INGI 1131 - Programming language concepts

INGI of Empires

Autors:
Léonard Debroux
Sébastien Wilmet

Teacher:
Prof. Peter Van Roy

December 12, 2011

We implemented all the features that were asked in the statements. All the code is split into several functors: one for each port object, the brain, the configuration containing the constants, the GUI, some utilities functions, and the "main" (`INGI-of-Empires.oz`). There are several brains for testing purposes.

1 General Design

To explain the general design of the code, we will go trough the "main function" and explain the important steps. First, we need map.

The map - The map is stored in a text file that we read and process to create a tuple `board` containing several tuples `row` that represent the lines of the map. This allows to easily know every square type.

Team Creation - To create teams, we use the function `GenerateTeams` that returns a tuple containing one record for each team. It knows how many teams to create by checking the number of homes on the map. Each team record contains the ID of the team (`TeamNum`), a reference to its home which will be explained next and the position of the home.

Home - When creating one team, the method creates one port object `Home` and binds it to the user interface (UI) in order to update the number of resources. The state of the `Home` contains only the number of resources. We "repeat" all the states in a port, and the UI listens the corresponding stream to update the numbers displayed as soon as they change.

The Graphical User Interface - The next step is to create the GUI using the class `Gui` we adapted. The `init` method needs the map and the number of teams.

The Square objects - One `Square` port object is created for each square of the map. To create them, we need the map and the teams. The state contains many things which will be explained later. When `CreateAllSquares` has finished, a tuple is created like we did for the map but with port objects. Then the squares are bound to the UI, like we did for the home resources.

Initialization of the game - The initial towers and players are created. To create a new player, we first create the `Player` port object with the function `Player.create`. Then we create the brain, with `Brain.createBrain`. Finally we bind the player with its brain with the function `Player.embody`. This function calls the brain, sends the action to the player, waits that the player has finished its action, and then recall the brain, and so on (all this in a separated thread obviously).

2 Port objects

Now that we explained how the program gets started, we'll detail a little the three port objects used in this program.

2.1 Home

Its purpose is to know the amount of resources a team possesses at any time. As explained above, the state contains only the number of resources. Only a player sends messages to a home. Here are the messages:

`getNbResources(?Res)`
`addResources(Res)`
`removeResources(Res ?OK)` Removes Res if they are available.

2.2 Player

The state of a player contains its current position, its bag, whether he has a weapon and whether he is dead.

The messages received come only from the function `Player.embody` (i.e. come from the brain). A message has the form:

`Action#?BrainEnv#?Dead`

After performing the action, we bind the two variables `BrainEnv` (the next brain environment), and `Dead` if the player died. The possible actions are those described by the brain specification available on iCampus.

2.3 Square

The state contains:

- A tuple containing the number of players for each team, split into two categories: with and without a weapon;
- If the square contains a home, the `Home` port object;
- If the square contains a tower, the tower description, which contains the owner (the `TeamNum`) and the number of points;
- A list of the "near" towers. A near tower can kill a player;
- A list of the visible towers;
- A tuple containing, for each team, a list of the players currently exploiting the resource available on the square.

The possible messages:

`playerIn(...)`	
`playerOut(...)`	
`buildTower(... ?OK)`	Try to build a tower and notify the neighbours.
`weakenTower(Strength ?OK)`	If the tower was already destroyed, bind OK to false.
`towerBuilt(...)`	A tower has been built in the neighbourhood.
`towerDestoryed(...)`	
`getVisibleTowers(...)`	
`beginExploit(...)`	A player begins to exploit the resource.
`endExploit()`	
`getHomePort(?HomePort)`	

2.4 Communication between port objects

Two simple protocols are used. The first one consist in sending a message and wait nothing in return, for example `playerOut()`.

The second protocol used is the RMI: when a message is sent, we give an unbound variable, and we wait that it is bound. All the messages with a parameter prefixed with "`?`" use the RMI protocol.