OGP Assignment 2011-2012: RoboRally (part 3)

This text describes the third part of the assignment for the course 'Object-oriented Programming'. The groups that were formed for the first and second part should preferably work together on the third part. Students that have worked alone on the second part must also complete the third part on their own. 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.

In this third part, we extend the program developed in parts one and two focussing particularly on inheritance.

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 implement the class `Robot`. In the second part, we added boards, walls, batteries and shooting. In this third part, each robot executes a program. Furthermore, the board can be populated with two new item types, namely repair kits and surprise boxes. Finally, each game object now reacts differently when being hit by a laser (instead of being terminated). 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 items. 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 items. However, a robot, wall, or item can be located on at most one board. Moreover, no board

contains the same robot, wall or item twice. All aspects related to the placement of walls, robots and items on boards should be worked out defensively.

Placing a robot, wall or item on a board at a certain position must take amortized constant time. Removing a particular robot, wall or item is preferably linear in the number of robots, walls or items located at the same position. Moreover, it should be possible to look up all robots, walls and items 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 items 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 item 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 items. When a board is terminated, all robots, walls and items 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 items.

The class Board must also offer a method that returns an iterator that will return all the elements on the board (not including items carried by robots) that satisfy a given condition. Examples of conditions are all elements that have an energy of at least 1000 Ws (the iterator will then not return walls nor surprise boxes, because they have no known energy), all elements in some sub range of the board, all items on the board, etc.

The documentation of the classes `Board`, `Wall`, `Battery`, `RepairKit` and `SurpriseBox` (and their super classes) must be worked out in a formal way. In addition, the documentation of the class `Robot` must be worked out both formally and informally. Classes for representing commands and conditions must only be worked out informally. Auxiliary classes need not be documented.

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, items 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 Items

Boards contain items which can be picked up, used and dropped by robots.

Each item has a certain weight expressed in gram (g). The weight of an item never exceeds `Integer.MAX_VALUE`. All aspects related to the weight of an item must be worked out in a total way.

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

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

We currently support three types of items: batteries, repair kits and surprise boxes. However, additional item types may be added in the future.

1.3.1 Batteries

Robots can use batteries to replenish their energy. Each battery holds a certain amount of energy. From this point on, it must be possible to express amounts of energy in different units, such as watt-second (Ws), Joule and KiloJoule. You may assume that the numerical value of any energy amount never exceeds `Double.MAX_VALUE`. 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.

1.3.2 Repair Kits

Robots can use repair kits to repair themselves (i.e. increase their maximum energy). Each repair kit holds a certain amount of energy. The energy stored in a repair kit never exceeds `Double.MAX_VALUE`. All aspects related to the energy of a repair kit should be worked out nominally.

A robot can use a repair kit to increase its maximum energy. Each watt-second stored in the repair kit increases the robot's maximum energy by 1/2 watt-second. The maximum energy of a robot never exceeds 20000 Ws. When a robot uses a repair kit, either the repair kit is drained or the robot's maximum energy reaches 20000 Ws and the excess energy remains inside the repair kit.

1.3.3 Surprise Boxes

When a robot uses a surprise box, one of the following events occur (selected at random): the surprise box explodes and the robot is hit, the robot is teleported to a random position, or the surprise box contains another item (either a battery, a repair kit or a new surprise box) and this item is added to the list of items carried by the robot. A surprise box can only be used once.

1.4 Programs

Robots can store and execute programs written in a simple programming language. This language has four basic commands: `move`, `turn`, `shoot` and finally `pickup-and-use`. These basic commands can be combined into sequences (`seq`), loops (`while`) and selections (`if`). For example, consider the program shown below:

```
(while
  (energy-at-least 1000)
  (seq
      (move)
      (turn clockwise)
  )
)
```

When a robot executes this program, it will run in circles as long as it has at least 1000 Ws.

In the remainder of this section, we discuss each construct of the programming language in more detail.

1.4.1 Commands

The language contains four basic commands: `move`, `turn`, `shoot` and finally `pickup-and-use`.

- **move** The robot moves one position in the direction it is currently facing. If the robot cannot move for some reason (e.g. insufficient energy), the move command has no effect.
- **turn** The robot turns 90 degrees in either clockwise or counter-clockwise direction. This direction is an argument of the turn command. If the robot cannot turn for some reason (e.g. insufficient energy), the turn command has no effect.
- **shoot** The robot shoots its laser in the direction it is currently facing. If the robot cannot shoot for some reason (e.g. insufficient energy), the shoot command has no effect.
- **pickup-and-use** The robot picks up an item and immediately uses this item. If no item is present, this command has no effect. If several items are present, one of them is selected randomly.

The basic commands can be combined into sequences, loops and selections.

- **sequence** A sequence is a list of commands. For example, the string

  ```
  (seq (move) (turn counterclockwise) (move))
  ```

represents a sequence consisting of a move, a turn and another move command.

- **while** A while loop consists of a condition and a command (corresponding to the body of the loop). For example, the string

  ```
  (while (can-hit-robot) (shoot))
  ```

  represents a loop with condition `(can-hit-robot)` and body `(shoot)`.

- **if** An if command consists of a condition and two commands corresponding to the then and else branch. For example, the string

  ```
  (if (can-hit-robot) (shoot) (turn clockwise))
  ```

  represents an if command with condition `(can-hit-robot)`. If the condition evaluates to true, then the robot shoots; otherwise, the robot turns.

Note that non-basic commands can arbitrarily be nested.

1.4.2 Conditions

The language has five basic conditions: `true`, `energy-at-least`, `at-item`, `can-hit-robot`, `wall`.

- **true** Always holds.
- **energy-at-least** Evaluates to true only if robot's current energy is at least equal to the given amount. The amount is an argument of this condition. For example, the string

  ```
  (energy-at-least 1000)
  ```

  represents a condition that evaluates to true only if the robot has at least 1000 Ws.

- **at-item** Evaluates to true only if the robot is located on a board and it can pick up an item.
- **can-hit-robot** Evaluates to true only if the robot can hit another robot by firing its laser.
- **wall** Evaluates to true only if there is a wall to the right of the robot.

These basic conditions can be combined into conjunctions, disjunctions and negations.

- **and** A conjunction is a list of conditions. A conjunction evaluates to true only if all sub conditions evaluates to true. For example, the string

```
(and (can-hit-robot) (energy-at-least 1000))
```

represents a conjunction that evaluates to true only if a robot can be hit and if the robot has at least 1000 Ws.

- **or** A disjuncton is a list of conditions. A conjunction evaluates to true only if at least one of its sub conditions evaluates to true. For example, the string

  ```
  (or (can-hit-robot) (energy-at-least 1000))
  ```

  represents a conjunction that evaluates to true only if either a robot can be hit or if the robot has at least 1000 Ws.

- **not** A negation has one argument . A negation evaluates to true only if its argument does not evaluate to true. For example, the string

  ```
  (not (energy-at-least 1000))
  ```

  represents a negation that evaluates to true only if the robot has less than 1000 Ws.

Note that non-basic conditions can arbitrarily be nested.

1.5 Robots

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

1.5.1 Energy

As before, each robot holds a certain amount of energy expressed in watt-second (Ws). As we will explain in Sections 1.5.2, 1.5.3 and 1.5.4, 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.

1.5.2 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.

1.5.3 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 -1. 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.

1.5.4 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 items is hit. A random robot, wall or item located at that position is hit.

Each game object reacts differently when being hit by a laser. When a robot is hit by a laser, its maximum energy decreases by 4000. If a robots maximum energy reaches 0, the robot is terminated. When a battery or repair kit is hit by a laser, its energy increases by 500 Ws. When a surprise box is hit by a robot, it explodes and hits all items located on adjacent squares. Finally, hitting a wall with a laser has no effect.

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

1.5.5 Picking up, using and dropping items

A robot can pick up, use and drop items. The class `Robot` must be generic in the type of items it can carry. For example, a robot of type `Robot<Battery>` can only carry batteries.

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. If the item the robot is trying to use is terminated, the item must be discarded by the robot. The effect of using an item is discussed in Section 1.3.

A robot can transfer all its item to another robot, provided the other robot is standing next to the robot on the same board.

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.

1.5.6 Programs

A robot can store and execute a program. Your solution must provide a method to execute a single step of a robot's program. That is, this method executes the program until it encounters a basic command or until it reaches a final state

(i.e. the empty sequence). After executing this basic command, the method returns (even if the basic command had no effect). Moreover, the next call to this method continues after this basic command[1].

Programs can be loaded from and saved to files on disk. Some example programs are included in the assignment.

2 Testing

Write a JUnit test suite for the classes `Robot` and `Board` that tests each public method. Testing the other classes 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 items. 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`, etc. 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 May 23th 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!

[1] This can for example be implemented by using a program counter or by rewriting the program during execution.