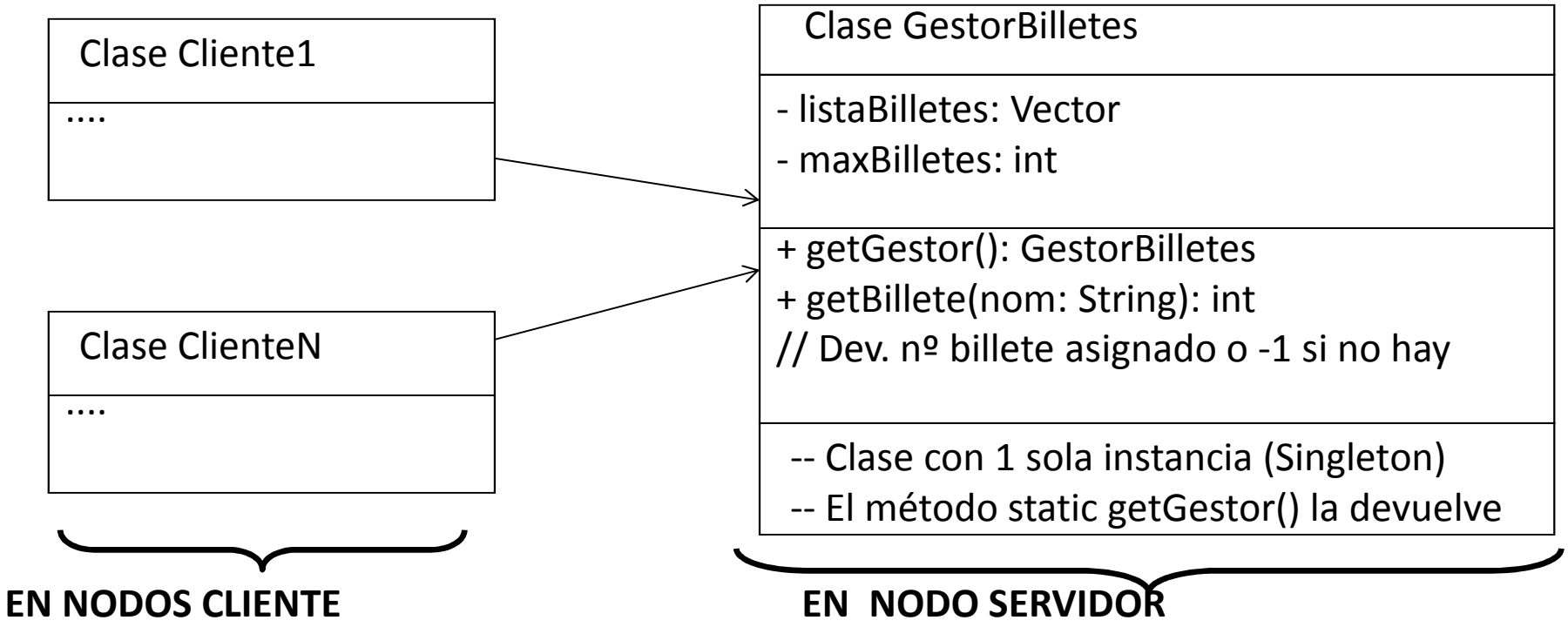
SI OO distribuidos

- RMI: Introducción
- Interfaz Remota
- Servidor Remoto
- Cliente
- Arquitectura RMI
- Evolución del Sistema
- SOAP



En una arquitectura distribuida no se puede hacer lo siguiente:

```
GestorBilletes g = GestorBilletes.getGestor();
return g.getBillete("Kepa Sola");
```

Ya que GestorBilletes es un objeto de la máquina virtual remota

Java Remote Method Invocation (RMI)

- API que proporciona Java (java.rmi)
- Conjunto de interfaces, clases y métodos que permiten desarrollar en Java aplicaciones distribuidas de forma sencilla
- Conserva la sintaxis y la semántica del modelo de objetos Java no distribuidos
- Permite la conexión con BD relacionales utilizando JDBC

- http://java.sun.com/docs/books/tutorial/rmi/overview.html
- http://www.chuidiang.com/java/rmi/rmi.php

Java Remote Method Invocation (RMI)

- Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente.
- Una servidor típico crea objetos remotos, hace accesibles las referencias a esos objetos y espera a que los clientes invoquen métodos a los objetos remotos
- Un cliente típico obtiene las referencias remotas a uno o más objetos remotos en el servidor e invoca métodos sobre ellos
- RMI proporciona el mecanismo por el que se comunican el cliente y el servidor para pasarse información.
- Este tipo de aplicaciones se denominan aplicaciones de objetos distribuidos

Ventajas de RMI

- Orientación a objetos: RMI permite pasar objetos como parámetros y tipos de retorno (no solamente tipos predefinidos)
- Seguridad: RMI usa los mecanismos de seguridad de Java
- Portabilidad: a otras JVM...
- Garbage Collection distribuida
- Paralelismo: el servidor RMI es multi-thread

Modelo de objetos distribuidos

- Los métodos de un objeto remoto pueden ser invocados por una máquina distinta a aquella que contiene el objeto

- Un objeto remoto se describe con una o más interfaces remotas. Estas interfaces se escriben en Java y declaran los métodos del Objeto Remoto

- Remote Method Invocation (RMI) es la acción de invocar un método de una interfaz remota en un objeto remoto

- La sintaxis de llamada a un objeto remoto es exactamente la misma que a un objeto local

Modelo de objetos distribuidos

- Para construir una aplicación cliente/servidor donde un cliente accede a un servicio remoto (proporcionada por una clase remota) usando RMI debemos:
  - Construir/Definir una interfaz remota
  - Implementar dicha interfaz remota (servidor RMI)
  - Implementar el cliente RMI que accede al servicio remoto

Modelo de objetos distribuidos

- Los clientes de un Objeto Remoto interaccionan con interfaces remotas (nunca con las clases de implementación de dichas interfaces)

- Los parámetros no-remotos y los resultados que devuelven las invocaciones a métodos remotos se pasan por copia (en el modelo no distribuido por referencia) utilizando la serialización de objetos (las clases tienen que implementar la interfaz *Serializable*)

- Un objeto remoto se pasa por referencia

- Los clientes de Objetos remotos deben tratar excepciones adicionales que pueden producirse debido a RMI

Localización de objetos remotos

- El sistema RMI ofrece un registro de nombres simple (rmiregistry) con el que pueden obtenerse las referencias a objetos remotos

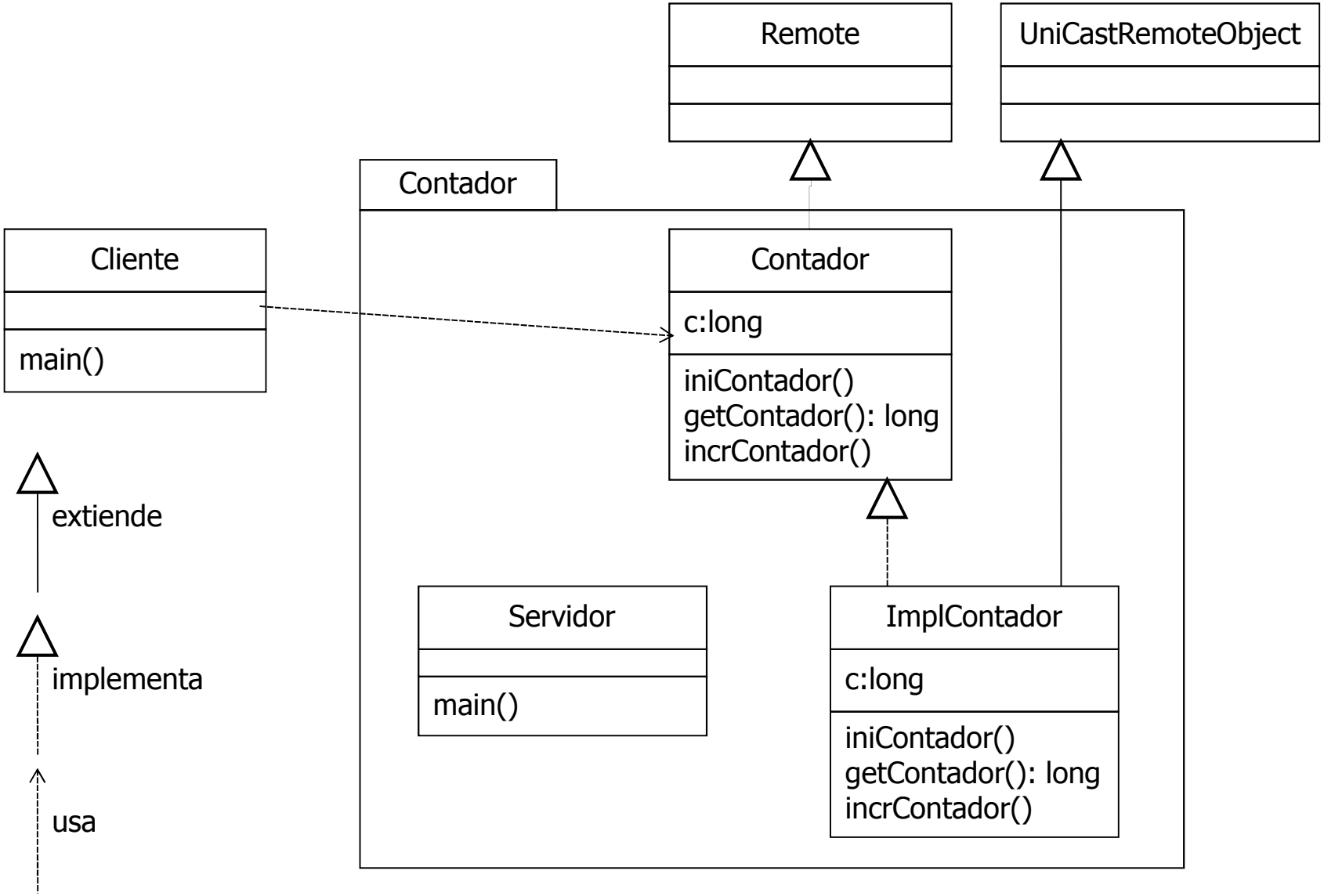
- El nombre de un objeto remoto es parecido a un URL (Uniform Resource Locator)

- //host[:port]/nombre
- El host es del registro de nombres
- port donde el registro espera recibir las invocaciones
- Nombre es el string que identifica el objeto dentro del registro

- El acceso al registro se realiza con los métodos de la clase java.rmi.Naming

Ejemplo: Contador no distribuido

Ejemplo: Contador distribuido



Interfaz Remota

- Una interfaz remota declara un conjunto de métodos que pueden ser invocados desde una JVM remota

- Una interfaz remota debe extender (directa o indirectamente) la interfaz java.rmi.Remote

- Las definiciones de métodos en las interfaces remotas deben contemplar la excepción java.rmi.RemoteException

Interfaz remota

```
package Contador;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Contador extends Remote {

    void iniContador() throws RemoteException;
    long getContador() throws RemoteException;
    void incrContador() throws RemoteException;

}
```

Implementación con Objetos "transitorios"

- La clase java.rmi.server.UnicastRemoteObject permite crear y exportar objetos remotos cuyas referencias únicamente son válidas durante la vida del proceso servidor que crea el objeto remoto (objetos "transitorios")

```
package Contador;

import java.rmi.RmoteException;
import java.rmi.server.UnicastRemoteObject;

public class ImplContador
    extends UnicastRemoteObject implements Contador {
    ...
    }
```

Implementación ServidorContador

- Es necesario un proceso servidor que cree el objeto remote y publique su referencia en el registro de nombres de RMI (Naming.rebind)

```
package Contador;
import java.rmi.Naming;
import java.rmi.RmoteException;
import java.rmi.server.UnicastRemoteObject;

public class Servidor {
    public static void main(String args[]) {
    ...
    ImplContador c = new ImplContador();  // crea el objeto contador
    Naming.rebind("//157.147.20.15/MiContador", c); //pone MiContador en reg. de nombres
    System.out.println("Contador creado y registrado.");
    ...
    }
```

Implementación Cliente

- El cliente invocará los métodos del objeto remoto un vez localizado su referencia en el registro de nombres del RMI (Naming.lookup)

```
package Contador;
import java.rmi.Naming;
import java.rmi.RmoteException;

public class Cliente {
    public static void main(String args[]) {
    ...
    // Obtiene MiContador del registro de nombres
    Contador c = (Contador)Naming.lookup("//157.147.20.15/MiContador");
    c.iniContador();
    for(int i=0;i<1000;i++) { c.incrContador() };
    System.out.println("El valor del contador es:"+c.getContador());
    ...
    }
```

Arquitectura RMI

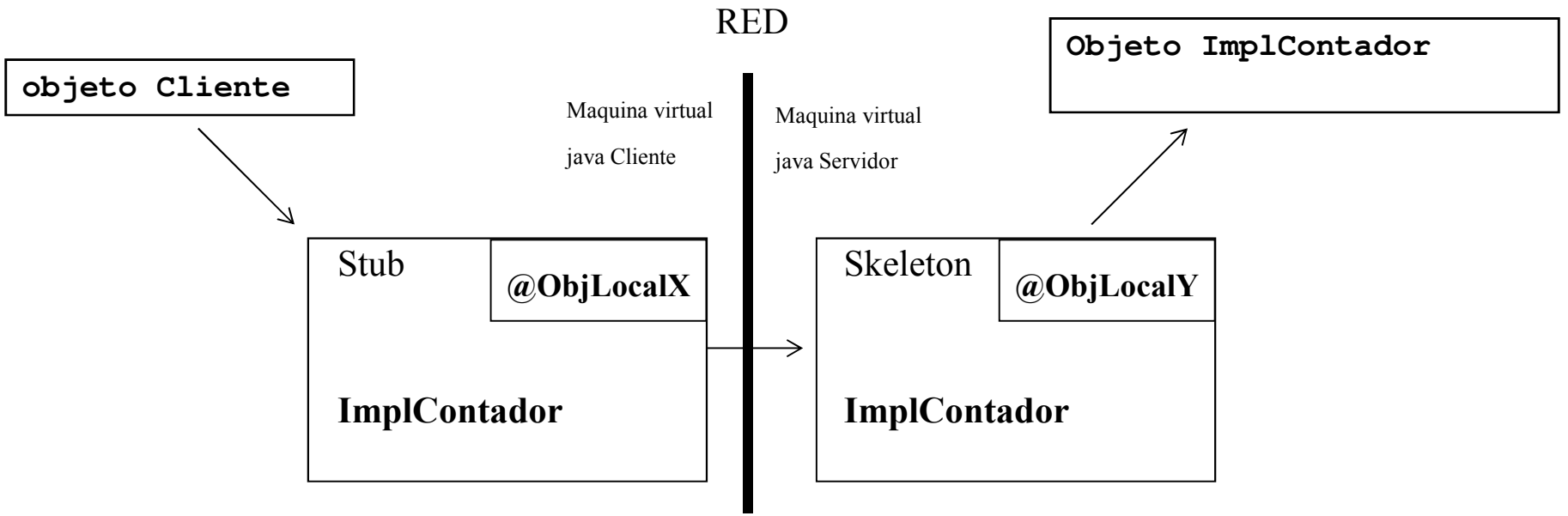
- Hay que conectar el objeto cliente con el objeto servidor para que las llamadas a métodos del primero sean ejecutadas por el segundo

- Hay que pasar los valores de los parámetros de los métodos del cliente al servidor

- Hay que pasar los resultados de los métodos del servidor al cliente

- Los objetos *Stub* (cliente) y *Skeleton* (servidor) se encargan de realizar la conexión y el paso de parámetros y resultados (desde JDK 1.2 la funcionalidad S*keleton* está en el servidor RMI)

Arquitectura RMI

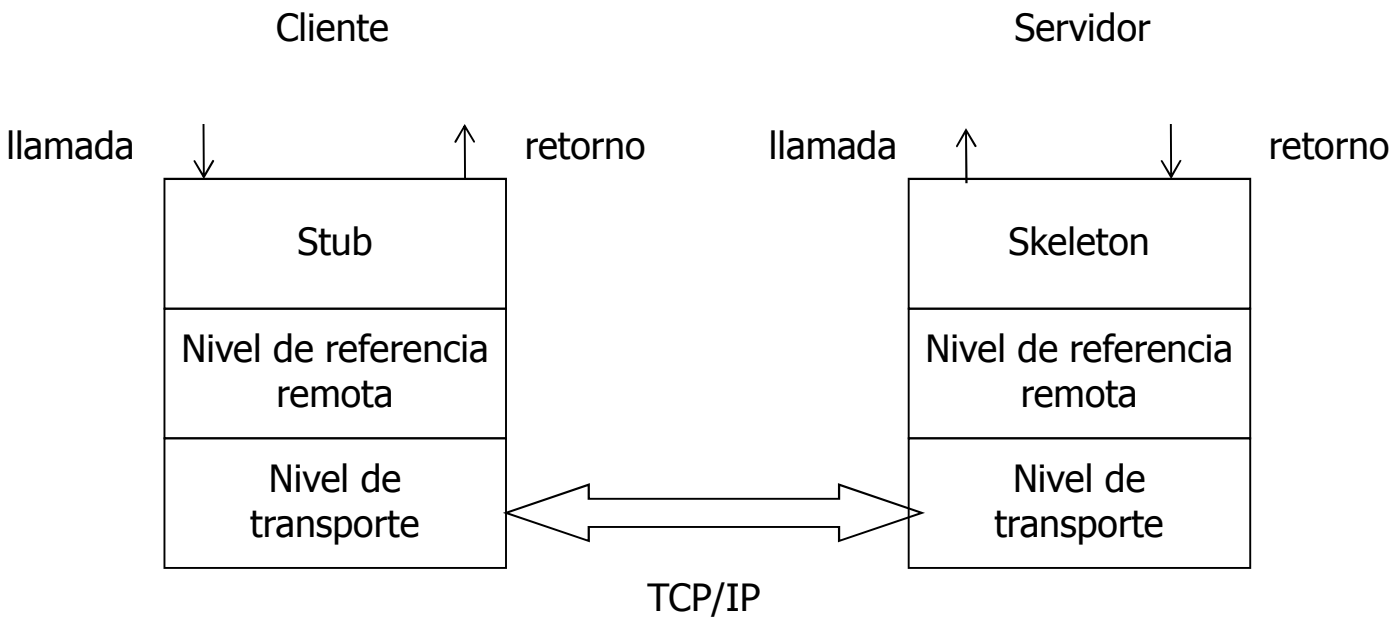


Stub ImplContador: Representante del objeto ImplContador en el cliente

Skeleton ImplContador: Representante del objeto ImplContador en el servidor

NOTA: Los objetos Stub, Skeleton y el objeto remoto comparten la misma interfaz

Arquitectura RMI



rmic: Generación de Stubs y Skeletons

- rmic: herramienta que se encarga de la generación de los stubs y skeletons a partir de la clase implementación

```
sistema> javac contador.java  (interfaz remota)
sistema> javac implContador.java  (clase remota)
sistema> javac servidor.java
sistema> rmic implContador
```

En tiempo de compilación se obtienen las clases:

implContador_Stub.class => debe quedar accesible al cliente!

implContador_Skeleton.class

rmiregistry

- Es un servidor de nombres que relaciona objetos con nombres
- Hay que lanzarlo como proceso independiente en la máquina servidor (en la misma máquina que el servidor RMI)
- rmiregistry [numPuerto]

- También se puede lanzar desde una aplicación Java (en el servidor RMI)
- Java.rmi.registry.LocateRegistry.createRegistry(p)

- Crea el preceso rmiregistry en el puerto p
- Rmiregistry no termina aunque acabe el servidor RMI
- Lanza una excepción si el puerto está ocupado

Política de seguridad

- Un programa java debe especificar un gestor de seguridad que determine su política de seguridad
- Algunas operaciones requieren que exista dicho gestor. En concreto, las RMI.
- RMI sólo cargará una clase serializable desde otra máquina si hay un gestor de seguridad que lo permita
- El gestor de seguridad por defecto de RMI sigue una política muy restrictiva (sólo permite ejecutar STUBs del CLASSPATH local)
- Para establecer un gestor de seguridad por defecto para RMI:
  - System.setSecurityManager(new RMISecurityManager());
- Para cambiar la política de seguridad, indicando otro fichero:
  - System.setProperty("java.security.policy","*pathficheropolicy*");

Política de seguridad

- También se puede indicar para cada ejecución cuál es la política de seguridad
- -Djava.security.policy=java.policy
- Para ello, el fichero java.policy hay que dejarlo en un directorio concreto:
  - Si la máquina virtual que se ejecuta es:
  - DIRECTORIO_JDK_O_JRE\bin\java.exe
  - El fichero java.policy tiene que estar en:

  DIRECTORIO_JDK_O_JRE\lib\security
- Así todas las aplicaciones lanzadas con esa máquina virtual, usarán dicha política de seguridad

Construcción aplicaciones RMI

Para construir una aplicación Cliente/Servidor donde un cliente acceda a un servicio remoto (clase remota) usando RMI hay que:

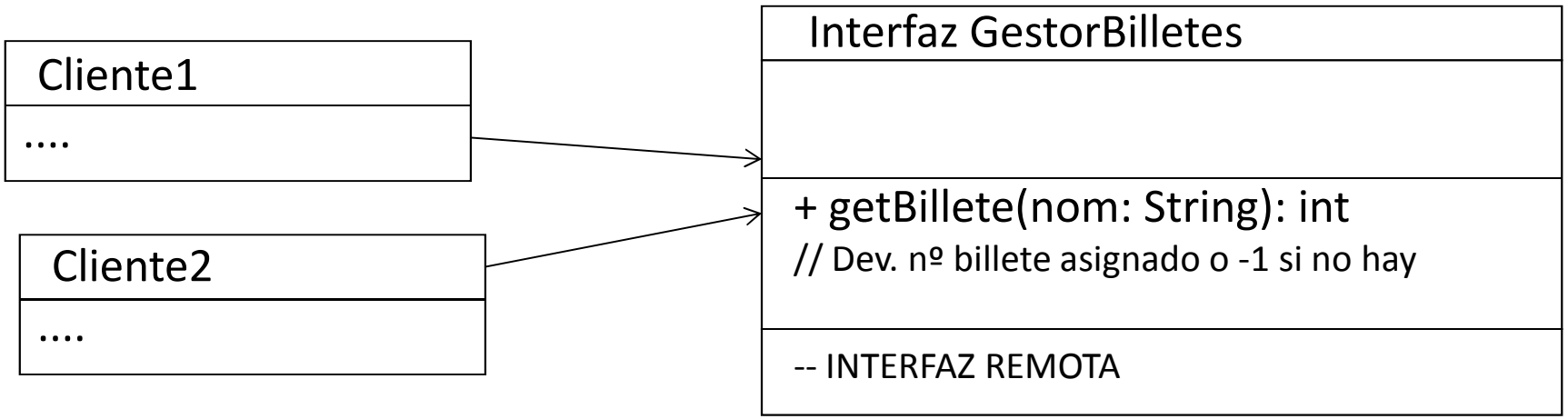
1.- Definir la interfaz remota

2.- Implementar la interfaz remota (clase remota)

3.- Registrar un objeto de la clase remota

4.- Localizar y ejecutar el objeto remoto

1. Definir la interfaz remota

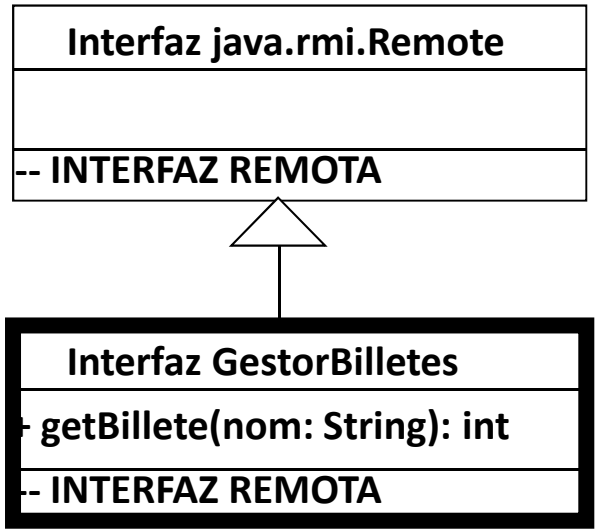


RMI permite invocar a un objeto remoto
Para ello hay que definir una interfaz remota
Así, un cliente puede hacer lo siguiente:

```
GestorBilletes g;
// Código para obtener la dirección del
// objeto remoto y dejarlo en g
return g.getBillete("Kepa Sola");
```

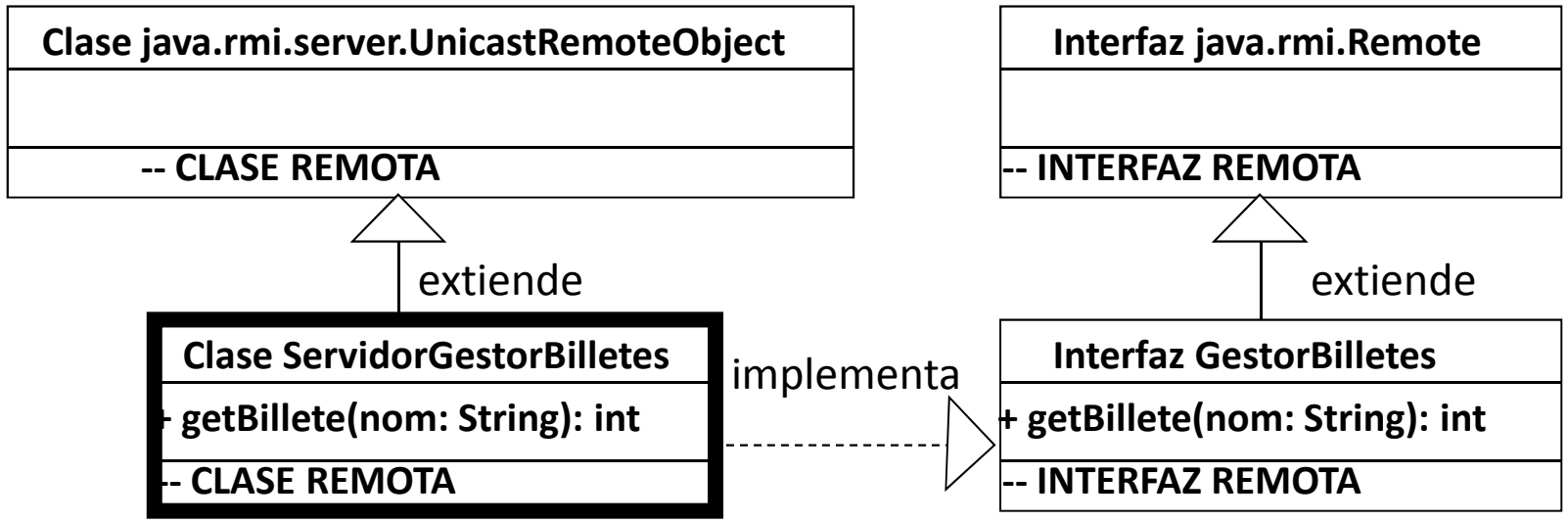
1. Definir la interfaz remota

```
// GestorBilletes.java
import java.rmi.*;
public interface GestorBilletes
        extends Remote
{
 public int getBillete(String nom)
        throws RemoteException;
}
```



La interfaz extiende **java.rmi.Remote**
Todos los métodos deben lanzar
java.rmi.RemoteException

2. Implementar la Interfaz Remota (La clase Remota)



El servidor implementa la interfaz remota
Extiende **`java.rmi.server.UnicastRemoteObject`**

2. Implementar la Interfaz Remota (La clase Remota)

```
// ServidorGestorBilletes.java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
public class ServidorGestorBilletes
                                extends UnicastRemoteObject
                                implements GestorBilletes{
  private Vector listaBilletes = new Vector();
  private static int maxBills = 50;
  public ServidorGestorBilletes() throws RemoteException{}

  public int getBillete(String nom) throws RemoteException{
        //lógica de negocio
        num=9999;
        return num;
  }
```

3. Registrar un objeto de la clase remota

Se crea un registro en el servidor (en el main() de la misma clase que el objeto u otra distinta)

```
java.rmi.registry.LocateRegistry.createRegistry(9999);
```

- Se crea un objeto de la clase remota

```
ServidorGestorBilletes objetoServidor =
                        new ServidorGestorBilletes();
```

- Se crea un nombre para ese servicio

```
String servicio = "rmi://localhost:9999//gestorBilletes";
```

- Se registra ese servicio con ese nombre

```
Naming.rebind(servicio,objetoServidor
```

3. Registrar un objeto de la clase remota

```
class ServidorRemoto
 public static void main(String[] args) {
 //Falta el java.policy
 try { java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception e)
{System.out.println("Rmiregistry ya lanzado"+e.toString());}
   try {
        ServidorGestorBilletes objetoServidor =
              new ServidorGestorBilletes();

        String servicio = "//localhost/gestorBilletes";
        //          "//localhost:NumPuerto/NombreServicio"
        // Registrar el servicio remoto
        Naming.rebind(servicio,objetoServidor);
   } catch (Exception e)
        {System.out.println("Error al lanzar el servidor");}
}
```

1

2

3

4

3. Registrar un objeto de la clase remota

java.rmi.registry.LocateRegistry.createRegistry(p)

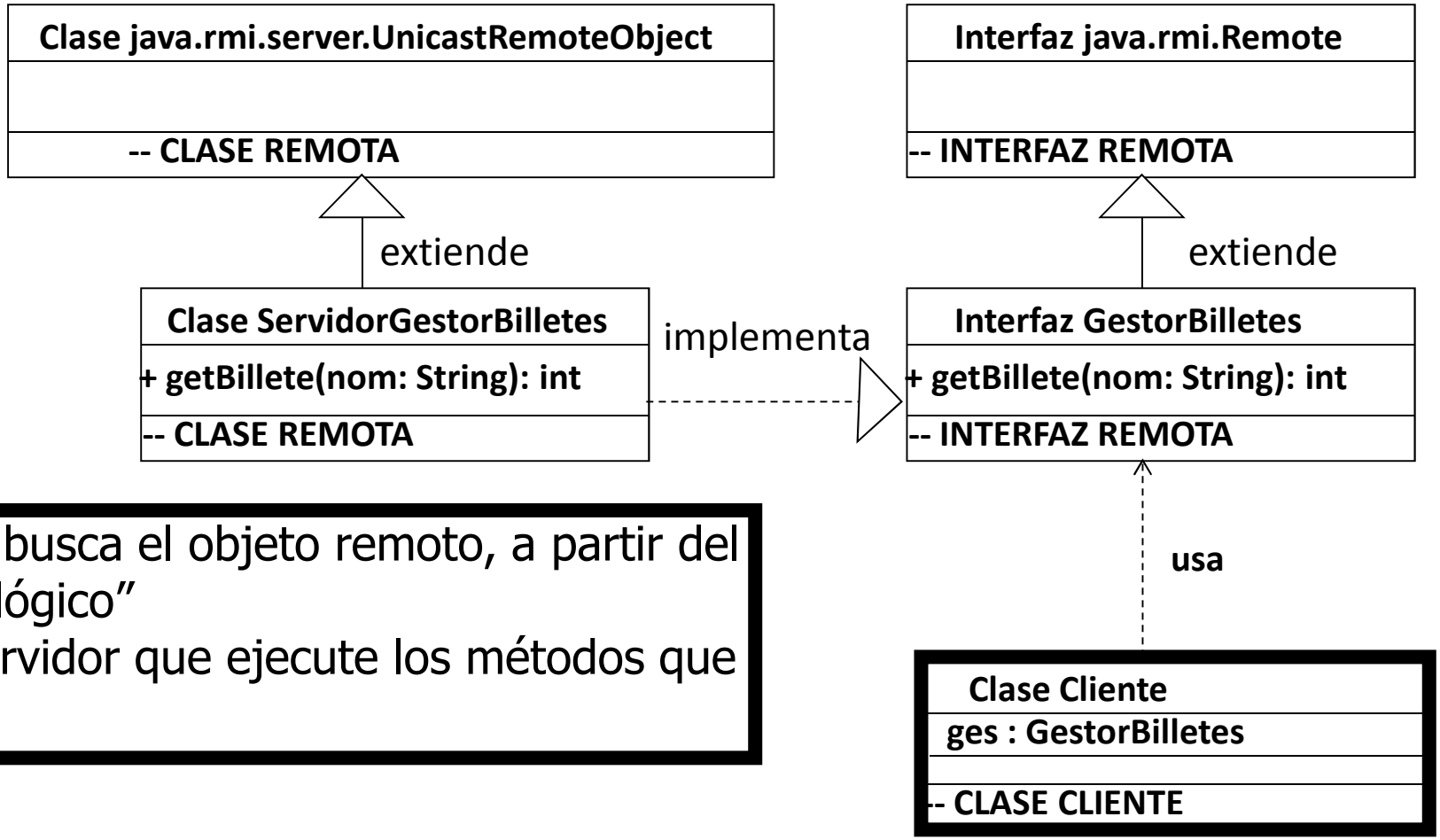
Crea el proceso rmiregistry en el puerto p.
El rmiregistry lanzado no acaba aunque acabe el servidor RMI
Lanza una excepción si el puerto está ocupado

```
try { java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception e)
{System.out.println("Rmiregistry ya lanzado"+e.toString());}
```

Código que lanza el rmiregistry y controla la excepción que se puede levantar al reejecutar varias veces el servidor RMI

Parar el rmiregistry:UnicastRemoteObject.*unexportObject*(registry, **true**);

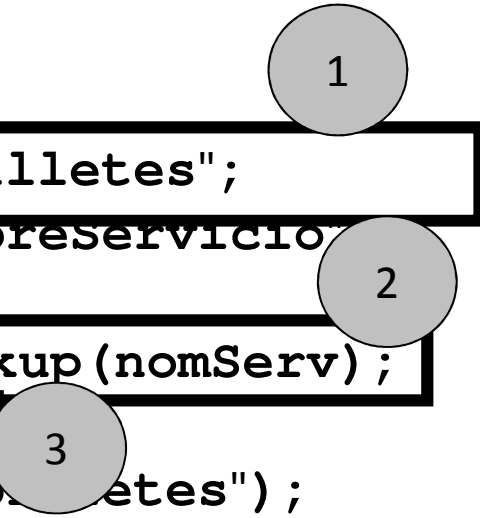
4. Localizar y ejecutar el objeto remoto



El cliente busca el objeto remoto, a partir del nombre "lógico"
Pide al servidor que ejecute los métodos que desee

4. Localizar y ejecutar el objeto remoto

```
import java.rmi.*;
public class Cliente {
 public static void main(String[] args) {
   // Falta gestion java.policy
     GestorBilletes objRemoto;
   String nomServ = "rmi://localhost/gestorBilletes";
   //          "rmi://DireccionIP:NumPuerto/NombreServicio"
 try {
      objRemoto = (GestorBilletes)Naming.lookup(nomServ);
      int b = objRemoto.getBillete(args[0]);
      if (b==-1) System.out.println("No hay billetes");
      else System.out.println("Obtenido : "+b);
 } catch (Exception e) { System.out.println("Error... ");}
}}
```



Evolución del sistema

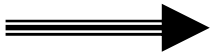
¿Qué cambios habría que realizar si…?

- Se quisiera cambiar el SGBD de Access a MySQL
- Si se quisiera cambiar la BD: por ejemplo tabla BILLETES => ENTRADAS
- Quisiéramos una nueva regla del negocio: no permitir comprar más de 6 entradas a la misma persona
- Se desea que la respuesta a “Pedir Billete” salga en otra ventana

Cambiar el SGBD: de Access a MySQL

```
public ServidorGestorBilletesBD() throws RemoteException{
try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 conexion=DriverManager.getConnection("jdbc:odbc:Billetes");
 conexion.setAutoCommit(false);
 sentencia=conexion.createStatement(); }
catch(Exception e)
     {System.out.println("Error:"+e.toString());}}
```

```
Class.forName("org.gjt.mm.mysql.Driver");
```



```
conexion=DriverManager.getConnection(
          "jdbc:mysql://localhost/Billetes");
```

Cambiar el SGBD: de Access a MySQL

En el SERVIDOR DE DATOS

- Hay que cambiar el nivel de datos (definir la BD MySQL)

En el SERVIDOR DE APLICACIONES

- Hay que instalar la clase org.gjt.mm.mysql.Driver en el CLASSPATH del servidor de aplicaciones
- Hay que cambiar la lógica del negocio, para que acceda al nuevo nivel de datos
  - Recompilar la clase ← **SE PUEDE EVITAR**
  - Volver a ejecutar la lógica del negocio (para que se haga el rebind y se exporte la nueva lógica del negocio)

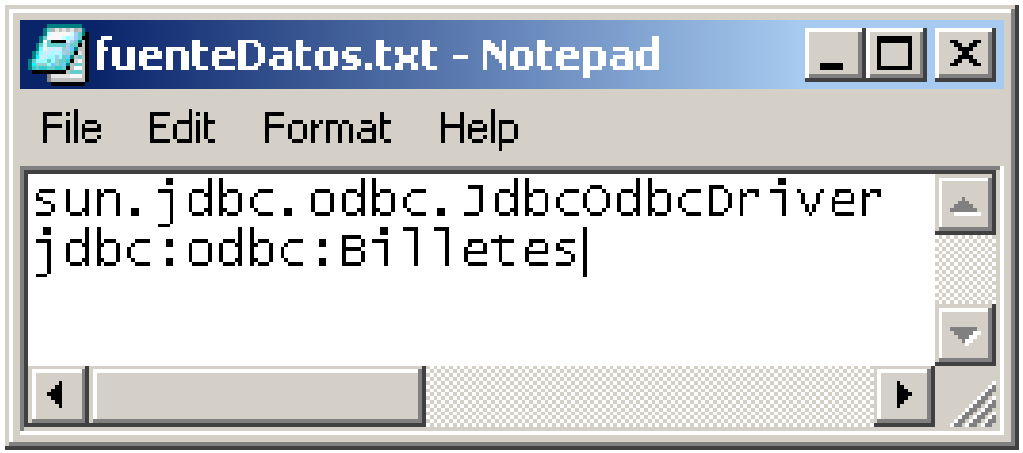
En el CLIENTE

- Volver a ejecutar el nivel de presentación para que haga el lookup y utilice la nueva lógica del negocio

SE PUEDE EVITAR

Para no tener que recompilar...

```
public ServidorGestorBilletesBD() throws RemoteException{
try {BufferedReader ent =
       new BufferedReader
       (new InputStreamReader(
          new FileInputStream("F:\\fuenteDatos.txt")));
Class.forName(ent.readLine());
conexion=DriverManager.getConnection(ent.readLine());
conexion.setAutoCommit(false);
sentencia=conexion.createStatement();}
catch(Exception e)
  { System.out.println("Error:"+e.toString());} }
```



Sólo configurar un fichero en el servidor de aplicaciones

Para no tener que relanzar la presentación…

```
void jButton1_actionPerformed(ActionEvent e) {
try{
 gestorBilletes=(GestorBilletes)Naming.lookup(
                    "rmi://localhost:1099/gestorBilletes");
 int res =
     gestorBilletes.getBillete(jTextField1.getText()).getNum();
 if (res<0) jTextArea1.append("Error al asignar billete");
 else jTextArea1.append("Asignado. \nReferencia: "+res+"\n");
} catch (Exception er) {System.out.println("Error: "+er.toString());
```

Obtener el objeto con la lógica del negocio SIEMPRE antes de usarla. Así se ejecutará la última versión.

Eso sí, será más ineficiente al solicitar el billete. Habrá que evaluar si es mejor tener que relanzar la presentación si se cambia la lógica del negocio…

Un cambio en el nivel de datos afecta principalmente al nivel de datos

Y obliga a reconfigurar el servidor de aplicaciones

Cambiar la base de datos: usar la tabla entradas en vez de billetes

```
public Billete getBillete(String nom)
  throws RemoteException {
 // Devuelve billete con nº billete, -1 si no hay, -2 problemas
  String pregSQL = "SELECT NUMERO FROM BILLETES"+
                   " WHERE ESTADO=\'LIBRE\'";
  try{ ResultSet rs = sentencia.executeQuery(pregSQL);
    if (rs.next()) {
        String num = rs.getString("NUMERO");
        int act = sentencia.executeUpdate("UPDATE BILLETES"+
                  " SET ESTADO='OCUPADO', NOMBRE = '"+nom+
                  "' WHERE NUMERO="+num+" AND ESTADO='LIBRE'");
        conexion.commit();
```

⟹ "SELECT NUMERO FROM **ENTRADAS**"

⟹ "UPDATE **ENTRADAS**"

Cambiar la base de datos: usar la tabla entradas en vez de billetes

En el SERVIDOR DE DATOS

- Hay que cambiar el nivel de datos (redefinir la tabla BILLETES por ENTRADAS)

En el SERVIDOR DE APLICACIONES

- Hay que cambiar la lógica del negocio, para que acceda al nuevo nivel de datos
  - Recompilar la clase ⟵ **SE PUEDE EVITAR**
  - Volver a ejecutar la lógica del negocio (para que se haga el rebind y se exporte la nueva lógica del negocio)

En el CLIENTE

- Volver a ejecutar el nivel de presentación para que haga el lookup y utilice la nueva lógica del negocio

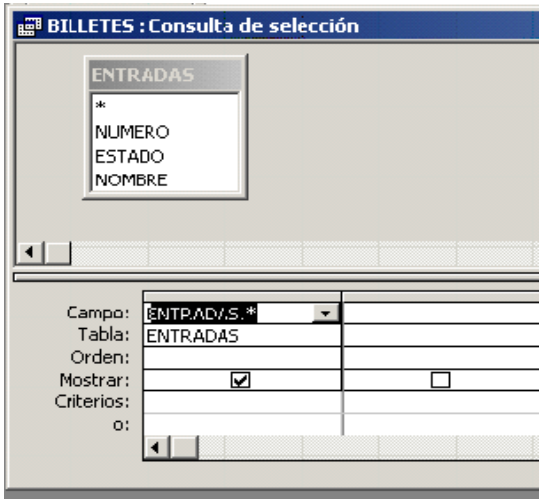
SE PUEDE EVITAR

En realidad basta con cambiar la base de datos

No hay que olvidar que las bases de datos proporcionan independencia lógica con las aplicaciones (por medio de las VISTAS)

Si una vez cambiada la base de datos se definiera una vista llamada BILLETES definida como SELECT * FROM ENTRADAS, entonces **no haría falta cambiar la lógica del negocio**

Esto no es siempre posible si la vista no es actualizable…



Un cambio en el nivel de datos que afecta sólo al nivel de datos

Nueva regla negocio: no más de 6 entradas por persona

```
public Billete getBillete(String nom) throws RemoteException {
// Devuelve billete con nº billete, -1 si no hay, -2 si hay probs
// Devuelve -3 si "nom" tiene ya 6 entradas
String pregSQL = "SELECT NUMERO FROM BILLETES"+
                          " WHERE ESTADO=\'LIBRE\'";
try{
 ResultSet rs = sentencia.executeQuery(
   "SELECT COUNT(*) FROM BILLETES WHERE NOMBRE='"+nom+"'");
 if ((rs.next()) && (rs.getInt(1)>=6))
     return new Billete(-3,"");
 rs = sentencia.executeQuery(pregSQL);
 if (rs.next()) {...
```

Un cambio en la lógica del negocio que afecta sólo al nivel de lógica del negocio

¿Habría que cambiar la presentación?

Si nos indican que se debe mostrar al usuario que no se da el billete porque ya tiene 6 billetes, entonces, tal y como está implementado SÍ HABRÍA QUE CAMBIARLA.

```
void jButton1_actionPerformed(ActionEvent e) {
  try{int res =
       gestorBilletes.getBillete(jTextField1.getText()).getNum();
   if (res==-3) jTextArea1.append("Error: máx. 6 entradas");
    if (res<0) jTextArea1.append("Error al asignar billete");
   else jTextArea1.append("Asignado. \nReferencia: "+res+"\n");
  } catch (Exception er)
       {System.out.println("Error: "+er.toString());
        jTextArea1.append("Error al asignar billete");}}
```

¿Habría que cambiar la presentación?

Para no tener que cambiar la presentación aunque hubiera nuevos tipos de error, la lógica del negocio debía haber sido distinta…

La clase Billete tendría que tener un atributo nuevo (el atributo **mensaje**)

```
 public Billete getBillete(String nom) throws RemoteException {
... try{ ResultSet rs = sentencia.executeQuery(
    "SELECT COUNT(*) FROM BILLETES WHERE NOMBRE='"+nom+"'");
 if ((rs.next()) && (rs.getInt(1)>=6))
     return new Billete(-3,"","Error: más de 6 billetes");
 rs = sentencia.executeQuery(pregSQL);
 ... if (act>0) return new Billete(n,nom,"Billete asignado");
     return new Billete(-2,"","Error, inténtelo de nuevo"); }
```

¿Habría que cambiar la presentación?

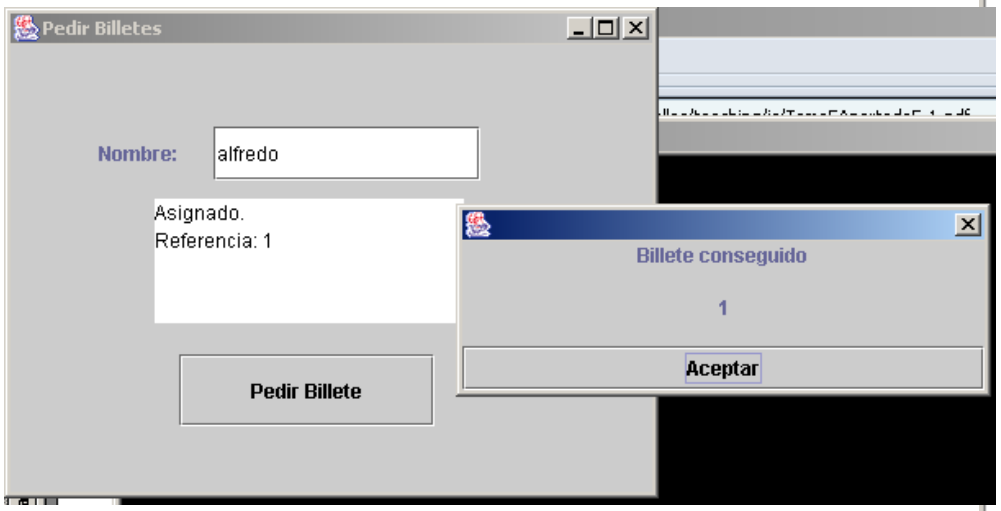
Y la presentación debía haber sido así:

```
void jButton1_actionPerformed(ActionEvent e) {
  try{Billete res =
       gestorBilletes.getBillete(jTextField1.getText()).getNum();
       jTextArea1.append(res.getMensaje()+
                         "\nReferencia: "+res.getNum()
  } catch (Exception er)
       {System.out.println("Error: "+er.toString());
        jTextArea1.append("Error al asignar billete");}}
```

Un cambio en la lógica del negocio se implementa en la lógica del negocio
El cambio en la presentación podría evitarse

Se desea que la respuesta a "Pedir Billete" salga en otra ventana

```
void jButton1_actionPerformed(ActionEvent e) {
int res = -4;
try{res =
     gestorBilletes.getBillete(jTextField1.getText()).getNum();
} catch (Exception er) {System.out.println("Error: "+er.toString());
                        res=-2;
 Mensaje m = new Mensaje(res);
 m.setVisible(true);
```




```
public class Mensaje extends JDialog
{
  private BorderLayout borderLayout1 = new BorderLayout();
  private JButton jButton1 = new JButton();
  private JLabel jLabel1 = new JLabel();
  private JLabel jLabel2 = new JLabel();
  public Mensaje()
  {
    this(null, "", true);
  }
  public Mensaje(int res)
  {
    this();
    if (res>=0)
    jLabel1.setText("Billete conseguido");
    else jLabel1.setText("Error al conseguir billete");
    jLabel2.setText(Integer.toString(res));
  }

  public Mensaje(Frame parent, String title, boolean modal)
  {
    super(parent, title, modal);
    try
```

Se desea que la respuesta a "Pedir Billete" salga en otra ventana

En el SERVIDOR DE DATOS

- No hay que cambiarlo

En el SERVIDOR DE APLICACIONES

- No hay que cambiar la lógica del negocio, para implementar la nueva regla del negocio
  - Ni recompilar la clase
  - Ni volver a ejecutar la lógica del negocio (para que se haga el rebind y se exporte la nueva lógica del negocio)

En el CLIENTE

- Hay que compilar la clase
- Volver a ejecutar el nivel de presentación

Un cambio en el nivel de presentación que afecta sólo al nivel de presentación

Simple Object Access Protocol (SOAP)

- SOAP es un simple protocolo en XML que permite a las aplicaciones intercambiar información mediante HTTP.

- SOAP es un protocolo para facilitar los Servicios Web

- http://www.w3schools.com/soap/default.asp
- http://www.w3schools.com/xml/default.asp

- SOAP se inició en 1999 por W3C.
- SOAP 1.0 estaba basado por entero en HTTP.
- La siguiente versión SOAP 1.1 (Mayo 2000) era más genérica e incluía otros protocolos de transporte.
- La versión actual de SOAP 1.2 (Junio 2003) ha sido promovida a "Recomendación".

Qué es SOAP?

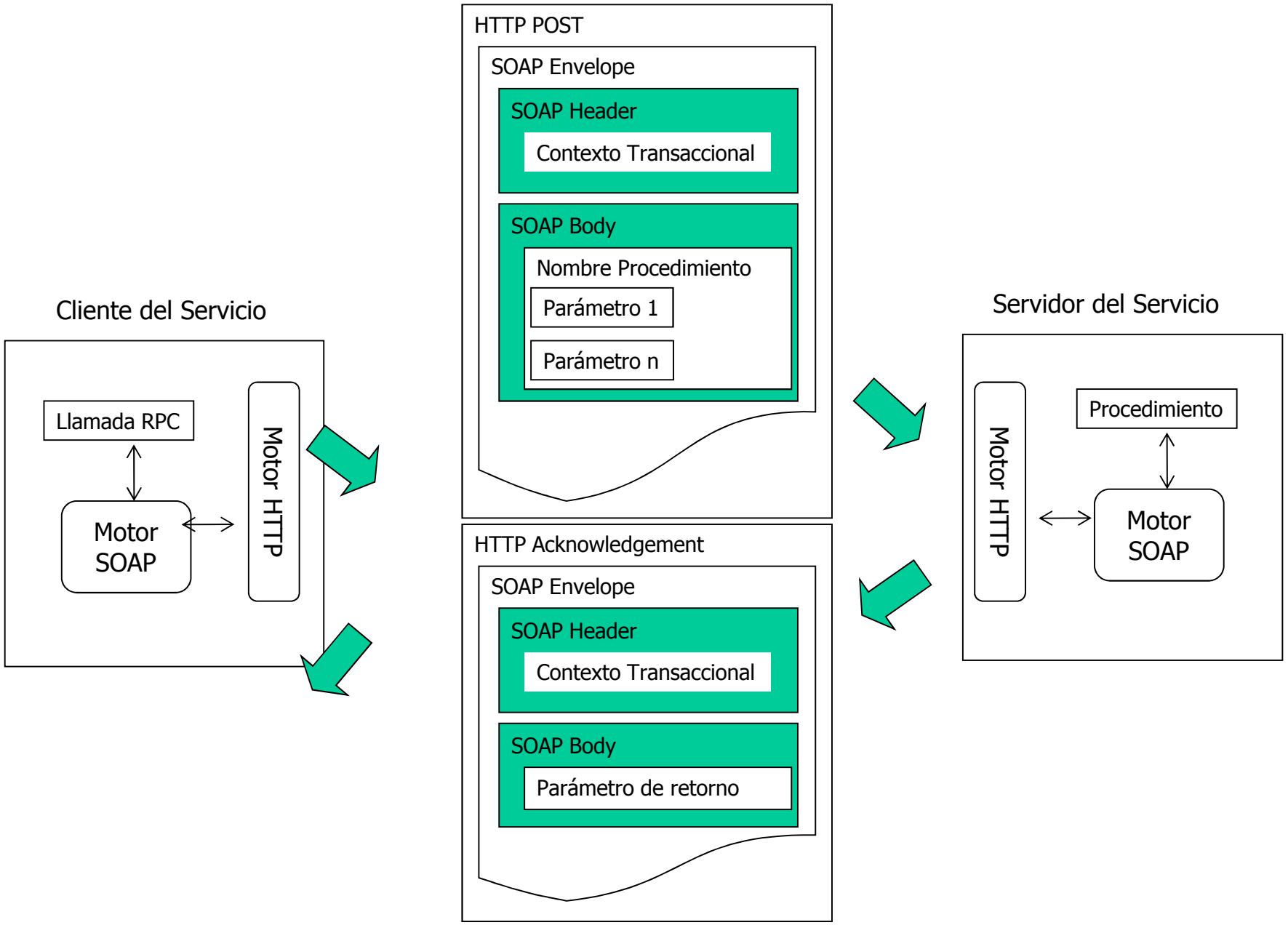
- SOAP significa **Simple Object Access Protocol**
- SOAP es un **protocolo de comunicación**
- SOAP permite la comunicación entre **aplicaciones**
- SOAP es un formato para **enviar mensajes**
- SOAP está diseñado para comunicarse via **Internet**
- SOAP es **independiente de la plataforma**
- SOAP es **independiente del lenguaje de programación**
- SOAP está **basado en XML**
- SOAP es **simple y extensible**
- SOAP evita **firewalls**
- SOAP es un **estándar W3C**

¿Por qué SOAP?

- Las aplicaciones se comunicaban usando Remote Procedure Calls (RPC) entre objetos como DCOM y CORBA.
- Sin embargo, RPC representa un problema de compatibilidad y seguridad: los firewalls y proxies normalmente bloquearán este tipo de tráfico.
- Es importante para el desarrollo de aplicaciones permitir la comunicación entre programas usando Internet.
- Una mejor forma de comunicar aplicaciones es usando HTTP, porque HTTP es soportada por todos los navegadores y servidores.
- Usa estándares: HTTP y XML.
- SOAP proporciona una forma de comunicación entre aplicaciones que pueden estar ejecutándose en distintos SO, distintas tecnologías y distintos lenguajes.

Componentes SOAP

- Un formato de mensaje en XML para comunicaciones de una sola dirección
- Una descripción de cómo un mensaje SOAP se transporta a través de la web (usando HTTP) o e-mail (usando SMTP).
- Un conjunto de reglas que seguir cuando se procesa un mensaje SOAP y una clasificación simple de las entidades involucradas en el proceso. También especifica qué partes del mensaje deben ser leídas por quién y cómo reaccionar en caso de que el contenido no ha sido entendido.
- Un conjunto de convenciones sobre cómo convertir llamadas y retornos tipo RPC en mensajes SOAP


HTTP POST
SOAP Envelope
SOAP Header
Contexto Transaccional
SOAP Body
Nombre Procedimiento
Parámetro 1
Parámetro n
Cliente del Servicio
Llamada RPC
Motor SOAP
Motor HTTP
Servidor del Servicio
Motor HTTP
Procedimiento
Motor SOAP
HTTP Acknowledgement
SOAP Envelope
SOAP Header
Contexto Transaccional
SOAP Body
Parámetro de retorno

Mensajes SOAP

- SOAP se basa en el intercambio de mensajes
- Los mensajes pueden verse como sobres donde las aplicaciones encapsulan los datos a ser enviados
- Un mensaje tiene dos partes: cabecera (header) y cuerpo (body), que pueden ser divididos en bloques. La cabecera es opcional y el cuerpo obligatorio
- El uso de la cabecera y el cuerpo es implícito. La cabecera es para los datos del nivel de infraestructura y el cuerpo para los datos del nivel de aplicación.

Sintaxis SOAP

- Un mensaje SOAP es un documento XML que contiene los siguientes elementos:

- Una parte obligatoria (Envelope) que identifica el documento XML como un mensaje SOAP
- Una cabecera opcional (Header) que contiene la información de la cabecera
- Un cuerpo obligatorio (Body) que contiene la información de la llamada y la respuesta
- Una sección opcional (Fault) que proporciona información sobre los errores que ocurren mientras se procesa el mensaje

Ejemplo SOAP: Petición

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="http://www.stock.org/stock">
      <m:GetStockPrice>
            <m:StockName>IBM</m:StockName>
      </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

Ejemplo SOAP: Respuesta

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: nnn
<?xml version="1.0"?>
  <soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPriceResponse>
          <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
    </soap:Body>
</soap:Envelope>
```

Sintaxis SOAP

```
<?xml version="1.0"?>
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope”
   soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Header> ... ... </soap:Header>
<soap:Body> ... ...
<soap:Fault> ... ... </soap:Fault>
</soap:Body>
</soap:Envelope>
```

Abstracción vs. Eficiencia