

Un ejemplo de sistema distribuido: **la agencia de robots**



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

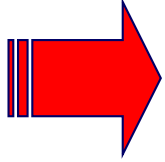


José Simó

DISCA / UPV

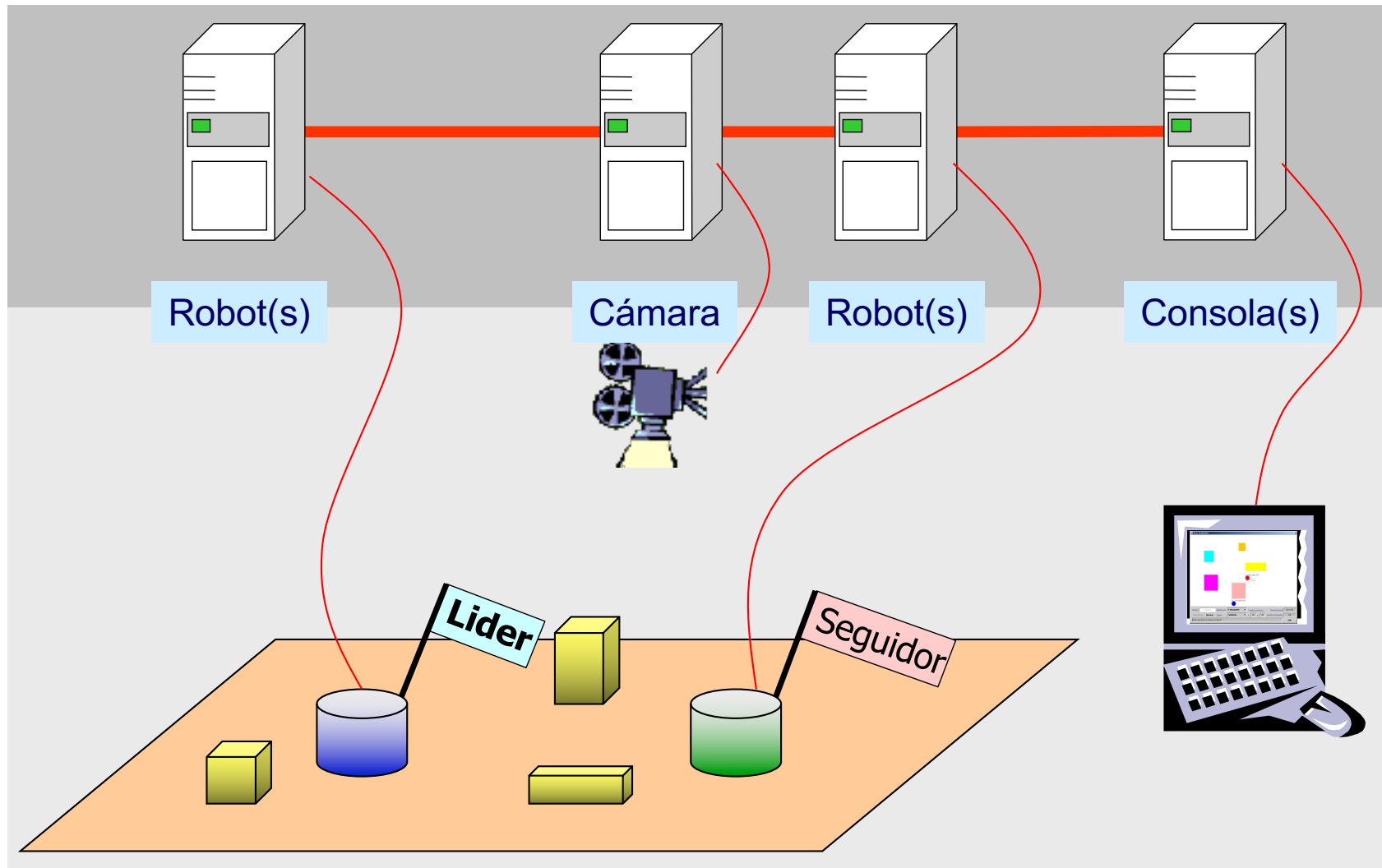
**Departament d'Informàtica de Sistemes i Computadors
Universitat Politècnica de València**

● Contenido



- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: ICE y difusión
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola

- La persecución de robots



- La persecución de robots

- Sobre un **escenario** con un conjunto de obstáculos se mueven una serie de **robots** cuya conducta es alcanzar un objetivo evitando los obstáculos del escenario. Un robot dos tipos de objetivos:
 - un punto aleatorio (robot líder)
 - otro robot (robot seguidor)
- La coordinación de los robots se basa en las imágenes que capta una **cámara** cenital que proporciona el *estado global del sistema*: posiciones de todos los robots y de los obstáculos, las cuales son difundidas a todos los componentes del sistema.
- En el sistema pueden ejecutarse diversas **consolas** en diferentes nodos cuya misión es monitorizar y controlar el sistema.
- El sistema es un *grupo dinámico* cuyos componentes son robots y consolas de las cuales pueden ejecutarse un número arbitrario de instancias. La cámara actúa de gestor de grupo:
 - Los componentes deben suscribirse en la cámara al comenzar su ejecución
 - La cámara monitoriza las posiciones de los robots inscritos (estado global) y las difunde a todos sus componentes.

● Componentes del sistema

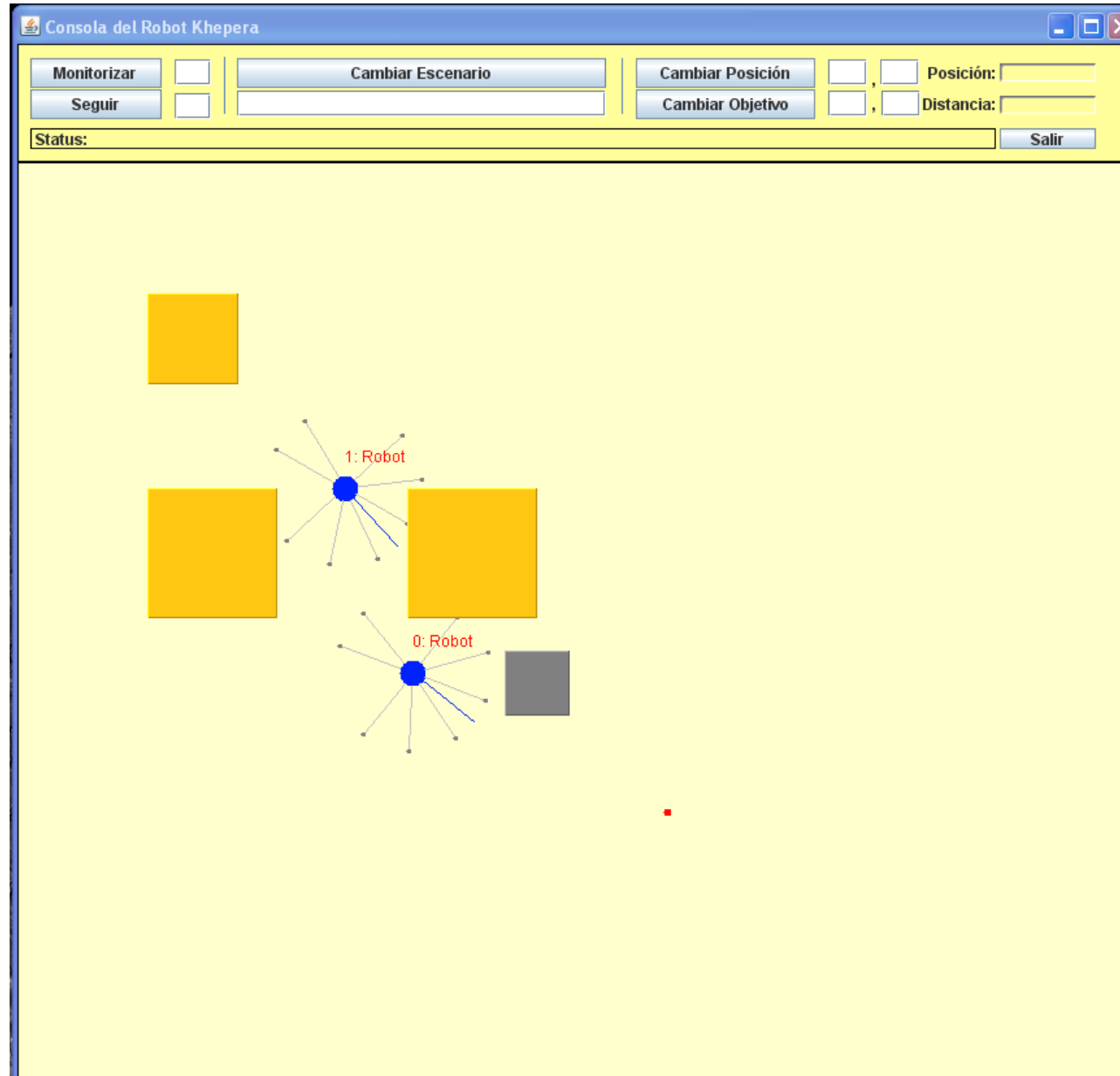
El sistema consta de tres tipos de componentes:

- **Robots:** cuya conducta es alcanzar un objetivo (robot o punto aleatorio) evitando los obstáculos del escenario
- **Cámara cenital:** tiene un doble objetivo:
 - Difundir **instantáneas** del estado global (posición de todos los robots) y gestionar e informar sobre los cambios del **escenario** de obstáculos.
 - Gestiona el canal de difusión: la dirección IP y el port para las difusiones del sistema, son establecidos como parámetros al lanzarla a ejecución.
 - Actuar de gestor del grupo, ocupándose de las altas y bajas de los **robots** y de las **consolas** en el grupo así como de detectar los “fallos” de los **robots**.
- **Consolas:** cuyo objetivo es monitorizar y controlar el sistema.

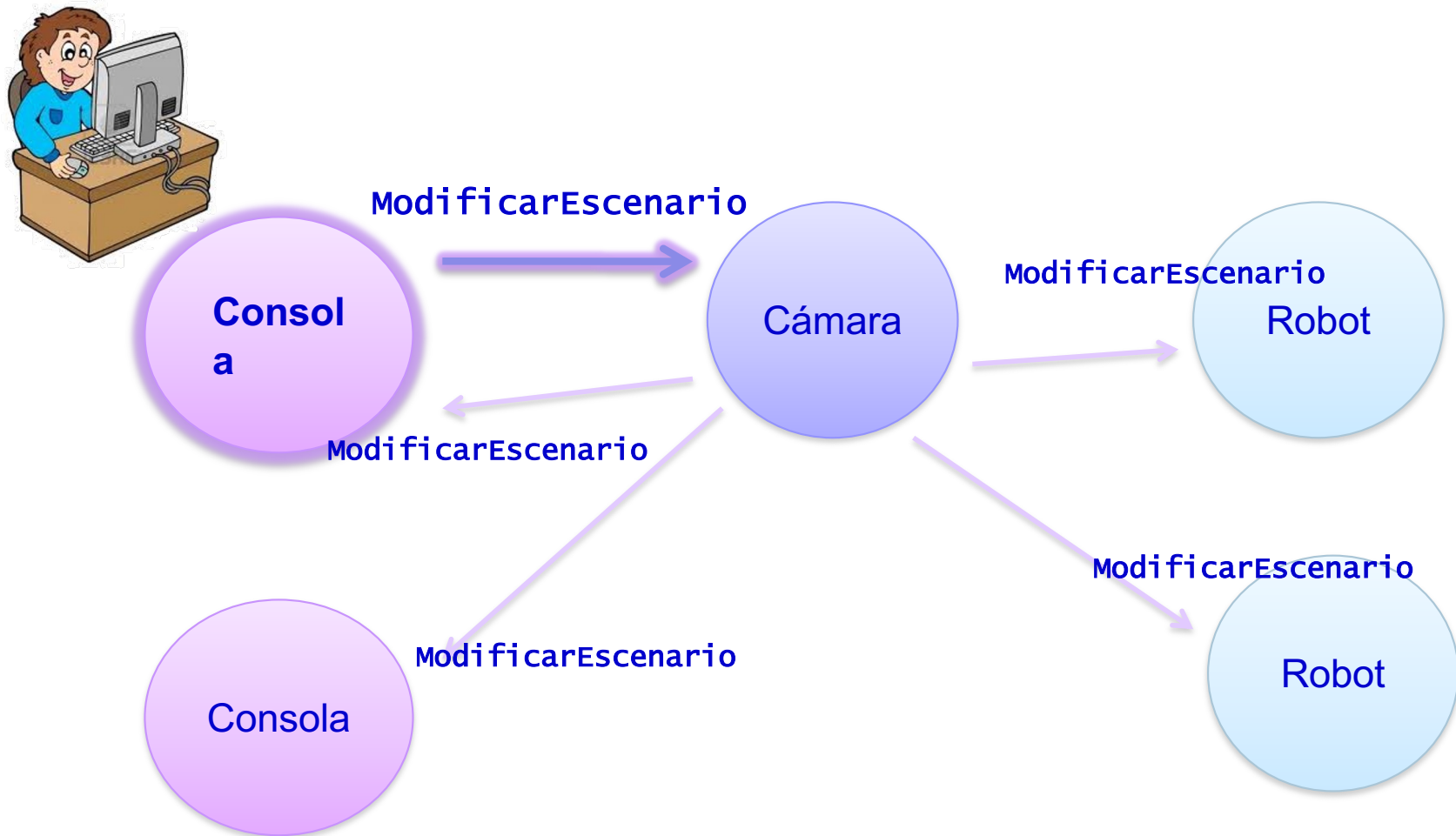
- Implementación mediante simulación
 - La **cámara** es *simulada*: como no existe posibilidad de que capte el estado global del sistema de forma real, recurre a la siguiente técnica:
 - Cada robot evalúa su posición en el sistema, sabiendo su punto de partida y sus movimientos
 - La cámara interroga periódicamente a todos los robots sobre su posición
 - Todos los cambios de escenario solicitados por las consolas son comunicados a la cámara que es la que gestiona el escenario actual.
 - Los **robots** son *simulados*: existe una biblioteca `khepera.jar` en el que se proporciona:
 - Una interfaz similar a la del robot Khepera para acceder los motores y los sensores.
 - Algoritmos de control para evitar obstáculos y encaminarse a un destino
 - La **consola** se proporciona *implementada*: existe una biblioteca `console.jar` que permite al operador:
 - Visualizar los robots y el escenario
 - Cambiar objetivos, posición de los robots y el escenario

SDI Definición del problema

- La interfaz gráfica (Consola)



- Protocolo de cambio de escenario



- Comunicación remota

- Comunicación cliente-servidor

- La comunicación se realizará utilizando ICE
 - Todos los objetos remotos adoptarán una interfaz predefinida (Slice).

- Comunicación de grupo

- Existe un concepto de estado global del sistema que comprende:
 - El estado de todos los robots (básicamente su posición)
 - Se denominará *instantánea* al estado global del sistema en un determinado instante.
 - La cámara gestionará un grupo de robots dinámico y difundirá una instantánea con el estado global del sistema al grupo

- Objetos con interfaz remota

Existen tres clases con interfaz “ICE” de los que pueden ejecutarse diferentes instancias en diferentes ubicaciones y que comunican vía invocaciones remotas o difusiones:

- **RobotSeguidor**

- Se pueden ejecutar tantas instancias como se desee en diferentes nodos
- Las instancias se registran en la **cámara** que actúa como gestor del grupo

- **Consola**

- Se pueden realizar tantas instancias como se desee en diferentes nodos
- Las instancias se registran en la **cámara** que actúa como gestor del grupo

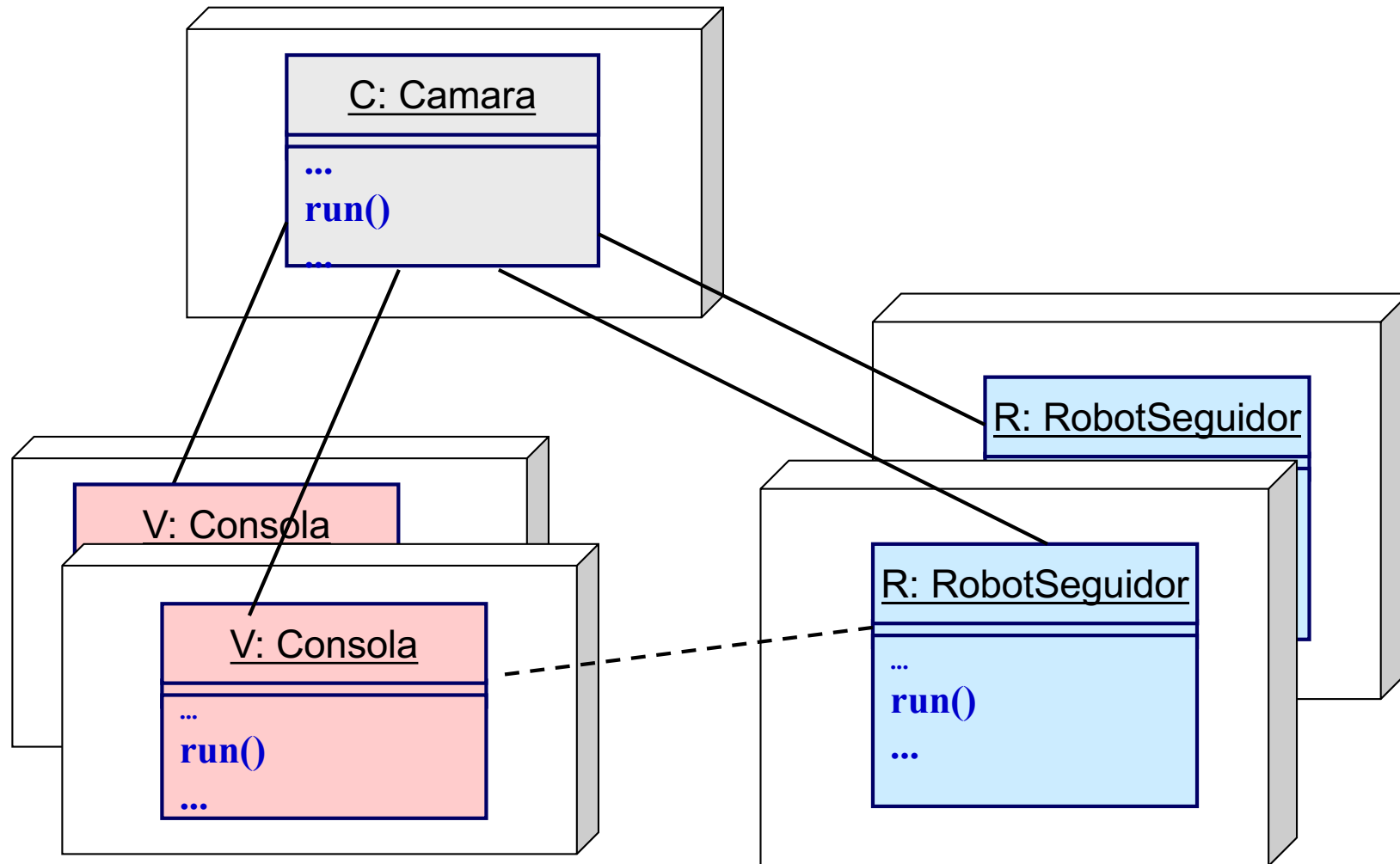
- **Camara**

- Una única instancia en todo el sistema
- Es la única inscrita en el servicio de nombres
- Es el primer componente que ha de lanzarse a ejecución.
- Actúa de gestor del grupo, ocupándose de las altas y bajas de los **robots** y de las **consolas** en el grupo

- Cada objeto remoto → un proyecto

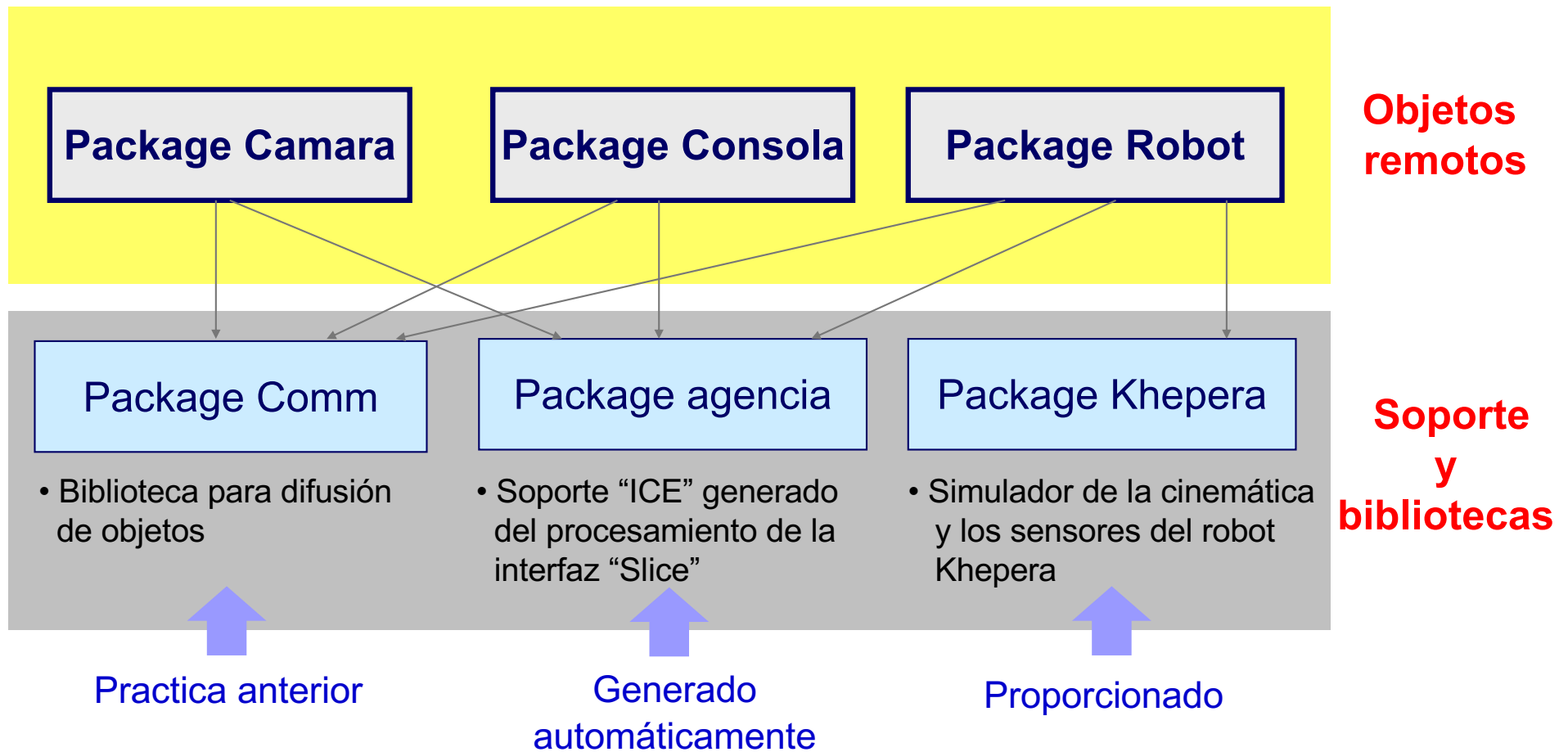
SDI Objetos remotos

- Objetos con interfaz remota



SDI Estructura de la aplicación

- Estructura de la implementación
 - Está estructurada en 6 paquetes
 - Cada paquete es, además, un proyecto que genera un fichero JAR con el código.



```
module agencia {
```

```
    ///Module agencia.datos
```

```
    module datos {
```

```
        struct Posicion {
```

```
            float x;
```

```
            float y;
```

```
        }
```

```
        ...
```

```
        ...
```

```
        ...
```

```
    }
```

```
    /// Module agencia.objetos
```

```
    module objetos {
```

```
        interface RobotSeguidor{
```

```
            agencia::datos::EstadoRobot ObtenerEstado( );
```

```
            void ModificarEscenario( agencia::datos::Escenario esc);
```

```
            void ModificarObjetivo( agencia::datos::Posicion NuevoObj);
```

```
            void ModificarPosicion( agencia::datos::Posicion npos);
```

```
            void ModificarLider( int idLider);
```

```
        };
```

```
    ...
```

```
    ...
```

```
    }
```

- Clases básicas

- Facilitan trabajar con puntos geométricos, distancias, ...

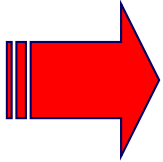
- **Posicion:** define una coordenada (x,y) con precisión de float y proporciona un método para calcular distancias con otra “Posicion”.
- **Polares:** define la posición del robot en coordenadas polares (centro, angulo_del_puntero). El ángulo del puntero se mide en radianes. Por cuestiones de eficiencia contiene también los vectores unitarios de la dirección del robot (evita recalcularlos frecuentemente).
- **IzqDer:** un par de enteros utilizados para determinar la velocidad de pulsos del motor izquierdo y del motor derecho.

```
class IzqDer {  
    int izq;  
    int der;  
}
```

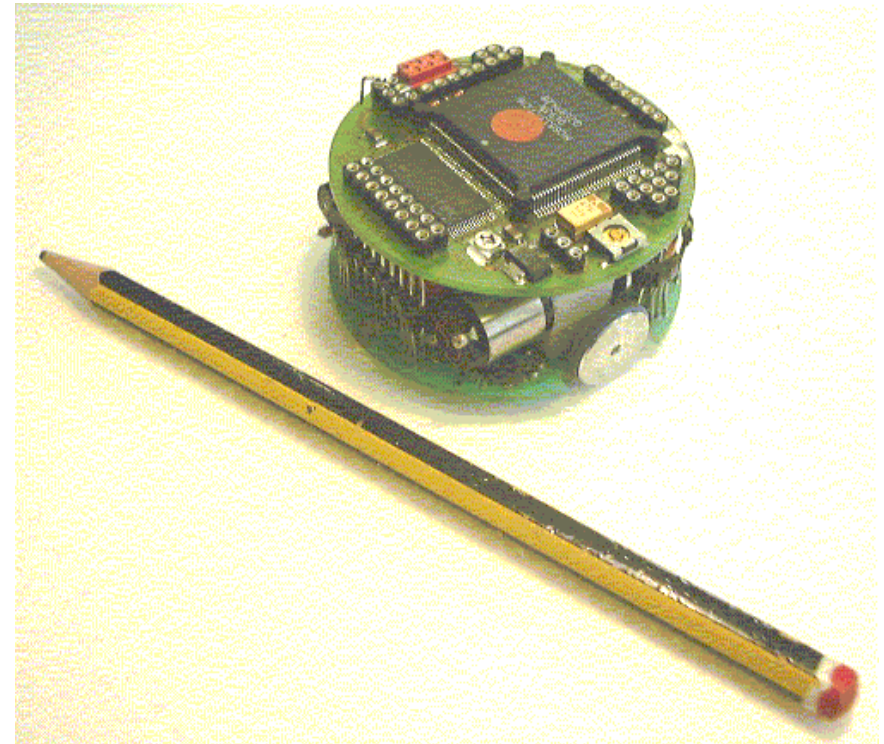
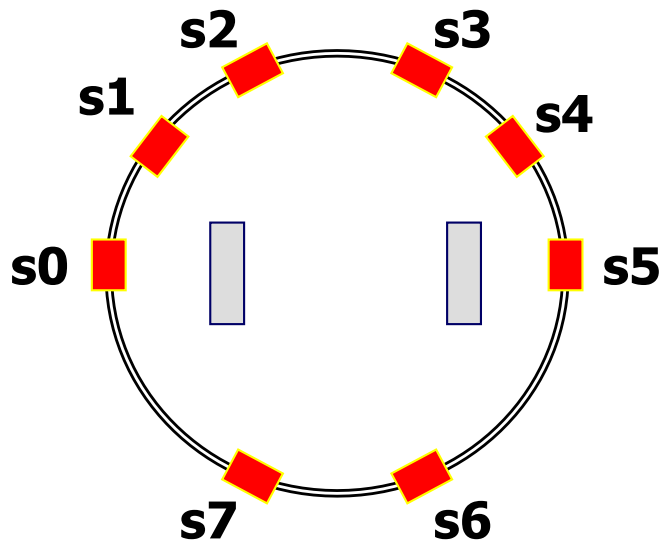
```
class Polares {  
    float x;  
    float y;  
    float angulo ; //en radianes  
    float ux; //componente unitaria x: sen(angulo)  
    float uy; //componente unitaria y: cos(angulo)  
}
```

● Contenido

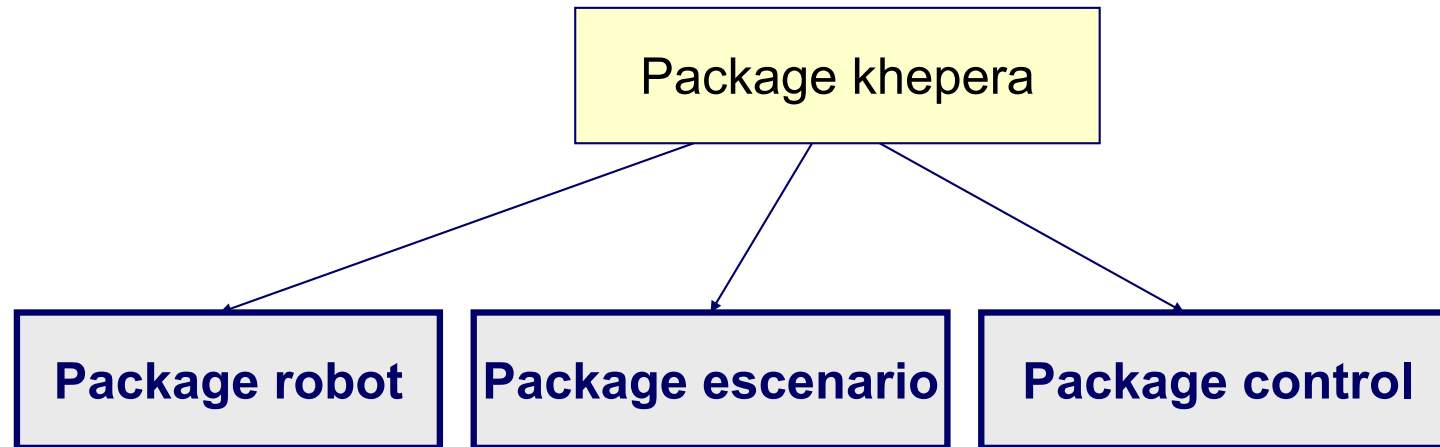
- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: ICE y difusión
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola



- Características
 - 2 motores CC
 - 8 sensores de proximidad de infrarrojos



- Estructura de la implementación



- La clase RobotKhepera

- Simula el comportamiento del robot Khepera tanto en lo que se refiere al movimiento del robot como a los sensores de proximidad.
- Proporciona una interfaz próxima a la lista de comandos que ofrece el robot Khepera a través de su interfaz serie:
 - Fijar la velocidad de las ruedas del motor
 - Leer los *encoders* de posición
 - Obtener las lecturas de los sensores.
- Es una clase “activa” (subclase de Thread) que recalcula la posición y las lecturas de los sensores con un periodo que se puede especificar.
 - Alternativamente, permite que se recalculen las velocidades y las lecturas de sensores invocando el método avanzar y especificando periodo=0.
- Requiere una instancia de **escenario** para su construcción.
- Permite realizar clases más “inteligentes” en las que la conducta del robot obedezca a una motivación o combinación de motivaciones (repeler obstáculos, encaminarse a un objetivo).

```
public class ConfRobot {  
    public final static int NSENSORES = 8; //el número de sensores  
    final static int diametro = 20; //diámetro del robot  
    final static int dEntreRuedas = 15; //distancia entre ruedas del robot (la real es 662 pulsos)  
    final static int alcanceSens = 50; //longitud máxima que alcanzan los sensores  
    //grados a que están situados los sensores (origen en eje vertical, sentido horario)  
    final static int[] gradosSens = {270,306,342,18,54,90,162,192,0};  
    //matriz de coeficientes Braitenberg de los sensores [motor][sensor].  
    //motor 0: motor izq. motor 1: motor der.  
    public final static int[][] KBrai={{2,-3},{4,-15},{6,-18},{-18,6},{-15,4},{-3,2},{5,3},{3,5}};  
    //valores a,b,c (ax+by+c) de una ecuación lineal aproximada de la respuesta de los sensores  
    final static float[] ecuacionSens = {-1023,-40,-51150};  
    //La constante de proporcionalidad entre pulsos y pixels (o cm)  
    //En realidad 1 pulso/10ms corresponde a 8 mm/seg o sea 100 pulsos son 8 mm  
    public final static float K=(float) .03;
```

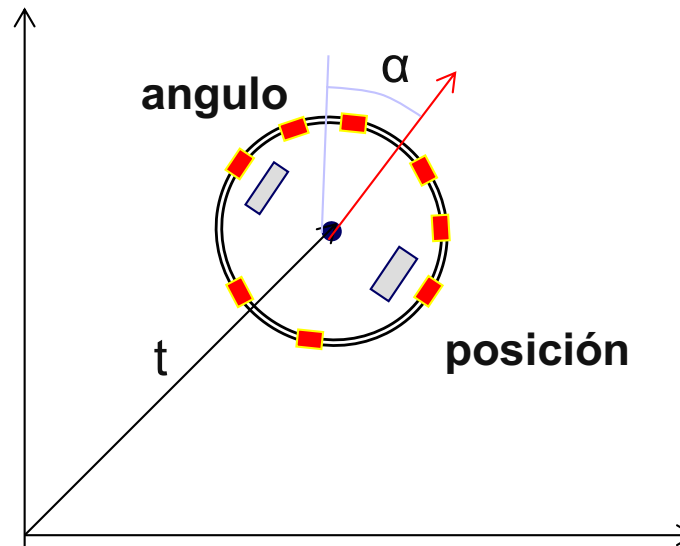
```
}
```

```
public interface KheperaInt {  
    // establecer/leer velocidad motores en pulsos.  
    // rango 0..127. La unidad es el pulso/10ms que corresponde a 8 mm/s  
    void fijarVelocidad(int velocidad_izq, int velocidad_der);  
    IzqDer leerVelocidad();  
    // establecer coordenadas cartesianas de posicion  
    void fijarPosicion(posicionD p);  
    // obtener coordenadas polares de posicion  
    Polares posicionPolares();  
    // devuelve un array de 8 floats con las lecturas de los sensores  
    float[] leerSensores();  
    // devuelve un array de 8 Posiciones con las coordenadas de los sensores + punt. posición  
    PuntosRobot posicionRobot();  
}
```

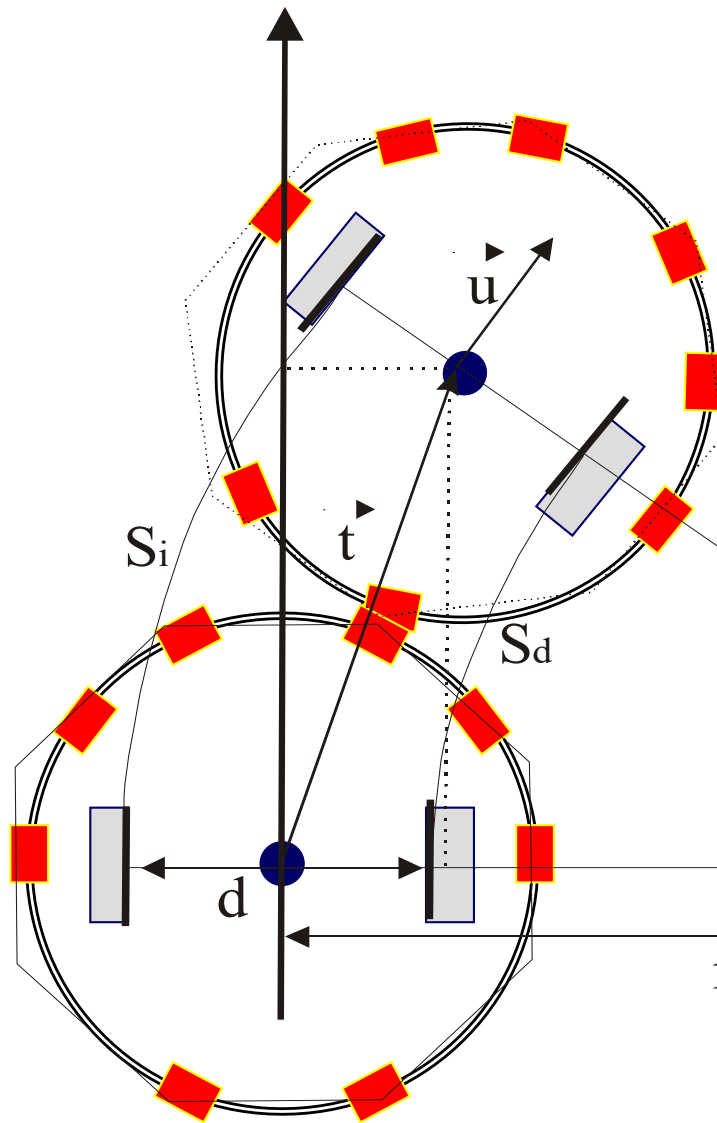
```
public class KheperaRobot implements Runnable, KheperaInt {  
    // Constructor:  
    public KheperaRobot(Posicion inicio, Escenario e, int periodo) {...}  
    // Posicion: posición inicial del robot  
    // Escenario: precisa haber creado un escenario  
    // int: periodo del thread que recalcula la posición del robot en mseg.  
    //      De el depende la velocidad. Valor recomendado 400  
    // periodo=0 permite recalcula su posición invocando el método:  
    public void avanzar() {...}  
    ...  
}
```

- Modelo cinemático

- Debe resolver el movimiento del robot a partir del movimiento de los motores.
- Entradas:
 - velocidad motores: **mi** y **md**
 - distancia entre ruedas: **dr**
- Salidas:
 - Las nuevas coordenadas polares del robot: **posición**, **angulo**



● Modelo cinemático

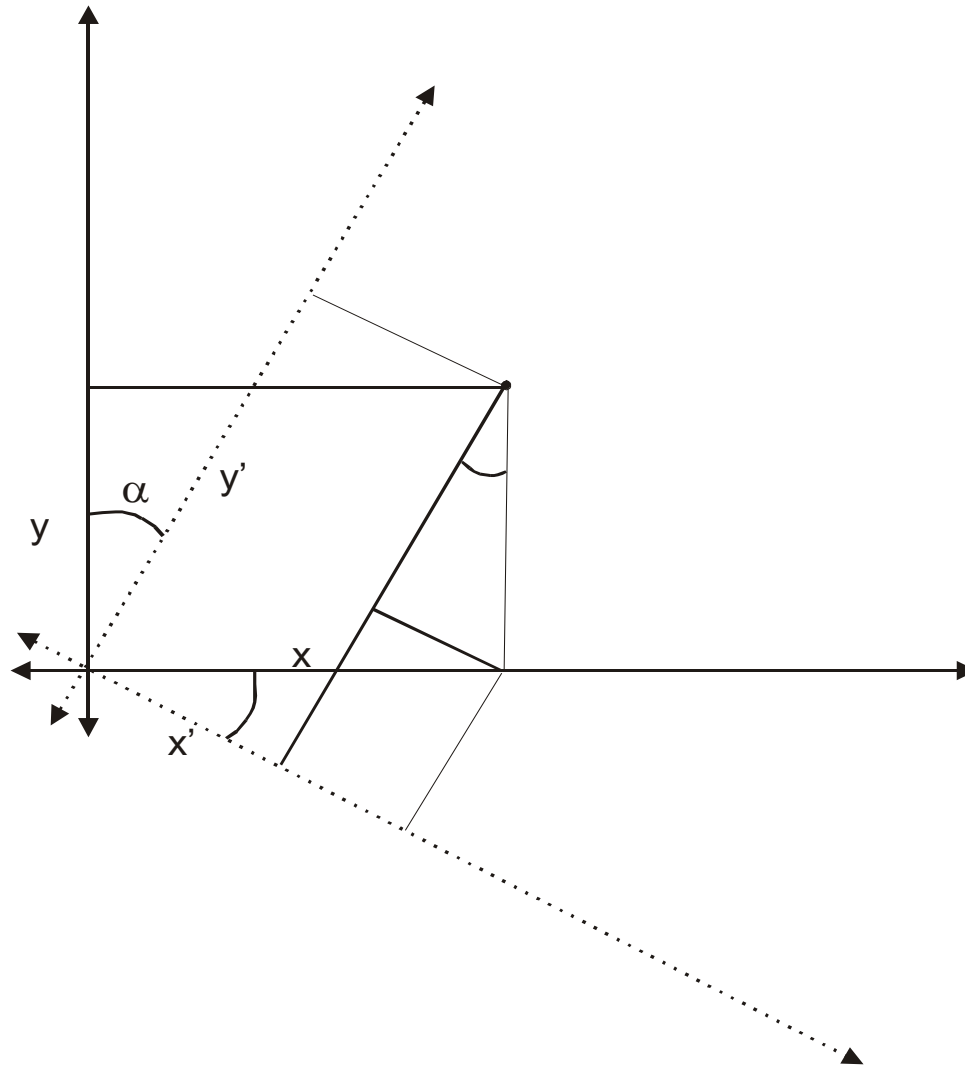


$$\begin{cases} S_i = \alpha(r + \frac{d}{2}) \\ S_d = \alpha(r - \frac{d}{2}) \end{cases} \quad \alpha = \frac{S_i - S_d}{d}$$

$$r = \frac{S_i + S_d}{S_i - S_d} \frac{d}{2}$$

$$\begin{cases} t_x = r - r \cos(\alpha) \\ t_y = r * \sin(\alpha) \end{cases}$$

● Modelo cinemático



$$x' = x \cos(\alpha) - y \sin(\alpha)$$

$$y' = x \sin(\alpha) + y \cos(\alpha)$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} x^{k+1} \\ y^{k+1} \end{pmatrix} = \begin{pmatrix} x^k \\ y^k \end{pmatrix} + \begin{pmatrix} u_x^k & -u_y^k \\ u_y^k & u_x^k \end{pmatrix} \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

$$\begin{pmatrix} u_x^{k+1} \\ u_y^{k+1} \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} u_x^k \\ u_y^k \end{pmatrix}$$

● Código Java del modelo cinemático

$$\begin{pmatrix} x^{k+1} \\ y^{k+1} \end{pmatrix} = \begin{pmatrix} x^k \\ y^k \end{pmatrix} + \begin{pmatrix} u_x^k & -u_y^k \\ u_y^k & u_x^k \end{pmatrix} \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

$$\begin{pmatrix} u_x^{k+1} \\ u_y^{k+1} \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} u_x^k \\ u_y^k \end{pmatrix}$$

```
public void tick()
```

```
{
    // actualiza la posicion del robot
    // aplicando la velocidad de las ruedas.
    // vectorU y posicion son variables globales
    // que almacenan el estado.
```

```
ParejaDoubles step = new ParejaDoubles();
double angulo, radio, forward, lateral, ux, uy;
```

```
// simulación
```

```
step.x = velocidadRuedas.x/5.0;
step.y = velocidadRuedas.y/5.0;
//////////
```

```
if (step.x != step.y)
{
```

```
    angulo = (step.y-step.x)/ DISTANCIA_EJES;
    radio = (DISTANCIA_EJES/2)*(step.y+step.x)/(step.y-step.x);
    forward = radio*java.lang.Math.sin(angulo); //tx
    lateral = radio * (1-java.lang.Math.cos(angulo)); //ty
```

```
} else {
    angulo = 0;
    forward = step.x;
    lateral = 0;
}
```

```
ux=vectorU.x; //ux
uy=vectorU.y; //uy
posicion.x += java.lang.Math.round(forward*ux - lateral*uy);
posicion.y += java.lang.Math.round(forward*uy + lateral*ux);
```

```
vectorU.x = ux*java.lang.Math.cos(angulo) -
            uy*java.lang.Math.sin(angulo);
vectorU.y = dx*java.lang.Math.sin(angulo) +
            uy*java.lang.Math.cos(angulo);
```

```
}
```

- Modelado de los sensores de infrarrojos

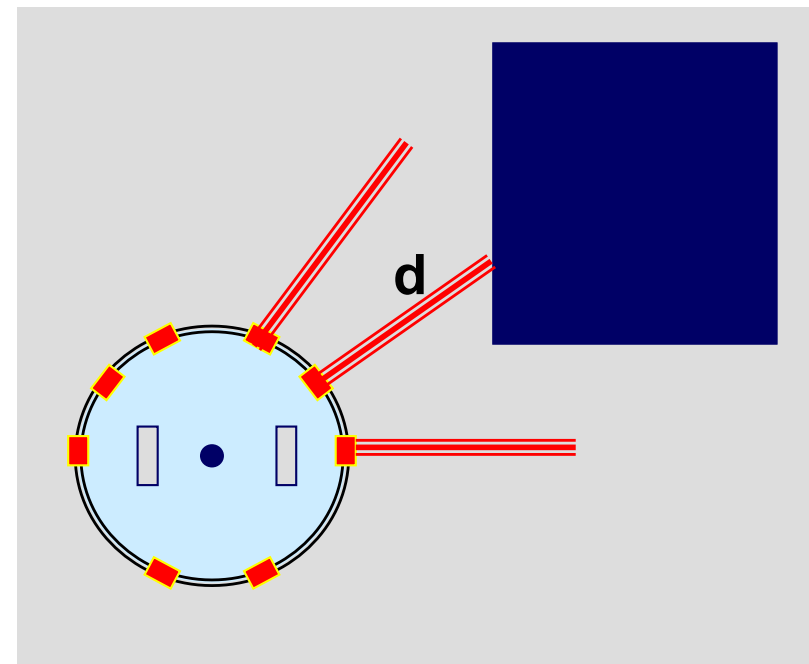
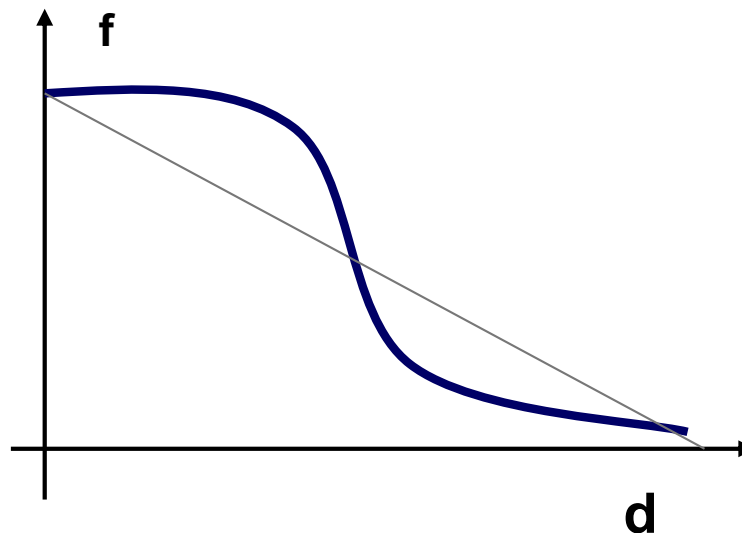
- La lectura responde a la ecuación:

$$f = 1 - 1/(1 + e^{\alpha + \beta d})$$

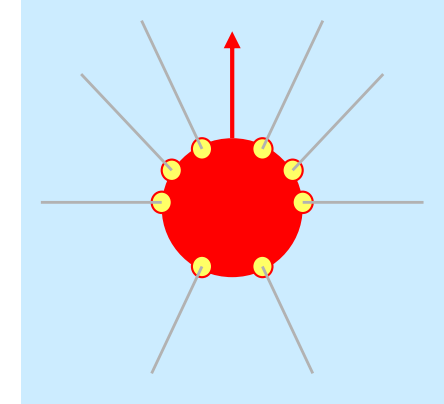
- Se aproximará por una recta:

$$a \cdot d + b \cdot f + c = 0;$$

- $a = -1023$
- $b = -40$
- $c = -51150$



- Representación gráfica del robot
 - El robot asume una representación gráfica por defecto como la indicada.
 - Para simplificar el dibujo de esta representación, la clase Khepera.Robot.Robot no solo proporciona la posición del centro del robot sino que también recalcula las coordenadas de todos los puntos de esta representación.
 - **Clase PuntosRobotD**
 - Derivada de la clase definida en IDL: **PuntosRobotD**
 - Define todos los puntos interesantes para representar gráficamente el robot: centro, puntos de inicio de los sensores (el puntero se considera el sensor 8), puntos finales de los sensores, intersección de los sensores con los obstáculos.

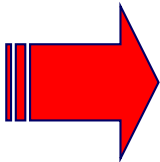


SDI Clase Khepera . Robot . PuntosRobot

```
final public class PuntosRobotD implements org.omg.CORBA.portable.IDLEntity
{
    public PuntosRobotD() { }
    public PuntosRobotD(corba.Khepera.Robot.PosicionD centro,
        corba.Khepera.Robot.PosicionD[] sens,
        corba.Khepera.Robot.PosicionD[] finsens,
        corba.Khepera.Robot.PosicionD[] inter)
    {
        this.centro = centro;
        this.sens = sens;
        this.finsens = finsens;
        this.inter = inter;
    }
    public corba.Khepera.Robot.PosicionD centro;
    public corba.Khepera.Robot.PosicionD[] sens;
    public corba.Khepera.Robot.PosicionD[] finsens;
    public corba.Khepera.Robot.PosicionD[] inter;
}
```

● Contenido

- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: CORBA y difusión
 - Interfaz IDL
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola



- La clase Escenario

Representa un espacio rectangular bidimensional en el que se mueven los robots.

- Define un conjunto de obstáculos, que son zonas en las que los robots no pueden penetrar.
 - Cada tipo de **obstáculo** (**rectángulo**, **círculo**, ...) se representa mediante una nueva clase y debe proporcionar métodos para calcular la intersección de un sensor (línea) con el obstáculo.
- La definición de los obstáculos se realiza a partir de un fichero y es gestionado por la cámara.
- Proporciona métodos para:
 - Definir los obstáculos a partir de un fichero
 - Detectar obstáculos: calcular la distancia al objeto más cercano del escenario.

```
public interface EscenarioInt {  
    // Sea un escenario con obstáculos  
    // Sea un segmento definido por los puntos "inicio" y "fin" que intersecta obstáculos  
    // Considere la intersección "X" del segmento con el obstáculo más cercano  
    // El siguiente procedimiento devuelve:  
    //     a) como valor de retorno: distancia a que esta X del inicio del segmento  
    //     b) en el parámetro inter: las coordenadas de X  
    float detectarObstaculos(Posicion inicio, Posicion fin, Posicion inter);  
}
```

```
public class Escenario implements EscenarioInt,Serializable,Cloneable {  
  
    // Constructores  
  
    // Constructor con un escenario predefinido  
    public Escenario() {...}  
  
    // Constructor a partir de la clase CORBA  
    public Escenario(EscenarioD esc) {...}  
  
    // Constructor con datos en fichero  
    public Escenario(String fichero) throws java.io.FileNotFoundException {...}  
  
    // Métodos públicos  
  
    public corba.Khepera.Escenario.EscenarioD toEscenarioD(){  
    public float detectarObstaculos(Posicion inicio, Posicion fin, Posicion inter);  
}
```



- Definición del escenario

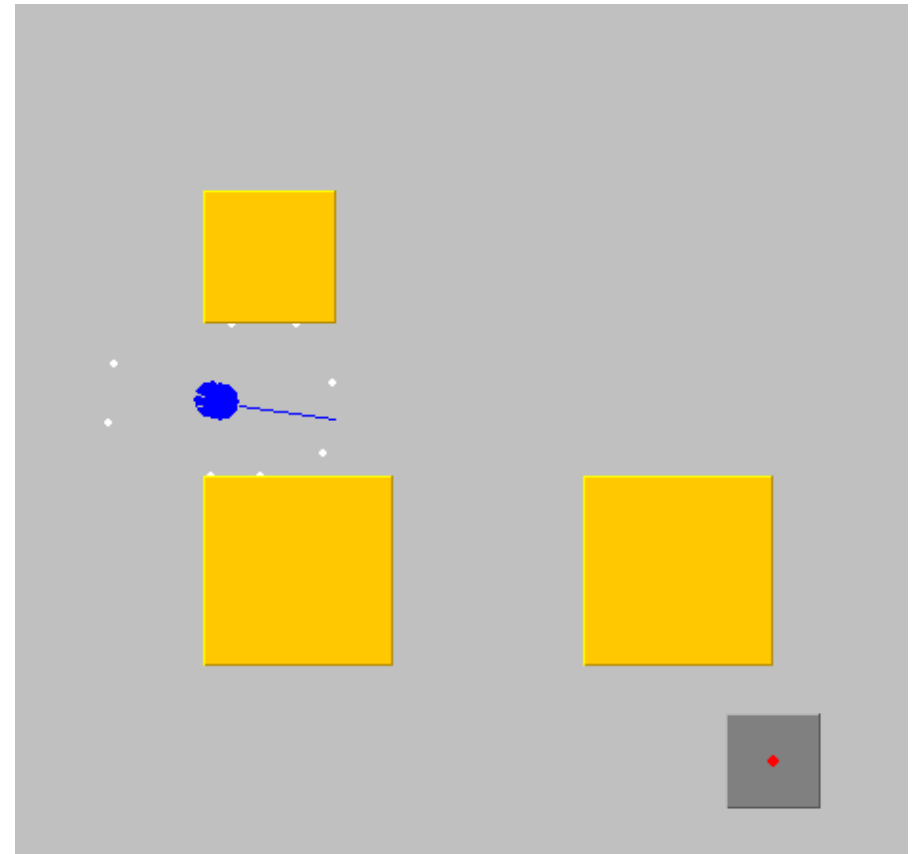
- El escenario es simulado y se define en un fichero cuyo formato es:

```
color tipo-figura punto-1 punto-2 ...
```

Ejemplo:

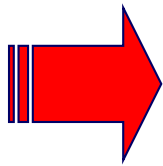
```
white rect 0 0 860 710  
orange rect 100 100 70 70  
orange rect 100 250 100 100  
orange rect 300 250 100 100  
gray rect 375 375 50 50
```

**El primero no se representa
y tiene las dimensiones del escenario**



● Contenido

- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: CORBA y difusión
 - Interfaz IDL
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola



● Evitación de obstáculos

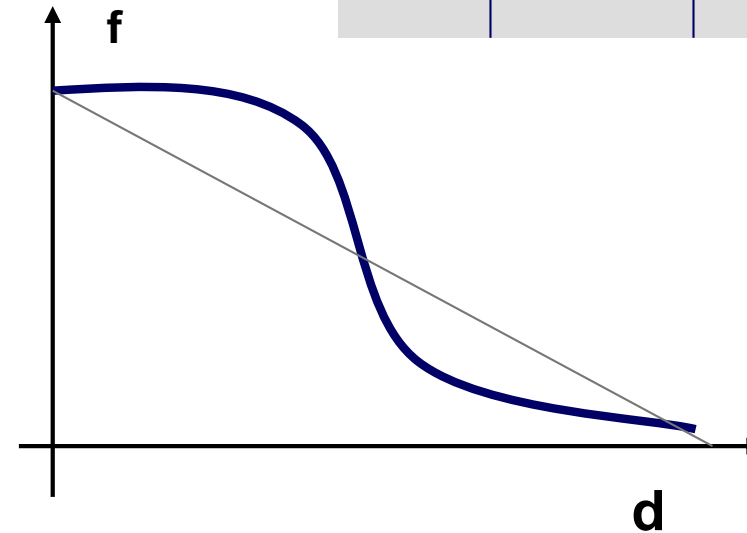
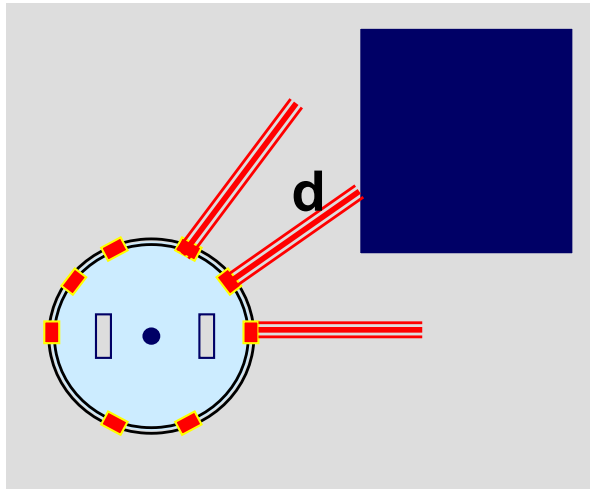
Algoritmo Braitenberg

La actuación sobre los motores es una suma ponderada de las lecturas de los sensores:

- $m_i = \sum K_{i,j} * f_j$
- $m_d = \sum K_{d,j} * f_j$

Constantes Braitenberg

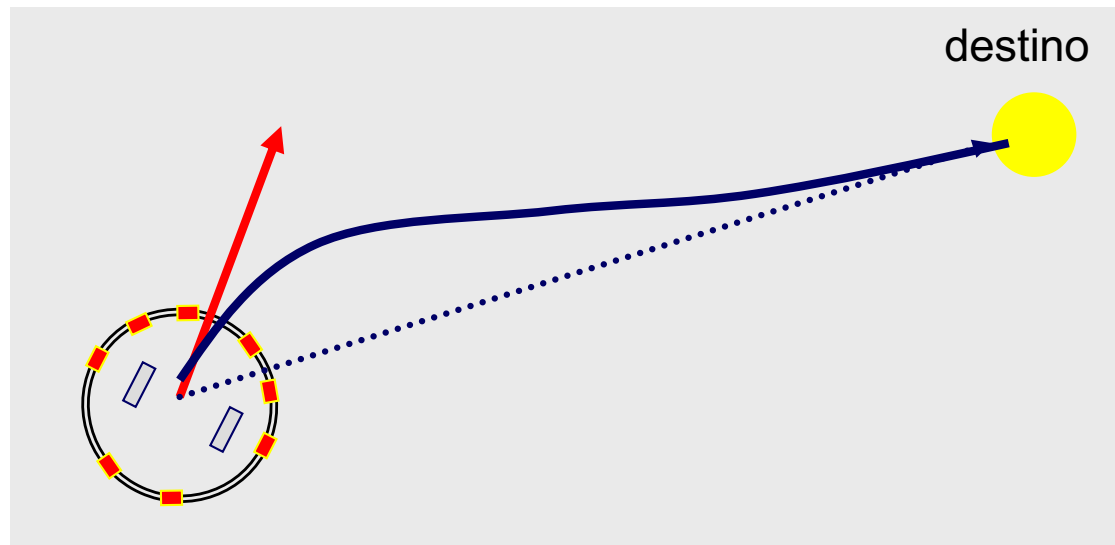
S	α	Ki	Kd
0	270	2	-3
1	306	4	-15
2	342	6	-18
3	18	-18	6
4	54	-15	4
5	90	-3	2
6	162	5	3
7	192	3	5



- Encaminamiento a un destino

Dos casos:

- Si esta “lejos”:
 - Describe una trayectoria suave (spline)
- Si está “cerca”:
 - orientarse (adoptar el ángulo) y
 - avanzar ambas ruedas a la misma velocidad



- Comportamiento global

La actuación sobre los motores es una suma ponderada de dos **motivaciones**:

- Evitar obstáculos: M1
- Dirigirse a un destino: M2

$$\text{Motor}_i = \sum K_{i,j} * M_i$$

$$\text{Motor}_d = \sum K_{d,j} * M_j$$

- Algoritmo de control
 - Implementa un determinado tipo de acción de control. Ejemplos:
 - Aproximación a un objetivo mediante una trayectoria suave (*spline*)
 - Repeler obstáculos mediante Braitenberg
 - Combinaciones de otros algoritmos.
 - Debe tener una interfaz bien definida para poder parametrizarlo.

Interfaz `Khepera.Control.Algoritmo`

```
public interface Algoritmo extends Serializable {  
    // Define el algoritmo de control (Braitenberg, Destino, ...)  
    // Devuelve la velocidad de las ruedas  
    // Toma como argumento un Object x que es distinto para cada algoritmo  
    // - para Destino es una Trayectoria  
    // - para Braitenberg es una vector de 8 floats con las lecturas de los sensores  
    IzqDer calcularVelocidad(Object x);  
}
```

SDI Clases que implementan Algoritmo

```
class Braitenberg implements Algoritmo {
```

```
    float sensor[] = new float[ConfRobot.NSENSORES];
```

```
    IzqDer vel = new IzqDer();
```

```
    public IzqDer calcularVelocidad(Object x){
```

```
        sensor= (float[]) x;
```

```
        vel.izq = 0;
```

```
        vel.der = 0;
```

```
        for (int i=0; i<ConfRobot.NSENSORES; i++){
```

```
            vel.izq += sensor[i] * ConfRobot.KBrai[i][0];
```

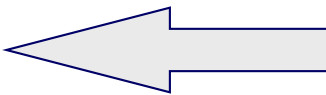
```
            vel.der += sensor[i] * ConfRobot.KBrai[i][1];
```

```
        }
```

```
        return vel;
```

```
    }
```

```
}
```

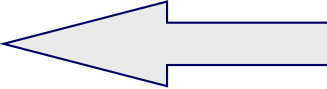


El objeto se interpreta como
un array con las lecturas
de los 8 sensores

SDI Clases que implementan Algoritmo

```
class Trayectoria{  
    Polares posicion = new Polares();  
    Posicion objetivo = new Posicion();  
    public Trayectoria(Polares pos, PosicionD obj){ ... }  
}
```

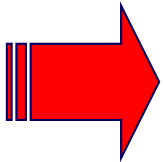
```
public class Destino implements Algoritmo{  
    Posicion objetivo = new Posicion();  
    Polares posicion = new Polares();  
    ...  
    public IzqDer calcularVelocidad(Object x) {  
        posicion= ((Trayectoria) x).posicion;  
        objetivo= ((Trayectoria) x).objetivo;  
        ...  
    }  
}
```



El objeto se interpreta como
la definición de una trayectoria

● Contenido

- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: CORBA y difusión
 - Interfaz IDL
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola



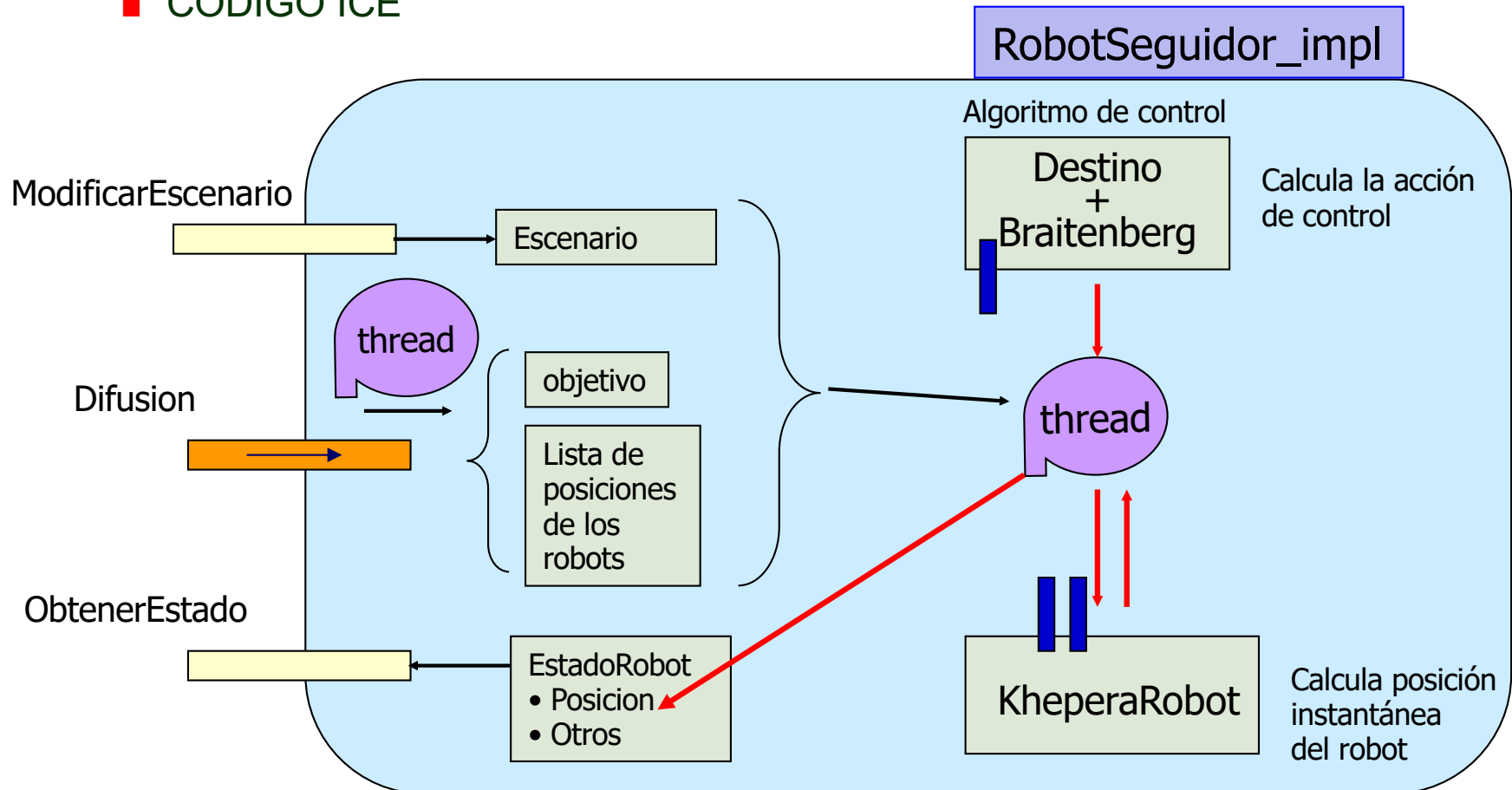
● RobotSeguidor

Es una especialización de **RobotKhepera**

- Su **comportamiento** consiste en intentar alcanzar un **objetivo** móvil sin salirse del escenario.
- El **objetivo** puede ser especificado al crear el robot o *en caliente* y puede ser de dos tipos:
 - Otro robot: es el caso de un *robot seguidor*
 - Un objetivo aleatorio: es el caso de un *robot líder*
- Realiza una **acción de control periódica**
 - Muestra las lecturas de los sensores
 - Calcula las velocidades de los motores utilizando el algoritmo de control
 - Programa las velocidades en los motores
- El **algoritmo** de control es fijo y es una combinación de “Braitenberg” y “Destino”.

- Estructura de la Implementación ICE

- RobotSeguidorI.java
 - CODIGO APLICACION
- RobotSeguidorIceServer.java
 - CODIGO ICE



Nota: Los dos threads pueden ser el mismo

- Detalles de implementación -> ALGORITMO DE CONTROL

```
private Khepera.escenario.EscenarioKhepera escenario;  
private RobotKhepera r;  
private Trayectoria tra;  
private Destino dst = new Destino();  
private Braitenberg bra = new Braitenberg();
```

```
escenario= new Escenario(sus.esc);  
r = new RobotKhepera(new PosicionD(0,0),escenario,0);
```

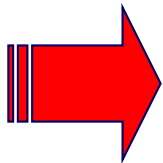
// ALGORITMO DE CONTROL

```
r.avanzar();  
mipos = r.posicionPolares();  
mipuntos = r.posicionRobot();
```

```
tra = new Trayectoria(mipos, miobj);  
float[] ls=r.leerSensores();  
nv = dst.calcularVelocidad((Object)tra);  
nv2 = bra.calcularVelocidad((Object) ls);  
nv.izq += nv2.izq/90; nv.der += nv2.der/90;  
r.fijarVelocidad(nv.izq,nv.der);
```

● Contenido

- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: CORBA y difusión
 - Interfaz IDL
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola



- La clase Cámara

Simula una cámara cenital con las siguientes funciones:

- **Gestor de grupo dinámico**

- Los robots buscan inicialmente una referencia a la cámara y se suscriben al grupo.
- La cámara proporciona, métodos para suscribirse y darse de baja del grupo.
- **Detecta fallos:** la cámara da automáticamente de baja aquellos robots que fallan.

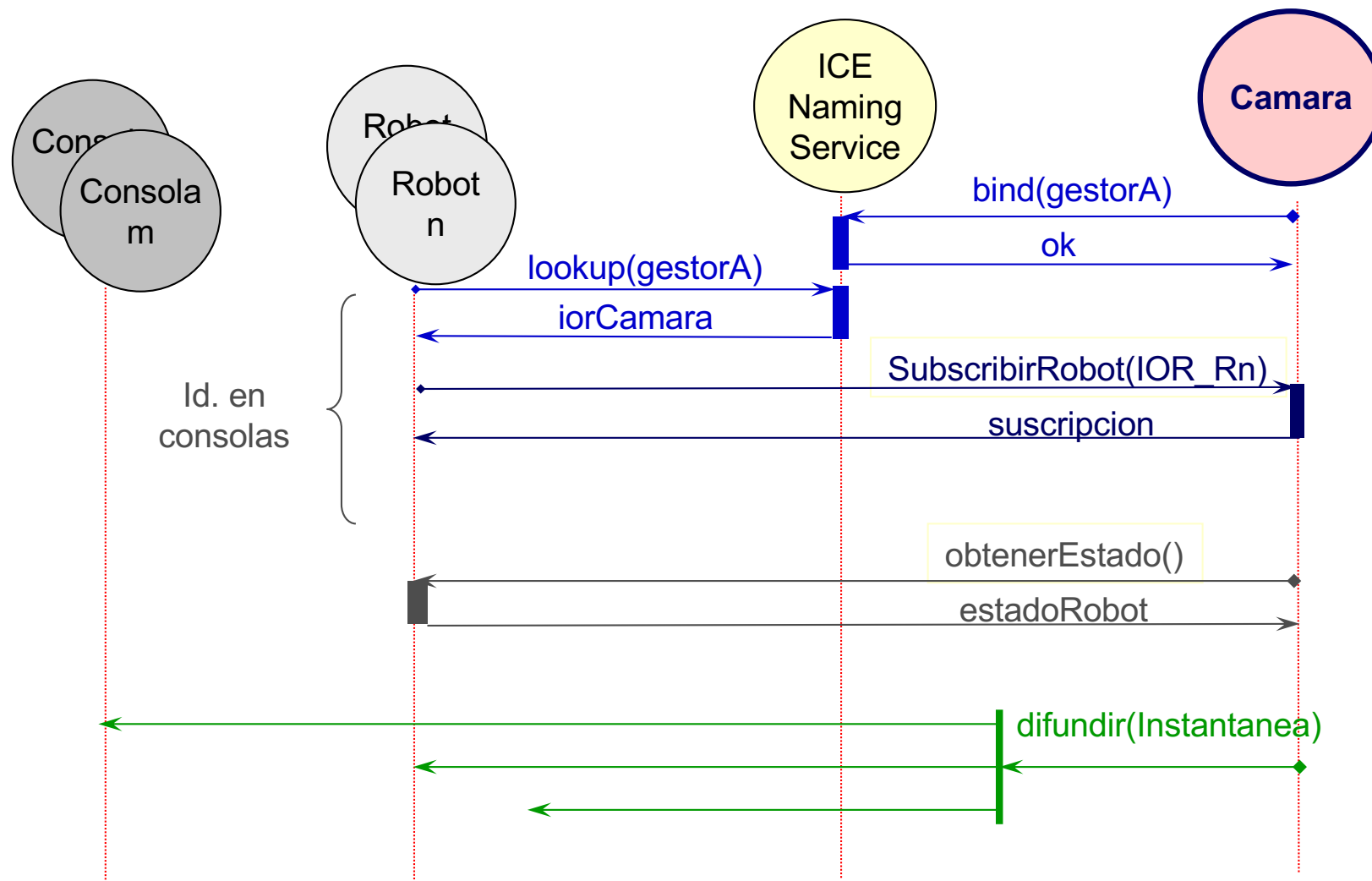
- **Difusor de imágenes:**

- Periódicamente muestrea las posiciones de los robots que integran el grupo y compone una **instantánea** con la posición de todos los robots en un instante de tiempo dado.
- Si durante este muestreo se observa que alguno de los robots no responde, entonces se excluye del grupo.
- Periódicamente difunde la instantánea del grupo a todos sus componentes. *Difundir la instantánea supone serializar una lista de objetos EstadoRobot.*

- **Gestión del escenario**

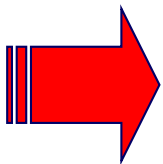
- Un usuario puede solicitar un cambio de escenario a una consola, la cual a su vez se lo comunica a la cámara vía la invocación de un método remoto.
- Si la cámara acepta el cambio de escenario, informa de ello a todos los componentes (Robots y Consolas) mediante la invocación de un método remoto. Puede requerir resetear todos los robots.

- Protocolo de gestión de componentes del grupo



● Contenido

- Definición del problema
 - Definición de componentes: objetos remotos
 - Comunicación remota: CORBA y difusión
 - Interfaz IDL
- El paquete Khepera
 - El robot Khepera
 - El escenario
 - El control del robot
- El paquete Robot
- El paquete Camara
- El paquete Consola

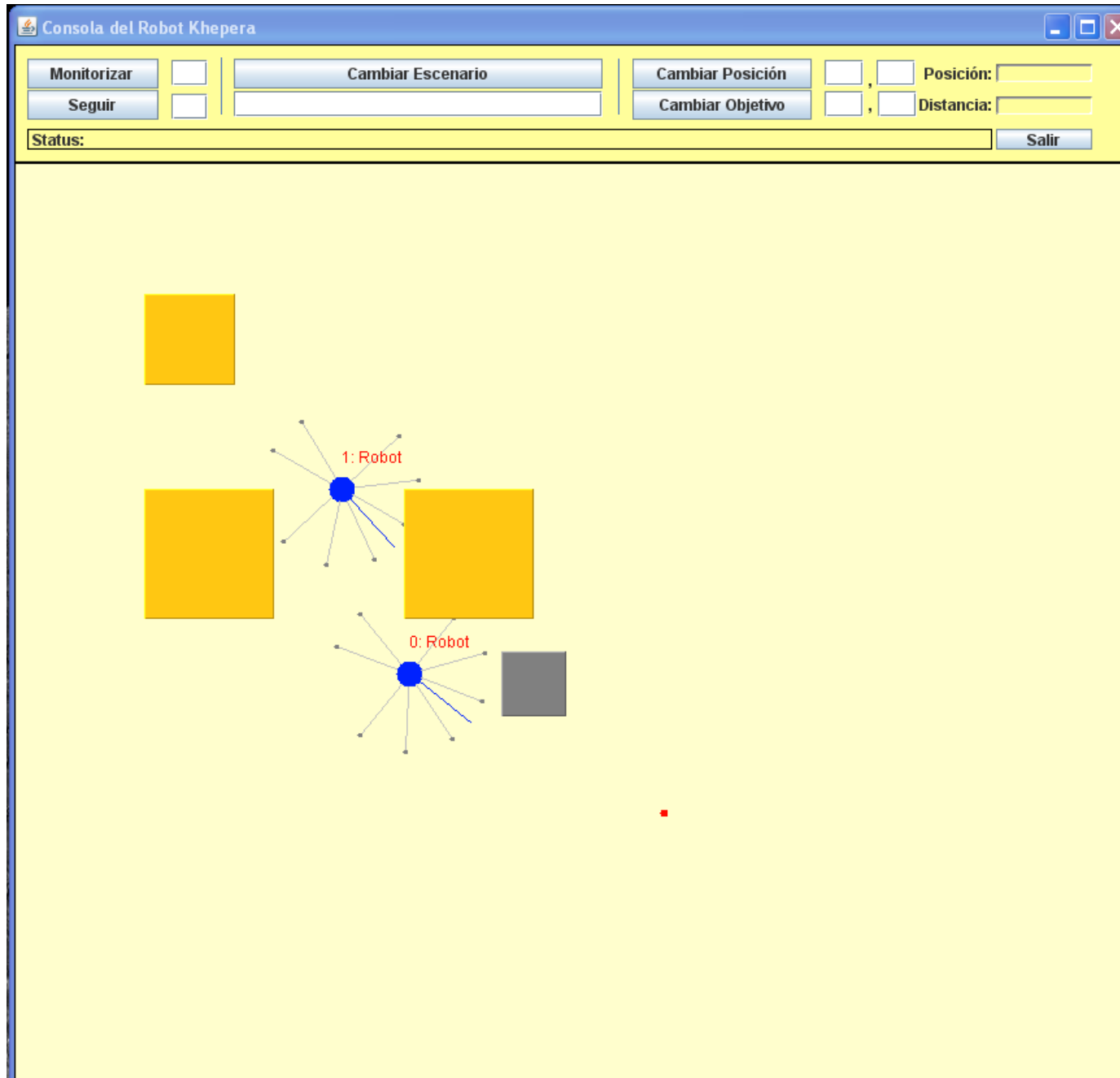


- Consola

Sistema gráfico que debe permitir las siguientes operaciones:

- Monitorizar el estado global del entorno.
- Monitorizar el estado de un robot concreto y cambiar su comportamiento.
- Mostrar:
 - El **escenario**
 - Los robots existentes
 - El **robot** monitorizado, su estado y objetivo
- También existen controles para especificar la ubicación de la cámara, el robot a monitorizar, cambiar el objetivo de un robot, ...
- Interfaz Web: Applet firmado

SDI La consola



- Interfaz de comunicación remota

- Asume que existe una cámara en la que se suscribe a la aplicación y a la que puede solicitar cambios de escenario:

```
interface Camara{  
    suscripcion SuscribirConsola(string IORcons);  
    void ModificarEscenario(in agencia::datos::Escenario::Escenario esc);  
}
```

- SuscripcionD le proporciona el canal de difusión y escenario inicial
- Asume que la cámara le notificará cambios de escenario (solicitados por otras consolas o la misma):

```
interface Consola{  
    void ModificarEscenario(in agencia::datos::Escenario::Escenario esc);  
};
```

- Asume que los datos a representar le llegarán vía difusión con el formato IDL

```
struct EstadoRobot {  
    string nombre;  
    string IORrob;  
    PuntosRobot puntrob;  
    agencia::datos::Posicion posObj;  
};  
struct Instantanea{  
    sequence<EstadoRobot> estadorobs;  
};
```

- Detalles de implementación
 - Realización de componentes gráficos (subclases de Component) para dar un aspecto gráfico a:
 - Robot → RobotGrafico
 - Escenario → EscenarioGráfico
- RobotGráfico
 - Representa el estado de un Robot remoto en la consola
 - Es un componente de la propia consola.
 - Funcionamiento:
 - Monitorización de la posición
 - Dibuja la imagen del robot en la posición obtenida a partir de la información difundida
 - Monitorización del estado
 - Solicita al robot remoto la información tanto de la posición y orientación, como del estado de los sensores en el instante actual.

- Robot / RobotSeguidor/ RobotGráfico

