CS 301 **Project Assignment 3** Tuesday, October 3, 2017

To Deliver: 1. Release 3.0 in your SVN repository

To Download: Robot.java and RobotDriver.java from blackboard site
Hard & final submission deadline: October 18, at midnight (11.59 PM), which is right after fall break, please try to finish the project before fall break.

Learning Objectives:
- Programming in Java: To implement classes for a given context, learn to work with exceptions
- SW Development tools: Working with the debugger, working with SVN
- SW Development: Exercise encapsulation, read existing code; explore an existing code base, get experience with the Pseudocode Programming Process(McConnell), refactor existing classes to adjust to a modified class design, experience in test-driven design and limitations of testing when it comes to randomized algorithms and graphics. Experience in establishing a hierarchical, layered design where each layer provides functionality in form of an API (an interface) to the layer above and with the help of own code and the API of the layer below.

Motivation/Background information

One way to structure large code bases is a layered architecture. A single layer often appears as an application programming interface (API) that is defined and established with the help of some low-level, basic classes. The API provides an abstraction and a single point of access (a facade) to some functionality. In this project, we will establish such an API inside the existing Maze codebase.

The Challenge:
There are many different ways to solve a Maze.

Aside (NOT for this project:)
for a nice summary click here and read the section on Maze Solving Algorithms.
This Wikipedia page is also a good start.

For this project, we want to prepare the maze code base to provide an API for maze exploration algorithms that will guide a "robot" out of the maze. The design idea is to have a layered architecture where robot driver algorithms operate on top of an abstract robot layer that operates on the overall maze.

Future robot driver algorithms will implement a RobotDriver interface (as given in file RobotDriver.java), for example, think of a naive random search algorithm.
Such driver algorithms will depend on a robot (a class that implements the Robot interface as given in file Robot.java) to explore a maze and find the exit. The robot interface follows from what a Lego robot may provide for as sensors to recognize distant objects (here walls) and the ability to move forward and to turn on the spot.

As a first step, we want to establish the Robot platform by providing some implementation of the Robot interface that works on the maze controller and the maze graphics (first person and map drawer classes).

To see if the concept works, we want to add a **manual** maze exploration with class ManualDriver.java that implements the RobotDriver with a class that operates on the robot implementation (and not directly on the maze object anymore). This will replace/extend the existing code to manually play the game.

Note: Any RobotDriver will drive the graphics (MapDrawer and FirstPersonDrawer) without being aware of this by operating a Robot which in turn operates the MazeController class which controls the graphics. The robot interface implementation needs to take care of this by operating on top of the MazeController class. As the robot driver is manual, however, it needs to be notified by the user interface if the user presses keys to navigate instead of a fully automatic exploration in its drive2Exit method. For this reason, the ManualDriver implementation needs to have a few extra methods in addition to what is listed in the RobotDriver interface in order to accept inputs for a single step to move forward or turning left or right. The ManualDriver has only a “fake” drive2Exit method that does not work (as the ManualDriver has no “automatic” mode).

Note: The class design is supposed to clearly separate concerns:
1. The driver algorithm, e.g. the ManualDriver or later an automated driver algorithm, implements the RobotDriver interface and interacts with an object that implements the Robot interface. So the RobotDriver basically collaborates with the Robot but not much else (the Manual driver also needs to receive keyboard input).
2. The class that implements the Robot interface is not necessarily aware of the driver class but it knows about the MazeController class to be able to perform move and rotate operations.
3. The MazeController class is aware of the graphics (MapDrawer and FirstPersonDrawer) and sends notifications if move or rotate operations require an update to the graphics.

The robot configuration that we will consider at this point is the following:
- It has distance sensors in all 4 directions and a room sensor such that it knows if it is inside a room or not.
- The robot can move forward and can turn.

Energy consumption: We also want to compare algorithms in terms of energy efficiency to operate the robot. So there is some energy consumption being involved in any movement and sensor operation and the robot draws energy from its battery on its journey. We need to modify the final screen to report on the length of the chosen path (number of steps) and the amount of energy consumed.
- The initial battery level is 3000.
- The energy costs are:
  - Distance sensing in one direction or checking if exit is visible in one direction: 1
  - Rotating left or right (quarter turn or 90 degrees): 3
  - Moving forward one step (one cell): 5

Design decision

To retain a reasonably uniform design, I would like to fix the following naming conventions for classes:
- The class that implements the Robot interface is named BasicRobot.java.

- The class that implements the key functionality for the manual operation of a robot is called ManualDriver.java. (You can use more classes if you like, but use this name for the one that has the responsibility to drive the robot).
- Classes that implement test cases for each class have the class name followed by suffix Test, e.g. BasicRobotTest.java for the BasicRobot.java class.

Note that we will only consider a default robot configuration with 4 distance sensors, one in each direction. However, it may be straightforward to have a class that is configurable to have more/less/other distance sensors. Provide the class with at least one constructor that has no parameters and creates a robot with the default configuration we consider.

Task list:

1) Prerequisite:
   1) Download the interface files Robot.java and RobotDriver.java from blackboard and import them into your src/falstad package.
   2) Reminder: In case you have mixed source code and test code, I would like to ask you to change this. In your Eclipse project, keep source code and test code separate. Use a separate source directory src for source code, another source directory test for test code. Both should then have the same package structure.

2) **Design:** Perform a class design with CRC cards.
   1) Note: there are several possible design solutions!
   2) Put your crc design in writing, simple list each class, its responsibilities and collaborators. Use this in the JavaDoc comments that go into the header of each class.

3) Implementation of Robot API:
   1) Implement BasicRobot.java and BasicRobotTest.java classes
   2) You can use as many additional other classes, methods as you like. You can change the existing code base to make the integration of BasicRobot.java easier.
   3) **WARNING:** perform the following sanity test:
      1) assign the starting location (0,0) to a maze when it is built in MazeBuilder, make the rotate method print its directions before and after the rotation, print for the initial location (0,0) in which direction potential walls on top/bottom/right/left are and compare this with the graphics on display in map mode.
      2) You will recognize an inconsistency in the interpretation of directions for rotate and the CW_TOP, CW_BOT and CardinalDirection directions used in MazeBuilder and Cells. BEFORE you make changes to the existing code, think why it can work the way it is and think twice before you want to stir things up …
   4) For testing: ask yourself what a correct robot implementation must be able to support.

4) Implementation of Manual Robot Driver:
   1) Implement the Manual Robot Driver algorithm with class ManualDriver.java and make this the one that is used by MazeApplication.java. Apply the Pseudocode Programming Process as it seems useful to you.
   2) You can use as many additional other classes, methods as you like.
   3) For testing: ask yourself what a correct robot driver implementation distinguishes from a faulty one. Manually exercise the game and see if you can operate the robot and play the game.

5) Implementation of the GUI changes
   1) Modify the last screen in the game to show path length and energy consumption.

6) Comments, Documentation:
   1) Use a good programming style towards self-documenting code

2) Provide comments in JavaDoc format that clarify intent, purpose of code
3) Use assert() statements in BasicRobot and ManualDriver to express correctness conditions

7) SVN:

1) Update your subversion repository whenever you made progress and got a particular feature or method going.
2) You are expected to have at least 10 additional revisions with meaningful comments in your SVN project's history.

8) Final submission:

1) When you have finished the project assignment, create a tag with a Release 3.0 before the deadline. Include your junit test cases in your release. Keep source code and test code in separate directories (src and test).

Grading:

You will receive points for:

1) **Your SVN Repository**:
   1) your subversion repository is set up correctly and the grader can download your release3.0 from its tags folder.
2) **Your implementation:**
   1) **Comments:**
      1) Code is expected to have comments, at least a statement per method (using the javadoc format with /** …. */) unless you are implementing interface methods where this does not apply.
      2) Code is expected to be self-documenting: Methods and variables are expected to be meaningful and communicating what a method does or what a variable stores. Methods are preferably short in length and simple in complexity (think: McCabe's cyclomatic complexity as discussed for structured basis testing).
   2) **Test code:** You will receive points for implementing test cases for the BasicRobot class. Test cases are expected to come with a comment that specifies what is tested, what is the correct behavior and (if not obvious) why that behavior is correct.
   3) **Production code:** You will receive points for your implementation of ManualDriver.java and BasicRobot.java.
      1) **Bonus points:** if you make use of assert() statements in a meaningful, non-trivial way to incorporate correctness considerations into your production code.
   4) **Code for BasicRobot passes Junit tests, code coverage:** You will receive points if test cases perform and your code passes our own tests. We will run and evaluate your tests as well as some additional ones of our own that will test BasicRobot.java based on the interface specification. We will evaluate your tests to see if they perform a rigid test and if they achieve a reasonable coverage on the tested classes. Coverage will be measured with Eclemma and the percentage for BasicRobot.java will count for points (the higher the better, granularity goes by 20% steps).
   5) **Application works, code passes acceptance test:** You will receive points if the overall MazeApplication runs with the manual operation operating on top of the robot class works. This includes correct operation of the maze graphics that is displayed in the first person perspective as well as the map perspective with the current location and the suggested path to the exit (based on the distance matrix). The last screen of the game needs to show the path length and energy consumption.