OGP Assignment 2011-2012: RoboRally (part 2)

This text describes the second part of the assignment for the course 'Object-oriented Programming'. The groups that were formed for the first part should preferably work together on the second and third part. Students that worked alone on the first part are encouraged to find another team member for the second part. The deadline for submitting a new team composition to ogp-project@cs.kuleuven.be is March 28 at 11.59pm. New groups cannot be created after this deadline for the second and third part of this assignment. It is not necessary to send an email if the composition of your group did not change. However, if you submit a new team composition, put all affected (old and new) team members in CC.

If problems arise within a group, the group can split and each team member is required to complete the assignment on its own. If a group splits, this must be reported to ogp-project@cs.kuleuven.be.

The first part of this assignment focussed on a nominal, defensive and total programming of methods within a single class. In the second part, we extend the first part with additional classes and relationships between these classes. Note that certain aspects of the class Robot described in part one change in part two.

1 Assignment

RoboRally is a board game where each player plays the role of a robot. The goal of the game is to move the robot from its starting position to the goal position as fast as possible. The first player to reach the goal wins the game. In this assignment, we will create a game loosely based on this board game.

The goal of the first part of the assignment was to program the class `Robot`. In this second part, we will add a game board, walls, batteries and shooting. Note that if the assignment does not specify how to work out a certain aspect of the game, select the option you prefer.

1.1 Boards

A board is two-dimensional, rectangular grid containing robots, walls and batteries. Each board has its own size, described by a width and height. The size

of a board cannot be modified after construction. Both the width and height must lie in the range 0 to `Long.MAX_VALUE` (both inclusive) for all boards. In the future, the upper bound on the width and height may decrease. However, all boards will share the same upper bound.

At each position, a board can contain robots, walls and batteries. However, a robot, wall, or battery can be located on at most one board. Moreover, no board contains the same robot, wall or battery twice. All aspects related to the placement of walls, robots and batteries on boards should be worked out defensively.

Placing a robot, wall or battery on a board at a certain position must take amortized constant time. Removing a particular robot, wall or battery is preferably linear in the number of robots, walls or batteries located at the same position. Moreover, it should be possible to look up all robots, walls and batteries at a particular position in constant time. The amount of memory required to store a `Board` should be proportional to the number of positions in use, not to the total size of the board.

The class `Board` should provide a method to merge a board with another board. Merging means that all robots, walls and batteries from the second board are moved (whenever possible) to the first board at the corresponding positions and that the second board is afterwards terminated. Each robot, wall and battery on the second board that does not have a corresponding position on the first board or that cannot be placed alongside an existing robot or wall on the corresponding position on the first board need not be moved and is instead terminated together with the second board.

It must be possible to terminate boards, robots, walls and batteries. When a board is terminated, all robots, walls and batteries on that board must be terminated as well. Similarly, when a robot is terminated, all items it is carrying are terminated as well. A board should not contain terminated robots, walls or batteries.

All documentation for the class `Board` must (only) be worked out in a formal way. In fact, this applies to all classes in the project, except for the class `Robot`, whose documentation must be worked out both formally and informally.

1.2 Walls

A wall is an object that can be placed at a certain position on the board. If the board contains a wall at certain position, then that position can contain no robots, batteries or other walls. The class `Wall` should provide a method to inspect its position. Note that it is possible for a wall not to be located on any board.

Walls must not be worked out by students who do the project on their own.

1.3 Batteries

Robots can use batteries to replenish their energy. Each battery holds a certain amount of energy expressed in watt-second (Ws). The maximum amount of

energy that can be stored in a battery is 5000 Ws. However, in the future this limit can change and need not remain the same for each battery. All aspects related to the energy of a battery should be worked out nominally.

Each battery has a certain weight expressed in gram (g). The weight does not depend on the amount of energy stored in the battery. The weight of an item never exceeds `Integer.MAX_VALUE`. All aspects related to the weight of a battery must be worked out in a total way.

Batteries can be placed on a board at a certain position or can be carried by a robot. However, a battery cannot both be owned by a robot and be located on a board at the same time.

Note that it is possible that a battery is not owned by any robot and is not placed on a board. The class Battery should provide a method to inspect its position on a board. However, there is no requirement for a method to query the robot that is currently carrying a battery. In fact, batteries may not store an explicit reference to the robot that is carrying them. Each battery can only be carried by a single robot.

1.4 Robots

Each player in our game plays the role of a robot.

1.4.1 Energy

As before, each robot holds a certain amount of energy expressed in watt-second (Ws). As we will explain in Sections 1.4.2, 1.4.3 and 1.4.4, turning, moving and shooting consumes energy. Currently, we will assume that the maximum amount of energy for each robot is 20000 Ws. However, in the future this limit can change and need not remain the same for each robot. All aspects related to energy of a robot should be worked out nominally.

The class `Robot` should provide methods to (1) query the amount of energy in watt-second, (2) query the amount of energy as a fraction of the maximum amount and (3) recharge the robot with a certain amount of energy.

1.4.2 Orientation

Each robot faces a certain direction: up, right, down or left. All aspects related to orientation should be worked out in a total way. The class `Robot` should provide a method to turn a robot 90 degrees in clockwise or counter-clockwise direction. Turning a robot 90 degrees consumes 100 watt-second. Moreover, the class `Robot` should provide a method to inspect its orientation.

1.4.3 Position

Robots can be placed at a certain position on a board. However, each position on the board can contain at most one robot. The class `Robot` should provide a method to inspect its position. It is possible for a robot not to be located on a board.

The class `Robot` should provide a method to move a robot forward. For example, if a robot is located on position (5, 10) facing left, then the `move` method should move the robot to position (4, 10). Moving a robot consumes 500 watt-second plus 50 watt-second per kilogram of items the robot is carrying.

In addition, there must be a method that returns the minimum amount of energy required to reach a particular position only by moving and turning taking into account the robot's current energy level, obstacles and the robot's current load. This method should not take into account shooting, picking up, using and dropping items. If the robot has insufficient energy to reach the position, this method must return -2. If you want to achieve a score of 17 or more, this method must take into account obstacles and the fact that turning requires energy; otherwise, you only have to minimize the number of moves (i.e. do not take into account the fact that turning requires energy) and you may assume (via a precondition) that there are no other robots or walls in the rectangle that covers both the robot and the target position. In any case, this method should return -1 if the robot is not placed on a board. If you want to achieve 17 or more, also return -1 if the target position is not reachable because of obstacles.

Finally, `Robot` should provide a method to move a robot next to another robot. When implementing this method, take into account the following constraints:

- Both robots should be located on the same board.
- The robots should move as close as possible (expressed as the manhattan distance) to each other (without ending up on the same position) given their current energy. Assuming that the robots have sufficient energy, they end up next to each other.
- Both robots can move and turn to make sure they end up next to each other. Shooting, picking up, using and dropping items is not allowed within this method.
- At the end of this method, the robots should be as close as possible (expressed as the manhattan distance) to each other given their current energy and load on the board. If you want to achieve a score of 18 or more, this method must take into account obstacles and the fact that turning requires energy. Thus, if there exist multiple, potential, final positions for which the distance is minimal, a pair must be selected that minimizes the energy consumption. Otherwise (less than 18), you only have to move as close as possible taking only into account the energy required for moving (i.e. do not take into account the fact that turning requires energy) and you may assume (via a precondition) that there are no other robots or walls in the rectangle that covers both the robot and the target position. In this case, if there exist multiple, potential, final positions for which the distance is minimal, a pair must be selected that minimizes the number of moves required.

1.4.4 Shooting

A robot can shoot its laser in the direction it is facing thereby consuming 1000 Ws. Starting from the robot and moving in the direction of its current orientation, the first position containing one or more robots, walls or batteries is hit. A random robot, wall or battery located at that position is destroyed and must be removed from the board. If the robot is not placed on a board, shooting has no effect.

Shooting must not be worked out by students who do the project on their own.

1.4.5 Picking up, using and dropping items

A robot can pick up, use and drop items. A robot can only pick up an item if the robot is placed on the board and the item is located at the same position on the board. When an item is picked up, it should be removed from the board and added to the possessions of the robot. Robots must be able to return the ith heaviest item they are carrying in constant time.

A robot can use any item it possesses. When a robot uses a battery, energy transfers from the battery to the robot. Either the battery is drained or the robot reaches its maximum energy. If the battery the robot is trying to use is terminated, the battery must be discarded by the robot.

A robot can drop an item. If the robot is placed on a board, the item must be placed at the position where the robot is located; otherwise, the item simply has no owner and is not located on any board.

You can select yourself how you work out aspects related to picking up, using and dropping items.

2 Testing

Write a JUnit test suite for the classes `Robot` and `Board` that tests each public method. Testing the classes `Wall` and `Battery` is optional. Include this test suite in your submission.

3 User Interface

We provide a new graphical user interface (GUI) to visualize the effects of various operations on board, robots, walls and batteries. The user interface is included in the assignment.

To connect your implementation to the GUI, write a class `Facade` that implements `IFacade`. `IFacade.java` contains additional instructions on how to implement the required methods. To start the program, run the `main` method in the class `RoboRally`. After starting the program, you can enter commands to modify the state of the program. These modifications are shown in a separate window.

You can freely modify the GUI as you see fit. However, the main focus of this assignment are the classes at the core of the application: `Robot`, `Board`, `Wall`, ... No additional grades will be awarded for changing the GUI.

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `IFacade`. As described in the documentation of `IFacade`, the methods of your facade implementation may print error messages to `System.err`, but should not throw exceptions (unless specified otherwise in the documentation of `IFacade`).

4 Submitting

The solution must be submitted via Toledo as a jar file individually by all team members before April 27th 2012 at 11.59 pm. You can generate a jar file on the command line or using eclipse (via `export`). Include all source files (including tests) and the generated class files. Include your name and studies in comments in your solution. When submitting via Toledo, make sure to press `OK` until your solution is submitted!