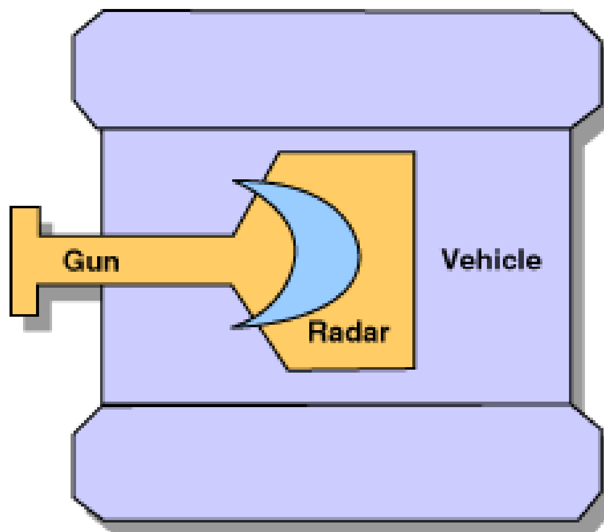
Práctica III: “RoboCode”

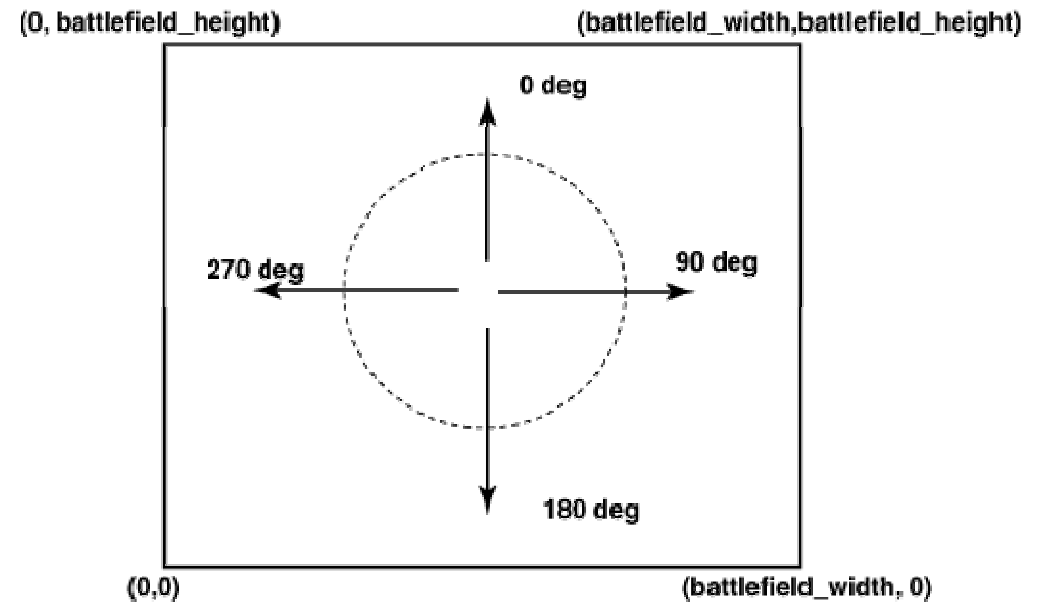
Introducción

- **Robocode** es un entorno de simulación de guerras de robots, desarrollado por Alphaworks de **IBM**®
- **Robocode** nos permite **programar tanques de combate en Java** para combatir en el campo de batalla contra tanques programados por otros jugadores.
- Existen dos modos de juego: batalla individual, en el que cada robot lucha contra todos los demás, y batalla en equipo, en el que un ejército de robots lucha por la victoria de modo colaborativo.
- En nuestro caso nos centraremos en la **batalla individual**.

Descripción del problema

- Anatomía de un robot y del terreno de combate:





Descripción del problema

- El objetivo del juego es generar la I.A. que permita a un robot derrotar a otros robots.
- **Robocode** cuenta con multitud de métodos para gestionar los disparos y los ataques a nuestros enemigos.
- Al principio de cada combate todo robot comienza con un nivel de energía por defecto, y un robot muere cuando su energía disminuye hasta 0.
- La energía de un robot puede disminuir por distintas causas:
  - Alcance de un disparo enemigo
  - Colisión con un robot enemigo
  - Colisión con paredes u obstáculos del campo de batalla
- Adicionalmente, la energía liberada en el disparo también resta energía al robot.

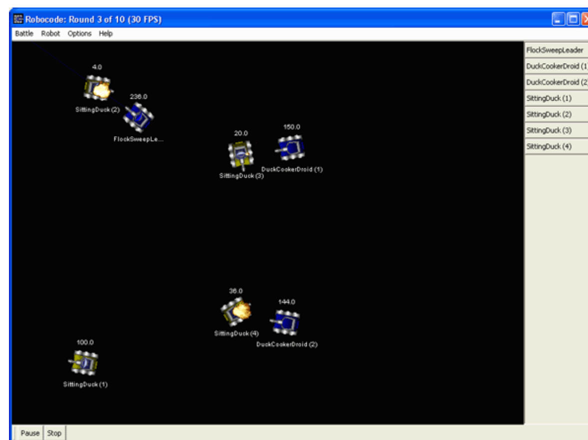
Descripción del problema

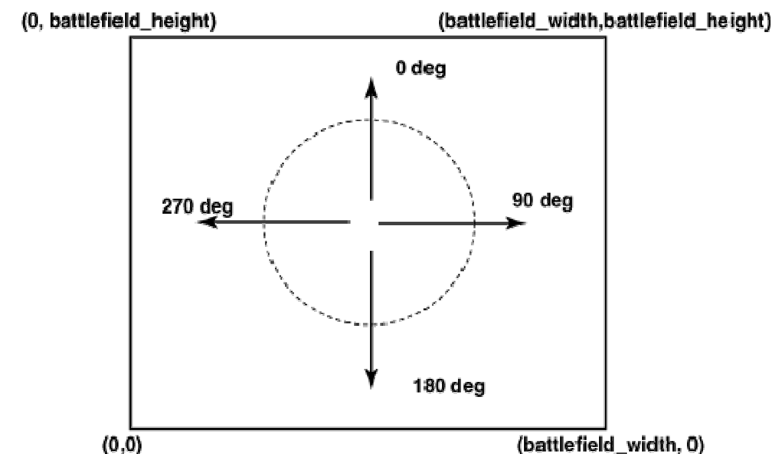
- Cuanta más energía se utilice en un disparo, mayor daño se infligirá al robot enemigo alcanzado, pero también más energía restará a nuestro tanque.

- Pero no todo son pérdidas de energía, también se recupera energía cuando se alcanza a algún enemigo, o bien, de forma constante por el efecto del "enfriamiento de los cañones".

Descripción del sistema

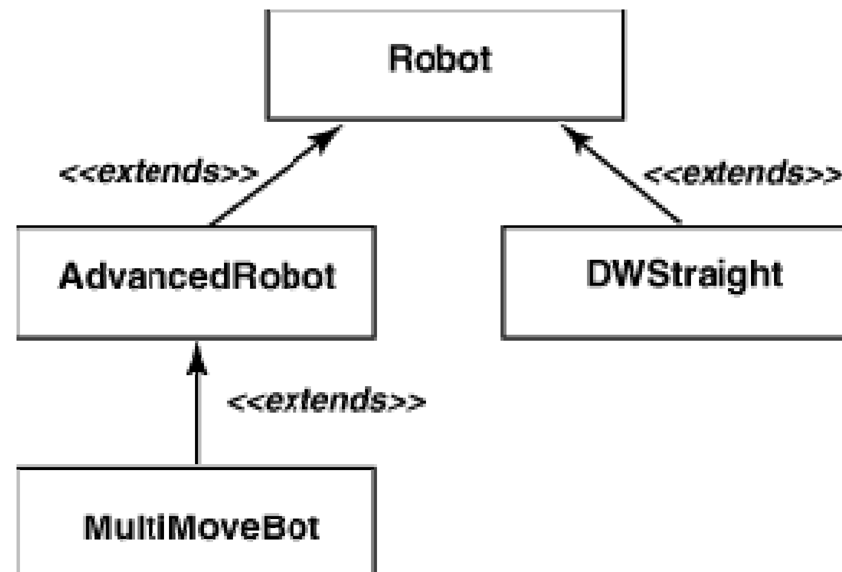
- En Robocode hay dos elementos principales, los **robots** y las **batallas**. Las batallas se llevan a cabo en el terreno de combate entre los robots, que juegan por sí mismos bajo una programación concreta que define su comportamiento.





Descripción del sistema

- Todas las clases generadas en Robocode extienden de la clase **`robocode.Robot`**, con métodos que permiten interactuar con el juego.



Ejemplo de robot

```
package <nombre equipo>;
import robocode.*;
import java.awt.Color;

public class TerminatorI extends Robot
{
      public void run() {
          setColors(Color.red,Color.blue,Color.green);
          while(true)
          {
                    // Remplazar las siguientes líneas por el comportamiento deseado
                    ahead(100);
                    turnGunRight(360);
                    back(100);
                    turnGunRight(360);
          }
      }

      public void onScannedRobot(ScannedRobotEvent e) {
          fire(1);
      }

      public void onHitByBullet(HitByBulletEvent e) {
          turnLeft(90 - e.getBearing());
      }
}
```

Descripción del sistema

- Durante el combate se generan "eventos", estos eventos son situaciones que se producen a partir de las cuales el robot puede decidir hacer unas cosas u otras.
- Un robot **básico** tiene por defecto manejadores de eventos.
- Todo tanque heredera de la clase **`Robot`**, estos métodos se pueden sobrescribir y así añadir funcionalidad a los robots para cuando dichos eventos se disparen.

Descripción del sistema

- A la hora de programar los robots hay que tener en cuenta 3 áreas fundamentales del código:
  1. Un área en la que se definen las variables de clase, disponibles dentro del método `run()`, así como en el resto implementados.
  2. El propio método `run()`, que es llamado por el gestor de combate para comenzar la vida del robot y típicamente se divide en dos áreas:
     - Área en la que se definen las cosas que sólo se harán una vez por cada instancia del robot.
     - Área dentro de un `while` infinito en la que se define la acción repetitiva en la que se verá envuelto el robot.
  3. Métodos auxiliares para usar por el robot dentro de su método *`run()`*. En esta zona también se ponen los manejadores de eventos que se quieran implementar.

Descripción del sistema

Órdenes que se pueden utilizar:

- public void **ahead**(double distancia)
  - Mueve el robot hacia delante la distancia pasada por parámetro. Se ejecuta inmediatamente y no devuelve nada hasta que haya finalizado el movimiento. En el caso de que el robot colisionara con algún obstáculo se detendría.
- public void **fire**(double potencia)
  - Dispara una bala, el rango válido de potencia va desde 0.1 a 3. La bala sigue el trayecto al que apunte el cañón. Si se alcanza al enemigo causará el daño equivalente a (4*potencia). Se ejecuta inmediatamente. Además, la energía que se le otorgue a la bala, se restará automáticamente de la energía del propio robot.

Descripción del sistema

Órdenes que se pueden utilizar:

- `public void turnLeft(double grados)`
  - Rota el tanque/robot "n" grados a la izquierda. Se ejecuta inmediatamente, no devuelve nada hasta que finaliza.
- `public void turnRight(double grados)`
  - Rota el tanque/robot "n" grados a la derecha. Se ejecuta inmediatamente, no devuelve nada hasta que finaliza.
- `public void turnGunLeft(double grados)`
  - Rota el cañón del robot y el radar "n" grados a la izquierda. Se ejecuta inmediatamente, no devuelve nada hasta que finaliza..
- `public void turnGunRight(double grados)`
  - Rota el cañón del robot y el radar "n" grados a la derecha. Se ejecuta inmediatamente, no devuelve nada hasta que finaliza.

Descripción del sistema

Órdenes que se pueden utilizar:

- `public double getHeading()`
  - Devuelve la dirección a la que apunta el frente del robot en grados (de 0 a 360).

Descripción del sistema

- Los eventos que se podrán utilizar son los siguientes:
  - **ScannedRobotEvent**: se dispara cuando el radar detecta un robot a su paso por una zona angular.
  - **BulletHitEvent:** se lanza cuando nuestro robot alcanza con un disparo a uno de los enemigos.
  - **HitRobotEvent**: se lanza cuando nuestro robot colisiona con un robot enemigo.
  - **HitWallEvent**: se dispara cuando el robot choca contra un muro de los que limitan el campo de batalla.
  - **HitByBulletEvent:** se lanza cuando nuestro robot es alcanzado por un disparo.

Descripción del sistema

De la clase `ScannedRobotEvent` se pueden utilizar:

- `public double` **`getBearing`**`()`
  - Devuelve el ángulo (en grados) del enemigo en relación al ángulo que apunta tu tanque/robot. Devuelve valores entre -180º a 180º. Por ejemplo, -90 grados (si mi tanque mira al norte, 0º), sería lo mismo que decir que lo tengo a la izquierda.

- `public double` **`getDistance`**`()`
  - Devuelve la distancia al enemigo.

Descripción del sistema

De la clase **`BulletHitEvent`** se puede utilizar:

- `public double` **`getEnergy`**`()`
  - Devuelve la energía restante del adversario descontada la del cañonazo.

Descripción del sistema

De la clase **`HitRobotEvent`** se pueden utilizar:

- `public double` **`getBearing()`**
  - Devuelve el ángulo (en grados) del enemigo contra el que se ha chocado en relación al ángulo que apunta tu tanque/robot. Devuelve valores entre -180º a 180º. Por ejemplo, -90 grados (si mi tanque mira al norte, 0º), sería lo mismo que decir que lo tengo a la izquierda.

- `public double` **`getEnergy`**`()`
  - Devuelve la energía del enemigo tras la colisión.

Descripción del sistema

De la clase **`HitWallEvent`** se pueden utilizar:

- `public double` **`getBearing`**`()`
  - Devuelve el ángulo (en grados) del obstáculo con el que se colisiona en relación al ángulo que apunta el tanque/robot. Devuelve valores entre -180º a 180º.

Descripción del sistema

De la clase **`HitByBulletEvent`** se pueden utilizar:

- `public double` **`getBearing`**`()`
  - Devuelve el ángulo (en grados) del enemigo en relación al ángulo que apunta tu tanque/robot. Devuelve valores entre -180º a 180º. Por ejemplo, -90 grados (si mi tanque mira al norte, 0º), sería lo mismo que decir que lo tengo a la izquierda.

Ejemplos

```
public void onScannedRobot(ScannedRobotEvent enemigo)
{   If (enemigo.getDistance() < 100)
       fire(3);
    else
       fire(1);
}

public void onHitWall(HitWallEvent e)
    {
       moveDirection *= -1;
    }
```

Descripción del sistema

Resultados:

Se almacenan en un archivo, el usuario decide cual al lanzar **robocode** con el parámetro –result <nombre>.

El formato es el siguiente:

- Total Score – total de puntos (sumado de todas las rondas) para cada robot ordenado de mejor a peor (daño infringido a los enemigos).
- Survival Score – cada vez que otro robot muere y el nuestro sobrevive, almacena 50 puntos automáticamente en esta cuenta.
- Last Survivor Bonus – el último robot en sobrevivir almacena 10 puntos por cada robot eliminado antes que él.
- Bullet Damage – se otorga un punto por cada punto de daño (disparo) que se realice a un enemigo.
- Bullet Damage Bonus – cuando un robot elimina a un enemigo suma un 20% del daño que le hizo a ese enemigo.

Ejemplo de robot

```
package <nombre equipo>;
import robocode.*;
import java.awt.Color;

public class TerminatorI extends Robot
{
      public void run() {
           setColors(Color.red,Color.blue,Color.green);
           while(true)
           {
                      // Remplazar las siguientes líneas por el comportamiento deseado
                     ahead(100);
                     turnGunRight(360);
                     back(100);
                     turnGunRight(360);
           }
      }

      public void onScannedRobot(ScannedRobotEvent e) {
           fire(1);
      }

      public void onHitByBullet(HitByBulletEvent e) {
           turnLeft(90 - e.getBearing());
      }
}
```

Objetivos

- **Objetivo**: se desea implementar un robot competitivo que sea capaz de derrotar a los adversarios independientemente de la estrategia que implementen.
- La modalidad de juego es “melee”: tu tanque compite contra otros 9 (pueden ser iguales o distintos). El campo de batalla será de 1000x1000.
- **Dudas:**
  - ¿Sería positivo implementar técnicas de IA para alcanzar al enemigo?
  - ¿Sería interesante desarrollar mecanismos para evadir cañonazos enemigos?
  - ¿Sería bueno basar nuestra estrategia en la cantidad de energía restante en cada momento?.
  - ¿Quedarse en el centro o en las esquinas?
  - **Etc..**

Instrucciones de instalación

1. Descargar **robocode** desde:

   http://et.evannai.inf.uc3m.es/docencia/cb/transparencias.html

2. Instalar el paquete:

3. Java –jar robocode-setup-1.2.2.jar

Instrucciones de instalación

4. Crear archivo .bat a partir de robocode.bat con el siguiente contenido:

a) Líneas de compilación de nuestros robots (adaptar path java)

.\jikes-1.22\bin\jikes.exe -deprecation -g -Xstdout +T4 -classpath "C:\Archivos de programa\Java\jre1.5.0/lib/rt.jar";robocode.jar;robots ".\robots\ysa\RamboI.java"

.\jikes-1.22\bin\jikes.exe -deprecation -g -Xstdout +T4 -classpath "C:\Archivos de programa\Java\jre1.5.0/lib/rt.jar";robocode.jar;robots ".\robots\ysa\TerminatorI.java"

b) Línea que lanza al simulador Robocode minimizado, en el terreno de juego yago.battle y que guarda los resultados en el fichero "resultadoCombate"

java -Xmx512M -Dsun.io.useCanonCaches=false -jar robocode.jar -battle ./battles/yago.battle -results resultadoCombate -minimize

5. Ejecutar .bat y observar que no da errores

Nota: el fichero yago.battle define el terreno de batalla y debe existir previamente

Instrucciones de instalación

6. A jugar!!!!

Práctica III: valoración

1. La valoración de la práctica consiste en probar el sistema de IA que presente el alumno frente a enemigos que presentará el profesor y frente al mejor enemigo encontrado por sus compañeros.
2. La nota final proviene de los resultados obtenidos (en varias batallas y contra otros robots) y de la valoración de los mecanismos implementados.
3. Fecha límite y día de la evaluación:

18 de enero de 2007

Práctica III

1. Resolver el problema mediante algunas de las técnicas vistas durante el curso.
2. Entregar una memoria breve comentando las técnicas implementadas, las distintas pruebas realizadas los problemas encontrados y los resultados obtenidos.
3. El lenguaje de programación será libremente seleccionado por el/los alumnos.

Normas de la asignatura

- Las prácticas se pueden realizar conjuntamente con un máximo de 2 alumnos por práctica.
- Cada alumno es responsable de la presentación de sus prácticas.
- Una práctica presentada fuera de plazo quedará automáticamente suspensa.

Normas entrega práctica

1. Entregar memoria antes de finalizar la clase.
2. Enviar código fuente por correo electrónico antes de las 17:15 del día de entrega a *yago.saez@uc3m.es* con las siguientes instrucciones:
   a) Asunto: nombre alumnos (practica3)
   b) Contenido email:
      i. Mejores resultados obtenidos en la práctica contra robots de prueba
      ii. Archivo Zip con el código fuente y leeme.txt con las instrucciones para compilarlo y ejecutarlo.

Ejemplo de llamada al Robocode

Ejemplo de llamada al Robocode (hay otras muchas formas)

```
public static void ejecutarBatalla() {

        Runtime r = Runtime.getRuntime();
        try {
                Process p=r.exec("C:\\robocode\\prueba.bat");
                StreamGobbler errorGobbler = new
                StreamGobbler(p.getErrorStream(), "ERROR");
                StreamGobbler outputGobbler = new StreamGobbler(p.getInputStream(), "OUTPUT");
                errorGobbler.start();
                outputGobbler.start();
                int exitVal = p.waitFor();
                if (exitVal != 0) System.out.println("Process exitValue: " + exitVal);
        } catch (Throwable t) {
                t.printStackTrace();
        }
    }
```

Ejemplo de llamada al Robocode

```
import java.io.*;
class StreamGobbler extends Thread {
    InputStream is;
    String type;
    OutputStream os;
    StreamGobbler(InputStream is, String type) {
        this(is, type, null);
    }
    StreamGobbler(InputStream is, String type, OutputStream redirect) {
        this.is = is;
        this.type = type;
        this.os = redirect;
    }
    public void run() {
        try {
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            while (br.readLine() != null) { }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }}}
```