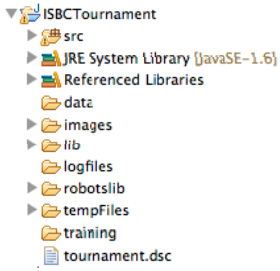
Programación en SBTournament

Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid

Preparación

- Descomprimir el proyecto de Eclipse e importarlo en Eclipse



- Para lanzar la aplicación:
  es.ucm.fdi.gaia.SBTournament.SBTournament

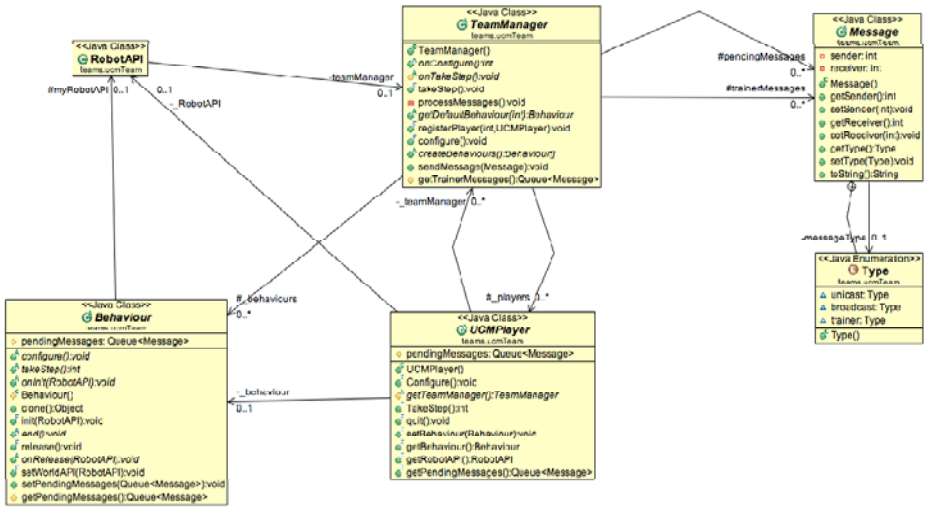
Importante:

- **El proyecto ha de estar en un directorio cuya ruta no tiene espacios en blanco ni caracteres especiales**
- **Cada robot estará en un paquete t1112<num. Grupo>**

Creación de robots

- En el SoccerBots original, los robots se programan extendiendo la clase SocSmallSS.
- Esta clase tiene métodos sensores y actuadores.
- En la práctica **no** extenderemos esta clase
  - Limita a que todos los robots se comporten igual
  - O, si no, tenemos que hacer un IF inicial para determinar el comportamiento.
- En su lugar utilizaremos unas clases propias: librería UCMTeam

Clases a utilizar



Clases a utilizar

- Para implementar los robots utilizaremos las clases del paquete teams.ucmTeam

Package teams.ucmTeam

Class Summary	
Behaviour	The interface that represents a robot behaviour.
Message	A generic message that can be sent by the UCMTeam robots.
RobotAPI	This is the robot's interface with the world.
TeamManager	The team manager is the object that controls the team.
UCMPlayer	UCMTeam is a Control System that represents a team whose robots are controlled based on predefined behaviours.

Enum Summary	
Message.Type	

- Un equipo se implementa extendiendo la clase **UCMPlayer**
- Define diferentes roles **Behaviour** para cada tipo de jugador
  - En esta clase irá todo el código del comportamiento
- La clase Behaviour tiene acceso al **RobotAPI**
  - RobotAPI contiene los métodos sensores y actuadores.
- El **TeamManager** se utiliza para cambiar el comportamiento de los jugadores y coordinar el juego en equipo

Pasos a realizar

1. Definir al menos un comportamiento
   - Extender de la clase Behaviour
2. Definir nuestro entrenador
   - Extender la clase TeamManager
   - Crear los comportamientos y asignarlos a los jugadores
     - createBehaviours()
     - getDefaultBehaviour(int id)
3. Definir nuestro equipo
   - Extender de la clase UCMPlayer
   - Crear el entrenador
     - getTeamManager()

1. Behaviour

- Los Behaviour implementan el comportamiento de un jugador

Method Summary	
java.lang.Object	clone()
abstract void	configure() Configures the role.
abstract void	end() Informs the behaviour the match is over.
void	init(RobotAPI r) Initializes a behaviour with the RobotAPI of the robot that the behaviour will control
abstract void	onInit(RobotAPI r) Abstract method executed during the init method.
abstract void	onRelease(RobotAPI r) Abstract method invoked by release method.
void	release() Releases the behaviour.
void	setPendingMessages(java.util.Queue<Message> pendingMessages) Updates the message queue including the new pending messages
void	setWorldAPI(RobotAPI r) Establishes the robot controlled by this behaviour
abstract int	takeStep() Decides the actions that the robot will perform during the current step.

- Internamente se dispone de un atributo RobotAPI que permite acceder a los sensores y actuadores.
- El método más importante es takeStep() ← comportamiento

1. Behaviour

- Es obligatorio implementar:
  - configure()
    - Se invoca al principio del partido.
    - No se dispone de RobotAPI
  - onInit(RobotAPI)
    - Se invoca cada vez que activamos / establecemos un comportamiento en un jugador
  - takeStep()
  - onRelease(RobotAPI)
    - Se invoca cada vez que desactivamos / cambiamo un comportamiento en un jugador
  - end
    - Se invoca al final del partido
    - No se dispone de RobotAPI

1. Behaviour

- Intenta colocarse detrás de la bola apuntando a la portería contraria.
- Si puede → dispara

Es un agente reflexivo, basado en reglas, sin estado, ni objetivos, ni aprendizaje

```
public class GoToBall extends Behaviour {

        public void configure() { }

        public int takeStep() {
            myRobotAPI.setBehindBall(myRobotAPI.getOpponentsGoal());
            if (myRobotAPI.canKick())
                myRobotAPI.kick();
            return myRobotAPI.ROBOT_OK;
        }

        public void onInit(RobotAPI r) {
                r.setDisplayString("goToBallBehaviour");
        }

        public void onRelease(RobotAPI r) {   }

        public void end() { }
}
```


2. TeamManager

- El TeamManager se encarga de:
  - Crear y almacenar los comportamientos
  - Asignar comportamientos a los jugadores
  - Gestionar el juego en equipo

Method Summary

void	**configure**() Initialization of the team manager.
abstract Behaviour[]	**createBehaviours**() Abstract method that creates an array with all the available behaviours that the team manager can use during a match.
abstract Behaviour	**getDefaultBehaviour**(int id) Returns the initial default behaviour of the player specified by the id.
abstract int	**onConfigure**() This method is invoked by `configure` method before any behaviour configuration.
void	**registerPlayer**(int id, UCMTeam r) Stores a reference of the robot that will work as the player identified by id in the team
void	**sendMessage**(Message message) Stores a message sent by a robot.
void	**takeStep**() Runs the team manager every time step

2. TeamManager

- Cada Behaviour tendrá acceso a esta clase a través de la RobotAPI
  - getTeamManager()
- Contiene
  - Array de jugadores (UCMPlayer [] _players)
  - Array de comportamientos (Behaviour[] _behaviours)
- Es obligatorio implementar:
  - onConfigure(): Se invoca durante el configure, justo antes de createBehaviours.
  - createBehaviours(): Ha de crear un array no vacío de comportamientos.
  - getDefaultBehaviour(int id): comportamiento inicial del jugador id
  - onTakeStep(): invocado desde el método takeStep

2. TeamManager

- Ejemplo de implementación de un TeamManager

```
public class Entrenador extends TeamManager {
        public int onConfigure() {
                return RobotAPI.ROBOT_OK;
        }

        public void onTakeStep() {
                // No hacemos nada
        }

        public Behaviour getDefaultBehaviour(int id) {
                return  behaviours[0];
        }

        public Behaviour[] createBehaviours() {
                Behaviour[] behav =new Behaviour[1];
                behav[0]=new GoToBall();
          return behav;
        }
}
```


3. UCMPlayer

- El equipo en sí se crea extendiendo de la clase UCMPlayer
- Hay que implementar el método abstracto que devuelve el TeamManager

Method Summary	
void	**Configure**() Configures the team and the robot.
Behaviour	**getBehaviour**() Gets a reference to the behaviour that controls this robot
java.util.Queue<Message>	**getPendingMessages**() Gets the queue of pending messages for this robot
RobotAPI	**getRobotAPI**() Gets a reference to the robotAPI employed to execute actions on this robot
void	**quit**()
void	**setBehaviour**(Behaviour b) Changes the behaviour on this robot
int	**TakeStep**() TakeStep delegates in the behaviour of the corresponding robot

3. UCMPlayer

- Ejemplo de implementación de un equipo

```
public class TestPlayer extends UCMPlayer {

        @Override
        protected TeamManager getTeamManager() {
                return new Entrenador();
        }
}
```


RobotAPI

- Un Behaviour tiene acceso a la RobotAPI del robot que está controlando
- RobotAPI permite:
  - Acceder a las dimensiones del campo, robots...
  - Acceder a información del partido: tiempo de juego, marcador...
  - Cambios de coordenadas egocéntricas ←→ globales
  - Acceder a sensores y actuadores
    - De bajo y alto nivel
    - Pases y heurísticas de pase = experimental
  - Acceder al TeamManager del equipo.
  - Crear / abrir ficheros
    - createFile(Class<?> classRequester,String filename)

Comunicación entre robots

- Message
  - Sender: id del robot que lo envía
  - Receiver: id del robot al que va dirigido (opcional)
  - Tipo de mensaje
    - Unicast: de sender a receiver
    - Broadcast: de sender al resto de robots
    - Trainer: de sender a trainer (TeamManager)

Extender de la clase Message para meter contenido del mensaje

Method Summary

int	**getReceiver**() Returns the id of the message receiver
int	**getSender**() Returns the id of the message sender
Message.Type	**getType**() Returns the message type (unicast,broadcast,trainer)
void	**setReceiver**(int receiver) Sets the id of the message receiver
void	**setSender**(int sender) Sets the message sender id
void	**setType**(Message.Type type) Sets the message type (unicast,broadcast,trainer)
java.lang.String	**toString**()

Comunicación entre robots

Comunicación entre robots

- Asíncronos
  - Se reciben en el siguiente takeStep
  - TeamManager hace de “estafeta de correos”
- Enviar un mensaje
  - Crear el mensaje y configurarlo → msg
  - sendMessage(msg)
    - Si lo envía el TeamManager
  - myRobotAPI.getTeamManager().sendMessage()
    - Si lo envía el Behaviour
- Procesar mensajes recibidos
  - En el takestep (del TeamManager o del Behaviour)
  - Queue<Message> getPendingMessages()

Orden de llamadas de nuestros métodos

- Inversión de control
  - SBTournament llama a los métodos de las clases que hemos creado
- Al comenzar un partido (solo una vez)
  - UCMPlayer::getTeamManager
  - TeamManager::onConfigure
  - TeamManager::createBehaviours
  - Behaviour::configure
    - Una vez por cada comportamiento creado
  - TeamManager::getDefaultBehaviour
    - Una vez por cada jugador
    - Behaviour::onInit

Orden de llamadas de nuestros métodos

- Cada vez que ejecutamos _players[N].setBehaviour
  - Si el comportamiento que queremos activar es distinto (equals) del comportamiento que ya está activado
    - Behaviour::onRelease del comportamiento que desactivamos
    - Behaviour::onInit del comportamiento que activamos
- Cada vez que se produce un ciclo de simulación
  - Se envían los mensajes a jugadores / entrenador
  - TeamManager::onTakeStep
  - Para cada uno de los comportamientos activos de los jugadores
    - Behaviour::takeStep
- Cuando termina el partido
  - Behaviour::end