Ingeniería Técnica en Informática de Gestión
Laboratorio de Programación Avanzada

Curso 2010/2011
Práctica Cocurrente N: 2

Autor: 09047323C - Lucena Pumar, Diego Antonio

Reservados algunos derechos de autor amparados en la Ley internacional del Copyright.
© Diego Antonio Lucena Pumar.
Puedes libremente distribuir este documento, y usarlo para tus propios cometidos, pero nunca serán el de lucro.
En la Villa de Meco a 27 de Diciembre de 2010.
Fecha de última revisión: Meco, 8 de Enero de 2011.
Editor: Diego Antonio Lucena Pumar.
Maquetado directamente en LaTeX 2_ε

A los que empiezan por los cimientos. A todos, GRACIAS.

"Java debe ser un territorio neutral, la Suiza de la tecnología informática."

James Gosling

Índice general

Índice de figuras

Capítulo 1

Análisis de Alto Nivel

1.1. ¿Cómo hemos desarrollado la Aplicación?

1.2. Requisitos

La apliación Ciudad Deportiva del Laboratorio de Programación Avanzada para el curso 2010/2011 en las disciplinas de Ingenierías Técnicas en Informática propuso las siguientes restricciones que debe cumplir la aplicación. A continuación las enumeramos:

1.2.1. Respecto al funcionamiento general de la aplicación

- El número total de Hilos que se ejecutan es de 49, cada uno de ellos identificado entre los número 51 y 99. Los que tienen número de identificación par son Hilos que corresponden a deportistas masculinos, por contra, los impares, corresponden a deportistas femeninos.

  Cada deportista se irá creando cada 0,1 segundos.

- En el complejo aplicación Ciudad Deportiva hay cuatro instalaciones: PistaTenis, SalaMusculacion, CampoGolf y Piscina.

- El funcionamiento de la PistaTenis tiene las siguientes restricciones:

  - Sólo puede haber jugando dos deportistas del mismo sexo.
  - Si la pista está vacía, entra la primera persona que llegue a la instalación y espera dentro del complejo a que llegue otra de su mismo sexo para empezar a jugar.
  - El partido dura entre 0,5 y 1 segundo.

    - Mientras se está jugando todos los deportistas que quieran jugar esperarán en la cola su turno por orden de llegada.

- La instalación SalaMusculación hay 15 aparatos, 9 de ellos para hombres y 6 de ellos para mujeres. Una vez alcanzado el máximo de su capacidad los deportistas de ambos sexos tendrán que esperar en una cola única su turno. Cada deportista estará en el aparato entre 0,8 y 1,5 segundos. Cuando termina deja sitio a otro.
- La instalación CampoGolf consta de cuatro hoyos que se juegan de manera individual. Si un deportistas encuentra en hoyo número uno vacío empieza a jugar por hoyo entre 0,4 y 0,6 segundos estando el resto de deportistas que solicitan en la cola hasta que quede de nuevo el hoyo uno vacío y comience a jugar otro Hilo. No pueden jugar, repetimos dos deportistas el mismo hoyo, por lo que si termina su ciclo y el siguiente está ocupado tendrá que esperar.
- La instalación Piscina es la primera que visitan los deportistas durante 5 segundos, dónde no hay limitaciones de capacidad.
- Cada deportista deberá usar todas las instalaciones una vez antes de terminar su vida útil en la aplicación.
- Al terminar el deportistas imprimirá un mensaje en consola indicando que ha terminado y su identificador.

1.2.2. Respecto al Módulo de Control

La apliación tendrá un módulo de control que se conectará al servidor (CiudadDeportiva) por la tecnología RMI. Tendrá las siguientes restricciones:

- Se implementará un método jugandoTenis que permita visualizar cuales jugadores están disputando en ese momento la partida de Tenis.
- Tendrá igualmente un método colaMusculacion que nos devolverá la lista de deportistas que están esperando en la cola de la SalaMusculacion.
- Implementará el método cerrarSalaMusuclacion que hará el cierre de la propia sala, con ello, haciendo que los hilos que se encuentran todavía en ella terminen su ciclo y el resto que intenta acceder espere en la cola hasta que se abrá de nuevo.
- Contendrá el métdo abrirSalaMusculacion para reanudad la actividad de la SalaMusculacion.

- El módulo de Control podrá ejecutarse desde una computadora diferente a la que se ejecuta el servidor. Ofercerá una interfaz gráfica con campos de texto y botones además de tres botones para operar con la SalaMusculacion además de actualizar los datos (refrescarDatos).

1.3. Respecto a la propia implementación

1.3.1. Tecnología empleada

Para diseñar el simulador se ha utilizado el Lenguaje de Programación Java. Concretamente sus tecnología en concurrencia y RMI. A continuación haremos un pequeño resumen descriptivo y bastante teórico sobre las mismas:

Concurrencia

"Los Hilos de ejecución de procesos son sin lugar a duda un hito en la computación, la posibilidad de compartir un espacio de direcciones en común, de proceso, y a la vez, ser unidades funcionales independientes dota a los sistemas distribuidos de gran versatilidad a la par de complejidad en el diseño, ya que como hemos dicho (repetimos), el hilo comparte, el espacio de memoria con otros hilos, asignados al proceso..." [Pum10].

Definición 1.3.1. Hacemos un distinción entre Hilo y Tarea. Un hilo puede se un conjunto de uno o varios Tareas. Puede ser con esto diseñado para una única función o desempeñar varias a los largo de su vida útilo de ejecución.

Definición 1.3.2. Para crear un Hilo en Java se necesita o bien de implementar la interfaz Runnable o bien extender la clase Thread.

Corolario 1.3.3. *La ventaja de utilizar la interfaz Runnable radica en su flexibilidad en la implementación del propio Hilo.*

Corolario 1.3.4. *Java dota al programador de una serie de útiles (métodos) para controlar el estado del Hilo.*

Comunicación entre Hilos

Definición 1.3.5. En un entorno de múltiples Hilos se emplean unas técnicas específicas para la comunicación.

Corolario 1.3.6. *Un ejemplo de estas son el algoritmo Productor/Consumidor.*

Corolario 1.3.7. *El repertorio del que dota Java al programador para el control del estado se basa en:*

$$\Sigma = \{wait(), notify()\} \tag{1.1}$$

Corolario 1.3.8. *Esta técnica consta de tres elementos principales [Qui05]:*

- *Productor: Generador de los elementos a controlar.*
- *Consumidor: Reductor de los elementos a controlar.*
- *Monitor: Control de los elementos desde el Productor hasta el Consumidor.*

RMI

Definición 1.3.9. La tecnología RMI de Java permite comunicar objetos de manera distribuida, es decir, estos se pueden ejecutar en máquinas distintas.

Corolario 1.3.10. *Se basa en una arquitectura de Cliente/Servidor.*

Corolario 1.3.11. *Para ello en necesario un registrador de objetos.*

Corolario 1.3.12. *La serialización es una técnica que permite que los objetos pueden ser intercambiados entre equipos sin perder la consistencia de los mismos.*

1.4. Elementos principales

1.5. La implementación de la Lógica Concurrente

A continuación haremos un desglose textual de la implementación que contiene cada una de las clases resaltando el código eje motor de la clase.

1.5.1. Clase PlantillaDeportista.java

```
import java.io.*;

public class PlantillaDeportista implements Serializable {

    private int identificador;
```

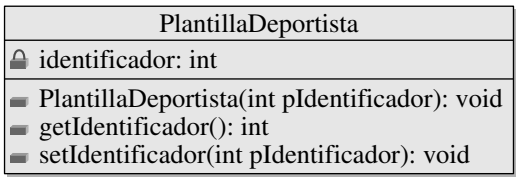

Figura 1.1: Representación UML de la clase PlantillaDeportista.java.

```
    PlantillaDeportista(int pIdentificador) {

        identificador = pIdentificador;
    }

    public int getIdentificador() {
        return identificador;
    }

    public void setIdentificador(int pIdentificador) {
        identificador = pIdentificador;
    }
}
```

Se trata la abstracción de un deportista. Únicamente nos interesa la variable identificador y su métodos de acceso. Esta, implementa la serialización ya que un conjunto de ellos serán transmitidos por RMI.

1.5.2. Clase Deportista.java

Se trata de un Hilo ya que deriva de la clase *Thread*.
Contiene las siguientes variables de sincronización:

```
private int varIdentificador;

private boolean estadoDetenido;
```

Que dan la funcionalidad, en el caso de identificador de contener el índice del Hilo. Y en el caso de estadoDetenido, se hace necesario para detener nuestro Hilos como monitor, como veremos en el método *pararEjecución()*.

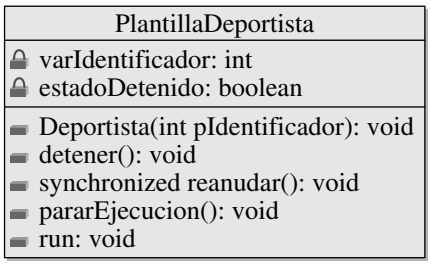

Figura 1.2: Representación UML de la clase Deportista.java.

El control de estado se hace para activar[1], por medio de:

```
public synchronized void reanudar() {
        estadoDetenido = false;
        notify();
    }
```

y para detener por medio de:

```
    public void pararEjecucion() {
        try {
            synchronized (this) {
                while (estadoDetenido == true) {
                    wait();
                }
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
```

Ambos métodos acceden a la variable booleana *estadoDetenido*. Lo hacen de manera sincronizada y emplean, en el caso de ser activado el control de *notify()* para monito y *wait()* para detener.

El método *run()* implementa un monitor para cada una de las instalaciones. Comienza creando un nuevo Hilo-Deportista:

[1]Dicho método es ejecutado por el botón de REANUDAR Hilos.

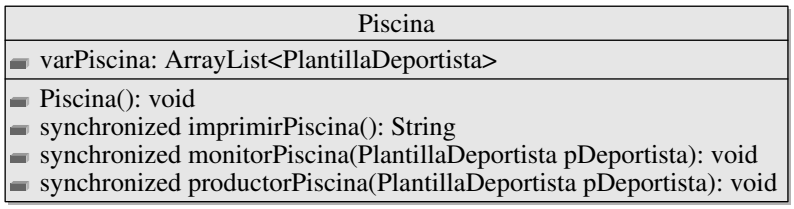

Figura 1.3: Representación UML de la clase Piscina.java.

```
PlantillaDeportista varPDeportista = new PlantillaDeportista(varIdentificador);
```

Para depués hacer que este haga la instalación piscina y mediante un *switch* y hasta que no haya pasado por todas y cada una de las instalaciones (cada elemento del switch con el monitor correspondiente), no termine:

```
//Finaliza el Hilo
        CiudadDeportiva.JTextFinal.setText("<" +
String.valueOf(varIdentificador) + ">");
        System.out.println("Finaliza el Deportista con identificador: " +
varIdentificador);
```

Nota: De modo genérico cada monitor se encarga de que secuencialmente haga cada una de las actividades con sus etapas controlando el flujo con Thread con pararEjecucion() en el monitor principal. Siendo el monitor de cada aplicación el que reanude su actividad.

1.5.3. Clase Piscina.java

Contiene un *ArrayList*:

```
private ArrayList<PlantillaDeportista> varPiscina;
```

dónde se almacenan los deportistas y, los siguientes métodos:

```
public synchronized String imprimirPiscina() {
//EL CODIGO SE ENCARGA DE MOSTRAR LOS DEPORTISTAS EN LA INSTALACION
}

//MONITOR PISCINA
public synchronized void monitorPiscina(PlantillaDeportista pDeportista) {
```

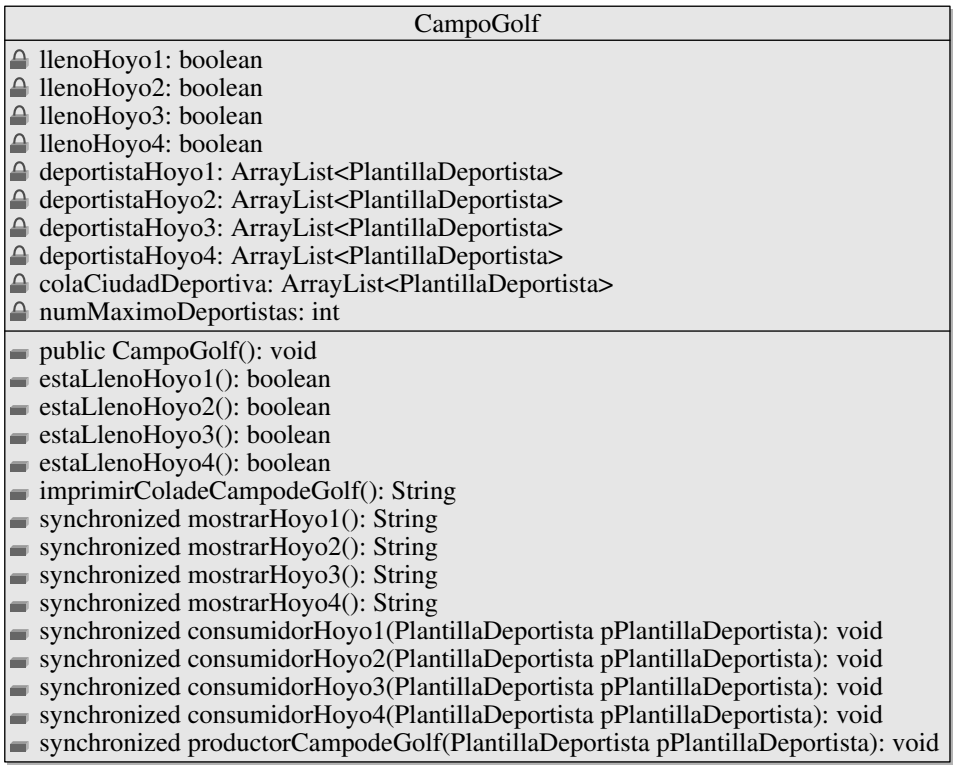

Figura 1.4: Representación UML de la clase CampoGolf.java.

```
   varPiscina.add(pDeportista);
}

//PRODUCTOR PISCINA
public synchronized void productorPiscina(PlantillaDeportista pDeportista) {
   varPiscina.remove(pDeportista);
}
```

El CONSUMIDOR está implementado de manera lógica en el *switch* antes citado de la clase Deportista.

1.5.4. Clase CampoGolf.java

Contiene las siguiente variables:

```
//Variables de sincronizacion para exclusi\'on mutua
private boolean llenoHoyo1;
private boolean llenoHoyo2;
private boolean llenoHoyo3;
private boolean llenoHoyo4;
//Almacenes para cada hoyo
private ArrayList<PlantillaDeportista> deportistaHoyo1;
private ArrayList<PlantillaDeportista> deportistaHoyo2;
private ArrayList<PlantillaDeportista> deportistaHoyo3;
private ArrayList<PlantillaDeportista> deportistaHoyo4;
//Alamacen para intalacion
private ArrayList<PlantillaDeportista> colaCiudadDeportiva;
//Numero total de deportistas por hoyo
private int numMaximoDeportistas = 1;
```

dónde:

- $llenoHoyo\lambda \Longrightarrow$ una variable candado para cada Hoyo.
- $deportistaHoyo\lambda \Longrightarrow$ un ArrayList por cada Hoyo que contiene a los deportistas.
- $colaCiudadDeportiva\lambda \Longrightarrow$ un ArrayList con los deportistas en espera.
- $numMaximoDeportistas \Longrightarrow$ el número máximo de deportistas por Hoyo[2].

Hay un método "cerrojo" por cada Hoyo, $estaLlenoHoyo\lambda()$. Y un conjunto de métodos para controlar la "salida por pantalla".

Por otra parte hay un CONSUMIDOR por cada Hoyo (Mostramos para el Hoyo 1):

```
public synchronized void consumidorHoyo1(PlantillaDeportista
pPlantillaDeportista) {
        //A\~{n}ade por defecto para espera
        colaCiudadDeportiva.add(pPlantillaDeportista);

CiudadDeportiva.JTextColaCampodeGolf.setText(imprimirColadeCampodeGolf());

        while (estaLlenoHoyo1() == true) {
            try {
```

[2]Esto está directamente implicado con el tamaño de $deportistaHoyo\lambda$.

```
                //Espera
                wait();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        //Activa
        notifyAll();

        //Inserta deportista en el hoyo
        colaCiudadDeportiva.remove(pPlantillaDeportista);
        deportistaHoyo1.add(pPlantillaDeportista);

        //Muestra texto

CiudadDeportiva.JTextColaCampodeGolf.setText(imprimirColadeCampodeGolf());
        CiudadDeportiva.JTextHoyo1.setText(mostrarHoyo1());
        try {
            //Descansa
            sleep((((long) (Math.random() * (200)) + 400)));
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

    }
```

Y un productor:

```
    public synchronized void productorCampodeGolf(PlantillaDeportista
pPlantillaDeportista) {
        deportistaHoyo4.remove(pPlantillaDeportista);
        CiudadDeportiva.JTextHoyo4.setText(mostrarHoyo4());
        notifyAll();
    }
```

Que devuelve al flujo principal el deportista.

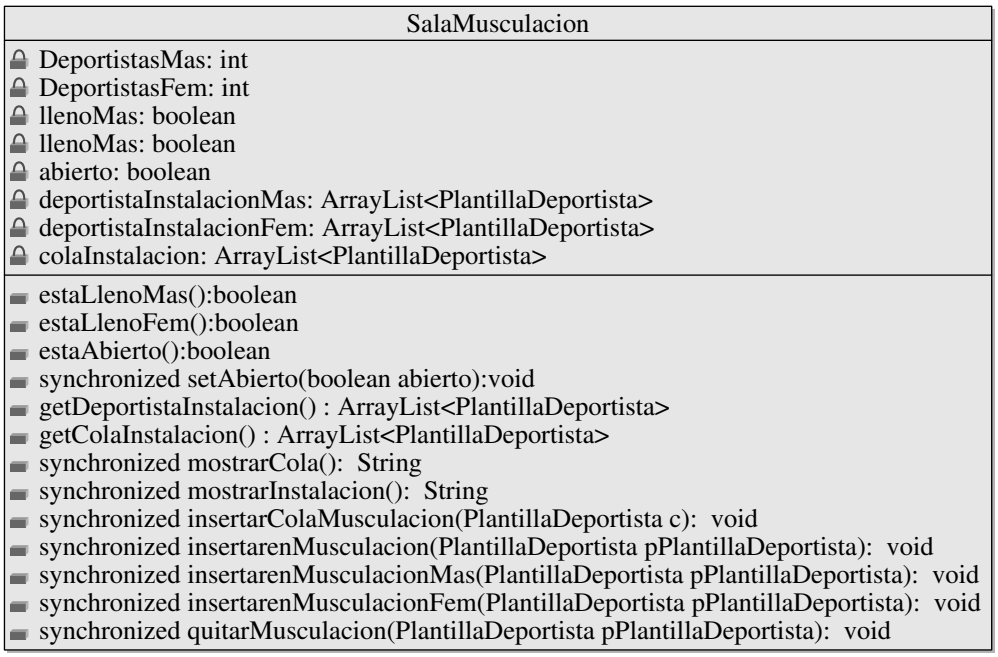

Figura 1.5: Representación UML de la clase SalaMusculacion.java.

1.5.5. Clase SalaMusculacion.java

El código que presenta está clase a nivel estructural se parece mucho a la antes citada CampoGolf ya que todo nuestro trabajo de concurrencia ha sido tratado mediante monitores. Por ello tenemos las variables:

```
private int DeportistasMas = 9;
private int DeportistasFem = 6;
private boolean llenoMas;
private boolean llenoFem;
private boolean abierto;
private ArrayList<PlantillaDeportista> deportistaInstalacionMas;
private ArrayList<PlantillaDeportista> deportistaInstalacionFem;
private ArrayList<PlantillaDeportista> colaInstalacion;
```

de "cerrojo" y Lista[3] para los deportistas.

Por otra parte tenemos l,os métdos para cada candado y la lógica de RMI:

```
public boolean estaAbierto() {
        return abierto;
    }

    /** M\'etodo que abre o cierra la Sala de Musculacion */
    public synchronized void setAbierto(boolean abierto) {
        if (abierto == true) {
            if ((this.abierto == false) && (deportistaInstalacionMas.size() ==
DeportistasMas) && (deportistaInstalacionMas.size() == DeportistasMas)) {
                System.out.println("Deportistas terminando el ejercicio...");
            } else {
                CiudadDeportiva.jLabelEstadoSala.setText("Estado de la Sala:
Abierta");
                this.abierto = abierto;
                notifyAll();
            }
        } else {
            CiudadDeportiva.jLabelEstadoSala.setText("Estado de la Sala:
Cerrada");
            this.abierto = abierto;
```

[3]La variable *abierto* es empleada para RMI.

```
            if (colaInstalacion.size() > 0) {
                llenoMas = false;
                llenoFem = false;
                notifyAll();
            }
        }
    }
```

Más lógica para "salida de datos". Y por último nuestros CONSUMIDORES que se diferencian por sexo y son llamados por:

```
public synchronized void insertarenMusculacion(PlantillaDeportista
pPlantillaDeportista) {
        if (pPlantillaDeportista.getIdentificador() % 2 == 0) {
            insertarenMusculacionMas(pPlantillaDeportista);
        } else {
            insertarenMusculacionFem(pPlantillaDeportista);
        }
    }
```

De modo genérico mostramos el consumidor de deportistas masculinos:

```
public synchronized void insertarenMusculacionMas(PlantillaDeportista
pPlantillaDeportista) {
        while ((llenoMas == true) || (abierto == false)) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        notify();
        colaInstalacion.remove(pPlantillaDeportista);
        deportistaInstalacionMas.add(pPlantillaDeportista);
        llenoMas = deportistaInstalacionMas.size() == DeportistasMas;
        CiudadDeportiva.JTextColaSaladeMusculacion.setText(mostrarCola());
        CiudadDeportiva.jTextSaladeMusculacion.setText(mostrarInstalacion());
    }
```

Figura 1.6: Representación UML de la clase CiudadDeportiva.java.

que comprueba los cerrojos, sino están disponibles espera y por último libera recurso. Y finalmente el productor:

```
public synchronized void quitarMusculacion(PlantillaDeportista
pPlantillaDeportista) {

        if (pPlantillaDeportista.getIdentificador() % 2 == 0) {
            deportistaInstalacionMas.remove(pPlantillaDeportista);
            llenoMas = false;
        } else {
            deportistaInstalacionFem.remove(pPlantillaDeportista);
            llenoFem = false;

        }
        CiudadDeportiva.jTextSaladeMusculacion.setText(mostrarInstalacion());
        notify();

    }
```

que tiene la función de devolver el deportista al flujo principal.

1.5.6. Clase PistaTenis.java

1.5.7. Clase CiudadDeportiva.java

La clase contiene:

```
static ArrayList<Deportista> arrayHilos;
```

Al que se asocian todos los deportistas (Hilos) para poder ser controlados a modo de conjunto.

Es por ello que esta clase incorporta el código para la interfaz gráfica (SWING) además de los botones de paro y reanudar de Hilos:

```
  //Bonton para detener procesos
    private void jButtonDetenerActionPerformed(java.awt.event.ActionEvent evt) {

        for (int cont = 0; cont < 49; cont++) {
            arrayHilos.get(cont).detener();
        }
    }

    //Boton para reanudar procesos
    private void jButtonReanudarActionPerformed(java.awt.event.ActionEvent evt)
{
        for (int cont = 0; cont < 49; cont++) {
            arrayHilos.get(cont).reanudar();
        }
```

Que como vemos recorren el conjunto de Hilos para ejercer el control, uno por uno, y:

```
 public void run() {
                new CiudadDeportiva().setVisible(true);
                arrayHilos = new ArrayList();
                piscina = new Piscina();
                campodeGolf = new CampoGolf();
                pistadeTenis = new PistaTenis();
                saladeMusculacion = new SalaMusculacion();

                //Iniciamos hilos
                for (int cont = 51; cont <= 99; cont++) {
                    Deportista c = new Deportista(cont);
                    arrayHilos.add(c);
                }
                //Ejecutamos hilos
                for (int cont = 0; cont < 49; cont++) {
```

```
                arrayHilos.get(cont).start();
            }
```

Para inicializar clases e Hilos como, por último, el *Naming.rebind* para RMI[4]:

```
//Iniciamos el Objeto Remoto en local
            try {
                CiudadRMI datosCiudad = new CiudadRMI();
                Naming.rebind("//127.0.0.1/ObjetoRMI", datosCiudad);
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
```

1.6. La implementación RMI

1.6.1. Apliación de la tecnología RMI en nuestro simulador

Definición 1.6.1. La tecnología RMI permite que objetos de distinta localización física puedan comunicarse.

Corolario 1.6.2. *Por ello se ha defenido la tecnología y un método de empleo.*

El método Java RMI (Algoritmo)

1. Se ha de definir una interfaz donde se declaran los métodos remotos.

2. Se ha de implementar una clase con la interfaz dónde se definen los métodos remotos.

3. El programa servidor genera los objetos remotos y los ofrece a través de RMI, por medio de la directiva: *Naming.rebind("dirección IP/Nombre",ObjetoRemoto);*.

4. Por último el programa cliente invoca el método remoto: *Naming.lookup(/dirección IP/Nombre");*.

[4]En este caso con la dirección, *localhost*.

Ejecución en entorno Linux

Para una misma computadora:

1. Se asigna la dirección local tanto a servidor como cliente.
2. Como Root se lanza:

   ```
   # rmiregistry
   ```

3. Se lanza el servidor.

   ```
   $ java (desde build) path hacia paquete .jar metapaquete/servidor o cliente
   ```

4. Se lanza igualmente el cliente.

Para distintas computadoras:
Se siguen los pasos anteriores salvo en el punto 1 para

Ejecución en entorno Windows

Fuente: [Mar10].
Si ejecutamos la aplicación en un solo ordenador debemos realizar los siguientes pasos:

1. Compilamos todas las clases y abrimos una ventana de consola.
2. Nos situamos en el directorio: path hacia paquete .jar (Donde NetBeans ha compilado la interfaz y las clases)
3. Con el classpath apuntando al directorio de trabajo (> set classpath=.) poner en marcha el rmiregistry (>start rmiregistry).
4. Ejecutamos el programa servidor (> start java saluda/Servidor).
5. Ejecutar el programa cliente (> metapaquete/cliente).

(Para cerrar el rmiregistry se debe pulsar CTRL+c. o cerrar la ventana de símbolo del sistema)
Si ejecutamos la aplicación en dos ordenadores debemos realizar los siguientes pasos:
Preparación del servidor:

1. Compilamos todas las clases y abrimos una ventana de consola.

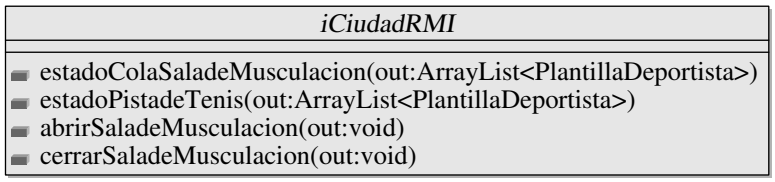

Figura 1.7: Representación UML de la interfaz iCiudadRMI.java.

2. Nos situamos en directorio: path hacia paquete .jar.

3. Con el classpath apuntando al directorio de trabajo (> set classpath=.) poner en marcha el rmiregistry (> start rmiregistry).

4. Ejecutar el servidor (> start java metapaquete/servidor).

Preparación del cliente:

1. Compilamos todas las clases y abrimos una ventana de consola.

2. Nos situamos en el directorio: path hacia paquete .jar.

3. Poner el classpath apuntando al directorio de trabajo (> set classpath=.).

4. Ejecutar el cliente (> java metapaquete/cliente).

Nota: Hay que tener en cuenta que debemos modificar el código del programa Cliente con la dirección IP del ordenador donde se ejecuta el programa Servidor.

1.6.2. Interfaz iCiudadRMI.java

Dicha interfaz implementa la lógica de prototipado para métdos remotos. Es en síntesis una abstracción para hacer más geneérica nuestra implementación:

```
public interface iCiudadRMI extends Remote {

    //Prototipos de metodos para RMI
    ArrayList<PlantillaDeportista> estadoColaSaladeMusculacion() throws
RemoteException;

    ArrayList<PlantillaDeportista> estadoPistadeTenis() throws RemoteException;
```

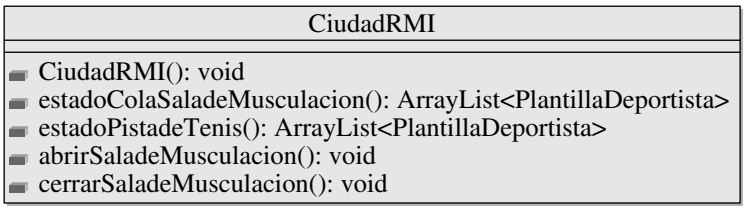

Figura 1.8: Representación UML de la clase CiudadRMI.java.

```
    void abrirSaladeMusculacion() throws RemoteException;

    void cerrarSaladeMusculacion() throws RemoteException;
}
```

1.6.3. Clase CiudadRMI.java

Implementa la interfaz antes citada y llama por cada método a la acción dentro del código de servidor, la ejecución del programa para ser enviada como flujo de bits.

```
public class CiudadRMI extends UnicastRemoteObject implements InterfazCiudadRMI
{

    //Constructor
    public CiudadRMI() throws RemoteException {
    }

    //Implementacion de metodos para RMI
    public ArrayList<PlantillaDeportista> estadoColaSaladeMusculacion() throws
RemoteException {
        return CiudadDeportiva.saladeMusculacion.getColaInstalacion();
    }

    public ArrayList<PlantillaDeportista> estadoPistadeTenis() throws
RemoteException {
        return CiudadDeportiva.pistadeTenis.getDeportistaInstalacion();
    }
```

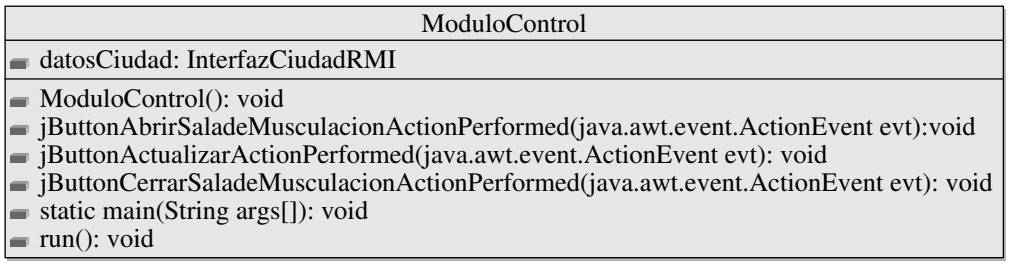

Figura 1.9: Representación UML de la clase ModuloControl.java.

```
    public void abrirSaladeMusculacion() throws RemoteException {
        CiudadDeportiva.saladeMusculacion.setAbierto(true);

    }

    public void cerrarSaladeMusculacion() throws RemoteException {
        CiudadDeportiva.saladeMusculacion.setAbierto(false);
    }
}
```

1.6.4. Clase ModuloControl.java

Contiene tres botones y su lógica para RMI: Abrir Sala de Musculación, Cerrar Sala de Musculación y Refrescar con sus correspondientes TextBox.

Por otra parte en el método principal (main) se hace la llamada al objeto remoto:

```
datosCiudad=(InterfazCiudadRMI)Naming.lookup("//127.0.0.1/ObjetoRMI");
```

para ser empleado en cada uno de los botones del módulo, como hemos dicho:

```
private void
jButtonAbrirSaladeMusculacionActionPerformed(java.awt.event.ActionEvent evt) {

        try {
            datosCiudad.abrirSaladeMusculacion();
        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }
```

```
    }

    private void jButtonActualizarActionPerformed(java.awt.event.ActionEvent
evt) {
        //Recibe el estado de la cola de musculacion y de la pista de tenis y
muestra todo por pantalla
        try {
            ArrayList<PlantillaDeportista> cola =
datosCiudad.estadoColaSaladeMusculacion();
            ArrayList<PlantillaDeportista> local =
datosCiudad.estadoPistadeTenis();
            String muestra = "";


            for (int cont = 0; cont
                    < cola.size(); cont++) {
                muestra = muestra + "Identificador: " +
String.valueOf(cola.get(cont).getIdentificador());
            }
            jTextSaladeMusculacion.setText(muestra);
            muestra =
                    "";


            for (int cont = 0; cont
                    < local.size(); cont++) {
                muestra = muestra + "Identificador: " +
String.valueOf(local.get(cont).getIdentificador());
            }
            jTextTenis.setText(muestra);


        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }

    }
```

```
    private void
jButtonCerrarSaladeMusculacionActionPerformed(java.awt.event.ActionEvent evt) {

        try {
            datosCiudad.cerrarSaladeMusculacion();


        } catch (RemoteException e) {
            System.out.println(e.getMessage());
        }
    }
    }
```

1.7. Análisis UML

1.8. Diagrama de Casos de Uso

Figura 1.10: Diagrama de Casos de Uso para Ciudad Deportiva.

Capítulo 2

Manual de Usuario

2.1. Cuadro de Mando General (CMG)

Diego's Software & Systems se complace en presentarle el simulador de una Ciudad Deportiva. A grandes rasgos podrá observar un comportamiento aleatorio en cada simulación con distintas opciones que da a la aplicación en si, gran versatilidad y realismo para probar este tipo de entornos.

El Cuadro de Mando General se muestra en la Figura 2.1. Diremos que tiene como función principal el proporcionar un total control de la ejecución del programa. Podrá observar dónde se encuentra cada doportista y como cumple sus objetivos, desde que acude a la Piscina hasta que finaliza su ciclo.

Figura 2.1: Cuadro de Mando General (CMG).

Figura 2.2: Vista detallada del botón Reanudar del CMG.

Figura 2.3: Vista detallada del botón Detener del CMG.

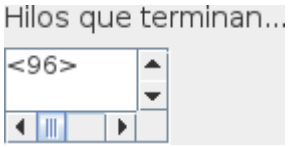

Figura 2.4: Vista detallada la finalización de la actividad de un deportista.

Apéndice A

El sistema de Tiempo Aleatorio

En este anexo haremos un inciso sobre el sistema que empleamos para determinar un tiempo aleatorio en Java.

Cuando es necesario que un determinado Hilo espere un tiempo invocamos al métdo *sleep(límite de tiempo)*.

En está práctica hemos utilizado un métdo alatorio de *random()* para la clase *Math*. La ecuación como se verá a continuación, sirve de ejemplo para determinar un tiempo aleatorio entre 800 y 1500 milisegundos:

$$RANDOM * (LIMITE - BASE) + BASE \tag{A.1}$$

Por lo que si *random()* es 0 dará la base y siendo uno será: 800 + 700 = 1500, que es el tiempo límite máximo.

Ahora mostramos un ejemplo de la utilidad:

```
try {
    sleep(((long) (Math.random() * (1500 - 800)) + 800));
    } catch (InterruptedException e) {
System.out.println(e.getMessage());
      }
```

Bibliografía

[Mar10] Luis Bengochea Martínez. *Apuntes para la asignatura: Laboratorio de Programación Avanzada.* UAH, 2010.

[Pum10] Diego Antonio Lucena Pumar. *Teoría de Sistemas Operativos Distribuidos.* Lulu, 2010.

[Qui05] Agustín Froufe Quintas. *Java 2: Manual de usuario y tutorial.* Ra-Ma, 2005.