y en segundo lugar, se terminaba modificando el código fuente directamente en la máquina remota por comodidad perdiendo así la coherencia entre los ficheros de la máquina local y la remota.

De esta manera surge la idea de encontrar una forma más cómoda de llevar a cabo estas ejecuciones y para ello se decide desarrollar un plugin para el IDE Eclipse. Un plugin es una funcionalidad extra que se le añade al IDE para mejorar la tarea del desarrollador y en este caso esa funcionalidad era permitir el desarrollo de aplicaciones Java y su ejecución en una máquina remota.

Capítulo 2

Objetivos

2.1. Objetivo principal

El objetivo principal de este proyecto es desarrollar una herramienta gracias a la cual el programador pueda ejecutar programas remotamente de una manera sencilla y eficiente.

2.2. Objetivos secundarios

2.2.1. Ejecución de aplicaciones remotas desde Eclipse

Uno de los objetivos es poder ejecutar aplicaciones java en una máquina remota desde el IDE Eclipse. Este objetivo deriva en que Eclipse es uno de los entornos de desarrollo más utilizados debido a su capacidad de ejecución en múltiples plataformas y la posibilidad de desarrollar programas en multitud de lenguajes.

2.2.2. Monitorización de las ejecuciones

De poco serviría poder ejecutar aplicaciones de manera remota si no se puede conocer cuál es el estado de la ejecución en un cierto momento. Con este segundo objetivo se pretende conseguir capturar la salida del programa en ejecución y conocer en todo momento cuál es el estado de la ejecución, así como los posibles errores que se deriven de la misma.

2.2.3. Depuración remota

Como en todo programa informático, es posible que se produzcan errores y debido a ello se pretende poder depurar el programa de forma remota. Teniendo lugar la depuración en la máquina remota pero a efectos del usuario sería como si se produjese en local.

2.2.4. Versionado

El último de los objetivos es el de poder *versionar* el código fuente. Cuando se lleve a cabo una ejecución de un determinado código fuente este será versionado, es decir, se guardarán todos los fuentes en el estado en el que se encuentren y se llevará a cabo la ejecución.

Por otra parte el programador podrá seguir desarrollando su aplicación y si en algún momento necesita saber que código fuente se ejecutó en cierto momento,se podrá acceder a el gracias a su versión.

Capítulo 3

Metodologías y Tecnologías

3.1. Metodologías

Una metodología de desarrollo es un modelo de ciclo de vida del software en el cuál se establecen las pautas para un correcto desarrollo de software. Existen multitud de metodologías como son la metodología en cascada, metodología en espiral, metodología incremental, etc.

La elección de una metodología depende mucho del proyecto qué se desarrolle y no tiene por que ser una única metodología en concreto. Para este proyecto se ha elegido una metodología mixta entre metodología en espiral, incremental y de prototipos.

3.1.1. Metodología en espiral

La metodología elegida para el desarrollo de este proyecto ha sido una metodología incremental en espiral con prototipos.Este tipo de metodología tiene fuertemente en cuenta el riesgo que aparece a la hora de desarrollar software.

Para ello se comienza estudiando las posibles alternativas de desarrollo, se opta por la que tenga un riesgo ínfimo y se hace un ciclo de la espiral. La Figura 3.1 muestra el esquema del modelo.

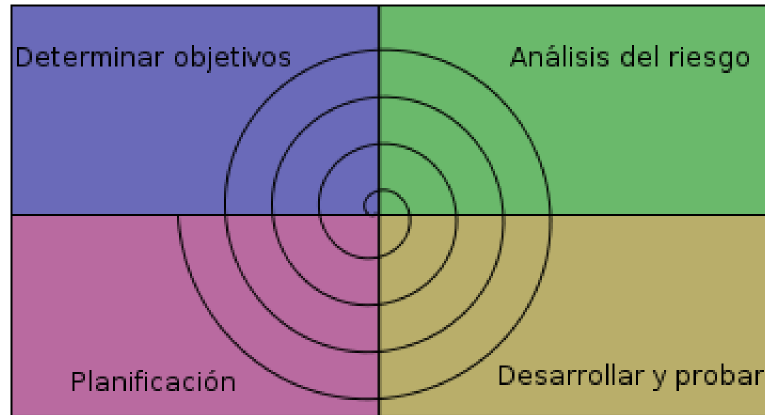

Figura 3.1: Esquema del modelo en espiral

Si se desea seguir haciendo mejoras en el software se vuelven a evaluar las distintas alternativas y riesgos y se realiza una nueva vuelta a la espiral. De esta manera se pueden realizar tantas vueltas como se quiera en la espiral hasta que llegue un momento en el que el software desarrollado sea aceptable y no necesite seguir siendo mejorado. En cada iteración o vuelta a la espiral se deben tener en cuenta los siguientes aspectos:

- Los objetivos o necesidades que debe cubrir el producto.
- Las alternativas o diferentes formas de conseguir los objetivos de forma exitosa. Estas alternativas pueden ser:
    1. Características de la alternativa (experiencia del personal, requisitos a cumplir...).
    2. Formas de gestión del sistema.
    3. Riesgo asumido en cada alternativa.
- Desarrollo y verificación mediante un prototipo.

Además, cada ciclo de la espiral se compone de las siguientes actividades:

1. **Determinar o fijar objetivos:** Consiste en determinar los requisitos de forma detallada, las restricciones que puedan haber, los riesgos y las alternativas a los mismos. Además, en el primer ciclo de la espiral se realizará una planificación inicial.

2. **Análisis del riesgo:** Consiste en estudiar los posibles riesgos existentes y las alternativas a los mismos.

3. **Desarrollar, verificar y validar:** Como su nombre indica consiste en desarrollar las funcionalidades fijadas en los objetivos, con su correspondiente verificación y validación.

4. **Planificar:** En esta actividad se comprueban los resultados obtenidos, se evalúan y se determina si se continúa con la siguiente fase, procediendo a su planificación.

3.2. Tecnologías

Las tecnologías empleadas durante el desarrollo de este proyecto han sido las siguientes:

- **Java:** Lenguaje de programación en el que ha sido desarrollado el plugin.
- **Rmi:** Mecanismo ofrecido por Java para invocar métodos de manera remota.
- **Eclipse:** Entorno de desarrollo del plugin
- **Java Development Tools (JDT):** Herramientas que proporciona Eclipse para el desarrollo de aplicaciones Java.
- **Standard Widget Toolkit (SWT):** Conjunto de componentes para construir interfaces gráficas en Java.
- **Plug-in Development Environment (PDE):** Conjunto de herramientas para la creación de plugins de Eclipse.
- **Subversion:** Software de sistema de control de versiones.

3.2.1. Java

Es el lenguaje de programación en el que ha sido desarrollado el plugin. La elección de este lenguaje se debe a que Eclipse es un IDE implementado en Java y por lo tanto sus plugins también lo son. Dado que Java es un lenguaje orientado a objetos a continuación se explican las características del lenguaje Java, qué es un lenguaje orientado a objetos y sus características principales.

Características de Java

El lenguaje Java se creó con cinco objetivos principales:

1. Debería usar la metodología de la programación orientada a objetos.
2. Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos (Independencia de la plataforma).
3. Debería incluir por defecto soporte para trabajo en red.
4. Debería diseñarse para ejecutar código en sistemas remotos de forma segura.
5. Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos, como C++.

A continuación se habla de las características y conceptos fundamentales de la programación orientada a objetos pero teniendo siempre en cuenta su orientación hacia el lenguaje Java.

Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones informáticas. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de 1990 y en la actualidad, existen variedad de lenguajes de programación que soportan la orientación a objetos.

Los objetos son entidades que combinan estado o *atributo*, comportamiento o *método* e identidad:

- El estado está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).

- El comportamiento está definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.

- La identidad es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador.

Un objeto contiene toda la información que permite definirse e identificarse frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos que favorecen la comunicación entre ellos y favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles en las que no se separa el estado y el comportamiento.

Los métodos y atributos están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

La POO difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Conceptos fundamentales de la POO

La programación orientada a objetos introduce nuevos conceptos que superan y amplían conceptos antiguos ya conocidos de la programación estructurada. Entre ellos destacan los siguientes:

- **Clase:** definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La sustanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- **Herencia:** Es la facilidad mediante la cual una clase hereda en ella cada uno de los atributos y operaciones de su clase padre, como si esos atributos y operaciones hubiesen sido definidos en la propia clase hija. Por lo tanto, puede usar los mismos métodos y variables públicas declaradas en la clase padre. Los componentes registrados como *privados* también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo se puede acceder a ellos a través de otros métodos públicos. Esto es así para mantener hegemónico el ideal de la POO.

- **Objeto:** Es una entidad provista de un conjunto de propiedades o atributos y de comportamiento o métodos, los cuales reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos de la aplicación. Es una instancia de una clase.

- **Método:** Es un algoritmo asociado a un objeto cuya ejecución se desencadena tras la recepción de un mensaje. Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un *evento* con un nuevo mensaje para otro objeto del sistema.

- **Evento:** Es un suceso en el sistema (una interacción del usuario con la máquina, un mensaje enviado por un objeto ...). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento a la reacción que puede desencadenar un objeto, es decir, la acción que genera.

- **Mensaje:** Es una comunicación dirigida a un objeto que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- **Atributo:** Es el contenedor de un tipo de datos asociados a un objeto que hace los datos visibles desde fuera del objeto y ésto se define como sus características predeterminadas y cuyo valor puede ser alterado por la ejecución de algún método.

- **Estado interno:** Es una variable que se declara privada y que únicamente puede ser accedida y alterada por un método del objeto. Se utiliza para indicar distintas situaciones posibles para el objeto y no es visible al programador que maneja una instancia de la clase.

- **Componentes de un objeto:** Son atributos, identidad, relaciones y métodos descritos anteriormente.

- **Identificación de un objeto:** Un objeto es representado por medio de una tabla o entidad que esta compuesta por sus atributos y funciones correspondientes.

Características de la POO

Las características mas importantes de la programación orientada a objetos son las siguientes:

- **Abstracción:** Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un *agente* abstracto que puede realizar trabajo, informar, cambiar su estado, y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.

- **Encapsulamiento:** Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Ésto permite aumentar la cohesión de los componentes del sistema.

- **Principio de ocultación:** Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interaccionar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas.

- **Polimorfismo:** Permite que comportamientos diferentes asociados a objetos distintos puedan compartir el mismo nombre. Al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando ésto ocurre en *tiempo de ejecución*, esta última característica se llama asignación tardía o asignación dinámica.

- **Herencia:** Las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir y extender su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y éstas en árboles que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

- **Recolección de basura:** La Recolección de basura o *Garbage Collector* es la técnica por la que el ambiente de objetos se encarga de destruir automáticamente, y por tanto de desasignar de la memoria, los objetos que hayan quedado sin ninguna referencia a ellos. Ésto significa que el programador no

debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando.

3.2.2. RMI

Introducción a RMI

RMI (*Java Remote Method Invocation*) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno de ejecución de Java y provee de un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java.

RMI se caracteriza por la facilidad de su uso en la programación ya que esta específicamente diseñado para Java y proporciona paso de objetos por referencia, recolección de basura distribuida (*Garbage Collector* distribuido) y paso de tipos arbitrarios.

Por medio de RMI, un programa Java puede exportar un objeto, lo que significa que éste queda accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de este momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto.

La invocación se compone de los siguientes pasos:

- Encapsulado o *marshalling* de los parámetros gracias a la funcionalidad de serialización de Java.
- Invocación del método por parte del cliente sobre el servidor, quedando el invocador en espera de una respuesta.
- Serialización del valor de retorno por parte del servidor al terminar la ejecución.

- El cliente recibe la respuesta y continúa como si la invocación hubiera tenido lugar en local.

Además, el uso de RMI resulta muy natural para todo aquel programador de Java ya que no tiene que aprender una nueva tecnología completamente distinta de aquella con la que se llevará a cabo el desarrollo de la aplicación. Sin embargo, RMI tiene algunas limitaciones debido a su estrecha integración con Java, la principal de ellas es que esta tecnología no permite la interacción con aplicaciones escritas en otro lenguaje.

RMI como extensión de Java, es una tecnología de programación que fue diseñada para resolver problemas escribiendo y organizando código ejecutable. Así, RMI constituye un punto específico en el espacio de las tecnologías de programación junto con C, C++, Smalltalk, etc.

Arquitectura

La arquitectura RMI puede verse como un modelo de cuatro capas:

- **Primera capa:** La primera capa es la de aplicación y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda *java.rmi.Remote*.

  Dicha interfaz se usa básicamente para marcar un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase *UnicastRemoteObject*, perteneciente al paquete *java.rmi.server*, o puede hacerse de forma explícita con una llamada al método *exportObject()* del mismo paquete.

- **Segunda capa:** La capa 2 es la capa proxy, o capa *stub-skeleton*. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas

a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

- **Tercera capa:** La capa 3 es la de referencia remota, y es la responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo stream (*stream-oriented connection*) desde la capa de transporte.

- **Cuarta capa:** La capa 4 es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (*Java Remote Method Protocol*), que solamente es utilizado por programas Java.

Elementos de una aplicación RMI

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un servidor que crea algunos objetos remotos, también crea referencias para hacerlos accesibles y espera a que el cliente los invoque.
- Un cliente que obtiene una referencia a los objetos remotos en el servidor y los invoca.

La Figura 3.2 representa el esquema de funcionamiento de la tecnología RMI.

3.2.3. Eclipse

Introducción

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama *Aplicaciones de Cliente Enriquecido*,

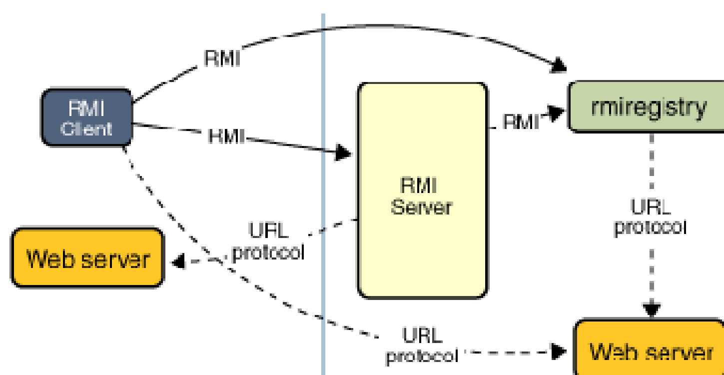

Figura 3.2: Funcionamiento de la tecnología RMI.

opuesto a las *Aplicaciones de cliente ligero* basadas en navegadores. Esta plataforma normalmente ha sido usada para desarrollar entornos de desarrollo integrados como el IDE de Java llamado *Java Development Toolkit* (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse y que son usados también para desarrollar el propio Eclipse.

Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para *VisualAge* pero actualmente es desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Arquitectura

La base para Eclipse es la Plataforma de cliente enriquecido (RCP). Los siguientes componentes constituyen la plataforma de cliente enriquecido:

- **Plataforma principal:** Lleva a cabo el inicio de Eclipse y la ejecución de plugins.
- **OSGi:** Es una plataforma para bundling estándar.

- **Standard Widget Toolkit (SWT):** Es un widget toolkit portátil.
- **JFace:** Permite el manejo de archivos, manejo de texto, editores de texto.
- **Workbench de Eclipse:** Son las vistas, editores, perspectivas, asistentes.

Los *widgets* de Eclipse están implementados por una herramienta de *widget* para Java llamada SWT, a diferencia de la mayoría de las aplicaciones Java, que usan las opciones estándar Abstract Window Toolkit (AWT) o Swing. La interfaz de usuario de Eclipse también tiene una capa GUI intermedia llamada JFace, que simplifica la construcción de aplicaciones basadas en SWT.

El entorno de desarrollo integrado (IDE) de Eclipse emplea módulos para proporcionar toda su funcionalidad al frente de la plataforma de cliente rico, a diferencia de otros entornos donde las funcionalidades están todas incluidas las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software.

Además, Eclipse puede extenderse para soportar el desarrollo de aplicaciones en otros lenguajes de programación como C/C++ o Python, lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet, sistemas de gestión de bases de datos o sistemas de control de versiones como CVS o SVN. La arquitectura plugin permite añadir cualquier extensión deseada en el entorno.

La definición que da el proyecto Eclipse acerca de su software es: *una especie de herramienta universal - un IDE abierto y extensible para todo y nada en particular.*

En cuanto a las aplicaciones clientes, eclipse provee al programador con *frameworks* muy ricos para el desarrollo de aplicaciones gráficas, definición y manipulación de modelos de software, aplicaciones web, etc. Por ejemplo, GEF (*Graphic Editing Framework* - Framework para la edición gráfica) es un plugin de Eclipse para el desarrollo de editores visuales que pueden ir desde procesadores de texto wysiwyg hasta editores de diagramas UML, interfaces gráficas para el usuario (GUI), etc. Dado que los editores realizados con GEF *viven* dentro de Eclipse, además de poder ser usados conjuntamente con otros plugins, hacen uso de su interfaz gráfica personalizable y profesional.

El SDK de Eclipse incluye las herramientas de desarrollo de Java, ofreciendo un IDE con un compilador de Java interno y un modelo completo de los archivos fuente de Java. Esto permite técnicas avanzadas de refactorización y análisis de código. El IDE también hace uso de un espacio de trabajo, *workspace*, en el cual existen un grupo de metadatos en un espacio para archivos planos, permitiendo modificaciones externas a los archivos en cuanto se refresque el workspace correspondiente.

La Figura 3.3 representa la arquitectura de la plataforma Eclipse.

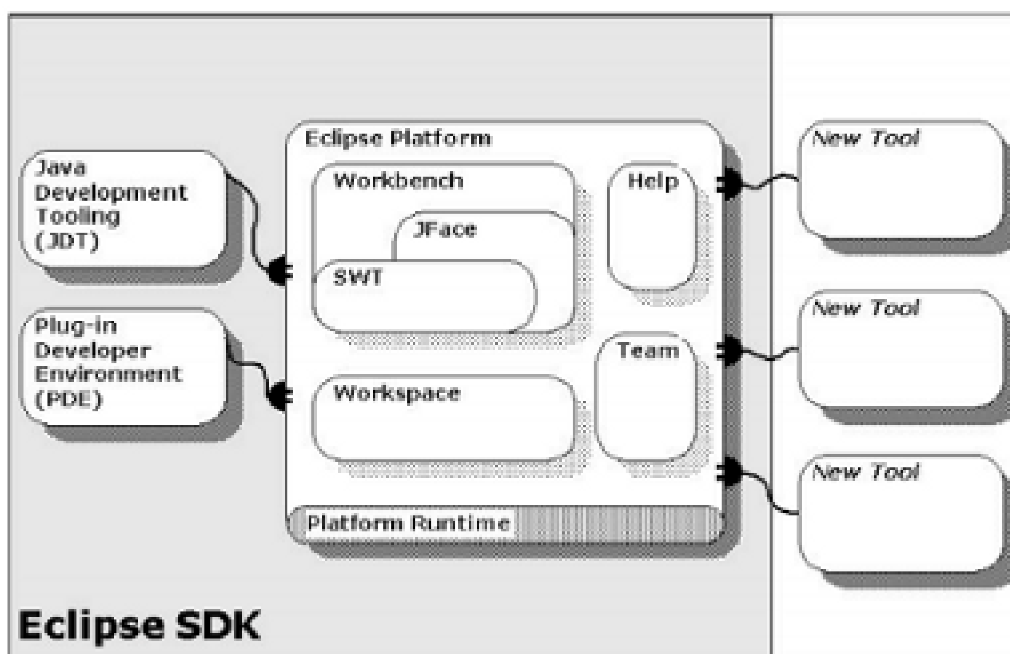

Figura 3.3: Arquitectura de la plataforma Eclipse.

Características

Eclipse dispone de un editor de texto con resaltado de sintaxis, compilación en tiempo real, pruebas unitarias con JUnit, control de versiones con CVS, integración con Ant, asistentes para creación de proyectos, clases, tests, etc.

Asimismo, a través de *plugins* libremente disponibles es posible añadir cualquier funcionalidad que el programador desee.

3.2.4. Java Development Tools (JDT)

Java Development Tools o JDT es uno de los plugins de la plataforma Eclipse que especializa las características del *workbench* de Eclipse en cuanto a edición,compilación,ejecución... de código Java.

3.2.5. Standard Widget Toolkit(SWT)

SWT o *Standard Widget Toolkit* es un conjunto de componentes para construir interfaces gráficas en Java, más concretamente aplicaciones desarrollados por el proyecto Eclipse. Recupera la idea original de la biblioteca AWT de utilizar componentes nativos, con lo que adopta un estilo más consistente en todas las plataformas, pero evita caer en las limitaciones de ésta.

La biblioteca Swing, por otro lado, está codificada enteramente en Java y frecuentemente se le acusa de no brindar una experiencia idéntica a la de una aplicación nativa. Sin embargo, el precio a pagar por esa mejora es la dependencia (a nivel de aspecto visual y no de interfaz de programación) de la aplicación resultante del sistema operativo sobre el que se ejecuta. La interfaz del *workbench* de eclipse también depende de una capa intermedia de interfaz gráfica de usuario (GUI) llamada JFace que simplifica la construcción de aplicaciones basadas en SWT.

3.2.6. Plug-in Development Environment (PDE)

El PDE es un plugin que proporciona, al igual que el JDT, una especialización del entorno. En este caso proporciona herramientas para automatizar la creación, manipulación, depuración y despliegue de nuevos plugins.

3.2.7. Subversion

Introducción

Subversion es un software de sistema de control de versiones diseñado específicamente para reemplazar al popular CVS. Es un software libre bajo una licencia de tipo Apache/BSD y se le conoce también como SVN.

Características

Una característica importante de Subversion es que, a diferencia de CVS, los archivos versionados no tienen cada uno un número de revisión independiente. En cambio, todo el repositorio tiene un único número de versión que identifica un estado común de todos los archivos del repositorio en un instante determinado.

Subversion puede acceder al repositorio a través de redes, lo que le permite ser usado por personas que se encuentran en distintos ordenadores. A cierto nivel, la posibilidad de que varias personas puedan modificar y administrar el mismo conjunto de datos desde sus respectivas ubicaciones fomenta la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones. Y puesto que el trabajo se encuentra bajo el control de versiones, no hay razón para temer que la calidad del mismo vaya a verse afectada.

Ventajas e inconvenientes

Las ventajas e inconvenientes de utilizar Subversion son las siguientes:

Ventajas:

- Se sigue la historia de los archivos y directorios a través de copias y renombrados.
- Las modificaciones son atómicas.

- La creación de ramas y etiquetas es una operación más eficiente que en CVS. Tiene complejidad constante O(1) frente a la complejidad lineal O(n) de CVS.
- Se envían sólo las diferencias en ambas direcciones, mientras que en CVS siempre se envían al servidor archivos completos.
- Puede ser proporcionado desde un servidor Apache lo que permite mayor transparencia con algunos clientes.
- Maneja eficientemente archivos binarios a diferencia de CVS, que los trata internamente como si fueran de texto.
- Permite selectívamente el bloqueo de archivos. Se usa en archivos binarios que, al no poder fusionarse fácilmente, conviene que no sean editados por más de una persona a la vez.
- Cuando se usa integrado junto a Apache permite utilizar todas las opciones que este servidor provee a la hora de autentificar archivos (SQL, LDAP, PAM, etc.).

Inconvenientes:

- El manejo de cambio de nombres de archivos no es completo. Lo maneja como la suma de una operación de copia y una de borrado.

Clientes

Existen multitud de interfaces o clientes de acceso a Subversion como son TortoiseSVN, ViewVC, RapidSVN, etc. Ya que este proyecto ha consistido en el desarrollo de un plugin para eclipse se ha decidido utilizar Subversive, que es un plugin para eclipse que proporciona la interfaz de acceso a Subversion.

Capítulo 4

Manual de usuario

En este capítulo se pretende explicar cómo debe utilizar un usuario cualquiera el plugin desarrollado en este proyecto.

Tras haber instalado correctamente el plugin (Capitulo 5) el usuario podrá elegir entre las siguientes opciones:

- Ejecución o depuración local de una aplicación.
- Ejecución o depuración remota de una aplicación.
- Recuperar el estado de una ejecución.

4.1. Ejecución o depuración de una aplicación en local

Un usuario podrá ejecutar o depurar una aplicación de tres maneras distintas, las cuales se describen a continuación:

- Desde el editor de texto de eclipse. El usuario deberá seguir los siguientes pasos:

- Click con el botón derecho del ratón sobre el editor.
- Si se desea ejecutar la aplicación:
  - Click con el botón izquierdo sobre el menú *Run as*
- Si se desea depurar la aplicación:
  - Click sobre el menú *Debug as*
- Click sobre *Local Java Versioned Application*

- Desde el explorador de paquetes de Eclipse. Se deberán seguir los siguientes pasos:
  - Click con el botón derecho del ratón sobre la clase a ejecutar o depurar.
  - Si se desea ejecutar la aplicación:
    - Click sobre el menú *Run as*
  - Si se desea depurar la aplicación:
    - Click sobre el menú *Debug as*
  - Click sobre *Local Java Versioned Application*
- Desde el menú de Eclipse. Los pasos serán los descritos a continuación:
  - Click con el botón derecho del ratón sobre la clase a ejecutar o depurar.
  - Si se desea ejecutar la aplicación:
    - Click sobre el menú *Run as*
  - Si se desea depurar la aplicación:
    - Click sobre el menú *Debug as*
  - Click sobre *Local Java Versioned Application*

Las siguientes figuras 4.1 y 4.2 muestran cómo sería la depuración en local de una aplicación Java versionada. En la primera de ellas se observa cómo acceder al *shortcut* que permite iniciar la depuración de la aplicación. En la segunda se observan las dos ventanas con código fuente que aparecen durante la ejecución. La ventana situada a la izquierda muestra el código fuente que se encuentra en

el *workspace* de Eclipse y la ventana situada a la derecha el código fuente que se encuentra en el fichero zip generado, y por lo tanto es el que se está depurando.

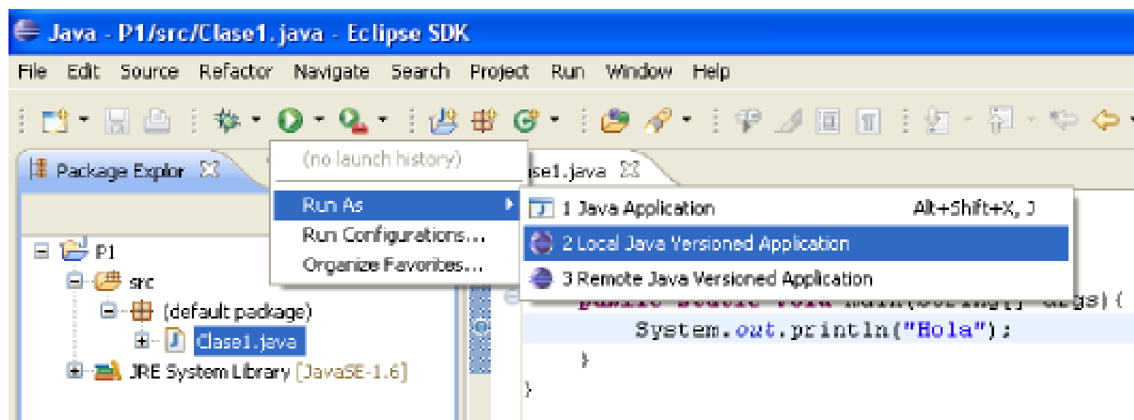

Figura 4.1: Depuración en local de una aplicación Java versionada.

Figura 4.2: Perspectiva de depuración de Eclipse.

4.2. Ejecución o depuración de una aplicación en remoto

Para la ejecución remota de una aplicación se necesitan una serie de parámetros para la conexión con el servidor remoto, así que los pasos a seguir serán ligeramente distintos. El usuario podrá elegir, al igual que anteriormente ejecutar la aplicación desde el editor de texto, el explorador de paquetes o el menú de Eclipse. En este caso deberá elegir la opción *Remote Java Versioned Application* en vez de la opción *Local Java Versioned Application*.

La Figura 4.3 muestra cómo acceder al menú de depuración remota de una aplicación java versionada.

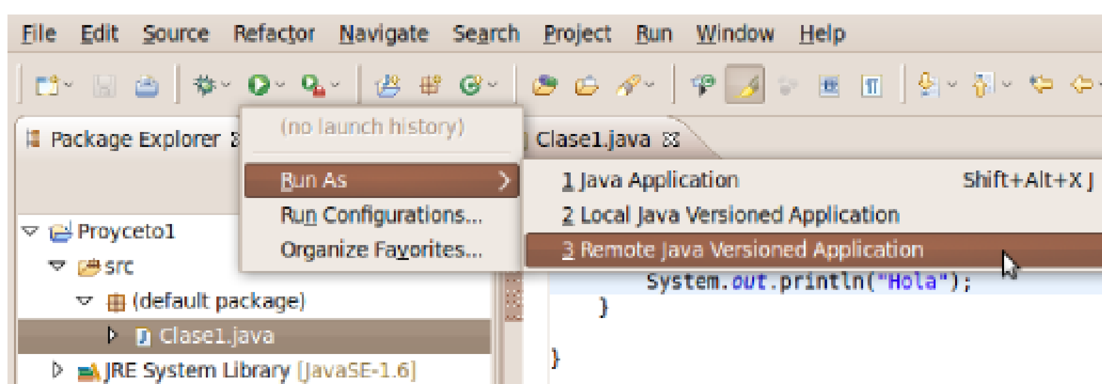

Figura 4.3: Menú de ejecución remota de una aplicación java versionada.

Una vez seleccionada la opción aparecerá una pequeña interfaz de usuario en la que se tendrán que introducir los siguientes parámetros de manera obligatoria:

- Contraseña de acceso.
- Host de acceso.
- Puerto de acceso.

De manera opcional el usuario podrá especificar estos otros:

- Parámetros del programa.
- Parámetros de la máquina virtual.

La Figura 4.4 muestra la ventana descrita anteriormente.

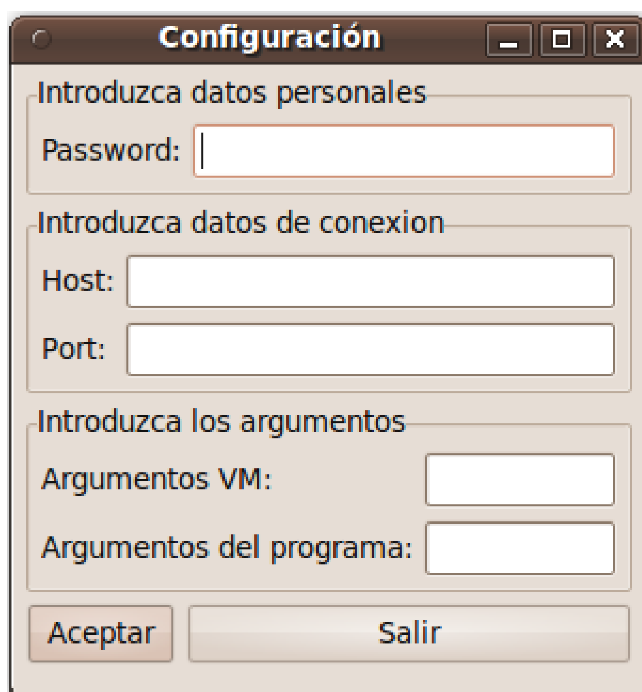

Figura 4.4: Panel o ventana principal del acceso remoto.

Una vez introducidos todos los parámetros habrá que hacer click con el botón izquierdo del ratón sobre el botón *Aceptar*. Eclipse pasará a mostrar su perspectiva de depuración o *Debug* donde se podrá seguir la ejecución de la aplicación detalladamente.

La Figura 4.5 muestra parte de la depuración de una aplicación java.

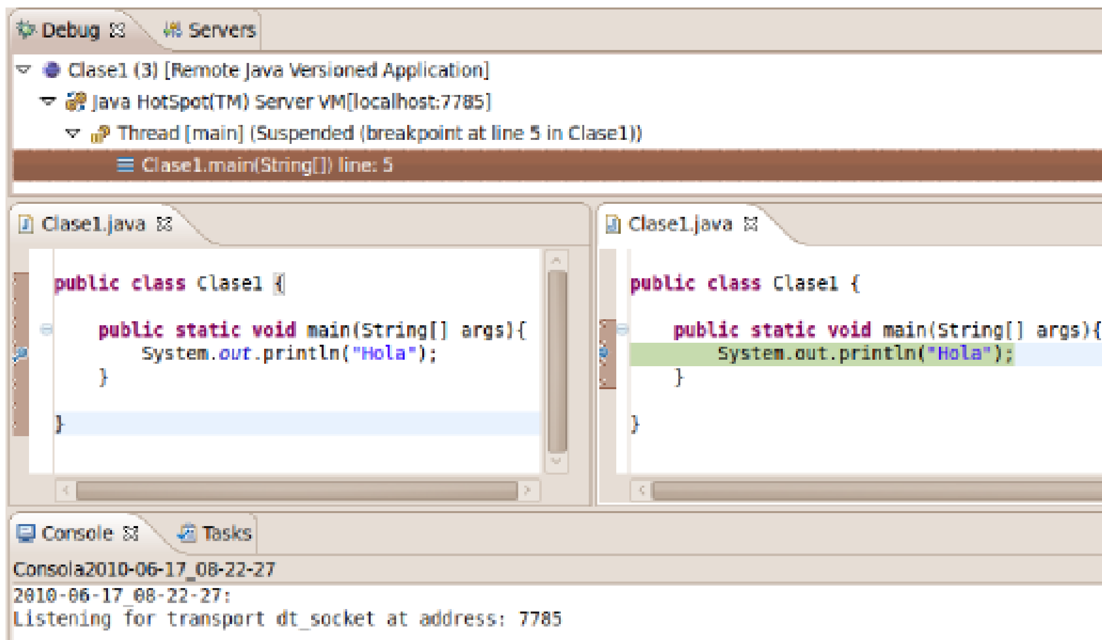

Figura 4.5: Perspectiva de depuración de Eclipse y salida por consola.

4.3. Recuperar el estado de una ejecución

Para visualizar y recuperar el estado de una ejecución existe una vista, *Optsicom View*, desde la que se puede observar en todo momento el estado de las ejecuciones.

Para acceder a la vista nombrada anteriormente los pasos serán los siguientes:

- Click en el menú *Window*
- Click en el menú *Show View*
- Click en el menú *Other...*

Tras seguir estos pasos aparecerá una ventana en la que se deberá buscar la carpeta *Optsicom Category* y tras hacer doble click en ella o desplegarla habrá que

seleccionar *Optsicom View* y hacer click en el botón Aceptar.

Esta ventana está compuesta de una tabla formada por tres columnas y cuya información es:

- **Proceso:** Informa del nombre del proceso ejecutado. Será la fecha concatenada con la hora de la ejecución.
- **Host:** Informa de la máquina en la que se ha ejecutado la aplicación.
- **State:** Indica el estado actual de la ejecución. Este puede ser *Finished*, *Running* o *Undetermined* en caso de que no se pueda conocer el estado actual.

La Figura 4.6 muestra la vista descrita anteriormente.

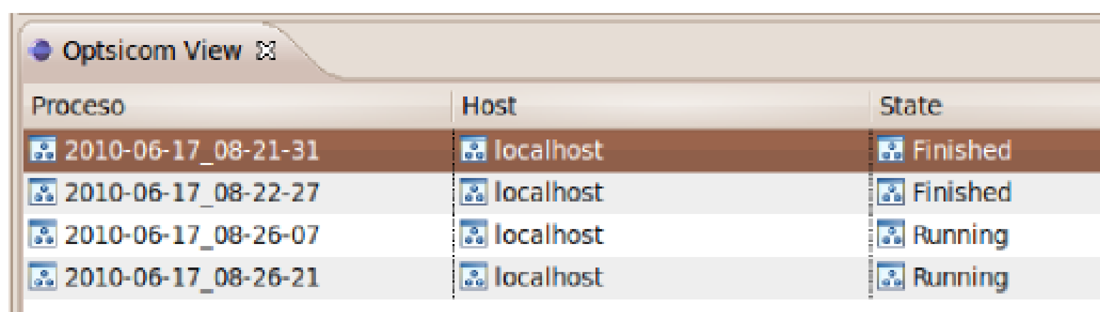

Figura 4.6: Vista Optsicom View.

Esta ventana también implementa la funcionalidad para observar la salida de la ejecución, eliminar una tarea de la tabla y actualizar el estado de las tareas. Para poder acceder a esta funcionalidad habrá que hacer click con el botón derecho sobre la tabla y aparecerán las siguientes opciones:

- **Ejecutar Tarea:** Al hacer click sobre ella aparecerá una consola que mostrará la salida estándar por consola de la ejecución.
- **Eliminar Tarea:** Al hacer click sobre ella eliminará la tarea de la tabla.

- **Actualizar estado:** Al hacer click sobre esta opción se actualizarán los estados de todos los procesos mostrados en la tabla.

La Figura 4.7 muestra la funcionalidad que proporciona la vista.

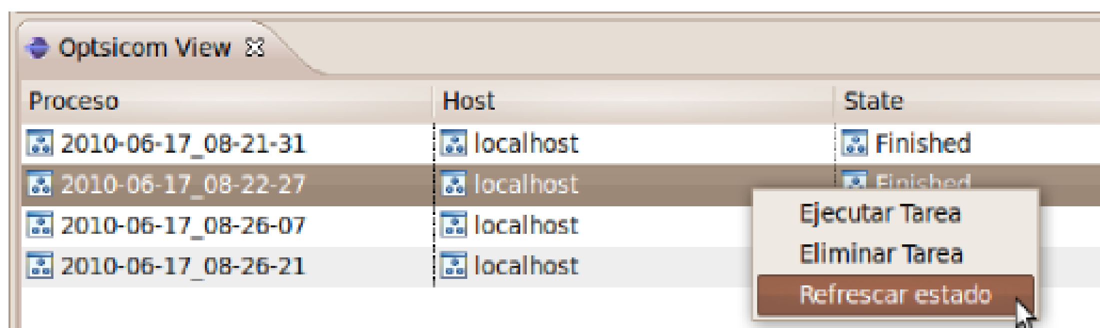

Figura 4.7: Funcionalidad de la vista Optsicom View.

Capítulo 5

Manual de instalación

En este capítulo se explicará cómo un usuario debe instalar el plugin desarrollado en este proyecto. Este manual de instalación está pensando para la instalación sobre *Eclipse Galileo*.

5.1. Requisitos

Los requisitos indispensables para la instalación del plugin son los siguientes:

- Tener instalado el Java Development Kit o (JDK). En caso de no tenerlo se podrá obtener desde la página oficial de su proveedor *http://java.sun.com/*.

- Tener instalado el entorno de desarrollo (IDE) Eclipse. En caso de no tenerlo se podrá instalar accediendo a la pagina *http://www.eclipse.org/* y descargarlo desde allí.

5.2. Procedimiento

Una vez cumplidos los requisitos el plugin se instalará siguiendo los siguientes pasos:

1. Abrir el entorno de desarrollo Eclipse.

2. En primer lugar habrá que seleccionar un *workspace* en caso de que no se tenga.

   La Figura 5.1 muestra cómo seleccionar un nuevo workspace. Además, se podrá elegir la opción *Use this as default and do not ask again* para que no vuelva a preguntarse por el workspace.

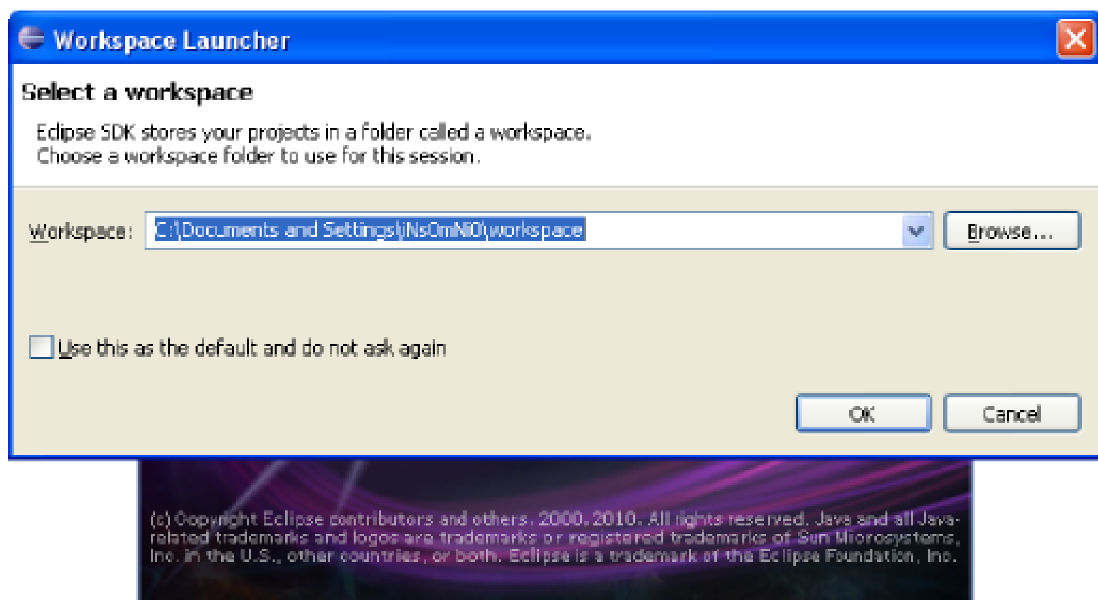

Figura 5.1: Selección del workspace de eclipse

3. Posteriormente habrá que seleccionar la opción *Install new Software...* perteneciente al menú *Help* que se puede encontrar en la parte superior de Eclipse. Figura 5.2.

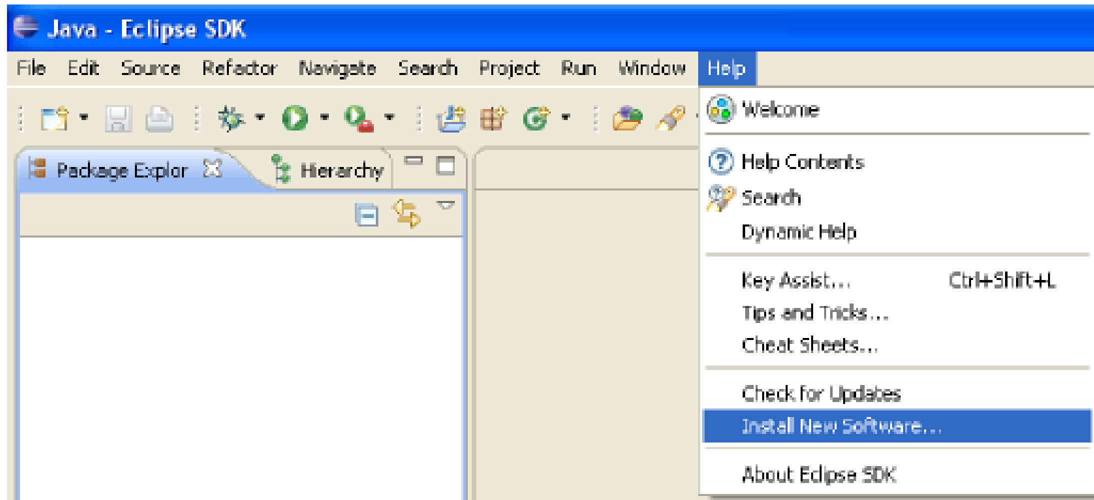

Figura 5.2: Install new Software...

4. A continuación aparecerá una ventana en la que se deberá hacer click con el botón izquierdo del ratón para añadir un sitio web desde el cual descargar el plugin. Figura 5.3. Aparecerá una nueva ventana que nos permitirá escribir el nombre del sitio y su dirección web mediante los campos *Name* y *Location*. Ademas permitirá elegir un directorio o un fichero local para la instalación del plugin mediante los botones *Local...* y *Archive...*.

5. En este caso el plugin se deberá instalar a través de una dirección web así que habrá que completar los campos *Name: Optsicom RES* y *Location: http://www.sidelab.es/files/res/update-site/* y hacer click con el botón izquierdo del ratón en *OK*.

6. Ahora en la ventana se podrá ver el plugin a instalar. Figura 5.4. Si se hace click sobre el símbolo + se desplegará y se podrá ver la versión del plugin. Tras hacer click en la *checkbox* de la última versión del plugin habrá que hacer click con el botón izquierdo del ratón sobre *Next*.

   Para avanzar por las siguientes ventanas bastará con hacer click con el botón izquierdo del ratón sobre *Next* hasta llegar a la última ventana en la que habrá que hacer click sobre *Finish*.

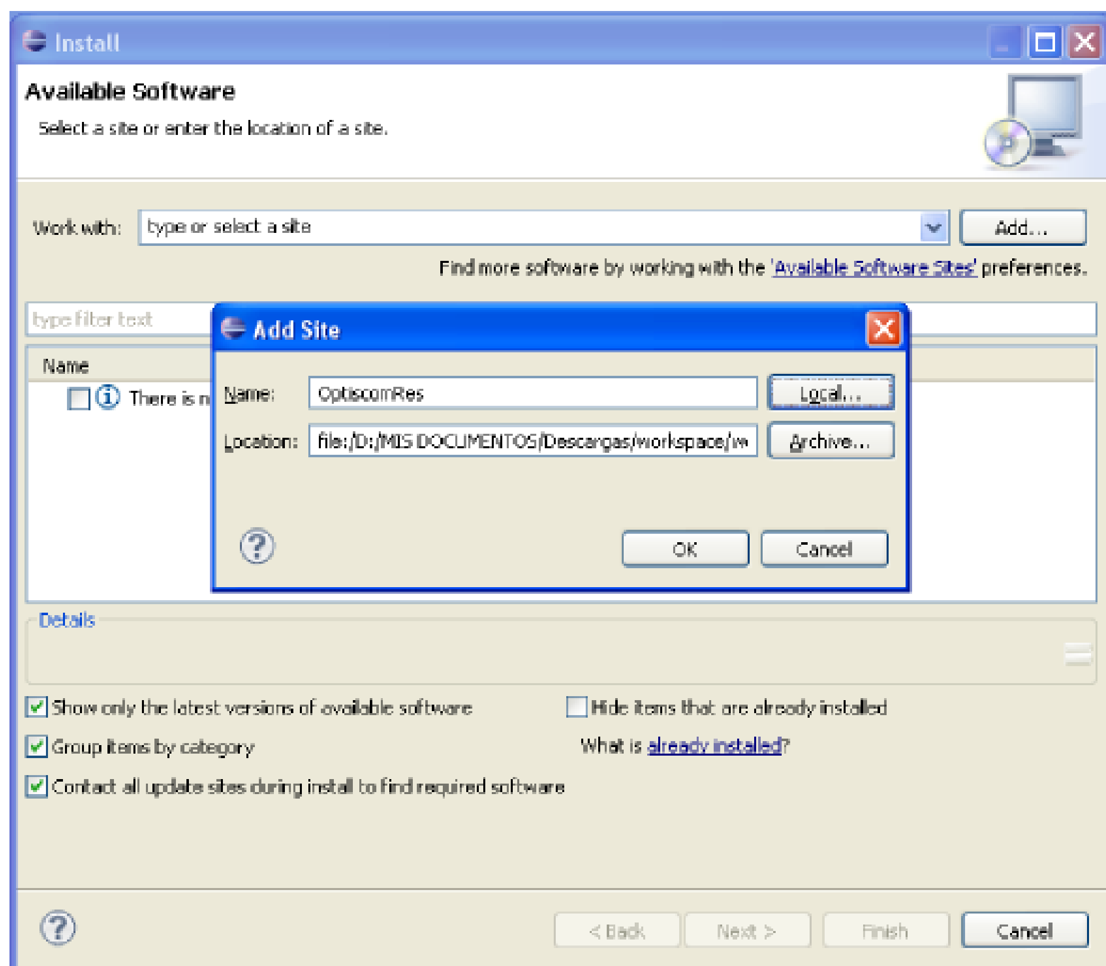

Figura 5.3: Ventana de instalación 1.

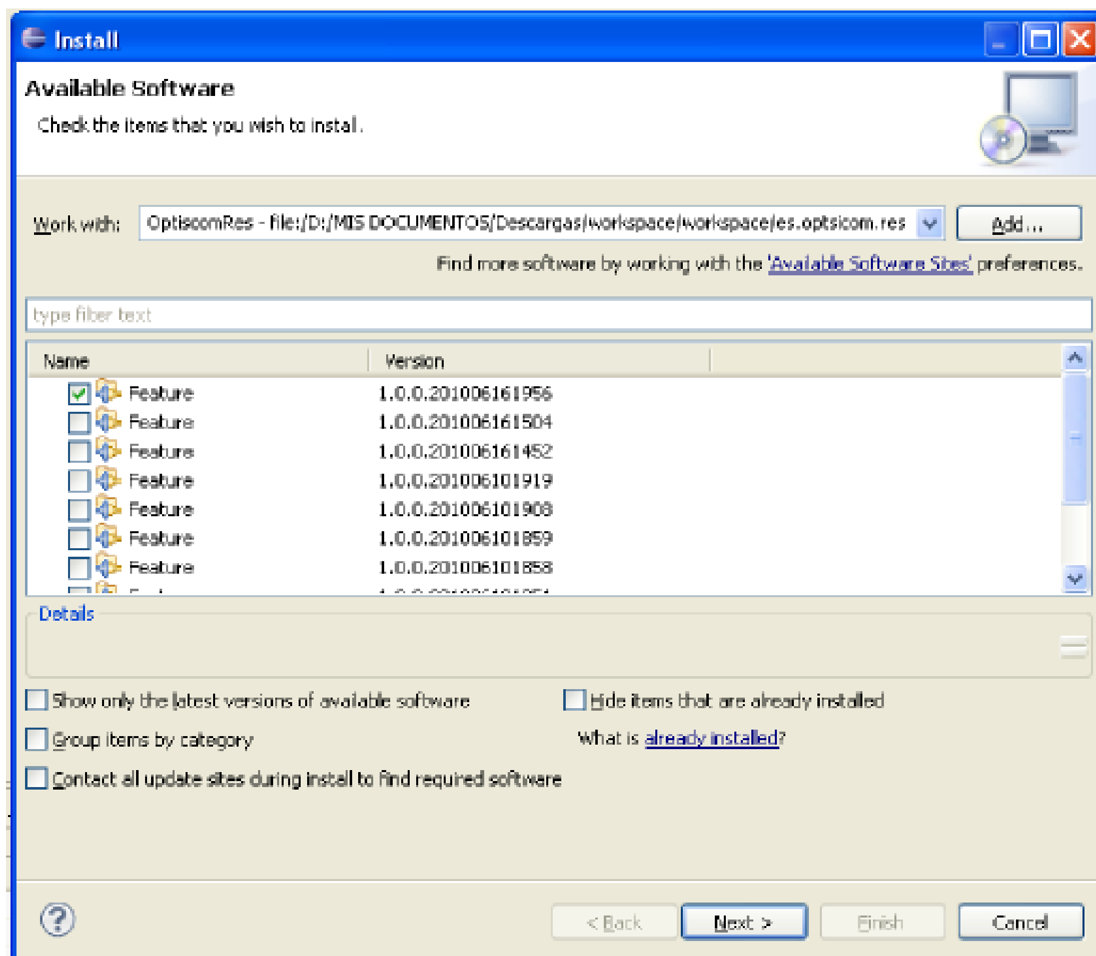

Figura 5.4: Ventana de instalación 2.

Capítulo 6

Descripción Informática

6.1. Especificación de requisitos

6.1.1. Requisitos funcionales

Se pueden diferenciar dos grandes apartados a la hora de definir los requisitos:

- Requisitos del cliente o plugin.
- Requisitos del servidor.

Requisitos del cliente

- **RF 1:** El plugin debe permitir la ejecución, depuración y versionado de aplicaciones java de manera local.
- **RF 2:** El plugin debe permitir la ejecución, depuración y versionado de aplicaciones java de manera remota.
- **RF 2.1:** El plugin debe generar y enviar al servidor un fichero *.zip* con todos los ficheros de un proyecto Java.

- **RF 2.1.1:** El plugin debe ser capaz de reconocer el proyecto que contiene el código fuente y obtener todos sus ficheros.
- **RF 2.1.2:** El plugin debe ser capaz de detectar dependencias entre el proyecto a ejecutar y otros proyectos existentes en el *workspace* de Eclipse y resolverlas sin problema.
- **RF 2.2:** El plugin debe permitir establecer las opciones de lanzamiento de la maquina virtual remota en la que se va a ejecutar la aplicación.
- **RF 2.3:** El plugin debe permitir establecer los argumentos de entrada del programa que se va a ejecutar.
- **RF 2.4:** El plugin debe permitir especificar el *host* al que se va a conectar remotamente para ejecutar la aplicación.
- **RF 2.5:** El plugin debe permitir especificar el puerto donde se puede conectar el depurador en caso de que se desee depurar una aplicación remotamente.
- **RF 2.6:** El plugin deberá establecer un identificador de trabajo para cada ejecución. Este identificador será la fecha y hora de ejecución del programa concatenadas con el siguiente formato: AAAA-MM-DD_HH-MM-SS
- **RF 2.7:** El plugin deberá guardar el estado de las ejecuciones realizadas incluso cuando éstas hayan terminado.
- **RF 2.7.1:** El plugin deberá permitir eliminar el estado de cualquiera de las ejecuciones guardadas anteriormente.
- **RF 2.7.2:** El plugin deberá permitir actualizar el estado de cualquiera de las ejecuciones guardadas anteriormente.
- **RF 2.7.3:** El plugin deberá recuperar el estado de cualquiera de las ejecuciones guardadas anteriormente.
- **RF 3:** El plugin deberá capturar la salida estándar de la ejecución de una aplicación.
- **RF 3.1:** El plugin deberá mostrar en una consola la salida estándar de la ejecución de una aplicación.

- **RF 3.2:** El plugin deberá generar un fichero de texto con la salida estándar de la ejecución de una aplicación.
- **RF 4:** El plugin deberá integrar las ejecuciones o depuraciones lanzadas con el *launching* de Eclipse, es decir, deberá ser posible guardar y recuperar cada una de las configuraciones con las que fue lanzada una aplicación.

Requisitos del servidor

- **RF 5:** El servidor debe poder autenticar a los usuarios que se conecten mediante una password.
- **RF 6:** El servidor podrá recibir ficheros *.zip* a través de un socket.
- **RF 6.1:** Los sockets por los que se recibe la información deben ser sockets seguros (SSL).
- **RF 6.2:** El servidor debe poder descomprimir los ficheros *.zip* recibidos anteriormente.
- **RF 7:** El servidor debe permitir la ejecución de aplicaciones Java.
- **RF 8:** El servidor debe capturar la salida estándar de la ejecución de cada aplicación Java ejecutada.
- **RF 8.1:** El servidor debe generar un fichero con la salida estándar de la ejecución de cada aplicación Java ejecutada.
- **RF 8.2:** El servidor debe ser capaz de enviar a través de un socket el fichero generado anteriormente.
- **RF 9:** El servidor debe estar protegido ante posibles errores.
- **RF 9.1:** El servidor debe estar protegido ante posibles cortes de luz y deberá guardar en el disco duro el estado de todas las ejecuciones efectuadas.
- **RF 9.2:** El servidor debe poder recuperarse de los errores y mantener el estado que tenía antes de producirse el error cuando se reconecte.

6.1.2. Requisitos no funcionales

- **RNF 1:** El plugin debe ser implementado de tal manera que su ejecución sea posible en cualquier plataforma, ya sea Windows, Unix ...

- **RNF 2:** El servidor también deberá ser multiplataforma, aunque no necesariamente deberá disponer de las mismas funciones en ambas plataformas.

6.2. Diseño

6.2.1. Casos de uso

La Figura 6.1 representa los posibles casos de uso de la aplicación. Estas serían las acciones que podría efectuar un usuario una vez instalado el plugin.

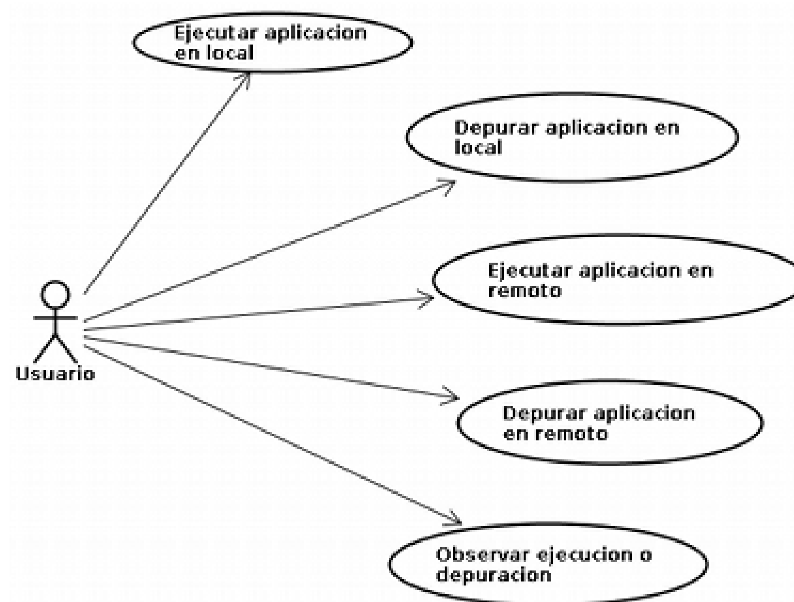

Figura 6.1: Casos de uso

6.2.2. Diagrama de clases

Dentro del proyecto se pueden diferenciar tres paquetes muy claramente, el paquete que contiene todos los fuentes del servidor, el que contiene los del cliente y el que contiene los fuentes de las interfaces que permiten la comunicación entre el cliente y el servidor. A continuación se procederá a comentar de manera más detallada las clases que componen los paquetes de la Figura 6.2.

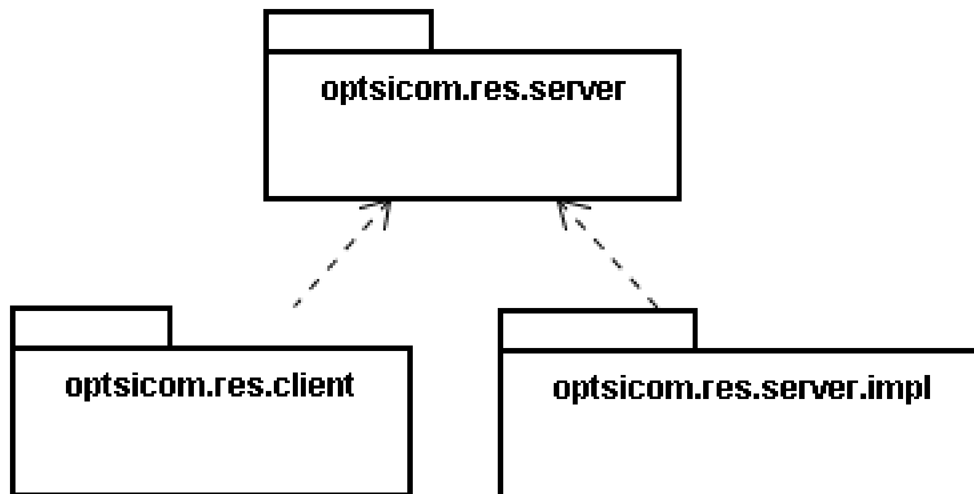

Figura 6.2: Paquetes principales de la aplicación

Paquetes optsicom.res.server.impl y optsicom.res.server

La Figura 6.3 muestra un diagrama con las clases principales y las interfaces que se utilizan en los paquetes *optsicom.res.server.impl* y *optsicom.res.server.*

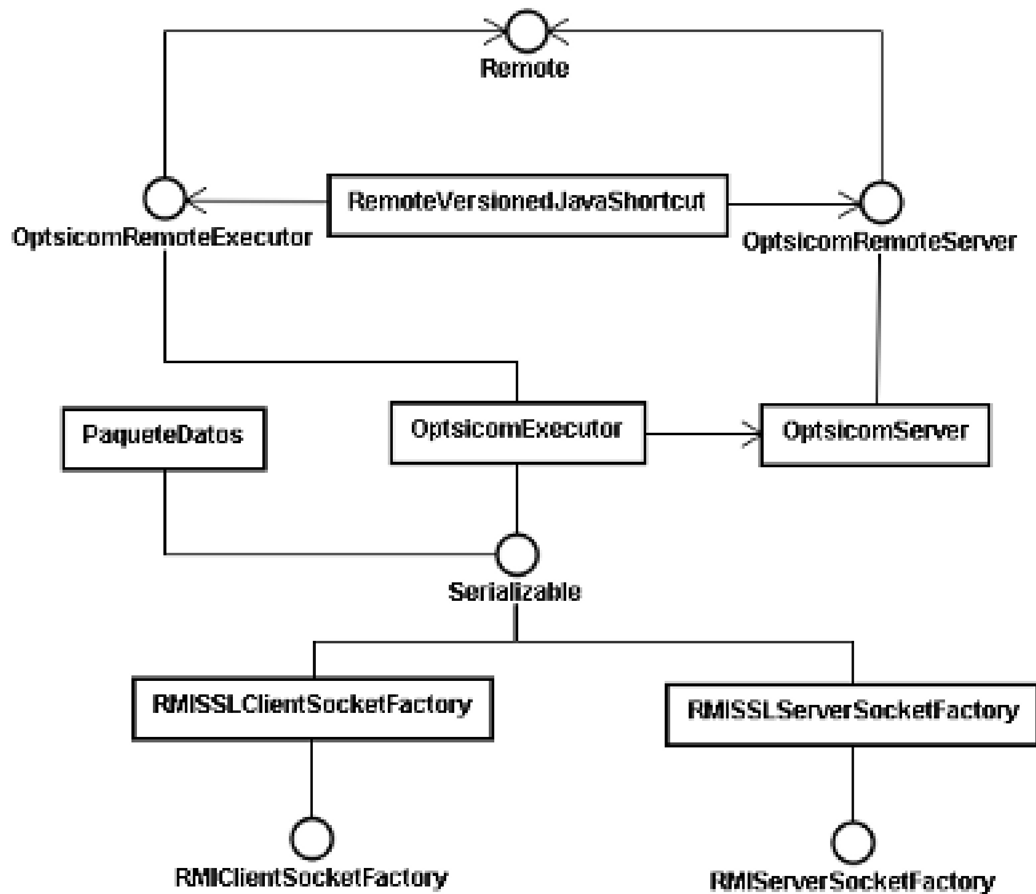

Figura 6.3: Clases de los paquetes *optsicom.res.server.impl* y *optsicom.res.server*

Estos paquetes agrupan toda la funcionalidad referente al servidor. El paquete *optsicom.res.server.impl* implementa toda la funcionalidad que proporciona el paquete de interfaces *optsicom.res.server*. Éstas son las clases que componen los paquetes:

- **UnicastRemoteObjet:** Clase que pertenece a la API de Java que proporciona el JRE (*Java Runtime Extension*) de la que heredan *OptsicomExecutor* y *OptsicomServer*. Esta clase permite tratar remotamente las instancias de las clases que hereden de ella.

- **OptsicomRemoteExecutor:** Es la interfaz de la clase *OptsicomExecutor*. Podría considerarse como un contrato entre el cliente y el servidor y expone la funcionalidad que posee la clase *OptsicomExecutor*.

- **OptsicomExecutor:** Es la clase que implementa la funcionalidad del Executor. La clase Executor será creada por el servidor en la primera conexión del cliente y posteriormente se encargará de la comunicación entre el cliente y el servidor. La clase Executor no mantiene ningún tipo de estado, es decir, no guarda nada importante, en caso de que el objeto Executor se pierda o muera la ejecución podría seguir perfectamente invocando a un nuevo objeto Executor

- **OptsicomRemoteServer:** Es la interfaz de la clase OptsicomServer. Podría considerarse como un contrato entre el cliente y el servidor y expone la funcionalidad que posee la clase *OptsicomServer*.

- **OptsicomServer:** Es la clase que implementa la funcionalidad del servidor. Se encarga de crear los sockets seguros para la transmisión de datos, guardar la información de cada ejecución, autenticar a los usuarios...

- **RMISSLServerSocketFactory:** Es la clase que se encarga de generar los sockets seguros con los que se comunicará el servidor a través de la tecnología SSL (*Socket Secure Layer*).

- **RMISSLClientSocketFactory:** Es la clase que se encarga de generar los sockets seguros con los que se comunicará el cliente a través de la tecnología SSL (*Socket Secure Layer*).

- **Serializable:** Es la clase proporcionada por Java que permite serializar objetos y que éstos se envíen por la red.

Paquete optsicom.res.client

Este paquete contiene toda la funcionalidad referente al plugin. A su vez se subdivide en dos paquetes principales: *optsicom.res.client.local* y *optsicom.res.client.remote*. Las siguientes figuras: 6.4, 6.5, 6.6 representan las clases e interfaces que participan en estos paquetes.

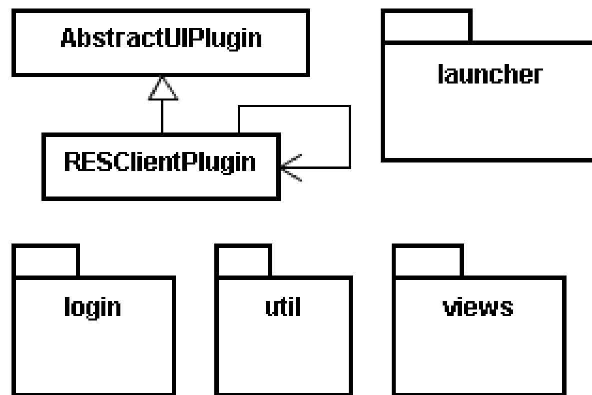

Figura 6.4: Clases del paquete *optsicom.res.client*

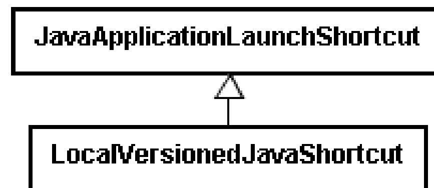

Figura 6.5: Clases del paquete *optsicom.res.client.local*

Estas son las clases que componen los paquetes *optsicom.res.client*,*optsicom.res.client.local* y *optsicom.res.client.remote*:

- **AbstractUIPlugin:** Es una clase abstracta básica diseñada para permitir la integración de los plugins en la plataforma Eclipse.

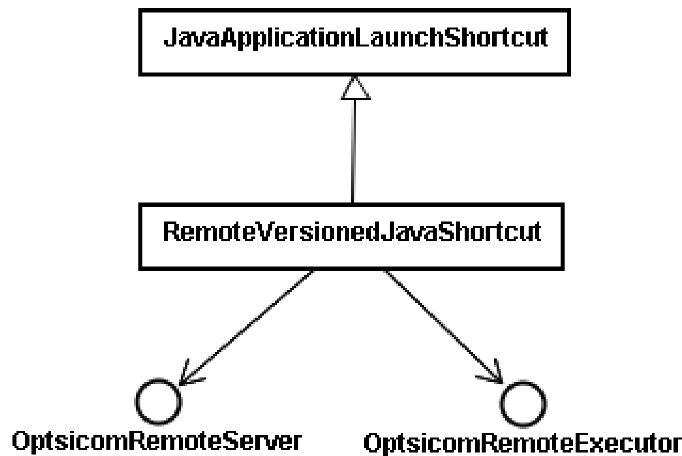

Figura 6.6: Clases del paquete *optsicom.res.client.remote*

- **RESClientPlugin:** Clase que hereda de *AbstractUIPlugin* y que se encarga de iniciar y detener el plugin cuando corresponda.

- **JavaApplicationLaunchShortcut:** Es la clase que representa el acceso directo al lanzamiento de una aplicación Java. Se llama desde Eclipse cuando se ejecuta o depura una clase y se encarga de recopilar toda la información necesaria para su correcta ejecución.

- **LocalVersionedJavaShortcut:** Clase que hereda de *JavaApplication-LaunchShortcut* y la que contiene toda la funcionalidad referente a los sucesos que tienen lugar cuando se ejecuta una *Local Java Versioned Application.*

- **RemoteVersionedJavaShortcut:** Clase que hereda de *JavaApplication-LaunchShortcut* y la que contiene toda la funcionalidad referente a los sucesos que tienen lugar cuando se ejecuta una *Remote Java Versioned Application*

Además existen otros paquetes como *optsicom.res.client.login*, *optsicom.res.client.util* y *optsicom.res.client.views* que contienen funcionalidad de importancia para el plugin y la que se describe a continuación:

- **optsicom.res.client.login:** Este paquete contiene una clase que se encarga de crear una interfaz gráfica de usuario para cuando se ejecuta o depura remotamente una aplicación. Esta clase presenta un formulario con los campos necesarios para la ejecución remota como son:
    - Contraseña del usuario
    - Host remoto.
    - Puerto remoto.
    - Parámetros del programa a ejecutar.
    - Parámetros de la maquina virtual.
- **optsicom.res.client.util:** Este paquete contiene una clase que se encarga de la generación de los ficheros *.zip*. Esta clase se encarga de buscar el proyecto que se tiene que ejecutar, buscar las posibles dependencias del proyecto y generar el fichero *.zip* de manera correcta.

6.2.3. Diagramas de secuencia

Ejecución local de una aplicación

La Figura 6.7 representa la secuencia de acciones que se producen al ejecutar una aplicación en local. Se puede observar cómo el cliente se comunica con el *Shortcut* para que este configure correctamente la ejecución y genere el fichero *.zip* para posteriormente ceder el flujo de la ejecución al jdt.

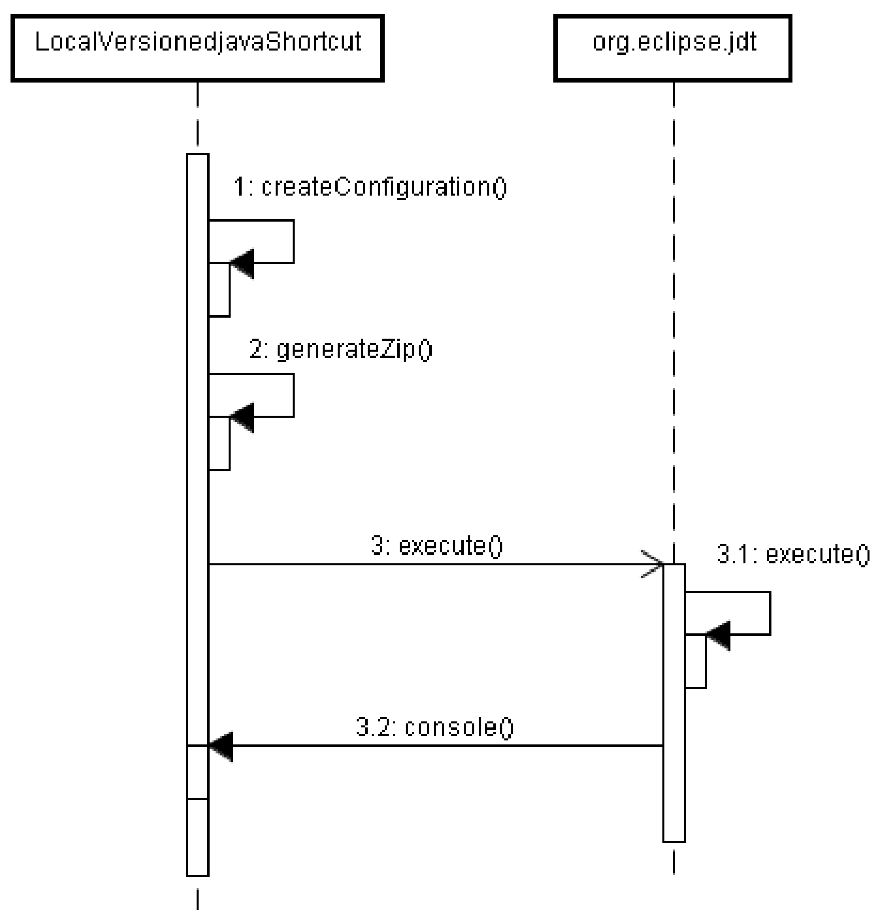

Figura 6.7: Ejecución local

Ejecución remota de una aplicación

La Figura 6.8 representa la secuencia de acciones que se producen al ejecutar una aplicación de forma remota. Se puede observar cómo el cliente únicamente se comunica con el servidor para pedirle un objeto *Executor* y una vez obtenido, todas las acciones para comunicarse con el servidor se llevan a cabo a través del *Executor*.

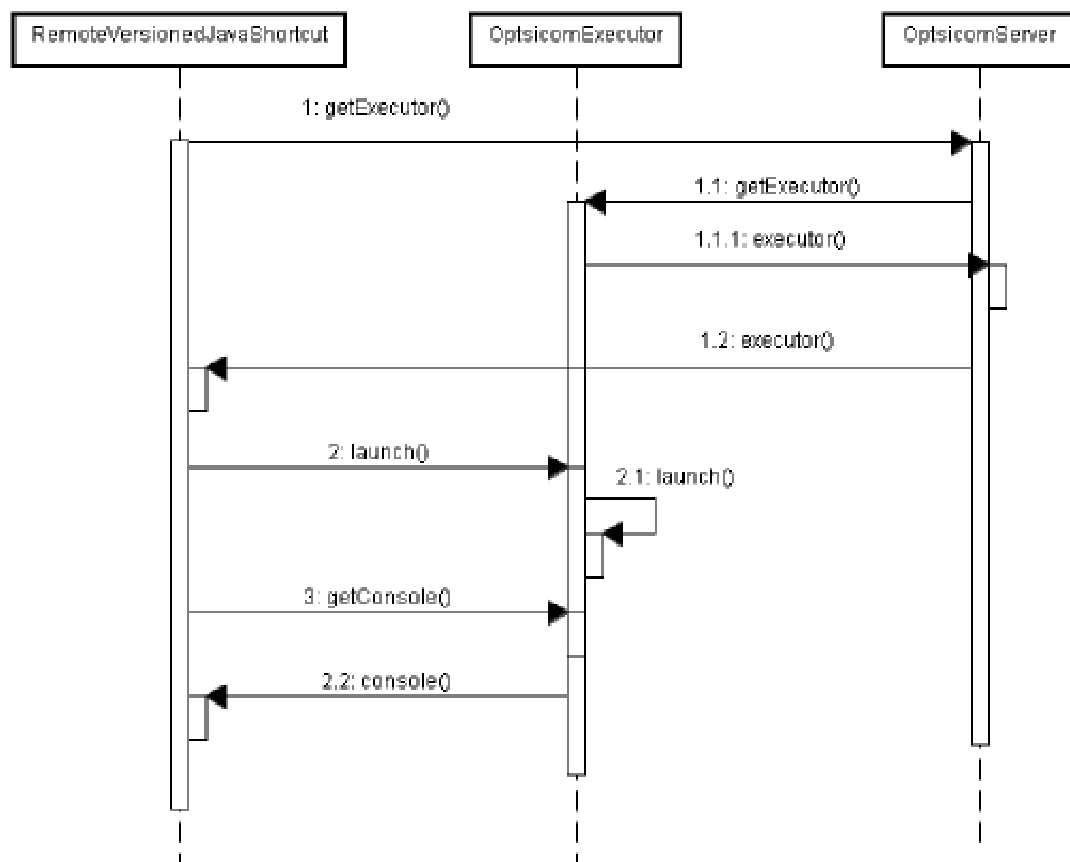

Figura 6.8: Ejecución remota

Desconexión en la ejecución remota de una aplicación

La Figura 6.9 representa la secuencia de acciones que se producen cuando tiene lugar una desconexión por parte del cliente. Aquí se puede observar perfectamente cómo el cliente se comunica de nuevo con el servidor para pedir un objeto *OptsicomExecutor*, ya que es el que le permitirá continuar con su ejecución.

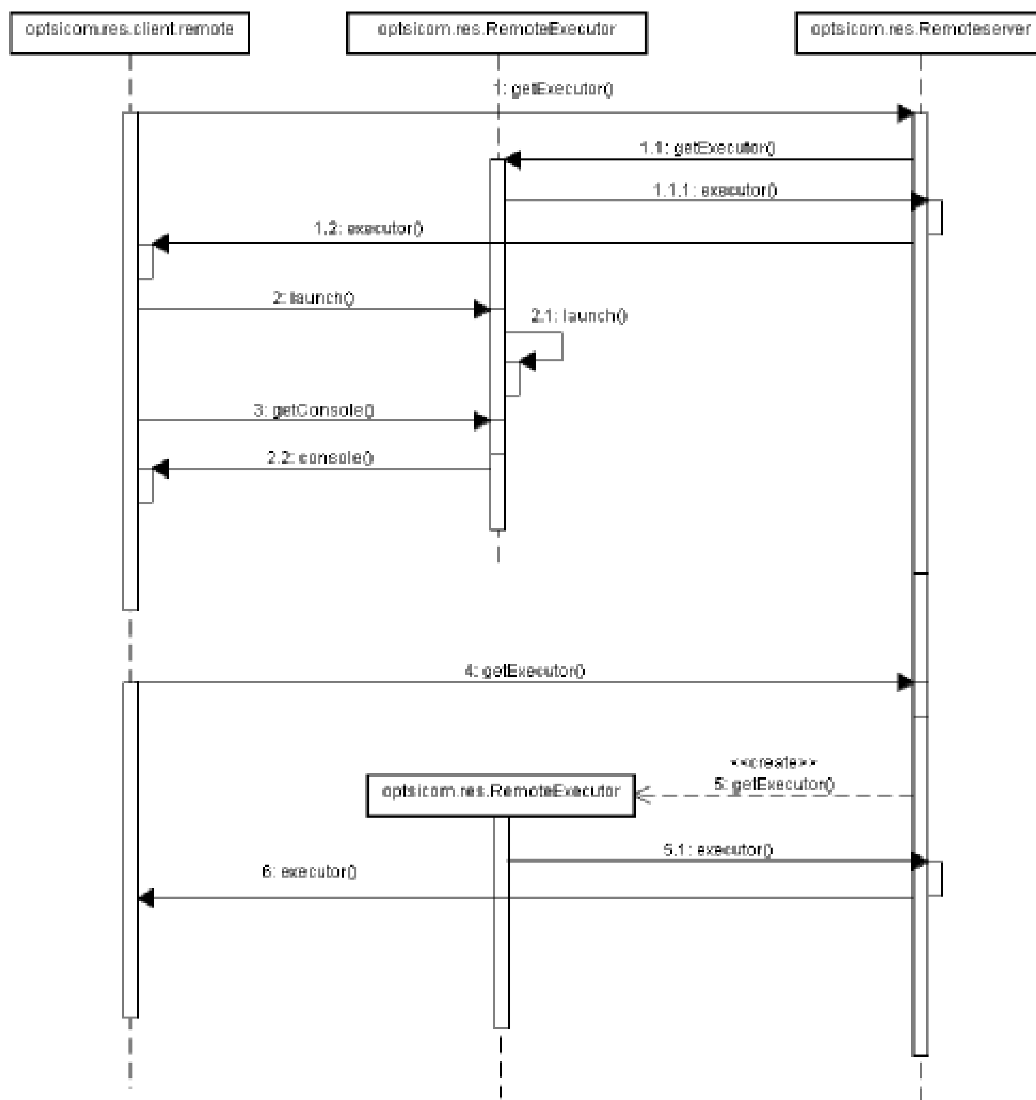

Figura 6.9: Desconexión de una ejecución remota

6.3. Implementación

Para poder desarrollar un plugin, en primer lugar hay que crear el tipo de proyecto de Eclipse apropiado mediante el menú *File->New->Project->Plug-in Development->Plug-in Project*. Lo mas cómodo para crear el proyecto es elegir *Custom Plug-in wizard* como template ya que proporciona una pequeña estructura base para comenzar.

Existe un fichero *plugin.xml* que es el más importante en un plugin y es donde se van a definir las extensiones del plugin. Este plugin lo que realmente quiere es definir un *Launcher*. Un launcher permite configurar cómo ejecutar y depurar programas y eso es precisamente lo que desea hacer este plugin. Todo esto se hará desde la pestaña *Extensions*. Las figuras 6.10 y 6.11 muestran las dos pestañas mas importantes en el desarrollo de un plugin.

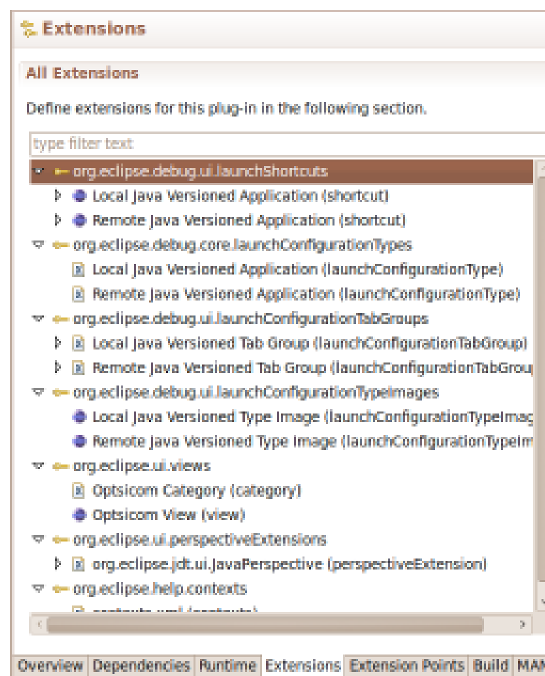

Figura 6.10: Pestaña Extensions

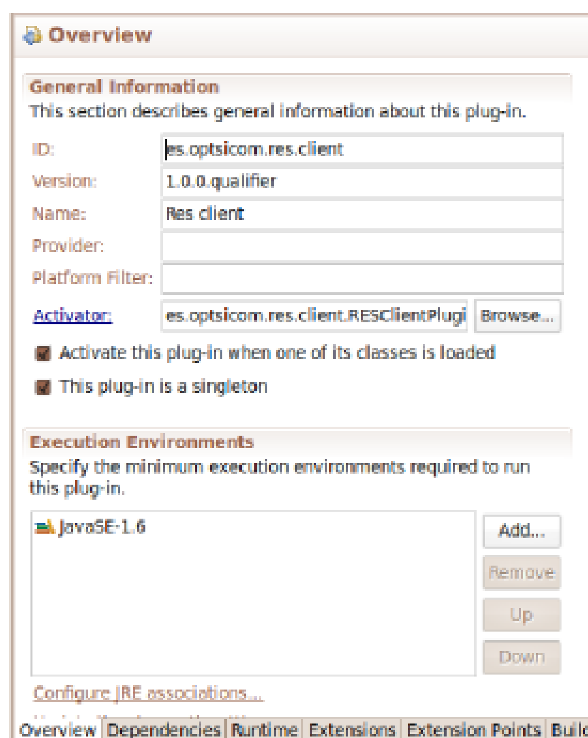

Figura 6.11: Pestaña Overview

6.3.1. Extensiones (Extensions)

Las siguientes extensiones han sido definidas para el desarrollo del plugin:

- org.eclipse.debug.ui.launchShortcuts
- org.eclipse.debug.ui.launchConfigurationTypes
- org.eclipse.debug.ui.launchConfigurationTabGroup
- org.eclipse.debug.ui.launchConfigurationTypeImages
- org.eclipse.ui.views
- org.eclipse.ui.perspectiveExtensions

org.eclipse.debug.ui.launchShortcuts

Esta extensión que proporciona Eclipse es la encargada de instanciar a través del *launchConfigurationType* la configuración para el lanzamiento de aplicaciones Java. Debido a que el fin de este plugin es poder ejecutar aplicaciones Java, basta con reutilizar el *Shortcut* que proporciona Eclipse para el lanzamiento de estas aplicaciones. Así pues los *Shortcuts* implementados heredan de la clase *JavaLaunchShortcut*.

Para este plugin se han definido dos *Shortcuts*, uno por cada tipo de ejecucion existente (local y remota), como se puede ver en la figura 6.10. Estos *Shortcuts* son los siguientes:

- **Local Java Versioned Application (shorcut):** Este Shortcut ha sido configurado para permitir la ejecución y depuración en local de aplicaciones java versionadas. Es el que genera el fichero *.zip* con todos los fuentes y configura las propiedades de lanzamiento.
- **Remote Java Versioned Application (shorcut):** Este Shortcut ha sido configurado para permitir la ejecución y depuración remotas de aplicaciones java versionadas. Se encarga de generar el fichero comprimido *.zip* y de enviar los parámetros necesarios al servidor para que sea éste último el que ejecute la aplicación Java.

org.eclipse.debug.core.launchConfigurationTypes

Este punto de extensión proporciona un mecanismo configurable para el lanzamiento de aplicaciones. Cada configuración del lanzamiento tendrá un nombre, un modo de lanzamiento (run or debug), y especifica un delegado responsable de la implementación del lanzamiento de la aplicación. Al igual que en el caso del Shortcut, eclipse ya proporciona un Delegate para aplicaciones java, así que el Delegate que hay que implementar podrá heredar de *AbstractJavaLaunchConfigurationDelegate*.

Puesto que el plugin podrá ejecutar y depurar aplicaciones tanto de forma local como remota se han desarrollado dos *launchConfigurationTypes*, uno para la configuración del lanzamiento en local y la otra para los lanzamientos en remoto:

- *Local Java Versioned Application (launchConfigurationType).*
- *Remote Java Versioned Application (launchConfigurationType).*

org.eclipse.debug.ui.launchConfigurationTabGroup

Este punto de extensión proporciona las clases que se encargarán de crear las diferentes pestañas de configuración del launcher. De nuevo, como se quieren lanzar aplicaciones Java, las extensiones definidas en este punto podrán utilizar directamente de las clases que proporciona eclipse para estas pestañas:

- *LocalJavaApplicationTabGroup.*
- *RemoteJavaApplicationTabGroup.*

org.eclipse.debug.ui.launchConfigurationTypeImages

Este punto de extensión permite definir un icono para la configuración creada. No es una cosa indispensable pero siempre ayuda identificar una imagen junto al texto relativo al lanzamiento de la aplicación. Tanto para la ejecución de aplicaciones en local como en remoto se ha elegido el icono de Eclipse.

org.eclipse.ui.views

Este punto de extensión permite definir una nueva vista. La vista definida en este punto ha sido llamada *Optsicom View* y es la que permite observar el estado de las ejecuciones en remoto. Esta ventana contiene una tabla en la que se muestra la información sobre el nombre del proceso, la máquina en la que se lanzó y el estado actual de su ejecución.

org.eclipse.ui.perspectiveExtensions

En este punto de extensión se pueden asociar vistas con *Perspectivas* del entorno Eclipse. Únicamente lo que se ha hecho es asociar la vista *Optsicom View* a la perspectiva Java que presenta Eclipse durante el desarrollo de aplicaciones Java para poder utilizar la vista.

6.3.2. Generación de ficheros .zip

Otra de las partes importantes en la implementación ha sido la generación de los ficheros *.zip* con todos los ficheros de un proyecto en concreto. La complicación de esta parte era identificar el proyecto a ejecutar y crear un fichero *.zip* con todos los ficheros contenidos.

Parece una cosa sencilla pero los proyectos no son sólo unas cuantas clases sueltas, todo lo contrario, cada proyecto puede tener multitud de paquetes y clases y además pueden existir dependencias de librerías, de ficheros *.jar*, de otros proyectos, etc.

Así que había que solucionar el problema de encontrar todas esas dependencias y generar un fichero *.zip* que contuviese todos los ficheros del proyecto y todas las dependencias del mismo.

De la misma manera al tener dependencias de otros proyectos había que solucionar posibles problemas de repetición de ficheros así que se determinó que únicamente se incluiría en el *.zip* el primero de todos los ficheros que se encontrasen duplicados.

6.3.3. Socket Secure Layer (SSL) y RMI

Otra de las partes importante del proyecto fue la comunicación entre el servidor y el cliente. Éstos se comunican a través de la tecnología RMI, la cual utiliza sockets para la comunicación, pero estos sockets utilizados por RMI no son

sockets seguros, así que se decidió implementar unos sockets seguros mediante la tecnología SSL. Java proporcionaba unas clases para la creación de sockets seguros así que se desarrollaron dos clases que heredaban de las proporcionadas por Java: *SSLClientSocket* y *SSLServerSocket*.

6.3.4. Desarrollo de la *feature* y del *update_site*

En un principio la depuración del plugin se realizó desde el propio entorno de desarrollo gracias a la funcionalidad que proporciona el *PDE* y que permite generar una nueva instancia de Eclipse en la que probar el plugin.

Posteriormente, se desarrolló una *feature* y un *update_site*. La *feature* es un proyecto Java que contiene el plugin y sus versiones. El *update_site* es otro tipo de proyecto que permite generar un sitio desde el que descargar el plugin para poder instalarlo en otras instancias de Eclipse.

Dicho plugin se instalo en otro Eclipse y se comenzó a hacer pruebas obteniendo resultados distintos a los obtenidos durante la depuración desde el propio entorno de desarrollo, apareciendo incluso problemas que anteriormente no aparecían. Esto ocasionó que se dejara de depurar el plugin desde el propio entorno de desarrollo y que de aquí en adelante todas las pruebas se realizasen instalando el plugin en otro Eclipse.

6.4. Pruebas

Se han realizado las siguientes pruebas para comprobar el correcto funcionamiento de la aplicación.

6.4.1. Generación correcta de los ficheros .zip

Para comprobar la correcta generación de los ficheros *.zip* se llevaron a cabo la pruebas con los siguientes tipos de proyectos Java:

- Proyecto con varios paquetes y varias clases y sin ningún tipo de dependencia.
- Proyecto con varios paquetes y clases con dependencia de un segundo proyecto.
- Proyecto con varios paquetes y clases con dependencia de librerías y diversos ficheros *.jar*.

Al finalizar todas estas pruebas con resultado positivo se llevó a cabo una prueba ejecutando un proyecto con dependencia de un segundo proyecto, el cual a su vez contenía dependencias de librerías y de un tercer proyecto que contenía dependencias de diversos ficheros *.jar*. Esta última prueba al igual que las anteriores finalizo con resultado positivo y una correcta generación del fichero *.zip*.

6.4.2. Ejecución y depuración de una aplicación en local

Para estas pruebas se utilizó uno de los proyectos descritos anteriormente. En primer lugar se ejecutó la aplicación en local sólo para comprobar que la ejecución era correcta y que el fichero *.zip* se generaba correctamente. En segundo lugar se depuró la aplicación para comprobar que el fichero *.zip* se descomprimía de manera correcta y la plataforma Eclipse era capaz de acceder a los fuentes contenidos en el *.zip*.

6.4.3. Ejecución y depuración de una aplicación en remoto

Para estas pruebas se utilizó de nuevo un proyecto entre los descritos anteriormente. En este caso se comprobó la correcta generación del *.zip*, su envío a

través del socket seguro, la descompresión y ejecución por parte del servidor y la generación del fichero con la salida por consola del proceso. Además, también se comprobó que el fichero era enviado correctamente de vuelta y mostrado por la consola de Eclipse.

6.4.4. Visualización del estado de una ejecución en remoto

Para estas pruebas se ejecutaron varios de los proyectos descritos anteriormente unas cuantas veces. La intención de estas pruebas era observar la ventana *Optsicom View* y comprobar cómo se actualiza el estado de las ejecuciones mientras éstas iban finalizando.

Capítulo 7

Conclusiones

7.1. Logros alcanzados

Se puede deducir que los objetivos citados al comienzo de esta memoria han sido logrados con éxito. Se ha conseguido desarrollar un plugin que permita la ejecución en máquinas remotas de aplicaciones Java versionadas. Además, se ha conseguido que ésto sea una tarea sencilla y transparente para el desarrollador.

La realización de este proyecto ha sido una experiencia muy positiva para mi, ya que me ha enseñado a trabajar de una manera diferente, que no tiene nada en común con la que he llevado a cabo durante mis cinco años de carrera.

Elegí este proyecto porque ya conocía el lenguaje Java y me resultó interesante la idea de desarrollar un plugin para Eclipse puesto que es el entorno de desarrollo que he utilizado durante todos estos años.

Al comienzo del proyecto me encontraba un poco perdido ya que nunca había desarrollado un plugin para Eclipse, pero gracias a la documentación y a la ayuda de mi tutor conseguí entender cómo funcionan los plugins de Eclipse y cómo desarrollar uno correctamente. Al final ha sido un trabajo laborioso pero a la vez entretenido y satisfactorio.

7.2. Trabajos futuros

El plugin desarrollado actualmente permite la ejecución de aplicaciones Java. Una de las futuras lineas de trabajo podría ser la de ampliar la funcionalidad del plugin para que se pudieran ejecutar aplicaciones desarrolladas en otros lenguajes (C, C++, Python, Haskell ...).

Otra posible linea de trabajo consistiría en aumentar la información que ofrece la vista *Optsicom View*. Actualmente muestra información acerca del nombre del proceso, la máquina remota en la que se lanzó y el estado de su ejecución y podría mejorarse añadiendo información acerca de los fuentes con los que se lanzó la ejecución, la ruta en disco de los mismos, el tiempo que tardó la ejecución, los atributos del programa y de la maquina virtual con los que fueron lanzados...

Por último sería interesante investigar la forma de detectar procesos en *Windows*, ya que actualmente alguna de las funcionalidades que posee el servidor en *Linux* no están operativas en *Windows*.

Bibliografía

[1] Dan Rubel Eric Clayberg. *Eclipse Plug-ins.* Addison-Wesley Professional, 2008.

[2] Kathleen McNiff Esmond Pitt. *java(TM).rmi: The Remote Method Invocation Guide.*

[3] The Eclipse Foundation. Eclipse - http://www.eclipse.org/, 2010.

[4] The Eclipse Foundation. Eclipse java development tools (jdt) - http://www.eclipse.org/jdt/, 2010.

[5] The Eclipse Foundation. Plug-in development environment (pde) - http://www.eclipse.org/pde/, 2010.

[6] William Grosso. *Java RMI.* O'Reilly Media, 2001.

[7] Stanford Ng Laurent Mihalkovic Matthew Scarpino, Atephen Holder. *SWT/J-Face in Action: GUI Design with Eclipse 3.0 (In Action series).* Manning Publications, 2004.

[8] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform (2º Edition).* Addison-Wesley Professional, 2010.

[9] Wikilibros. Manual de latex, 2010.