Guía de uso de UCMTeam

UCMTeam es un paquete de clases desarrollados por miembros del grupo de investigación GAIA de la Facultad de Informática de la Universidad Complutense de Madrid y becarios asociados al Proyecto de Innovación y Mejora de la Calidad Docente (PIMCD 179 – 2011). Este paquete permite construir robots que pueden ser utilizados en Soccerbots, un entorno de simulación de partidos de fútbol entre robots. Entre las principales características de UCMTeam destacan la siguiente:

- Los robots de UCMTeam se basan en comportamientos intercambiables durante la ejecución del partido.
- Un equipo UCMTeam tiene un Team manager, responsable de crear todos los comportamientos disponibles durante la ejecución de un partido, así como de asignar dichos comportamientos a los distintos robots del equipo durante el partido.
- Existe una comunicación entre los integrantes del equipo mediante envío de mensajes, tanto dirigidos al Team manager como a los distintos robots que componen el equipo.

Creación de un equipo 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	

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

1. Primero hay que definir los diferentes comportamientos que los jugadores del equipo podrán tener. Esto se realiza creando subclases de la clase Behaviour. En esta clase irá todo el código del comportamiento de un robot individual haciendo uso de la RobotAPI, que contiene los métodos sensores y actuadores.
2. El TeamManager se utiliza para crear los comportamientos, coordinar el juego del equipo y para asignar los comportamientos a los robots. Para ello, crearemos una subclase de TeamManager e implementaremos sus métodos abstractos.
3. Por último, crearemos una subclase de UCMPlayer. Esta clase se encarga de crear nuestro TeamManager.

Vamos a ver más en detalle cada uno de estos pasos.

Creación de las subclases de Behaviour

Los Behaviours implementan el comportamiento del 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 un Behaviour dispone de un atributo myRobotAPI que permite acceder a los sensores y actuadores del robot que está controlando en un determinado momento. Los métodos de la clase Behaviour se invocan en el siguiente orden:

- El método configure se llama antes de empezar el partido. En este momento podemos hacer inicialiaciones asociadas al comportamiento. Hay que tener en cuenta que en este momento **no se tiene acceso a la RobotAPI** por lo que cualquier invocación de un método de la misma, fallará. Este método se llama una sola vez.
- El método init se ejecuta cuando se asigna un comportamiento a un robot concreto. En este caso, realiza una inicialización del comportamiento que puede tener en cuenta el estado del robot, ya que tiene acceso a la RobotAPI. Este método puede se llamar múltiples veces durante la ejecución de un partido, cada vez que se cambia el comportamiento de los robots.
- El método más importante es takeStep() ya que es donde se implementa el comportamiento. Se invoca en cada ciclo de simulación una vez que el comportamiento ha sido asignado a un robot concreto.
- El método release se ejecuta cuando se quita el comportamiento de un robot. Puede hacer uso de la información del estado del robot durante este proceso. Este método puede se llamar múltiples veces durante la ejecución de un partido, cada vez que se cambia el comportamiento de los robots.
- El método end se ejecuta una única vez, una vez finalizada la ejecución del partido.

Para crear una subclase de Behaviour hay que implementar sus métodos abstractos, que son:

- configure.
- onInit: llamado desde el método init
- takeStep.
- onRelease: llamado desde el método release

- end

Los comportamientos se crean haciendo uso de los sensores y actuadores que proporciona RobotAPI. Consulta la documentación de esa clase para conocer más al respecto.

Veamos un ejemplo de comportamiento:

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

```
public class GoToBall extends Behaviour{

     @Override
     public void configure() {
          // No hacemos nada
     }

     @Override
     public int takeStep() {
          myRobotAPI.setBehindBall(myRobotAPI.getOpponentsGoal());
          if (myRobotAPI.canKick())
               myRobotAPI.kick();
          return myRobotAPI.ROBOT_OK;
     }
     @Override
     public void onInit(RobotAPI r) {
          r.setDisplayString("goToBallBehaviour");
     }

     @Override
     public void end() {
          // No hacemos nada
     }

     @Override
     public void onRelease(RobotAPI r) {
          // No hacemos nada
     }
}
```

Creación de un Team Manager

El TeamManager se encarga de crear los comportamientos y gestionar el juego en equipo asignando los comportamientos creados a los robots. Cada Behaviour tendrá acceso al Team Manager del equipo a través de su RobotAPI usando el método getTeamManager(). Los comportamientos creados por el TeamManager se guardan en el array _behaviours mientras que las referencias a los jugadores se guardan en _players.

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

Los métodos de la clase TeamManager se invocan en el siguiente orden:

- El método configure se invoca una sola vez antes de empezar el partido.
- Durante la configuración se crearán los comportamientos invocando el método createBehaviours. En este método es donde se creará un array con todos los posibles comportamientos que los jugadores del equipo podrán tener.
- Cuando se crea un robot se registrará en el TeamManager mediante el método registerPlayer. Además se invocará al método getDefaultBehaviour para establecer el comportamiento que el jugador registrado tendrá por defecto al principio del partido.
- El método takeStep que se invoca en cada ciclo de la simulación (antes de llamar al takeStep de cada uno de los jugadores del equipo). En él se pueden cambia los comportamientos de los jugadores usando el método setBehaviour de UCMPlayer.

Una subclase de TeamManager ha de implementar los siguientes métodos:

- onConfigure: Se invoca durante el configure, justo antes de createBehaviours.
- createBehaviours. Ha de crear un array no vacío de comportamientos.
- getDefaultBehaviour(int)
- onTakeStep: invocado desde el método takeStep

Veamos un ejemplo de implementación de TeamManager. En este ejemplo el TeamManager no realiza ninguna acción, sino que simplemente crea un comportamiento por defecto para todos jugadores del equipo (GoToBall).

```
public class Entrenador extends TeamManager {

    @Override
    public int onConfigure() {
        return RobotAPI.ROBOT_OK;
    }

    @Override
    public void onTakeStep() {
        // No hacemos nada
```

```
    }

    @Override
    public Behaviour getDefaultBehaviour(int id) {
        return _behaviours[0];
    }

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

}
```

Creación del UCMPlayer

Para terminar el equipo, crearemos una subclase de UCMPlayer. En ella solo tenemos que implementar el método abstracto responsable de crear el TeamManager (getTeamManager).

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

Aquí vemos un ejemplo de esta subclase:

```
public class TestPlayer extends UCMPlayer {

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

Ejemplo de un equipo con varios comportamientos

Vamos a hacer un pequeño ejemplo con un equipo que usa varios comportamientos. Primeramente creamos un nuevo comportamiento

```
public class Blocker extends Behaviour{
```

```
	@Override
	public void configure() {
		// No hacemos nada
	}

	@Override
	public int takeStep() {
		myRobotAPI.blockGoalKeeper();
		return RobotAPI.ROBOT_OK;
	}

	@Override
	public void onInit(RobotAPI r) {
		r.setDisplayString("BlockerBehaviour");
	}

	@Override
	public void end() {
		// No hacemos nada
	}

	@Override
	public void onRelease(RobotAPI r) {
		// No hacemos nada
	}
}
```

A continuación, modificaremos el takeStep() de nuestra clase Entrenador para que cambie el comportamiento de uno de los jugadores en función de su posición en el campo. Recordemos que también tendrá que crear el nuevo comportamiento:

```
public class Entrenador extends TeamManager {

	@Override
	public int onConfigure() {
		return RobotAPI.ROBOT_OK;
	}

	@Override
	public void onTakeStep() {
		// Si el jugador esta en su campo --> GoToBall
		RobotAPI robot = _players[2].getRobotAPI();
		if (robot.getPosition().x * robot.getFieldSide()>=0)
			_players[2].setBehaviour(_behaviours[0]);
		else
		// E.o.c. --> Blocker
			_players[2].setBehaviour(_behaviours[1]);
	}

	@Override
	public Behaviour getDefaultBehaviour(int id) {
```

```
            return _behaviours[0];
        }

        @Override
        public Behaviour[] createBehaviours() {
                  return new Behaviour[] {new GoToBall(),
                                          new Blocker()};
        }
}
```

Comunicación entre robots

UCMTeam dispone de comunicación entre los robots y el entrenador mediante el paso de mensajes. Los tipos de mensajes que se pueden enviar son personalizables. Para ello es necesario extender la clase Message

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**()

El emisor y (opcionalmente) el receptor son el id del robot que envía / recibe el mensaje. El tipo del mensaje puede ser:

- Unicast: El mensaje se envía exclusivamente al robot identificado por id.
- Broadcast: El mensaje se envía a todos los robots.
- Trainer: El mensaje se envía exclusivamente al TeamManager

El envío de un mensaje se realiza a través del TeamManager y su método sendMessage(). El envío de mensajes es asíncrono, es decir, el mensaje no llega inmediatamente al receptor sino que queda guardado en el TeamManager hasta la próxima vez que éste ejecute su takeStep.

Los mensajes recibidos por un robot se almacenan en una cola. El behaviour tiene acceso a ellos mediante el método getPendingMessages. Así mismo, existe un método similar en TeamManager para procesar los mensajes cuyo destinatario era el TeamManager. La cola de mensajes se vacía automáticamente al finalizar cada takeStep.

Como ejemplo vamos a crear un mensaje de prueba que contiene una posición de interés:

```
public class MensajePosicion extends Message{

    private Vec2 posicion;
```

```
     public Vec2 getPosicion() {
          return posicion;
     }

     public void setPosicion(Vec2 posicion) {
          this.posicion = posicion;
     }
}
```

El entrenador puede enviar un mensaje a alguno de sus jugadores diciendo a qué posición tiene que ir:

```
public void onTakeStep() {
     // Manda al jugador 0 a la portería contraria
     RobotAPI robot0 = _players[0].getRobotAPI();
     MensajePosicion message = new MensajePosicion();
     message.setReceiver(0);
     message.setType(Type.unicast);
     message.setPosicion(robot0.toFieldCoordinates(robot0.getOpponents
Goal()));
     sendMessage(message);
}
public Behaviour getDefaultBehaviour(int id) {
     if (id==0)
          return _behaviours[2];
     else
          return _behaviours[0];
}
public Behaviour[] createBehaviours() {
     return new Behaviour[] {new GoToBall(),
                             new Blocker()
                             new GoToPosition()};
};
```

Ahora solo nos faltaría crear un nuevo comportamiento GotoPosition que procesa la cola de mensajes y, si recibe el mensaje de ir a una posición, va hacia esa posición.

```
public class GoToPosition extends Behaviour {
     public int takeStep() {
          Queue<Message> pendingMessages = this.getPendingMessages();
          if (!pendingMessages.isEmpty()) {
               for (Message m:pendingMessages) {
                    if (m instanceof MensajePosicion){
                       Vec2 pos = (((MensajePosicion)m).getPosicion());
                       myRobotAPI.setSteerHeading(pos.t);
                       myRobotAPI.setSpeed(1.0);
                       // Miro cómo de lejos estoy
                       Vec2 dist = pos;
                       dist.sub(myRobotAPI.getPosition());
                       if (dist.r<0.2){
```

```
                    myRobotAPI.setSpeed(0.0);
                    myRobotAPI.setDisplayString("Arrive");
                  }
                }
              }
          }
          return 0;
    }

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

Resumen de tareas a realizar

Como toma de contacto con el desarrollo de robots para SBTournament te recomendamos que hagas las siguientes tareas:

1. Descarga el material de la práctica: proyecto de eclipse del CV + guía de uso de UCMTeam.
2. Define una subclase de Behaviour para el comportamiento GotoBall (el código de esta clase está en la pg. 3 de la guía)
3. Define una clase Entrenador que extienda el TeamManager y asigne el comportamiento GotoBall a todos los jugadores (pg 4)
4. Define la clase del equipo que extiende UCMPlayer y que se encarga de crear el entrenador (pg 5)
5. Prueba este equipo en el entorno SBTournament
6. Haz algunos cambios en este comportamiento, por ejemplo, cambiar la velocidad de los jugadores, o añadir el control de bloqueos
7. Define nuevos comportamientos. Tienes como ejemplo el comportamiento Blocker en la guía (pg 5 y 6) pero puedes añadir cualquier otro comportamiento sencillo que se te ocurra.
8. Modifica la clase Entrenador según se indica en la guía (pg 6) para que cambie el comportamiento de uno de sus jugadores: por ejemplo, para el jugador 2 si está en su campo se comportará según GotoBall, y si está en el campo contrario será Blocker.
9. Modifica la clase Entrenador para que cambie el comportamiento de todos sus jugadores según alguna condición.
10. Define un nuevo comportamiento que utilice el paso de mensajes entre el entrenador y los robots. Por ejemplo, el comportamiento GotoPosition incluido en la pg 8 de la guía de uso.
11. Prueba el equipo desarrollado en esta sesión creando un torneo (usa muy pocos equipos para que sea corto). ¿Cómo ha quedado el equipo en el torneo? ¿Puedes hacerlo mejor? Demuestralo.