Máster Universitario en Ingeniería de Sistemas y Control

PROYECTO FIN DE MÁSTER

CONCEPTUALIZACIÓN DE MODELOS Y VISTAS DE SIMULACIONES INTERACTIVAS

Autor:

Almudena Ruiz Sáez

Directores:

Francisco Esquembre Martínez
y
Sebastián Dormido Bencomo

Curso 2010/11

Convocatoria Septiembre 2011

Máster Universitario en Ingeniería de Sistemas y Control

PROYECTO FIN DE MÁSTER

CONCEPTUALIZACIÓN DE MODELOS
Y
VISTAS DE SIMULACIONES INTERACTIVAS

Tipo de proyecto B: Proyecto específico propuesto por el alumno

Autor:

Almudena Ruiz Sáez

Directores:

Francisco Esquembre Martínez
y
Sebastián Dormido Bencomo

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: **Almudena Ruiz Sáez**

Firma del alumno

Quisiera aprovechar esta ocasión para dar mi más sincero agradecimiento a todos aquellos profesores que han participado en mi formación académica. En especial, a todos los que me han guiado por el fantástico mundo de las Matemáticas, puesto que sin ellos no habría sido capaz de cumplir uno de mis sueños.
De entre todos ellos, me gustaría destacar al profesor Francisco Esquembre Martínez por su interés y ayuda prestada en estos últimos años, y por mostrarme que las Matemáticas van más allá de la teoría.

En último lugar y no por ello menos importante, quisiera agradecer a mi familia y amigos su apoyo incondicional durante todos estos años.
Y, sobre todo, a mis padres, Pedro y Carmen, porque sin ellos nada de esto habría sido posible.

Gracias a todos.

Almudena Ruiz

Resumen

En este proyecto fin de master titulado *"Conceptualización de modelos y vistas de simulaciones interactivas"* se ha tratado de definir qué es un modelo, qué es una vista y qué relación existe entre ellos dentro de una misma simulación.

El objetivo que se pretende conseguir con esta conceptualización es el diseño de una interfaz de comunicación entre ambas partes con el fin de que puedan ser ejecutadas independientemente, dando origen a las conocidas aplicaciones distribuidas.

En este contexto, nuestras aplicaciones distribuidas harán referencia al uso de objetos por parte de otros que pueden estar situados o no en la misma máquina, de forma casi transparente. También se hablará de objetos *servidores*, y de objetos *cliente*, que los usan, aunque esta distinción carece de sentido en sistemas distribuidos, donde un objeto puede adoptar ambos roles como se podrá observar en dicho trabajo.

Para el desarrollo de estas aplicaciones distribuidas nos hemos apoyado en las arquitecturas RMI y CORBA, siendo esta última la que albergará mayor relevancia.

En el primer capítulo se detallarán las características básicas de estas arquitecturas, las ventajas y desventajas que nos pueden aportar, y se expondrán algunos ejemplos básicos que nos ilustrarán su mecanismo de funcionamiento. En el segundo y tercer capítulo se abordará el concepto de modelo y vista para simulaciones no interactivas e interactivas respectivamente. Finalmente, se expondrán las conclusiones obtenidas y se comentarán algunos de los problemas fundamentales que existen en este tipo de simulaciones, los cuales se desarrollarán en futuros trabajos.

Lista de palabras clave

- Aplicaciones Distribuidas
- Cliente
- CORBA
- IDL
- JAVA
- Modelo
- RMI
- Servidor
- Vista

Índice general

Introducción **1**

1. Sistemas de comunicación de procesos **5**
1.1. RMI 5
1.1.1. Aplicación *Eco* con RMI 6
1.1.2. Aplicación de EJS con RMI 7
1.2. CORBA 7
1.2.1. Aplicación *Hello* con CORBA 9
1.2.2. Aplicación *Echo* con CORBA 10
1.3. ¿RMI o CORBA? 10

2. Modelos con vistas no interactivas **13**

3. Modelos con vistas interactivas **21**
3.1. Interfaz Vista Interactiva 22
3.2. Interfaz Modelo Interactivo 22
3.3. Diseño de una aplicación interactiva 23

Conclusiones **25**

A. El lenguaje de especificación IDL **27**

B. Códigos del Capítulo 1 **31**
B.1. Código de la aplicación 1.1.1 31
B.2. Código de la aplicación 1.1.2 32
B.3. Código de la aplicación 1.2.1 33
B.4. Código de la aplicación 1.2.2 35

C. Códigos del Capítulo 2 **39**
C.1. Implementación de la clase VISTA_impl.java 39
C.2. Implementación de la clase VistaServer.java 43
C.3. Implementación de la clase ModelClient.java 44

D. Códigos del Capítulo 3 **47**
D.1. Implementación de VistaInt_impl.java 47
D.2. Implementación de VistaServerInt.java 52
D.3. Implementación de ModelInt_impl.java 53
D.4. Implementación de ModelClientInt.java 56

Bibliografía **60**

Listado de abreviaturas **61**

Índice de figuras

1.1. Respuesta servidor(izquierda) y cliente (derecha) de la aplicación Eco. . 7
1.2. Respuesta servidor(izquierda) y cliente (derecha) de la aplicación de EJS. 8
1.3. Definición de la interfaz IDL para la aplicación 1.2.1. 9
1.4. Respuesta servidor de la aplicación de Hello con CORBA. 10
1.5. Definición de la interfaz IDL para la aplicación 1.2.2. 11
1.6. Respuesta servidor(izquierda) y cliente (derecha) de la aplicación de Echo. 11

2.1. Respuesta del diseño de la interfaz de comunicación. 19

3.1. Ejecución simulación interactiva: Péndulo. 24

Listings

B.1. Clase EcoRMI.java 31
B.2. Clase EcoServer.java 31
B.3. Clase EcoClient.java 32
B.4. Clase ViewServer 32
B.5. Clase ClientModel 33
B.6. Clase Hello_impl.java 33
B.7. Clase HelloServer.java 33
B.8. Clase HelloClient.java 34
B.9. Clase Echo_impl.java 35
B.10. Clase EchoServer.java 35
B.11. Clase EchoClient.java 36
C.1. Clase VISTA_impl.java 39
C.2. Clase VistaServer.java 43
C.3. Clase ModelClient.java 44
D.1. Clase VISTAInt_impl.java 47
D.2. Clase VISTAServerInt.java 52
D.3. Clase ModelInt_impl.java 53
D.4. Clase ModelClientInt.java 56

Introducción

Desde los principios de la historia, el ser humano ha intentado conseguir un mecanismo perfecto, por el cual sea capaz de simular la realidad que le rodea. En estas condiciones ha pasado de unas teorías empíricas a unos formalismos teóricos, con lo que se pretende interpretar la realidad sin necesidad de tener unas exhaustivas medidas experimentales. Partiendo de esta base, y teniendo en cuenta que la mayoría de las simulaciones tienden a representar el mundo real, es interesante detenerse, por un instante, en la problemática que surge a raíz de dicho tema de estudio.

El avance que se ha producido en el diseño, desarrollo y construcción de máquinas que ayudan al hombre a simular la realidad ha pasado por distintas etapas. Desde los primeros mecanismos que intentaban imitar el comportamiento de los animales, hemos llegado hasta los actuales ordenadores capaces de simular sistemas complejos. En cada paso del camino recorrido, se han ido aprendiendo, mejorando y perfeccionando las técnicas que nos permiten representar la realidad, llegando a aplicarse en la actualidad complejos desarrollos matemáticos para construir modelos que representan al mundo que nos rodea, o al menos a una parte de él.

Todo ello ha sido posible con la ayuda de los computadores, y por supuesto, debido al desarrollo que se ha alcanzado en su diseño y construcción, haciéndolos más rápidos y con más potencia de cálculo por unidad de tiempo. Pero aun así, hay problemas que requieren de más rapidez o más potencia; ¿qué hacer cuando el objetivo es solucionar una cuestión de este tipo?. No nos queda más remedio que recurrir al trabajo en equipo, o lo que es igual a un sistema de computadores trabajando de forma colaborativa sobre un conjunto de datos. Es esto precisamente a lo que denominamos Arquitecturas Distribuidas. Estos sistemas cuentan además de con los computadores, con un soporte de transmisión e intercambio de información, que generalmente consiste en una red de comunicaciones de alta velocidad, cuyo objetivo es conseguir el paso de información de la manera más ágil, rápida y fiable entre

los distintos elementos del proceso.

Como es natural, los avances producidos en el diseño de computadores y en las tecnologías y protocolos de comunicaciones, han conseguido que sistemas tan complejos como la interacción molecular a nivel de átomos, el movimiento de sistemas autónomos por la superficie de otros planetas, o el comportamiento en tiempo real de una planta industrial al completo, sean representables y simulables mediante Arquitecturas Distribuidas.

El uso de este tipo de arquitecturas proporciona una nueva visión de dónde y cómo debe ser tratada la información que compone la base de toda simulación, y se puede utilizar no sólo para la simulación propiamente dicha, sino también para mostrar los resultados de dicho proceso de una forma más agradable para el usuario.

Obviamente, la simulación en tiempo real es una cuestión ya conocida, ampliamente estudiada y en la que el problema fundamental no es la complejidad de los algoritmos que se utilicen para representar el modelo, sino la potencia del computador sobre el que se ejecuta la simulación, que condiciona aspectos fundamentales como son la visualización de resultados en tiempo real, la posibilidad de modificar parámetros de ejecución,. . . . Para solucionar estos problemas se han venido utilizando métodos tradicionales como son los optimizadores de código, las librerías y aceleradores de gráficos, y por supuesto la mejora de las prestaciones de los equipos a utilizar.

El problema es que se puede llegar al límite actual en estos campos y aún así no haber conseguido los objetivos de la simulación.

Todo este planteamiento conlleva una serie de problemas de diseño a la hora de realizar la aplicación que ejecutará la simulación. Aunque existen herramientas para trabajar sobre Sistemas basados en Arquitecturas Distribuidas, son de propósito general, y por lo tanto no suelen adaptarse a las necesidades requeridas, al igual que para cada modelo habrá que realizar una aplicación completa que refleje sus características particulares [14].

Por lo tanto, dada una simulación, el objetivo de este proyecto es conceptualizar qué es su modelo y qué es su vista. Y, sobre todo, cuál es la función que desempeñan cada una de estas partes dentro de la simulación.

Llegados a este punto, utilizaremos dichos conceptos en el diseño de una interfaz de comunicación. De esta manera, el modelo y la vista pasarán a ser totalmente independientes una de la otra, podrán definirse en máquinas distintas y podrán comunicarse haciendo uso de dicha interfaz, dando lugar

entonces a una simulación completa.

La interfaz de comunicación que se irá desarrollando en el transcurso de este proyecto, estará apoyada en la arquitectura CORBA. Esta arquitectura permite la realización de aplicaciones distribuidas y será descrita en el capítulo 1.

Una vez conceptualizado el modelo y la vista de una simulación y desarrollada su interfaz de comunicación se aplicará todo ello en el diseño de algunas simulaciones básicas que nos permitirán comprobar si hemos cumplido los objetivos fijados al inicio de este proyecto.

En trabajos futuros nos centraremos en abordar los siguientes aspectos:

- Extender los resultados obtenidos a otros lenguajes de programación como C,C++ para comprobar la eficacia de la interfaz definida entre distintos lenguajes de programación.
- Estudiar algunos de los problemas fundamentales que surgen en el desarrollo de aplicaciones distribuidas como son básicamente el problema de sincronización y el paralelismo.

El proyecto se estructura de la siguiente manera:

- En un primer capítulo se tratará con detalle los sistemas de comunicación de procesos. En concreto nos centraremos en las arquitecturas RMI y CORBA dando una descripción detallada de cada una de ellas y exponiendo algunos ejemplos básicos que nos permitirán comprender con más exactitud su funcionamiento.
- En el segundo capítulo se presentará con detalle el concepto de un modelo y de una vista no interactiva. Se diseñará su interfaz de comunicación y se aplicará en el diseño de algunas simulaciones ilustrativas.
- En el tercer capítulo basándonos en el capítulo anterior se dará el concepto de un modelo y una vista interactiva. Se reutilizará la interfaz definida anteriormente adaptándola a la nueva situación y de forma similar se aplicará en el diseño de alguna simulación interactiva.
- Finalmente, se detallarán las conclusiones obtenidas en el desarrollo de dicho proyecto y se expondrán algunos de los problemas fundamentales

que surgen en el desarrollo de simulaciones distribuidas y que nos darán lugar a futuras líneas de trabajo.

Capítulo 1

Sistemas de comunicación de procesos

En los últimos años se está viviendo un auge en el diseño y desarrollo de Sistemas Distribuidos. Más allá de las cuestiones clásicas de diseño de los mismos (acceso y localización transparentes, estabilidad, tolerancia a fallos, rendimientos, etc.) se plantea qué tecnología utilizar para su desarrollo. Se presenta aquí una introducción a los sistemas distribuidos en Java, incluyendo el estándar CORBA del OMG y el sistema propio de Java, RMI, con un componente crítico que permita discernir su aplicabilidad y adecuación al desarrollo de aplicaciones distribuidas.

El desarrollo de este capítulo está basado en las siguientes referencias bibliográficas:
[16],[11],[6],[13],[20],[18], [15],[8],[17],[5],[4],[3],[19].

1.1. RMI

RMI (*Remote Method Invocation*) fue el primer framework para crear sistemas distribuidos que apareció para Java. Viene integrado en cualquier máquina virtual Java posterior a la 1.0 y está pensado para hacer fácil la creación de sistemas distribuidos a partir de una aplicación cuyas clases ya estén implementadas.

A diferencia de otras tecnologías de sistemas distribuidos, RMI no busca la colaboración de objetos de distintas plataformas, programados en diferentes lenguajes, Java se encarga de solucionar los problemas de heterogeneidad. Así, su API es más sencillo y natural al no contemplar que tipos de datos (y

sus tamaños) existan o no en los lenguajes en los que se implementan cliente y servidor.

La gestión de la concurrencia en RMI es extraordinariamente sencilla e inflexible. Para cada cliente que trate de acceder a un objeto remoto, el servidor creará un nuevo hilo que se encargará de darle servicio. Si varios hilos del mismo cliente realizan distintas peticiones al servidor, compartirán un mismo hilo en el servicio.

RMI soporta que objetos clientes puedan emplear objetos remotos por valor y por referencia. El uso de un objeto remoto por referencia no es nada nuevo, la novedad es el paso de objetos por valor, para lo que se emplea la librería de serialización del API de Java. La ventaja del paso por valor en un sistema distribuido radica en que se minimiza el tráfico en la red, ya que, una vez recibido el objeto, todas las comunicaciones con él serán locales.

Por defecto, RMI no incluye ninguna facilidad para restringir el uso de las clases desde clientes no autorizados, aunque el usuario siempre puede suplir parte de esta funcionalidad usando los servicios del sistema operativo, o haciendo ese trabajo él mismo.

RMI ha sido implementado en tres capas:

- Un programa *stub* del lado del cliente y un correspondiente *skeleton* del lado del servidor. El *stub* aparenta ser el programa llamado para brindar el servicio, ante el programa que solicita el servicio.
- Una capa de referencia remota que puede comportarse de diferentes maneras dependiendo de los parámetros pasados por el programa que hace la llamada. Por ejemplo, esta capa puede determinar si la solicitud invoca un solo servicio remoto o a múltiples programas remotos así como a un multicast.
- Una capa de conexión de Transporte, que establece y administra la solicitud.

A continuación se presentan dos simulaciones distribuidas utilizando la arquitectura que acabamos de describir.

1.1.1. Aplicación *Eco* con RMI

En este primer ejemplo se pretente mostrar cómo se utiliza RMI para diseñar una aplicación distribuida básica como puede ser una aplicación Eco,

es decir, el cliente mandará un mensaje al servidor y este lo mostrará por pantalla. El código correspondiente a esta simulación puede verse en la sección B.1 del Apéndice B.

Para comprobar dicha simulación ejecutaremos en primer lugar el código del servidor y a continuación el del cliente. Entonces, el cliente esperará a que escribamos un mensajes que automáticamente mandará al servidor. Los resultados de esta primera simulación se muestran en la Figura 1.1.

```
EcoRMI RmiIsStarted at port 2009
EcoServer Islistening at 2009
Recibido: Hola
Recibido: ¿Cómo estás?
```

```
Eco>Hola
Hola
Eco>¿Cómo estás?
¿Cómo estás?
Eco>
```

Figura 1.1: Respuesta servidor(izquierda) y cliente (derecha) de la aplicación Eco.

1.1.2. Aplicación de EJS con RMI

Para la realización de esta segunda aplicación nos hemos apoyado en la herramienta de software EJS (*Easy Java Simulations*) diseñada por el profesor Francisco Esquembre de la Universidad de Murcia. En [10] se detalla cómo construir una simulación utilizando el software EJS. En la sección B.2 del Apéndice B encontrará parte del código empleado en el desarrollo de esta aplicación.

El cliente de esta aplicación resuelve un sistema de ecuaciones diferenciales ordinarias mediante el método de Runge Kutta y va enviando los datos obtenidos al servidor. El servidor los representa gráficamente y los imprime por pantalla. Los resultados de esta simulación se muestran en la Figura 1.2.

1.2. CORBA

CORBA, *Common Object Request Broker Architecture* es una arquitéctura estándar para crear aplicaciones distribuidas, creada por un conjunto de más de 700 compañías y organizaciones, agrupados bajo el nombre OMG, *Object Management Group*

El estándar CORBA define qué ha de incluir una implementación estándar, pero no cómo se han de hacer. Esta tarea se deja de la mano de los diferentes usuarios. Esta es una de las principales características de CORBA:

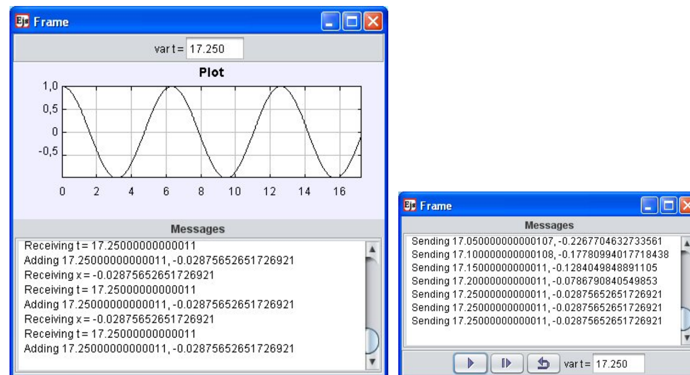


Figura 1.2: Respuesta servidor(izquierda) y cliente (derecha) de la aplicación de EJS.

permite una total libertad a los implementadores siempre que estos respeten unos mínimos orientados a la *interoperabilidad* entre implementaciones.

CORBA ofrece algunas de las ventajas que se enumeran a continuación:

1. Disponibilidad y Versatilidad: Muchas arquitecturas y sistemas operativos cuentan con una implementación de CORBA.

2. Eficiencia: La libertad de desarrollo ha favorecido la existencia de diferentes implementaciones del estándar que se adaptan a multitud de posibles necesidades de los usuarios.

3. Adaptación a Lenguajes de Programación: Es posible emplear los servicios de CORBA desde cualquier lenguaje de programación.

4. CORBA es neutral respecto al protocolo de red utilizado para comunicar cliente y servidor. Para ello especifica el GIOP *General Inter ORB Protocol* que define a muy alto nivel la comunicación entre ORBs diferentes.

5. Participación en múltiples interacciones simultáneamente: El papel del cliente y servidor puede intercambiarse temporalmente.

Pero, también tenemos algunos inconvenientes al usar esta arquitectura:

1. Complejidad: Permitir la interoperabilidad de distintos lenguajes, arquitecturas y sistemas operativos hace que sea un estándar bastante complejo, y su uso no sea tan transparente al programador como sería deseable:

- Hay que usar un compilador que traduce una serie de tipos de datos estándares a los tipos del lenguaje en el que se vaya a programar (IDL) *véase* Apéndice A.
- A la hora de diseñar hay que decidir qué objetos van a ser remotos y cuáles no.
- Los objetos remotos se pueden usar por referencia, pero no por valor.
- Es necesario emplear tipos de datos que no son los que proporcionan de manera habitual el lenguaje de programación.

2. Incompatibilidad entre implementaciones.

Una parte fundamental de la arquitectura CORBA es el ORB (*Object Request Broker*). El ORB es el servicio distribuido que implementa la petición al objeto remoto. Localiza el objeto remoto en la red, le comunica la petición, espera a los resultados y cuándo están disponibles se los devuelve al cliente. El ORB implementa transparencia de localización e independencia del lenguaje de programanción para las peticiones.

En los siguientes apartados se muestran algunas aplicaciones basadas en CORBA.

1.2.1. Aplicación *Hello* con CORBA

El ejemplo que se describe en esta sección es una versión especial de una simulación cliente-servidor. A continuación se van a describir los pasos a seguir para escribir esta aplicación basada en CORBA:

1. Crear la interfaz IDL para nuestra aplicación como se muestra en la Figura 1.3 y guardarlo con el nombre Hello.idl.

```
interface Hello{
        void say_hello();

};
```

Figura 1.3: Definición de la interfaz IDL para la aplicación 1.2.1.

2. Compilar el archivo Hello.idl con el compilador IDL de Java usando el siguiente comando:

jidl Hello.idl

Este comando creará los archivos *Hello.java, HelloHelper.java, HelloHolder.java, HelloOperations.java, HelloPOA.java* y *_HelloStub.java*

3. Implementación del servidor: En primer lugar tendremos que crear la clase *Hello_impl.java* como se muestra en la Lista B.6. Una vez creada esta clase, implementamos el código de nuestro servidor que se muestra en la Lista B.7. Ambas Listas se encuentran en la sección B.3.

4. Implementación del cliente: Dicha implementación puede verse en la Lista B.8 de la sección misma sección nombrada anteriormente.

5. Ejecutamos la simulación: Nuestra aplicación se divide en el programa servidor y el programa cliente. En primer lugar, ejecutamos el servidor ya que debe crear el archivo *Hello.ref* que el cliente necesita para poder conectarse al servidor. Una vez que el servidor esté listo, arrancamos el cliente. Si todo está bien hecho, el mensaje “Hello” aparecerá en la pantalla del servidor como se muestra en la Figura 1.4.

```
Servidor listo:
¡¡¡Hello!!!
```

Figura 1.4: Respuesta servidor de la aplicación de Hello con CORBA.

1.2.2. Aplicación *Echo* con CORBA

Siguiendo los pasos descritos en la sección 1.2.1 vamos a rediseñar la aplicación Eco vista en la sección 1.1.1 pero esta vez utilizando la arquitectura CORBA y la llamaremos *Echo*.

La interfaz IDL de esta aplicación se muestra en la Figura 1.5 y el código para generar el servidor y el cliente se puede encontrar en la sección B.4 del Apéndice B.

La solución que obtenemos al ejecutar esta aplicación será similar a la mostrada en la Figura 1.6

1.3. ¿RMI o CORBA?

Si tanto el cliente como el servidor van a usar Java, se quiere una solución sencilla para crear un sistema distribuido y valen las opciones por defecto de

```
module corba {
        interface Echo {
                string echo_(in string x);
        };
};
```

Figura 1.5: Definición de la interfaz IDL para la aplicación 1.2.2.

```
Servidor listo y en espera
La cadena es <Hola>
La cadena es <¿Cómo estás?>
```

```
Escribe un mensaje que desees enviar al servidor:
Hola
¿Cómo estás?
```

Figura 1.6: Respuesta servidor(izquierda) y cliente (derecha) de la aplicación de Echo.

los diseñadores de RMI, esta es la mejor elección. RMI es sin duda la opción más integrada en Java, la de uso más natural para un programador de Java y que además aporta un buen número de utilidades. Sin embargo, su sencillez se puede transformar en inflexibilidad si se quieren hacer cosas que los creadores de RMI no contemplan.

Por otro lado, si se quiere garantizar una casi absoluta capacidad de elección para cualquier parámetro del sistema distribuido o si no vale Java para ambos extremos de la comunicación, la solución es CORBA.

En definitiva, CORBA es casi lo opuesto a RMI. Donde este aporta sencillez, CORBA aporta un cierto nivel de complejidad. El premio a este esfuerzo es un sistema flexible y que se adapta a casi cualquier entorno de computación existente. Por este motivo, a partir de este momento se utilizará la arquitectura CORBA como la base del dicho proyecto.

Capítulo 2

Modelos con vistas no interactivas

En este capítulo se va a definir el concepto de modelo y vista dentro de una simulación no interactiva. Dadas estas conceptualizaciones se describirá con detalle la interfaz de comunicación diseñada entre ellas.

Con esta interfaz de comunicación se pretende que el modelo y la vista sean partes totalmente diferenciadas e independientes y que se puedan relacionar en un momento dado para formar una simulación completa. Ambas partes podrán ejecutarse en una misma máquina o en máquinas distintas, comunicándose entonces a través de la red.

Un modelo es una representación simplificada de la realidad que contiene la esencia del sistema, con el detalle suficiente como para que pueda utilizarse en la investigación y la experimentación en lugar del sistema real, con menos riesgo, tiempo y coste.

Un modelo está constituido por:

- Un conjunto de variables que representan las magnitudes físicas del sistema que se está modelando.
- Un conjunto de relaciones entre las variables del modelo que describen el comportamiento de las variables ante una cierta clase de situaciones.

Por lo tanto, el modelo es la parte de la simulación que se encarga de generar los datos del sistema real que estemos analizando. Dentro de la interfaz de comunicación diseñada, el modelo constituirá la parte *cliente*.

La vista es la parte de la simulación que se encarga de representar los datos del sistema real hallados por el modelo. Sin lugar a dudas, la creación de la interfaz gráfica, o vista, es la parte de la simulación que precisa de

mayor conocimiento de técnicas avanzadas de programación. Por lo general, las herramientas gráficas proporcionadas por un lenguaje de programación de alto nivel son de alta aplicabilidad y, por ello, poco especializadas en alguna tarea en concreto. Esto causa que se precise de un considerable esfuerzo técnico para adecuar dichas herramientas a un uso tan específico como el que deseemos hacer nosotros.

Por este motivo, en la definición de nuestra interfaz se ha tratado de simplificar dicha cuestión pudiendo obtener la visualización de nuestros datos de la manera más sencilla posible.

Nuestra vista desarrollará el papel del *servidor* dentro de nuestra interfaz de comunicación.

A continuación se va a pasar a describir en detalle la interfaz de comunicación entre la vista (servidor) y el modelo (cliente).

Esta interfaz está basada en la arquitectura CORBA descrita en la sección 1.2. Como vimos allí, la base de toda aplicación CORBA es la definición de la interfaz IDL. Esta interfaz contiene todos los métodos que el servidor ofrece al cliente. En nuestro caso, serán todos los métodos que conforman nuestra vista y que estarán disponibles para que el modelo pueda usarlos.

Nuestra interfaz IDL contendrá la siguiente lista de métodos:

- **void sendDouble(in string name, in double value)**:Método para enviar un double desde el cliente al servidor.
- **void sendInt(in string name, in long value)**:Método para enviar un valor entero desde el cliente al servidor.
- **void sendString(in string name, in string value)**:Método para enviar un String desde el cliente al servidor.
- **void sendBoolean(in string name, in boolean value)**:Método para enviar un valor booleano desde el cliente al servidor.
- **void sendStringArray(in string name, in StringArray aString)**:Método para enviar un array de String desde el cliente al servidor.
- **void sendDoubleVector(in string name, in DoubleVector value)**:Método para enviar un array de doubles desde el cliente al servidor.

- **void sendIntVector(in string name, in IntVector)**:Método para enviar un array de enteros desde el cliente al servidor.
- **void sendBooleanArray(in string name, in BooleanArray aBoolean)**:Método para enviar un array de booleanos desde el cliente al servidor.
- **void sendDoubleMatriz(in string name, in DoubleMatriz value)**:Método para enviar una matriz de doubles desde el cliente al servidor.
- **void sendIntMatriz(in string name, in IntMatriz value)**:Método para enviar una matriz de enteros desde el cliente al servidor.
- **void reset()**: Inicializa todas las variables de la vista al valor establecido en las tablas de variables y limpia la vista. Esto devuelve a la simulación completamente a su estado inicial.
- **void initialize()**: Este método puede considerarse como una inicialización blanda. No devuelve a las variables el valor que tenían en sus tablas de variables. Esto puede resultar útil para permitir inicializar la simulación respetando los valores de algunas variables que el usuario haya establecido mediante la interfaz de la simulación.
- **void update()**: Método para actualizar los posibles cambios que hayan sufrido las variables de la vista.
- **void clearMessages()**: Este método limpia el área de texto de la vista. Si ésta no incluye un elemento de este tipo, el método no tiene ningún efecto.
- **void clearView()**: Este método limpia todos los elementos de la vista.
- **void print(in string txt)**: Este método imprime el texto indicado en la vista de la simulación siempre que esta contenga algún elemento destinado para ello. En caso contrario, el mensaje aparecerá en la consola del sistema desde el que se lanzó el programa.
- **void println(in string txt)**: Este método imprime el texto indicado seguido de un retorno de carro, lo que significa que el siguiente mensaje que se imprima aparecerá en una nueva línea. El texto aparecerá como en el método print().
- **void newLine()**: Este método imprime un línea en blanco seguida de un retorno de carro.

- **void getListSpecifications()**: Este método imprime una lista con los métodos específicos que tiene disponibles.
- **void actionsNoStandar(in string name,in string value)**: Este método sirve para invocar a uno de los métodos específicos disponibles.
- **void alert(in string title, in string value)**: Este método resulta útil para mostrar mensajes de error o avisos que el usuario no pueda ignorar.
- **void switchPanel(in long numberPanel)**: Este método nos permite seleccionar un panel en el caso de que hayan varios disponibles a la vez.
- **void switchColor(in long color)**: Este método nos permite seleccionar el color para dibujar una curva o escribir un texto.
- **void dibujarCurva(in string name, in boolean ejes, in DoubleMatriz value)**: Este método nos abre un panel de dibujo con título *name*, en el que se pueden dibujar los ejes de coordenadas o no dependiendo del valor del booleano *ejes* y donde se dibujarán los puntos dados en la matriz *value*.

En la definición de algunos de estos métodos ha sido necesario definir nuevos tipos de datos con la función *typedef*. Los datos predefinidos han sido los siguientes:

- Arrays de doubles, de String, de int y de boolean: double[], String[], int[], boolean[].
- Dobles arrays de int y doubles: int[][] y double[][].

El motivo de hacer estas definiciones ha sido que en IDL es imposible utilizar arrays sin prefijar previamente sus dimensiones. Por ello, se ha utilizado el tipo *sequence* en lugar de *array*. Para saber más sobre las características del lenguaje IDL véase Apéndice A.

Una vez diseñada la interfaz *VISTA.idl* se compila el archivo con el compilador de Java para IDL, *jidl*, con el siguiente comando:

jidl VISTA.idl

Este comando nos creará automáticamente una carpeta llamada *ArchivosVISTA* que contendrá los siguientes archivos:

- VISTAPOA.java

- VISTAOperations.java
- VISTAHolder.java
- VISTAHelper.java
- VISTA.java
- _VISTAStub.java

El siguiente paso será la implementación de nuestro servidor, es decir, de nuestra clase Vista. Dicha implementación se divide en dos partes:

1. Implementación de la clase *Vista_impl.java*: En ella se implementarán todos los métodos definidos en la interfaz IDL. Dicho código puede consultarse en la sección C.1 del Apéndice C.

2. Implementación de la clase servidor, *VistaServer.java*: Esta implementación sigue los siguientes pasos:

   - Inicia el ORB.
   - Obtiene una referencia al objeto RootPOA.
   - Activa el POAManager.
   - Instancia el objeto server.
   - Pasa una referencia al ORB.
   - Obtiene una referencia para el server.
   - Obtiene el contexto de nombres iniciales.
   - Registra el server en el servicio de nombres.
   - Espera invocaciones en un bucle.

   El código puede consultarse en la sección C.2 del Apéndice C.

Seguidamente diseñaremos nuestra clase cliente, *ModelClient.java*. Dicha implementación sigue los siguientes pasos:

- Iniciar el ORB.
- Obtener el contexto de nombres inicial.
- Obtener una referencia al server.
- Escribir la parte correspondiente al modelo de la simulación que estemos diseñando invocando las operaciones del servidor que necesitemos.

El código correspondiente puede consultarse en la sección C.3 del Apéndice C.

Finalmente, ya estamos en condiciones para poder ejecutar la aplicación que hemos diseñado a lo largo de este capítulo. Para ello, seguiremos los siguientes pasos:

1. Ejecutamos la clase *VistaServer.java*.
2. Ejecutamos la clase *ModelClient.java*.

Si queremos ejecutar estas clases desde máquinas diferentes debemos hacer lo siguiente:

1. Desde la ventana de MD-DOS (en Widnows) arrancamos el servidor de nombres de Java IDL:

   tnameserv -ORBInitialPort nameserverport .

   Observa que *nameserverport* es el puerto en el que queremos que funcione el servidor.

2. Desde una segunda ventana ejecutamos el servidor:

   java nameServer -ORBInitialHost nameserverhost -ORBInitialPort nameserverport

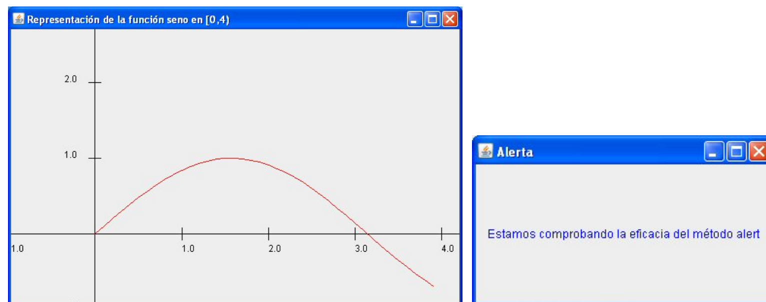
   Observa que *nameserverhost* es el nombre del host en el que se está ejecutando el servidor de nombres IDL. Podemos omitir *-ORBInitialHost nameserverhost* si el servidor de nombres se está ejecutando en el mismo host que el servidor.

3. Desde otra ventana, ejecutamos la aplicación cliente:

   java nameClient -ORBInitialHost nameserverhost -ORBInitialPort nameserverport

Los resultados obtenidos de nuestra aplicación se muestran en la Figura 2.1. Como podemos observar se han probado distintos métodos de la interfaz diseñada: sendDouble, sendString, getListSpecificActions, update, newLine, alert, println, newLine y dibujarCurva.

Para el desarrollo de este capítulo se ha tenido en cuenta las conclusiones obtenidas en el Capítulo 1 y las referencias bibliográficas [12], [7], [9].




```
Servidor Vista listo
Asignar a <x> el valor de 25.0
Error: La variable nombre no existe.
Los métodos específicos disponibles son:
alert
switchPanel
switchColor
Actualización de las variables
x = 25.0
y = 0.0
t = 0.0

Representación de la función seno en el intervalo [0,4)
Los puntos calculados son los siguientes:

Asignar a <x> el valor de 0.0
Asignar a <y> el valor de 0.0
Asignar a <t> el valor de 0.1
Actualización de las variables
x = 0.0
y = 0.0
t = 0.1
...
```

Figura 2.1: Respuesta del diseño de la interfaz de comunicación.

Capítulo 3

Modelos con vistas interactivas

En este capítulo se van a extender los resultados obtenidos en el capítulo 2 pero ahora nos centraremos en el diseño de simulaciones interactivas.

Una simulación interactiva puede definirse como aquella que ofrece la posibilidad de la creación gráfica de modelos de simulación, permite mostrar por pantalla dinámicamente el sistema simulado, así como la interacción entre el usuario y el programa en ejecución. La interacción implica que o bien se detiene la simulación y solicita información al usuario, o bien que esta puede parar la simulación a su voluntad e interaccionar con el mencionado programa.

En las simulaciones no interactivas la vista tenía el papel de servidor y el modelo el del cliente. En este capítulo, ambos papeles irán intercambiándose durante la ejecución de la simulación, es decir, la vista y el modelo serán servidor y cliente en una misma aplicación.

A continuación se va a pasar a describir en detalle las interfaces de comunicación entre la vista (servidor) y el modelo (cliente), y entre el modelo (servidor) y la vista (cliente).

Estas interfaces están basadas en la arquitectura CORBA descrita en la sección 1.2. Como vimos allí, la base de toda aplicación CORBA es la definición de la interfaz IDL. Esta interfaz contiene todos los métodos que el servidor ofrece al cliente. En nuestro caso, serán todos los métodos que la vista ofrecerá al modelo y todos los que el modelo ofrecerá a la vista.

3.1. Interfaz Vista Interactiva

Esta interfaz contiene todos los métodos de la interfaz Vista para simulaciones no interactivas descritos en el capítulo 2 página 14, más dos métodos propios, específicos para las simulaciones interactivas. Estos métodos nuevos se describen a continuación:

- **void dibujarPunto(in string color, in boolean ejes, in double x, in double y)**: Método para dibujar en la interfaz gráfica un punto dado pudiendo además especificar su color y si queremos que se dibujen o no los ejes de coordenadas.
- **void setModel(in String refFile)**: Método que asigna al modelo el papel de servidor y a la vista el de cliente.

Una vez diseñada la interfaz *VISTA_INT.idl* se compila el archivo con el compilador de Java para IDL, *jidl*, con el siguiente comando:

```
jidl VISTA_INT.idl
```

Este comando nos creará automáticamente los archivos necesarios para diseñar una aplicación CORBA.

3.2. Interfaz Modelo Interactivo

La interfaz IDL de un modelo interactivo contiene todos los métodos que dicho modelo puede ofrecer como servidor. Dichos métodos se describen a continuación:

- **void accion(in string name)**: Método para invocar a la acción seleccionada desde la vista de la simulación.
- **void play()**: Método que pone en marcha la simulación.
- **void pause()**: Método que detiene la evolución de la simulación hasta que se vuelva a seleccionar el método play().
- **void stop()**: Método que detiene la evolución de la simulación.
- **void step(in double value, in double longPaso)**: Método que da un paso en la ejecución de la simulación.
- **void reset()**: Método que inicializa todas las variables del modelo al valor establecido en las tablas de variables, hace un reset en la vista y actualiza dichos cambios.

- **void initialize()**: Inicializa el modelo, la vista y actualiza dichos cambios.
- **void update()**: Método para actualizar los posibles cambios que hayan sufrido las variables del modelo.
- **void setVista()**: Método que asigna a la vista el papel de servidor y al modelo el de cliente.

Una vez definida se compila de manera análoga al caso de la Interfaz Vista_INT.idl pero en este caso con la interfaz Modelo_INT.idl.

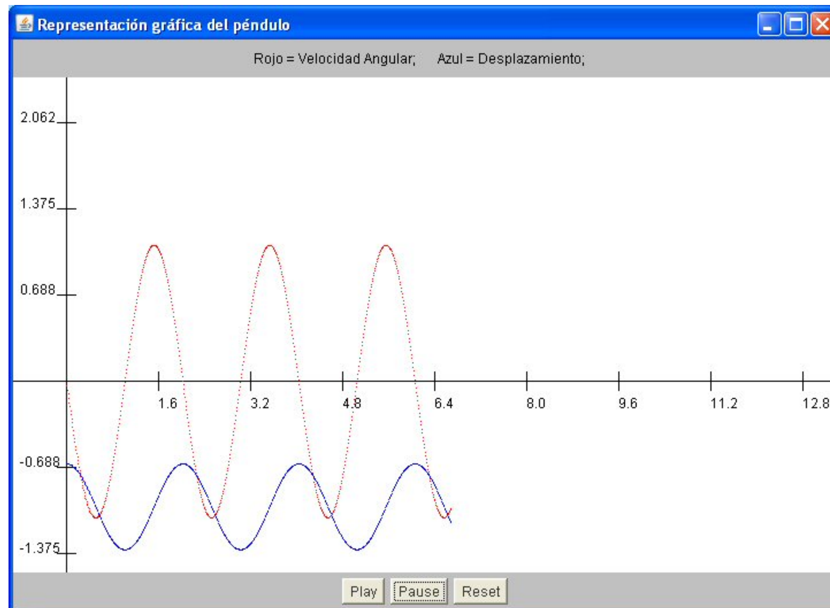
3.3. Diseño de una aplicación interactiva

Tras definir las interfaces IDL en las secciones anteriores ya estamos en condiciones de aplicarlas en el diseño de una simulación interactiva.

La simulación elegida estudia el movimiento de un péndulo simple. Esta consiste en una masa puntual que oscila suspendida por un hilo de masa despreciable y longitud fija, bajo la acción de la gravedad.
Dicho modelo se presenta como un sistema de ecuaciones diferenciales que se resolverá haciendo uso del método de resolución de Runge-Kutta.

Las clases que conforman esta simulación están detalladas en el Apéndice D. Como podremos observar en ellas, el diseño de una simulación interactiva tiene mayor complejidad puesto que tenemos que trabajar con un servidor y un cliente que pueden ir intercambiando sus papeles conforme se desarrolla la simulación. El resto del proceso es análogo al caso no interactivo descrito en el capítulo 2.

Los resultados obtenidos al ejecutar dicha aplicación se muestran en la Figura 3.1.




```
Servidor Vista Interactiva listo
Actualización de las variables
theta = 0.3488981024061307, omega = −0.03354702099450415, t = 0.0
Actualización de las variables
theta = 0.3483950130659937, omega = −0.06706311660043274, t = 0.01
Actualización de las variables
theta = 0.34755704619476235, omega = −0.10051737863988877, t = 0.02
Actualización de las variables
theta = 0.34638497453245515, omega = −0.1338789334437834, t = 0.03
Actualización de las variables
theta = 0.34487987931052516, omega = −0.16711695932599943, t = 0.04
Actualización de las variables
theta = 0.34304314964360577, omega = −0.2002007043188212, t = 0.05
Actualización de las variables
theta = 0.3408764817422487, omega = −0.23309950425265843, t = 0.060000000000000005
Actualización de las variables
theta = 0.33838187794281, omega = −0.26578280125964115, t = 0.07
...
```

Figura 3.1: Ejecución simulación interactiva: Péndulo.

Conclusiones

Se concluye del trabajo realizado los siguientes aspectos:

- La conceptualización básica de un modelo y una vista dentro de una simulación.
- La simplicidad de las interfaces de comunicación diseñadas para el desarrollo de simulaciones interactivas y no interactivas basadas en las conceptualizaciones anteriores.

Con los aspectos mencionados anteriormente se ha conseguido que el modelo y la vista sean dos partes totalmente independientes de una misma simulación y que se puedan comunicar incluso desde máquinas distintas, haciendo uso de las interfaces de comunicación diseñadas y dando lugar entonces a una simulación totalmente completa.

El lenguaje de programación utilizado ha sido Java y las arquitecturas de comunicación han sido RMI y CORBA.Estas nos han permitido diseñar cada componente del sistema por separado y comunicarlas de manera independiente dando origen a una aplicación distribuida.

Finalmente, podemos observar que el rendimiento de las interfaces descritas en este proyecto se puede mejorar abordando los problemas de sincronización y paralelismo que pueden surgir en el desarrollo de simulaciones de mayor complejidad.

Evidentemente, el tratamiento de estos problemas fundamentales es técnicamente elaborado y será objeto de estudio en futuras líneas de trabajo. En la cita bibliográfica [2] se puede encontrar información relativa a estos problemas.

Apéndice A

El lenguaje de especificación IDL

El lenguaje de definición de interfaz o IDL (*Interface Definition Language*), es un lenguaje de programación pensado exclusivamente para especificar las interfaces de las clases cuyas instancias queremos hacer públicas a objetos remotos que las usarán como clientes. Es importante destacar que IDL sólo puede definir interfaces, no implementaciones, por lo tanto es un lenguaje puramente declarativo. No hay forma de escribir sentencias ejecutables en IDL y no hay forma de decir nada sobre el estado del objeto.

La necesidad de un IDL viene dada por la independencia de CORBA respecto a la arquitectura y al lenguaje de programación. Distintos lenguajes soportan diferentes tipos de datos y tienen distintas formas de especificar clases. IDL pone de acuerdo a distintos lenguajes en el formato y tamaño de sus especificaciones. El compilador de IDL transforma una especificación neutral para la plataforma y el lenguaje en otra que puedan entender dicho lenguaje y plataforma.

Describiendo las interfaces IDL, un ORB genera automáticamente código en el lenguaje seleccionado para realizar la integración de las aplicaciones distribuidas. Evidentemente, puesto que sólo describe interfaces, todas las tareas complejas relacionadas con los lenguajes de programación, como control de flujo, gestión de memoria, composición funcional, no aparecerán en IDL.

En la siguiente tabla se representan los tipos predefinidos de IDL y su correspondencia con Java:

IDL	**Java**
boolean	boolean
char/wchar	char
octet	byte
short	short
long	int
long long	long
float	float
double	double
string/wstring	String

Además de proporcionar los tipos básicos predefinidos, IDL permite definir tipos compuestos: *enumerados, estructuras, uniones, secuencias, y arrays*. Se puede usar la palabra reservada *typedef* para dar nombre a un tipo de forma implícita.

Las secuencias son arrays de longitud variable. Pueden contener cualquier tipo de elementos y pueden ser acotadas o no acotadas. IDL no tiene punteros pero permite definir estructuras recursivas. Se pueden definir *constantes* de cualquier tipo predefinido (excepto any) como se muestra a continuación:

const float PI = 3.1415926;

Una interfaz IDL presenta las siguientes características:

- Los objetos remotos se pasan por referencia.
- Pueden devolver excepciones.
- Emplea atributos direccionales:
    - in: Indica que el parámetro se envía del cliente al servidor.
    - out: Indica que el parámetro se envía del servidor al cliente.
    - inout: Indica un parámetro que es iniciado por el cliente y enviado al servidor. El servidor puede modificar el valor del parámetro, de forma que, después de que se ha completado la operación, el valor del parámetro proporcionado por el cliente puede haber sido cambiado por el servidor.
- Los atributos direccionales son necesarios por dos razones:

  - Por eficiencia: Sin ellos, no habría forma de que el compilador del IDL pudiera saber si un valor de un parámetro se envía del cliente al servidor o viceversa. Además, permiten ahorros en el coste de transmisión. Por ejemplo, un parámetro *in* se envía únicamente del cliente al servidor.
  - Determinan la responsibilidad de la gestión de memoria: Controlan si es el cliente o el servidor el responsable de asignar y liberar la memoria para los parámetros.
- El nombre de las operaciones está limitado por la interfaz que los alberga y deben ser únicos dentro de esta interfaz, por lo cual, sobrecargar las operaciones es imposible.

En Resumen el IDL es el mecanismo dependiente del lenguaje de CORBA para definir tipos de datos e interfaces de objeto. IDL desacopla las implementaciones de los clientes de las de los servidores y establece el contrato al que se adhieren los clientes y los servidores. Las especificaciones IDL son traducidas por un compilador en stubs y esqueletos específicos para cada lenguaje. Los stubs y los esqueletos proporcionan las API del lado del cliente y del lado del servidor para dar soporte a las implementaciones en un lenguaje particular.

IDL proporciona un conjunto de tipos predefinidos que se pueden traducir fácilmente a la mayoría de los lenguajes de programación. El conjunto de tipos predefinidos se puede incrementar con tipos definidos por el usuario, como estructuras o secuencias. El IDL proporciona orientación a objetos a través de la herencia de interfaz que, además, establece compatibilidad de tipos y polimorfismo. Las excepciones sirven como un mecanismo uniforme de gestión de errores y los módulos proporcionan una construcción de agrupamiento para prevenir la contaminación del espacio de nombres. Los ID del Repositorio proporcionan nombres únicos internos para los tipos del IDL.

El desarrollo de este capítulo está basado en [11]y [1].

Apéndice B

Códigos del Capítulo 1

B.1. Código de la aplicación 1.1.1

Listing B.1: Clase EcoRMI.java

```
public interface EcoRMI extends java.rmi.Remote

{
  public String eco (String mensaje) throws java.rmi.RemoteException;
}
```

Listing B.2: Clase EcoServer.java

```
import java.rmi.*;
import java.rmi.registry.Registry;
import java.rmi.server.*;

public class EcoServer implements EcoRMI {
  static public final int REGISTRY_PORT = 2009;

  static private Registry registry;

  public static void main (String args[]){
    try { // Start the registry
      registry = java.rmi.registry.LocateRegistry.createRegistry(REGISTRY_PORT);
      System.out.println("EcoRMI RmiIsStarted at port "+REGISTRY_PORT);
    }
    catch (Exception e) {
      System.err.println("Warning: Can't start RMI registry at port: "+REGISTRY_PORT);
      e.printStackTrace();
    }
    try { // Register the console as the EjsConsoleServer
      EcoRMI stub = (EcoRMI) UnicastRemoteObject.exportObject(new EcoServer(), 0);
      registry.bind("EcoServer", stub);
      System.out.println("EcoServer Islistening at "+REGISTRY_PORT);
    }
    catch (Exception e) {
      System.err.println("Warning: Eco server can't listen at port: "+REGISTRY_PORT);
      e.printStackTrace();
    }
  }

  public EcoServer() {

  }

  public String eco(String mensaje) throws RemoteException {
    System.out.println("Recibido: " + mensaje);

    return mensaje;
  }
}
```

Listing B.3: Clase EcoClient.java

```
public class EcoClient {

  public static void main (String args[]){
    String mensajeEnviado;
    String mensajeRecibido;
    DataInputStream dataIn = new DataInputStream(System.in);
    BufferedReader in = new BufferedReader(new InputStreamReader(dataIn));

    try { // See if it is already running
      Registry reg = LocateRegistry.getRegistry(EcoServer.REGISTRY_PORT);
      EcoRMI stub = (EcoRMI) reg.lookup("EcoServer");

      // Hace bucle hasta el final de la entrada
      System.out.print("Eco>");
      while((mensajeEnviado = in.readLine())!= null){

        mensajeRecibido = stub.eco(mensajeEnviado);
        System.out.println(mensajeRecibido);
        System.out.print("Eco>");
      }
    }
    catch (Exception exc) {
      exc.printStackTrace();
    } // Do nothing

    System.exit(0);
  }
}
```

B.2. Código de la aplicación 1.1.2

Listing B.4: Clase ViewServer

```
public class ViewServer implements rmi_view_server.RMIViewServer {
  private boolean blockCommunication = false;

  public boolean getDouble (String name, double value) throws java.rmi.RemoteException {
    if (blockCommunication) return false;
    if ("abcisa".equals(name)) {
      _println ("Receiving t = "+value);
      t = value; return true;
    }
    else if ("ordenada".equals(name)) {
      _println ("Receiving x = "+value);
      x = value; return true;
    }
    return false;
  }

  public void update() throws java.rmi.RemoteException {
    blockCommunication = true;
    _view.trail.addPoint(t,x);
    _println ("Adding "+t+", "+x);
    _simulation.update();
    blockCommunication = false;
  }

  public void reset() throws java.rmi.RemoteException {
    _simulation.reset();
  }

}

static private final int REGISTRY_PORT = 2011;

static ViewServer server;

private void startServer () {
  if (server!=null) return; // Start server only once
  java.rmi.registry.Registry registry=null;
  try {
    // Start the registry
    registry = java.rmi.registry.LocateRegistry.createRegistry(REGISTRY_PORT);
    _println("ViewServer Rmi listening at port "+REGISTRY_PORT);
  }
  catch (Exception e) {
```

```
    _println("Warning: Can't start RMI registry at port: "+REGISTRY_PORT);
    e.printStackTrace();
    return;
  }
  try {
    // Register as the ViewServer
    server = new ViewServer();
    rmi_view_server.RMIViewServer stub = (rmi_view_server.RMIViewServer)
      java.rmi.server.UnicastRemoteObject.exportObject(server, 0);
    registry.bind("ViewServer", stub);
    _println("ViewServer is listening at "+REGISTRY_PORT);
  }
  catch (Exception e) {
    server = null;
    _println("Warning: Eco server can't listen at port: "+REGISTRY_PORT);
    e.printStackTrace();
  }
}
```

Listing B.5: Clase ClientModel

```
try {
  // See if it is already running
  java.rmi.registry.Registry reg = java.rmi.registry.LocateRegistry.getRegistry(2011);
  stub = (RMIViewServer) reg.lookup("ViewServer");
  stub.reset();
}
catch (Exception exc) {
  _println ("Server not responding or communication error!");
  exc.printStackTrace();
}
```

B.3. Código de la aplicación 1.2.1

Listing B.6: Clase Hello_impl.java

```
package server;

import hello.*;

public class Hello_impl extends HelloPOA {

  public void say_hello() {
    System.err.println ("¡¡¡Hello!!!");
  }
}
```

Listing B.7: Clase HelloServer.java

```
package server;

import hello.*;

public class HelloServer {

  // Inicialización/terminación de un programa CORBA Standalone
  // Idéntico para cliente y servidor

  public static void main(String args[]) {
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
      status = run(orb);
    } catch(Exception ex) {
      ex.printStackTrace();
      status = 1;
    }
```

```
    if(orb != null) {
      // Since the standard ORB.destroy() method is not present in
      // JDK 1.2.x, we must cast to com.ooc.CORBA.ORB so that this
      // will compile with all JDK versions
      try {
          ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
          ex.printStackTrace();
          status = 1;
      }
    }
    System.exit(status);
  }

  // El código del servidor

  static int run(org.omg.CORBA.ORB orb)
            throws org.omg.CORBA.UserException {

    // Buscar el POA (Adaptador de Objetos Portable para conectarse al bus ORB)
    org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
    org.omg.PortableServer.POA rootPOA =
                              org.omg.PortableServer.POAHelper.narrow(obj);

    // Obtener una referencia al POA manager
    org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

    // Crear una instancia del objeto
    Hello_impl helloImpl = new Hello_impl();
    Hello hello = helloImpl._this(orb);


    // Salvar la referencia en un fichero
    try {
         String ref = orb.object_to_string(hello);
         String refFile = "Hello.ref";
         java.io.PrintWriter out = new java.io.PrintWriter
         (new java.io.FileOutputStream(refFile));

         System.out.println(" Servidor listo: ");
         out.println(ref);
         out.close();

    } catch(java.io.IOException ex) {
         ex.printStackTrace();
         return 1;
    }

    manager.activate();
    orb.run();
    return 0;
  }

}
```

Listing B.8: Clase HelloClient.java

```
package client;
import hello.*;

public class HelloClient {

  //
  // Inicialización/terminación de un programa CORBA Standalone
  // Idéntico para cliente y servidor
  //
  public static void main(String args[]) {


    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
             "com.ooc.CORBA.ORBSingleton");
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
      status = run(orb);
    } catch(Exception ex) {
      ex.printStackTrace();
```

```
      status = 1;
    }

    if(orb != null) {
      // Since the standard ORB.destroy() method is not present in
      // JDK 1.2.x, we must cast to com.ooc.CORBA.ORB so that this
      // will compile with all JDK versions
      try {
          ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
          ex.printStackTrace();
          status = 1;
      }
    }
    System.exit(status);
  }

  // El código del cliente

  static int run(org.omg.CORBA.ORB orb){

    // Obtener una referencia al objeto "hello"
    // a partir de un fichero con el IOR en forma de string
    org.omg.CORBA.Object obj = null;

    //IOR del fichero Hello.ref
    try{
      String refFile ="Hello.ref";
      java.io.BufferedReader in= new java.io.BufferedReader(
          new java.io.FileReader(refFile));
      String ref = in.readLine();
      obj = orb.string_to_object(ref);

    }catch(java.io.IOException ex){

      ex.printStackTrace();
      return 1;

    }
    // Narrowing del IOR a tipo Hello
    Hello hello = HelloHelper.narrow(obj);

    // Bucle principal del cliente
    // Leer de teclado, invocar el servidor e imprimir por pantalla

    System.out.println("Hemos conectado con el servidor y nos dice Hello");
    hello.say_hello();
    return 0;
  }
}
```

B.4. Código de la aplicación 1.2.2

Listing B.9: Clase Echo_impl.java

```
package server;

import corba.*;

public class Echo_impl extends EchoPOA {
  private static EchoObject eo = new EchoObject();


  public String echo_(String aString) {
    System.out.println ("La cadena es <"+aString+">");
   return aString;
  }
}
```

Listing B.10: Clase EchoServer.java

```
package server;

import corba.*;
public class EchoServer {
```

```
  // Inicialización/terminación de un programa CORBA Standalone
  // Idéntico para cliente y servidor

  public static void main(String args[]) {
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
            status = run(orb, args);
          } catch(Exception ex) {
            ex.printStackTrace();
            status = 1;
          }

          if(orb != null) {
      // Since the standard ORB.destroy() method is not present in
      // JDK 1.2.x, we must cast to com.ooc.CORBA.ORB so that this
      // will compile with all JDK versions
      try {
                      ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
                      ex.printStackTrace();
                      status = 1;
      }
          }
    System.exit(status);
  }

  // El código del servidor

  static int run(org.omg.CORBA.ORB orb, String[] args)
            throws org.omg.CORBA.UserException {

    // Buscar el POA (Adaptador de Objetos Portable para conectarse al bus ORB)
    org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
    org.omg.PortableServer.POA rootPOA =
                              org.omg.PortableServer.POAHelper.narrow(obj);

    // Obtener una referencia al POA manager
    org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

    // Crear una instancia del objeto
    Echo_impl echoImpl = new Echo_impl();
    Echo echo = echoImpl._this(orb);

    // Salvar la referencia en un fichero
    try {
         String ref = orb.object_to_string(echo);
         String refFile = "Echo.ref";
         //IOR en el fichero Echo.ref
         java.io.PrintWriter out = new java.io.PrintWriter
         (new java.io.FileOutputStream(refFile));

         System.out.println("Servidor listo y en espera");
         out.println(ref);
         out.close();

    } catch(java.io.IOException ex) {
         System.err.println("echo.Server: can't write to '" +
         ex.getMessage() + "'");
         return 1;
    }

    manager.activate();
    orb.run();
    return 0;
  }
}
```

Listing B.11: Clase EchoClient.java

```
package client;
import java.io.*;
import corba.*;
```

```
public class EchoClient {

  // Inicialización/terminación de un programa CORBA Standalone
  // Idéntico para cliente y servidor

  public static void main(String args[]) {
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
            status = run(orb, args);
          } catch(Exception ex) {
            ex.printStackTrace();
            status = 1;
          }
          if(orb != null) {
      // Since the standard ORB.destroy() method is not present in
      // JDK 1.2.x, we must cast to com.ooc.CORBA.ORB so that this
      // will compile with all JDK versions
      try {
                      ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
                      ex.printStackTrace();
                      status = 1;
      }
          }
    System.exit(status);
  }

  // El código del cliente

  static int run(org.omg.CORBA.ORB orb, String[] args)
             throws org.omg.CORBA.UserException {

    // Obtener una referencia al objeto "echo"
    // a partir de un fichero con el IOR en forma de string
          org.omg.CORBA.Object obj = null;

    //IOR del fichero Echo.ref
          try{
            String refFile ="Echo.ref";
            java.io.BufferedReader in= new java.io.BufferedReader(
                new java.io.FileReader(refFile));
            String ref = in.readLine();
            obj = orb.string_to_object(ref);
          }catch(java.io.IOException ex){
            ex.printStackTrace();
            return 1;

          }
    // Narrowing del IOR a tipo Echo
          Echo echo = EchoHelper.narrow(obj);

    // Bucle principal del cliente
    // Leer de teclado, invocar el servidor e imprimir por pantalla

    try {
        System.out.println("Escribe un mensaje que desees enviar al servidor:");
        String input;
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));
        do {
              input = in.readLine();
              echo.echo_(input);
             } while(!input.equals("x"));
          }catch(IOException ex) {
             System.err.println("Can't read from `" + ex.getMessage() + "'");
             return 1;
          }
          return 0;
  }
}
```

Apéndice C

Códigos del Capítulo 2

C.1. Implementación de la clase VISTA_impl.java

Listing C.1: Clase VISTA_impl.java

```
package server;

import archivosVISTA.*;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import server.PanelDibujoSimplificado;

public class Vista_impl extends VISTAPOA {

  double miX = 0;
  double miY = 0;
  double miT = 0.0;

  //Definición de las variables de la vista
  double vistaX = 0;
  double vistaY = 0;

  //Definición de las tablas para los tipos de variables definidas:doubles, int, String...
  Hashtable<String,Double> tableDoubles = new Hashtable<String,Double>();
  Hashtable<String,Integer> tableInt = new Hashtable<String,Integer>();
  Hashtable<String, String> tableString = new Hashtable<String,String>();
  Hashtable<String, Boolean> tableBoolean = new Hashtable<String,Boolean>();
  Hashtable<String, String[]> tableStringArray = new Hashtable<String,String[]>();
  Hashtable<String, double[]> tableDoubleVector = new Hashtable<String,double[]>();
  Hashtable<String, int[]> tableIntVector = new Hashtable<String,int[]>();
  Hashtable<String, boolean[]> tableBooleanArray = new Hashtable<String,boolean[]>();
  Hashtable<String, double[][]> tableDoubleMatriz = new Hashtable<String,double[][]>();
  Hashtable<String, int[][]> tableIntMatriz = new Hashtable<String,int[][]>();

  String [] specificActions = {"alert","switchPanel","switchColor"};

  //Implementación de las variables en su tabla correspondiente
  public Vista_impl() {
    tableDoubles.put("x", miX);
    tableDoubles.put("y", miY);
    tableDoubles.put("t", miT);
  }

  // Método para enviar un double
  public void sendDouble(String name, double value){
    if (tableDoubles.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      System.out.println ("Asignar a <"+name+"> el valor de "+value);
      tableDoubles.put(name, value);
    }
  }
```

```
  // Método para enviar un int
  public void sendInt(String name, int value){
    if (tableInt.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> el valor de "+value);
      tableInt.put(name, value);
    }
  }

  // Método para enviar un String
  public void sendString(String name, String aString){
    if (tableString.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> el nombre de "+ aString);
      tableString.put(name, aString);
    }
  }

  //Método para enviar un boolean
  public void sendBoolean(String name, boolean value){
    if (tableBoolean.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> el valor "+ value);
      tableBoolean.put(name, value);
    }
  }

  //Método para enviar un array de String
  public void sendStringArray(String name, String[] aString){
    if (tableStringArray.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> la siguiente cadena: ");
      for(int i=0; i<aString.length;i++){
        System.out.print(aString[i]+ ", ");
      }
      System.out.println();
      tableStringArray.put(name, aString);
    }
  }

  //Método para enviar un vector de doubles, double[]
  public void sendDoubleVector(String name, double[] value){
    if (tableDoubleVector.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
    println ("Asignar a <"+name+"> los siguientes valores: ");
    for(int i=0; i<value.length;i++){
       print(value[i]+ ", ");
      }
      newLine();

    tableDoubleVector.put(name, value);
    }
  }

  //Método para enviar un vector de int, int[]
  public void sendIntVector(String name, int[] value){
    if (tableIntVector.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
    println ("Asignar a <"+name+"> los siguientes valores: ");
    for(int i=0; i<value.length;i++){
       print(value[i]+ ", ");
      }
      newLine();

    tableIntVector.put(name, value);
    }
  }

//Método para enviar un array de boolean, boolean[]
  public void sendBooleanArray(String name, boolean[] value){
    if (tableBooleanArray.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> los siguientes valores: ");
      for(int i=0; i<value.length;i++){
        print(value[i]+ ", ");
```

```
      }
      newLine();

      tableBooleanArray.put(name, value);
    }
  }

  //Método para enviar una matriz double[][]
  public void sendDoubleMatriz(String name, double[][] value){
    if (tableDoubleMatriz.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> los siguientes valores: ");
      for(int i=0; i<value.length;i++){
        for (int j=0; j<value[i].length; j++){
          print(value[i][j]+ ", ");
        }
        newLine();
      }
      newLine();
      tableDoubleMatriz.put(name, value);
    }

  }

  //Método para enviar una matriz int[][]
  public void sendIntMatriz(String name, int[][] value){
    if (tableIntMatriz.get(name)==null)
      println("Error: La variable "+name+ " no existe.");
    else{
      println ("Asignar a <"+name+"> los siguientes valores: ");
      for(int i=0; i<value.length;i++){
        for (int j=0; j<value[i].length; j++){
          print(value[i][j]+ ", ");
        }
        newLine();
      }
      newLine();
      tableIntMatriz.put(name, value);
     }
  }

  // Método reset: Deja la vista de la simulación como al inicio.
  public void reset(){
   println("Reset");
   tableDoubles.put("x", vistaX);
   tableDoubles.put("y", vistaY);

  }

  // Método para inicializar las variables
  public void initialize(){
  println("Inicialize");
  tableDoubles.put("t", 0.0);

  }

  // Método para actualizar las variables
  public void update() {
    println("Actualización de las variables");
    miX = tableDoubles.get("x");
    System.out.println("x = "+miX);
    miY = tableDoubles.get("y");
    System.out.println("y = "+miY);
    miT = tableDoubles.get("t");
    System.out.println("t = "+miT);

  }

  public void clearMessages(){}

  public void clearView(){}

  public void print(String _txt){
    System.out.print(_txt);
  }

  public void println(String _txt){
    System.out.println(_txt);
  }
  public void newLine(){
    System.out.println();
  }
```

```
// Método que nos muestra una lista con los métodos específicos disponibles

public void getListSpecificActions(){
  println("Los métodos específicos disponibles son:");
  for(int i=0; i< specificActions.length; i++){
    println(specificActions[i]);
  }
}

      /*
       * El método actionsNoStandar llama a la acción específica elegida, por ejemplo
 * alert, switchPanel,...
 */

public void actionsNoStandar(String name, String value){

  if (name.equals(specificActions[0])) alert("Alerta",value);
  else if (name.equals(specificActions[1])) switchPanel(Integer.parseInt(value));
  else  switchColor(Integer.parseInt(value));
}

//El método alert muestra un mensaje en una ventana gráfica
public void alert(String title, String message){

  PanelDibujoSimplificado alerta = new PanelDibujoSimplificado(title);
  alerta.addEjesCoordenados(false);

  Label texto = new Label(message);
  texto.setAlignment(Label.CENTER);
  texto.setForeground(Color.BLUE);
  alerta.add ("Center",texto);
  alerta.repaint();
};

/*
 * El método switchPanel te permite elegir el panel que deseas cuando hay más
 * de uno disponible
 */

      public void switchPanel(int numberPanel){};

 //El método switchColor te permite elegir un color

      public void switchColor(int numberPanel){};

 /*
  * Método DibujarCurva construye un panel de dibujo y dibuja
  * la colección de puntos dados.
  */

      public void dibujarCurva(String name, boolean ejes, double[][] value){

   PanelDibujoSimplificado pd = new PanelDibujoSimplificado(name);
   pd.addEjesCoordenados(ejes);
   pd.addCurva("rojo",value);
   pd.repaint();

   /*
    * Finalmente enviamos una orden para que la ventana abierta
    * se pueda cerrar correctamente y así detener la aplicación.
    */

   pd.addWindowListener(
       new WindowAdapter() {
   public void windowClosing(WindowEvent e) {
       System.exit(0);
   }
   }
);

 }
}
```

C.2. Implementación de la clase VistaServer.java

Listing C.2: Clase VistaServer.java

```
package server;
import archivosVISTA.*;

public class VistaServer {

  // Inicialización/terminación de un programa CORBA Standalone
  // Identico para cliente y servidor

  public static void main(String args[]) {
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");
    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
      status = run(orb, args);
    } catch(Exception ex) {
      ex.printStackTrace();
      status = 1;
    }
    if(orb != null) {

      try {
          ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
          ex.printStackTrace();
          status = 1;
      }
    }
    System.exit(status);
  }

  // El código del servidor

  static int run(org.omg.CORBA.ORB orb, String[] args)
  throws org.omg.CORBA.UserException {

    // Buscar el POA (Adaptador de Objetos Portable para conectarse al bus ORB)
    org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
    org.omg.PortableServer.POA rootPOA =
                              org.omg.PortableServer.POAHelper.narrow(obj);

    // Obtener una referencia al POA manager
    org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

    // Crear una instancia del objeto
    Vista_impl vistaImpl = new Vista_impl();
    VISTA vista = vistaImpl._this(orb);

    // Salvar la referencia en un fichero
    try {
         String ref = orb.object_to_string(vista);
         String refFile = "Vista.ref";
         java.io.PrintWriter out = new java.io.PrintWriter
         (new java.io.FileOutputStream(refFile));

         System.out.println("Servidor Vista listo");
         out.println(ref);
         out.close();

    } catch(java.io.IOException ex) {
         System.err.println("EjemploServer: can't write to '" +
         ex.getMessage() + "'");
         return 1;
    }

    manager.activate();
    orb.run();
    return 0;
  }
}
```

C.3. Implementación de la clase ModelClient.java

Listing C.3: Clase ModelClient.java

```
package client;

import archivosVISTA.*;


public class ModelClient {

    //
    // Inicialización/terminación de un programa CORBA Standalone
    // Identico para cliente y servidor
    //
    public static void main(String args[]) {
      int status = 0;
      org.omg.CORBA.ORB orb = null;

      java.util.Properties props = System.getProperties();
      props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass",
                "com.ooc.CORBA.ORBSingleton");

      try {
        orb = org.omg.CORBA.ORB.init(args, props);
        // Llamar al método run con las acciones propias del cliente o servidor
        status = run(orb,args);
      } catch(Exception ex) {
        ex.printStackTrace();
        status = 1;
      }

      if(orb != null) {

        try {
            ((com.ooc.CORBA.ORB)orb).destroy();
        } catch(Exception ex) {
            ex.printStackTrace();
            status = 1;
        }
      }
      System.exit(status);
    }

  //
  // El código del cliente
  //

  static int run(org.omg.CORBA.ORB orb, String[] args)
     throws org.omg.CORBA.UserException {
    // Obtener una referencia al objeto "Vista"
    // a partir de un fichero con el IOR en forma de string
    org.omg.CORBA.Object obj = null;

    // IOR del fichero Vista.ref
    try{
      String refFile ="Vista.ref";
      java.io.BufferedReader in= new java.io.BufferedReader(
          new java.io.FileReader(refFile));
      String ref = in.readLine();
      obj = orb.string_to_object(ref);
    }catch(java.io.IOException ex){
      ex.printStackTrace();
      return 1;

    }

    // Narrowing del IOR a tipo Vista
    VISTA vista = VISTAHelper.narrow(obj);

    // Bucle principal del cliente

    /*
    vista.sendDouble("x", 3);
    vista.sendDouble("y", 25);
    vista.sendString("nombre", "Almudena");
    vista.sendDouble("t", 0.5);
    vista.update();

    */
```

```
    /*
     * Comprobamos algunos métodos de la vista.
     */
    vista.sendDouble("x",25);
    vista.sendString("nombre", "Almudena");
    vista.getListSpecificActions();
    vista.update();
    vista.newLine();
    vista.actionsNoStandar("alert","Estamos comprobando la eficacia del método alert");

    /*
     * Dibujamos la función seno en el intervalo [0,4) e imprimimos sus puntos.
     */
    vista.newLine();
    vista.println("Representación de la función seno en el intervalo [0,4)");
    vista.println("Los puntos calculados son los siguientes:");
    vista.newLine();

    double t = 0.0;
    double x, y;
    double[][] puntos = new double[40][2];
    int i=0;
    while (t<4) {
      x = t;
      y = Math.sin(t);
      puntos[i][0]=x;
      puntos[i][1]=y;
      //t += 0.1;
      t= step(t,0.1);
      i += 1;
      vista.sendDouble("x", x);
      vista.sendDouble("y", y);
      vista.sendDouble("t", t);
      vista.update();
    }
    vista.dibujarCurva("Representación de la función seno en [0,4)",true, puntos);



    return 0;
  }

  /*
   * Este método da un salto en el cálculo de los datos
   */
   public static double step(double value, double longPaso) {
     return value += longPaso;
  }

}
```

Apéndice D

Códigos del Capítulo 3

D.1. Implementación de VistaInt_impl.java

Listing D.1: Clase VISTAInt_impl.java

```
package server;

import archivosMODELO_INT.MODELO_INT;
import archivosMODELO_INT.MODELO_INTHelper;
import archivosVISTA_INT.*;

import java.util.*;
import java.awt.*;
import java.awt.event.*;

import server.PanelDibujoSimplificado;

public class VistaInt_impl extends VISTA_INTPOA implements ActionListener{

  MODELO_INT modeloInt;
  double miX = 0;
  double miY = 0;
  double miT = 0.0;
  double miTheta = Math.PI/9.;
  double miOmega = 0;

  //Definición de las variables de la vista
  double vistaX = 0;
  double vistaY = 0;
  double vistaTheta = 0;
  double vistaOmega = 0;

  //Definición de las tablas para los tipos de variables definidas:doubles, int, String...
  Hashtable<String,Double> tableDoubles = new Hashtable<String,Double>();
  Hashtable<String,Integer> tableInt = new Hashtable<String,Integer>();
  Hashtable<String, String> tableString = new Hashtable<String,String>();
  Hashtable<String, Boolean> tableBoolean = new Hashtable<String,Boolean>();
  Hashtable<String, String[]> tableStringArray = new Hashtable<String,String[]>();
  Hashtable<String, double[]> tableDoubleVector = new Hashtable<String,double[]>();
  Hashtable<String, int[]> tableIntVector = new Hashtable<String,int[]>();
  Hashtable<String, boolean[]> tableBooleanArray = new Hashtable<String,boolean[]>();
  Hashtable<String, double[][]> tableDoubleMatriz = new Hashtable<String,double[][]>();
  Hashtable<String, int[][]> tableIntMatriz = new Hashtable<String,int[][]>();

  String [] specificActions = {"alert","switchPanel","switchColor"};

  //Implementación de las variables en su tabla correspondiente
  public VistaInt_impl() {
    tableDoubles.put("x", miX);
    tableDoubles.put("y", miY);
    tableDoubles.put("t", miT);
    tableDoubles.put("theta", miTheta);
    tableDoubles.put("omega", miOmega);
  }
```

```
// Método para enviar un double
public void sendDouble(String name, double value){
  if (tableDoubles.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    System.out.println ("Asignar a <"+name+"> el valor de "+value);
    tableDoubles.put(name, value);
  }
}

// Método para enviar un int
public void sendInt(String name, int value){
  if (tableInt.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> el valor de "+value);
    tableInt.put(name, value);
  }
}

// Método para enviar un String
public void sendString(String name, String aString){
  if (tableString.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> el nombre de "+ aString);
    tableString.put(name, aString);
  }
}

//Método para enviar un boolean
public void sendBoolean(String name, boolean value){
  if (tableBoolean.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> el valor "+ value);
    tableBoolean.put(name, value);
  }
}

//Método para enviar un array de String
public void sendStringArray(String name, String[] aString){
  if (tableStringArray.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> la siguiente cadena: ");
    for(int i=0; i<aString.length;i++){
      System.out.print(aString[i]+ ", ");
    }
    System.out.println();
    tableStringArray.put(name, aString);
  }
}

//Método para enviar un vector de doubles, double[]
public void sendDoubleVector(String name, double[] value){
  if (tableDoubleVector.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
  println ("Asignar a <"+name+"> los siguientes valores: ");
  for(int i=0; i<value.length;i++){
      print(value[i]+ ", ");
    }
    newLine();

  tableDoubleVector.put(name, value);
  }
}

//Método para enviar un vector de int, int[]
public void sendIntVector(String name, int[] value){
  if (tableIntVector.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
  println ("Asignar a <"+name+"> los siguientes valores: ");
  for(int i=0; i<value.length;i++){
      print(value[i]+ ", ");
    }
    newLine();

  tableIntVector.put(name, value);
  }
}
```

```
        //Método para enviar un array de boolean, boolean[]
public void sendBooleanArray(String name, boolean[] value){
  if (tableBooleanArray.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> los siguientes valores: ");
    for(int i=0; i<value.length;i++){
      print(value[i]+ ", ");
    }
    newLine();

    tableBooleanArray.put(name, value);
  }
}

//Método para enviar una matriz double[][]
public void sendDoubleMatriz(String name, double[][] value){
  if (tableDoubleMatriz.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> los siguientes valores: ");
    for(int i=0; i<value.length;i++){
      for (int j=0; j<value[i].length; j++){
        print(value[i][j]+ ", ");
      }
      newLine();
    }
    newLine();
    tableDoubleMatriz.put(name, value);
  }

}

//Método para enviar una matriz int[][]
public void sendIntMatriz(String name, int[][] value){
  if (tableIntMatriz.get(name)==null)
    println("Error: La variable "+name+ " no existe.");
  else{
    println ("Asignar a <"+name+"> los siguientes valores: ");
    for(int i=0; i<value.length;i++){
      for (int j=0; j<value[i].length; j++){
        print(value[i][j]+ ", ");
      }
      newLine();
    }
    newLine();
    tableIntMatriz.put(name, value);
   }
}

// Método reset deja la vista de la simulación como al inicio.
public void reset(){
 println("Reset");
 tableDoubles.put("x", vistaX);
 tableDoubles.put("y", vistaY);
 tableDoubles.put("t", 0.0);
 tableDoubles.put("theta", vistaTheta);
 tableDoubles.put("omega", vistaOmega);
 clearView();
}

// Método para inicializar las variables
public void initialize(){
println("Inicialize");
tableDoubles.put("t", 0.0);

}

// Método para actualizar las variables
public void update() {
  println("Actualización de las variables");
  /*
  miX = tableDoubles.get("x");
  System.out.println("x = "+miX);
  miY = tableDoubles.get("y");
  System.out.println("y = "+miY);
  miT = tableDoubles.get("t");
  System.out.println("t = "+miT);
  */
  miTheta = tableDoubles.get("theta");
  System.out.print("theta = "+miTheta+", ");
  miOmega = tableDoubles.get("omega");
```

```
    System.out.print("omega = "+miOmega+", ");
    miT = tableDoubles.get("t");
    System.out.println("t = "+miT);

  }

  public void clearMessages(){}

  public void clearView(){}

  public void print(String _txt){
    System.out.print(_txt);
  }

  public void println(String _txt){
    System.out.println(_txt);
  }
  public void newLine(){
    System.out.println();
  }

  // Método que nos muestra una lista con los métodos específicos disponibles

  public void getListSpecificActions(){
    println("Los métodos específicos disponibles son:");
    for(int i=0; i< specificActions.length; i++){
      println(specificActions[i]);
    }
  }

        /*
         * El método actionsNoStandar llama a la acción específica elegida, por ejemplo
   * alert, switchPanel,...
   */

  public void actionsNoStandar(String name, String value){

    if (name.equals(specificActions[0])) alert("Alerta",value);
    else if (name.equals(specificActions[1])) switchPanel(Integer.parseInt(value));
    else  switchColor(Integer.parseInt(value));
  }

  //El método alert muestra un mensaje en una ventana gráfica
  public void alert(String title, String message){

    PanelDibujoSimplificado alerta = new PanelDibujoSimplificado(title);
    alerta.addEjesCoordenados(false);

    Label texto = new Label(message);
    texto.setAlignment(Label.CENTER);
    texto.setForeground(Color.BLUE);
    alerta.add ("Center",texto);
    alerta.repaint();
  };

  /*
   * El método switchPanel te permite elegir el panel que deseas cuando hay más
   * de uno disponible
   */

  public void switchPanel(int numberPanel){};

  /*
   * El método switchColor te permite elegir un color
   */

  public void switchColor(int numberPanel){};

   /*
    * Método DibujarCurva dibuja la colección de puntos dados
    * en la interfaz gráfica.
    */

   public void dibujarCurva(boolean ejes, double[][] value){

     pd.addEjesCoordenados(ejes);
     pd.addCurva("rojo",value);
     pd.repaint();

      /*
       * Finalmente enviamos una orden para que la ventana abierta
       * se pueda cerrar correctamente y así detener la aplicación.
       */
```

```
    pd.addWindowListener(
        new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    }
 );
}

   /*
    * Método DibujarPunto dibuja en la interfaz gráfica el punto dado.
    */

    public void dibujarPunto(String color, boolean ejes, double x , double y){

      if (color.equals("rojo")) {

        pd.addCurva("rojo",x,y);
      }
      else pd.addCurva("azul", x,y);

      pd.addEjesCoordenados(ejes);
      pd.repaint();

      /*
       * Finalmente enviamos una orden para que la ventana abierta
       * se pueda cerrar correctamente y así detener la aplicación.
       */

      pd.addWindowListener(
          new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
          System.exit(0);
      }
      }
   );

  }

  /*
   * El método VistaInt arranca una gráfica inicialmente vacía dónde se representarán los
   * datos enviados
   */
   public PanelDibujoSimplificado VistaInt(String name){

    PanelDibujoSimplificado pd = new PanelDibujoSimplificado(name);
    pd.addEjesCoordenados(true, -1, 15, -2.5, 3,0,0);
    pd.setBackground(Color.white);
    Panel buttonPanel = new Panel (new FlowLayout(FlowLayout.CENTER));
    buttonPanel.setBackground(Color.LIGHT_GRAY);
    pd.add ("South",buttonPanel);
    Button b1 = new Button ("Play");
    buttonPanel.add (b1);
    Button b2 = new Button ("Pause");
    buttonPanel.add (b2);
    Button b3=new Button ("Reset");
    buttonPanel.add(b3);

    b1.addActionListener(new ActionListener() {
      // enviar mensaje al cliente
      public void actionPerformed( ActionEvent evento ){
       // modeloInt.accion("play");
        modeloInt.play();
      }
    }
    );
    b2.addActionListener(new ActionListener() {
      // enviar mensaje al cliente
      public void actionPerformed( ActionEvent evento ){
        modeloInt.accion("pause");
      }
  }
    );

    b3.addActionListener(new ActionListener() {
      // enviar mensaje al cliente
      public void actionPerformed( ActionEvent evento ){
        modeloInt.accion("reset");
      }
      }
    );
```

```
    Panel Etiquetas = new Panel (new FlowLayout(FlowLayout.CENTER));
    Etiquetas.setBackground(Color.LIGHT_GRAY);
    pd.add("North",Etiquetas);
    Label etiqueta1 = new Label("Rojo = Velocidad Angular;");
    etiqueta1.setForeground(Color.BLACK);
    Label etiqueta2 = new Label("Azul = Desplazamiento;");
    etiqueta2.setForeground(Color.BLACK);
    Etiquetas.add(etiqueta1);
    Etiquetas.add(etiqueta2);

    pd.repaint();
    return pd;
  }

    public void actionPerformed( ActionEvent evento ){};

    //Sentencia para crear la interfaz gráfica inicialmente vacía

    PanelDibujoSimplificado pd = VistaInt("Representación gráfica del péndulo");


    //Método que conecta el modelo como servidor y la vista como cliente.

    public void setModel(String refFile){
      // Obtener una referencia al objeto "ref"
     // a partir de un fichero con el IOR en forma de string
     org.omg.CORBA.Object obj = null;

     // IOR del fichero ref
     try{
       java.io.BufferedReader in= new java.io.BufferedReader(
           new java.io.FileReader(refFile));
       String ref = in.readLine();
       obj = this._orb().string_to_object(ref);
       modeloInt = MODELO_INTHelper.narrow(obj);
     }catch(java.io.IOException ex){
       ex.printStackTrace();
     }
    }
       }
```

D.2. Implementación de VistaServerInt.java

Listing D.2: Clase VISTAServerInt.java

```
package server;

import archivosVISTA_INT.*;

public class VistaServerInt {

  //
  // Inicialización/terminación de un programa CORBA Standalone
  // Idéntico para cliente y servidor
  //
  public static void main(String args[]) {
    int status = 0;
    org.omg.CORBA.ORB orb = null;

    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

    try {
      orb = org.omg.CORBA.ORB.init(args, props);
      // Llamar al método run con las acciones propias del cliente o servidor
      status = run(orb, args);
    } catch(Exception ex) {
      ex.printStackTrace();
      status = 1;
    }

    if(orb != null) {

      try {
          ((com.ooc.CORBA.ORB)orb).destroy();
      } catch(Exception ex) {
          ex.printStackTrace();
```

```
          status = 1;
      }
    }
    System.exit(status);
  }

  // El código del servidor

  static int run(org.omg.CORBA.ORB orb, String[] args)
  throws org.omg.CORBA.UserException {

    // Buscar el POA (Adaptador de Objetos Portable para conectarse al bus ORB)
    org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
    org.omg.PortableServer.POA rootPOA =
                                  org.omg.PortableServer.POAHelper.narrow(obj);

    // Obtener una referencia al POA manager
    org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

    // Crear una instancia del objeto
    VistaInt_impl vistaIntImpl = new VistaInt_impl();
    VISTA_INT vistaInt = vistaIntImpl._this(orb);

    // Salvar la referencia en un fichero
    try {
         String ref = orb.object_to_string(vistaInt);
         String refFile = "VistaInt.ref";
         java.io.PrintWriter out = new java.io.PrintWriter
         (new java.io.FileOutputStream(refFile));

         System.out.println("Servidor Vista Interactiva listo");
         out.println(ref);
         out.close();

    } catch(java.io.IOException ex) {
         System.err.println("EjemploServer: can't write to '" +
         ex.getMessage() + "'");
         return 1;
    }
    manager.activate();
    orb.run();
    return 0;
  }
}
```

D.3. Implementación de ModelInt_impl.java

Listing D.3: Clase ModelInt_impl.java

```
package client;

import java.util.Hashtable;
import client.MetodosAux.Metodos;
import archivosMODELO_INT.*;
import archivosVISTA_INT.*;

public class ModelInt_impl extends MODELO_INTPOA implements java.lang.Runnable {

    VISTA_INT vistaInt;
    String [] acciones = {"play","pause","reset"};

    //Definición de las variables necesarias para el modelo del Péndulo
    double theta = Math.PI/9.;
    double omega = 0;
    double t_0 = 0.0;
    double ModelTheta = 0;
    double ModelOmega = 0;
    Hashtable<String,Double> tableDoubles = new Hashtable<String,Double>();

    //Implementación de las variables en su tabla correspondiente
    public ModelInt_impl() {
      tableDoubles.put("theta", theta);
      tableDoubles.put("omega", omega);
      tableDoubles.put("t", t_0);
    }
```

```
  //Método para invocar a la acción seleccionada desde la vista
  public void accion(String name){
    System.out.println ("Mensaje recibido: "+name);
    if (name.equals(acciones[0])) play();
    else if (name.equals(acciones[1])) pause();
    else  reset();
  }

  //Método para comenzar la animación

  public synchronized void play() {
    setVista("VistaInt.ref");
    if(animationThread!=null) return; // the animation is running
      animationThread = new Thread(this);
      animationThread.setPriority(Thread.MIN_PRIORITY);
      animationThread.setDaemon(true);
      animationThread.start(); // start the animation

  }

  //Método para hacer una pausa en la animación.

  public synchronized void pause() {
    if(animationThread==null) return; // animation thread is already dead
    Thread tempThread = animationThread; // local reference
    animationThread = null; // signal the animation to stop
    if(Thread.currentThread()==tempThread) return; // cannot join with own thread so
    //return another thread has called this method in order to stop the animation
    try {                      // thread guard against an exception in applet mode
     tempThread.interrupt(); // get out of a sleep state
     tempThread.join(100);  // wait up to 1/100 second for animation thread to stop
    } catch(Exception e) {
     System.out.println("excetpion in stop animation"+e);
    }
  }

 //Método para finalizar la animación

  public synchronized void stop() {}

 //Método para dar un salto en el cálculo de los datos

  public synchronized double step(double value, double longPaso) {
      return value += longPaso;
  }

  /*
  * Este método hace un reset en el modelo, en la vista y
  * actualiza la vista
  */

  public synchronized void reset() {
    tableDoubles.put("theta", ModelTheta);
    tableDoubles.put("omega", ModelOmega);
    vistaInt.reset();
    vistaInt.update();
  }

  /*
   * Este método hace un initialize en el modelo, en la vista y
   * actualiza la vista
   */

  public synchronized void initialize() {
    tableDoubles.put("t", 0.0);
    vistaInt.initialize();
    vistaInt.update();
  }

 /*
  * Este método actualiza las variables del modelo
  */

  public synchronized void update() {}

  //Método que conecta la vista como servidor y el modelo como cliente

  public void setVista(String refFile){
    // Obtener una referencia al objeto "ref"
    // a partir de un fichero con el IOR en forma de string
   org.omg.CORBA.Object obj = null;
```

```
     // IOR del fichero ref
     try{
       java.io.BufferedReader in= new java.io.BufferedReader(
           new java.io.FileReader(refFile));
       String ref = in.readLine();
       obj = this._orb().string_to_object(ref);
       vistaInt = VISTA_INTHelper.narrow(obj);
     }catch(java.io.IOException ex){
       ex.printStackTrace();
     }
    }

    //Arrancamos un thread de Java
      private volatile java.lang.Thread animationThread = null;
      private long delay = 10; // The delay between successive steps

    /*
     * Implementación de la interfaz Runnable.
     * A este método no se accede directamente.
     */
      double x,y, h,l,g;
      double[] solucion;

      public void run() {
       long sleepTime = delay;
       while(animationThread==Thread.currentThread()) {
          long currentTime = System.currentTimeMillis();

          // step your model here ...
          h=0.01;
          l = 1;
          g = 9.81;
          double[] Y_0={theta, omega};
          /*LLamamos al método de RungeKutta para hallar la
           * solución del péndulo.
           */
          solucion= Metodos.RungeKutta(Y_0,t_0,h,g,l);
          vistaInt.sendDouble("theta",solucion[0]);
          vistaInt.sendDouble("omega",solucion[1]);
          vistaInt.sendDouble("t", t_0);
          vistaInt.update();
          theta =solucion[0];
          omega = solucion[1];
          tableDoubles.put("theta", theta);
          tableDoubles.put("omega", omega);
          tableDoubles.put("t", t_0);
          t_0=step(t_0,h);
          Y_0[0] = solucion[0];
          Y_0[1] = solucion[1];

          x = l*Math.sin(theta);
          y = -l*Math.cos(theta);

          //Representa el movimiento del péndulo
          /*
           * vistaInt.dibujarPunto(true,x,y);
          */

          //Representa la velocidad angular del péndulo.
          vistaInt.dibujarPunto("rojo",true,t_0,omega);

          //Representa el desplazamiento del péndulo.
          vistaInt.dibujarPunto("blue",true,t_0,x-l);

          // adjust the sleep time to try and achieve a constant animation rate
          // some VMs will hang if sleep time is less than 10
          sleepTime = delay-(System.currentTimeMillis()-currentTime);
          // Math.max(10, delay-(System.currentTimeMillis()-currentTime));
          if (sleepTime<10) Thread.yield();
          else {
            try { Thread.sleep(sleepTime); }
            catch(InterruptedException ie) {}
          }
       }
    }
}
```

D.4. Implementación de ModelClientInt.java

Listing D.4: Clase ModelClientInt.java

```
package client;

import archivosVISTA_INT.*;
import client.ModelInt_impl;
import archivosMODELO_INT.*;

public class ModelClientInt{

  static VISTA_INT vistaInt;

    // Inicialización/terminación de un programa CORBA Standalone
    // Idéntico para cliente y servidor

    public static void main(String args[]) {
      int status = 0;
      org.omg.CORBA.ORB orb = null;

      java.util.Properties props = System.getProperties();
      props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass",
                "com.ooc.CORBA.ORBSingleton");

      try {
        orb = org.omg.CORBA.ORB.init(args, props);
        // Llamar al método run con las acciones propias del cliente o servidor
        status = run(orb,args);
      } catch(Exception ex) {
        ex.printStackTrace();
        status = 1;
      }
      new ModelClientInt();

      }

    //
    // El código del cliente --> Nos conectamos como servidor
    //

    static int run(org.omg.CORBA.ORB orb, String[] args)
    throws org.omg.CORBA.UserException {

      // Buscar el POA (Adaptador de Objetos Portable para conectarse al bus ORB)
      org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
      org.omg.PortableServer.POA rootPOA =
                                  org.omg.PortableServer.POAHelper.narrow(obj);

      // Obtener una referencia al POA manager
      org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

      String refFile = "ModeloInt.ref";
      runServer(orb,args,refFile);
      runCliente(orb,args);
      vistaInt.setModel(refFile);

      manager.activate();
      orb.run();

      return 1;
    }

    static int runCliente(org.omg.CORBA.ORB orb, String[] args)
       throws org.omg.CORBA.UserException {
    // Obtener una referencia al objeto "VistaInt"
    // a partir de un fichero con el IOR en forma de string
      org.omg.CORBA.Object obj = null;

      // IOR del fichero VistaInt.ref
      try{
        String refFile ="VistaInt.ref";
        java.io.BufferedReader in= new java.io.BufferedReader(
          new java.io.FileReader(refFile));
        String ref = in.readLine();
        obj = orb.string_to_object(ref);

      }catch(java.io.IOException ex){
        ex.printStackTrace();
      return 1;
```

```
    }

    // Narrowing del IOR a tipo Vista
     vistaInt = VISTA_INTHelper.narrow(obj);

    // Bucle principal del cliente

       return 0;
  }

  // El código del servidor Modelo

  static int runServer(org.omg.CORBA.ORB orb, String[] args, String refFile)
  throws org.omg.CORBA.UserException {
    // Crear una instancia del objeto
    ModelInt_impl modeloIntImpl = new ModelInt_impl();
    MODELO_INT modelInt = modeloIntImpl._this(orb);

    // Salvar la referencia en un fichero
    try {
         String ref = orb.object_to_string(modelInt);
         java.io.PrintWriter out = new java.io.PrintWriter
         (new java.io.FileOutputStream(refFile));

         System.out.println("Servidor Modelo Interactivo listo");
         out.println(ref);
         out.close();

    } catch(java.io.IOException ex) {
         System.err.println("EjemploServer: can't write to '" +
         ex.getMessage() + "'");
         return 1;
    }
    return 0;
  }
}
```

Bibliografía

[1] *Idl del omg. capítulo 4: El lenguaje de definición de interfaces del omg*, http://antares.itmorelia.edu.mx/ fmorales/SisDisII/IDLdelOMG.pdf.

[2] *The java tutorials: Concurrency.*

[3] *Mi web: Diseño y aplicaciones de sistemas distribuidos*, http://web-sisop.disca.upv.es/dya/.

[4] *Orbacus for c++ and java*, http://futura.disca.upv.es/dya/downloads/sw-docs/docs/orbacus/OB-4.0.3.pdf.

[5] *Programación en java: Fundamentos de programación y principios de diseño*, http://elvex.ugr.es/decsai/java/index.html.

[6] *Tutorial de corba desde java*, http://www.terra.es/personal2/monkiki/doc/CORBA/.

[7] Joshuaa Bluch, *Effective java programming language guide*, ADDISON-WESLEY, 2002.

[8] Santi Caballé and Fatus Xhafa, *Aplicaciones distribuidas en java con tecnología rmi*, Delta Publicaciones, 2008.

[9] Wolfgang Christian, *Open source physics. a user's guide with examples*, Pearson; Addison-Wesley, 2007.

[10] Francisco Esquembre, *Creación de simulaciones interactivas en java. aplicación a la enseñanza de la física*, Pearson; Prentice Hall, 2005.

[11] Alvaro Rendón Gallón, *Claves para la interoperabilidad en corba*, Master's thesis, Universidad de Canca, ftp://jano.unicauca.edu.co/cursos/SistDistrib/Corba/ClavesInterop.pdf, 2002.

[12] Ducan Grisby, Apasphere Ltd, Sai-LaiLo, David Riddoch, and AT&T Laboratories Cambridge, *The omniorb versión 4.1. user's guide*, http://omniorb.sourceforge.net/omni4.1/omniORB.pdf, 2009.

[13] David Basanta Gutiérrez and Lourdes Tajes Matínez, *Tecnologías para el desarrollo de sistemas distribuidos: Java versus corba*, X Jornadas de paralelismo. La Manga del Mar Menor, Murcia (http://www.di.uniovi.es/ lourdes/publicaciones/bt99.pdf), 1999.

[14] M.Sanchez, J.M.García, A.F.Gómez, and H.Martínez, *Generación de simuladores eficientes para procesos complejos basados en arquitecturas distribuidas*, X Jornadas de paralelismo. La Manga del Mar Menor, Murcia (http://ditec.um.es/ jmgarcia/papers/acasanchez.pdf), 1999.

[15] ORACLE. Sun Developer Network (SDN), http://java.sun.com/developer/technicalArticles/Servlets/corba/ code1, *Corba comunication*.

[16] ORACLE. Sun Developer Network (SDN), http://java.sun.com/developer/onlineTraining/corba/corba.html, *Introduction to corba*.

[17] ORACLE. Sun Developer Network (SDN), http://java.sun.com/javase/technologies/core/corba/index.jsp, *Java se core tecnologies- corba/rmi-iiop*.

[18] ORACLE. Sun Developer Network (SDN), http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, *Remothe method invocation home (rmi)*.

[19] José Carlos Cortizo Pérez, *Corba*, Tech. report, Universidad Europea de Madrid, http://www.esi.uem.es/jccortizo/temasConcu/corba.pdf, 2009/2010.

[20] Ramón Jesús Millán Tejedor, *Programación de objetos distribuidos con corba*, http://www.ramonmillan.com/tutoriales/corba.php/Programacion, 1999.

Listado de abreviaturas

- API: Application programming interface
- CORBA: Common Object Request Broker Architecture
- EJS: Easy Java Simulations
- GIOP: General Inter ORB Protocol
- IDL: Interface Definition Language
- OMG: Object Management Group
- ORB: Object Request Broker
- RMI: Remothe Method Invocation