CS 301 Fall 2017 **Project Assignment 2**

To Deliver: 1. Release 2.0 in your SVN repository in a directory tags

To Download: P2.zip from blackboard site
Hard & final submission deadline: October 2, at midnight (11.59 PM)

Learning Objectives:
- Programming in Java: To implement classes for a given context; using inheritance to share code across classes that solve the same problem in different ways; implementing an existing algorithm in Java
- SW Development tools: To learn unit testing with Junit, measuring code coverage with Eclemma, working with the debugger, working with SVN,
- SW Development: To read existing code; explore an existing code base, to get experience with the Pseudocode Programming Process(McConnell), exercising a test-driven approach when extending an existing code base

Motivation/Background information

The first project assignment you worked on was a small-scale stand-alone problem. For the rest of the class and further assignments, we will work on a single different problem which is about a Maze game.

We will use an existing implementation by Paul Falstad (www.falstad.com) whose code is readily available over the internet. Note that the code is copyrighted. I contacted Paul Falstad and he gave us the permission to use and modify his code for teaching purposes in CS 301. While the original code is available and you may want to read it for your information, we will work on a refactored version that I produced (so blame errors on me). I reorganized the code into a few more classes and added comments which I believe will make it easier to understand.

Important: Recognize and check if the code works. It should! The documentation came after the fact and need not be correct. If the documentation and code do not match, improve the documentation!

Status Quo: Working version of Maze game
After installation, read the MazeApplication.java code to see how the Maze program works.
1. If it is called with no parameters it runs like the original code by Falstad (see also: http://www.falstad.com/maze/ for documentation).
2. If it is called with parameter "Prim", it will use a randomized version of Prim's algorithm to generate the maze.
3. If it is called with a parameter that provides an input file in a particular XML format, then the code loads that maze from a file and operates on it. The class MazeFileWriter produces the particular XML format.

Hint: use Eclipse->right click project -> run as->run configurations, create one run configuration for each way to run the MazeApplication. A pregenerated maze resides in test/data/input.xml.

To see how you can further operate the application, read the MazeController.keyDown method to see what keyboard input the application reads and what it does with that.

The Challenge:
There are many different ways to generate a Maze, for a nice summary click here and read the section on Perfect Maze Creation Algorithms. For a nice description with example and code check this blog.

Falstad's code comes with one algorithm that creates a maze and allows for rooms. The algorithm is implemented in MazeBuilder.java. In a previous CS301 class, students added a randomized version of Prim's algorithm that you find in MazeBuilderPrim.java. Still, we would like to have more alternatives to generate a maze. For this project, you are tasked to extend it by Eller's algorithm (see section "Perfect Maze Creation Algorithm" here). We have one lecture about graph algorithms (Prim, Kruskal and Eller) that describe the algorithmic ideas. If you are not familiar with Eller's algorithm, I would recommend this blog which also explains other maze generation algorithms. As with most basic algorithms, there is for sure a place on the web with sample code. Before you go ahead and copy code from elsewhere, please recognize that the time that is necessary to understand, adjust, and debug other people's code is mostly likely about the same than implementing the algorithm yourself (and we are not talking about copyright issues here yet). To get a solid implementation, you cannot avoid the necessity to properly understand the algorithm. Feel free to reuse code, but don't complain if it did not save you time and if it did not work in the end.

Design decision

The most straightforward design is to implement your new algorithm in a subclass of the existing MazeBuilder class just like the MazeBuilderPrim class. Since inheritance allows us to formulate generic solutions in a superclass and then have subclasses for more specific cases, let's just add yet another algorithm by adding a MazeBuilderEller class that inherits from MazeBuilder.

You can add further methods and classes as you deem necessary. You can change existing classes and methods as well. Of course, modifying an existing code base comes with the risk to introduce many subtle errors. A suite of Junit test cases can help you do regression testing to see if your changes broke anything.

To do so, you need to create a junit test class MazeFactoryTest.java that exercises all functionality of MazeFactory.java to check that the computed MazeConfiguration (or MazeContainer) has all properties of a correct maze. This is a typical black box test that would apply to mazes regardless which generation algorithm is used. For white box testing, develop a new test class MazeFactoryEllerTest.java to test specifics of your new MazeBuilderEller class.

Hint: You may want to test in your MazeBuilderEllerTest class that your implementation of generatePathways() in MazeBuilderEller does not leave some enclosed areas in the maze. This is a possible pitfall for a generation algorithm: some enclosed area that has no path to the exit. This may cause problems for the subsequent computation of the distance matrix so one want to test this before distances get calculated.

Task list:

1) **Prerequisite**: Create a new Eclipse project name Maze and import files from P2.zip. Make sure you reproduce the following directory structure. Note, in each java file, the first line states the package name the code belongs to.
    1) Maze (project)
        1) src (source folder)
            1) generation (package): generation related java files from P2.zip
            2) falstad (package): gets all other java files from P2.zip
        2) test (source folder)
            1) generation (package): gets AllCellsTest.java, CellsTest.java and CellsTestIterator.java from P2.zip
            2) falstad (package): empty at this point
            3) data: gets file input.xml from P2.zip

    Adjust the build path to include src and test as source code, Junit4 library in libraries.
2) **Prerequisite**: Once you have code that compiles and runs, share it as a new SVN project in our SVN repository. This is important as it allows you to get back to this working version easily. You can share the same tags folder with the SlidingPuzzle project as the release numbers will be different.
3) **Prerequisite**: Play the Maze game for some small configurations, figure out how you can make the map and solution show up on the display (check MazeController.keyDown method) and setup 4 run configurations (one without parameters, one for Prim, one for input.xml, one for Eller, hint: use the Arguments tab and set Program arguments. Also set VM arguments to "-ea" to enable assertions)
4) **Prerequisite**: **Test first design:** The testing and debugging of code is difficult it failures are hard to reproduce. As the generation of random mazes is supposed to produce different mazes each time, this is such as bad case for debugging. To fix this, check the constructor in MazeBuilder.java for a deterministic maze, implement what is missing here (1 line of code, check the documentation for SingleRandom.java and Random.java) and use this constructor in the test code that you develop for Task 1.
5) **Task 1**: **Test first design: Test cases for MazeBuilder.java:**
    1) Add a new junit test class MazeFactoryTest.java to the generation package in the test directory.
    2) Develop a set of test cases for the MazeFactory to test the MazeBuilder class which you can then easily reuse and adjust for the new algorithm with the help of inheritance. These test cases are naturally blackbox tests that work for any maze, regardless which algorithm is used to compute it.
    3) Figure out what a correct maze looks like that the MazeBuilder generates.
        1) You need to think about properties that are necessary for a maze to be correct.
        2) For instance, a correct maze needs to have an exit that can be reached from anywhere in the maze. (Hint: you need not write a maze exploration algorithm yourself, check what the distance matrix in MazeContainer.mazedists contains and consider necessary and sufficient conditions for the existence of a path to the exit.)
    4) **WARNING**:
        1) there is one big challenge here: the MazeFactory operates a background thread such that you need to make the test wait for the termination of the MazeBuilder thread before the test proceeds. The Factory interface has a special method to help you with that.

        2) The MazeFactory relies on an Order object for communication. Figure out how the communication between MazeController, MazeFactory and MazeBuilder works.
        3) You will need a stub for an Order object to operate the MazeFactory in a testing environment and to be able to check the properties of a generated maze.
    5) Do not forget to commit your changes to the SVN repository in a regular manner! You are expected to produce at least 10 revisions (commit your changes) while working on this assignment.

6) **Task 2: Implement Eller's Algorithm**
    1) **Understand the problem:** Figure out how the algorithm works, (Hint: check this blog). Work through an example.
        1) Constraints: For grading purposes:
            1) MazeBuilderEller.java must have a constructor method with no parameters: MazeBuilderEller().
            2) The algorithm is implemented such that it starts with a maze where all walls are up and then it selectively tears down walls with the Cells.deleteWall() method.
        2) Simplification: Falstad's maze generation code has a concept of "rooms". Start with a first version without rooms. You can do this by overwriting a corresponding method from the MazeBuilder class in your new class. Once this works add the room concept to the algorithm. Mind the difference between borders and walls in Cells.java!
    2) **Make a plan:**
        1) For your MazeBuilderEller class: inherit from MazeBuilder.java, check which methods you need to overwrite (change). Break the overall calculation into a set of methods as necessary.
        2) Think of ways how you could test the methods before you code them, make adjustments as necessary to make testing easier. Come up with test cases for your new algorithm.
        3) Use the pseudo code programming process to start writing down how the methods would work inside your MazeBuilderEller class.
    3) **Carry out the plan:**
        1) Implement the test cases in a separate MazeBuilderEllerTest.java class.
        2) Implement functionality in a piecemeal manner and make it pass test cases for the MazeFactoryTests and the MazeBuilderEllerTests.
        3) Take full advantage of existing classes and libraries.
        4) Use assert() statements in code to express your expectations on properties to hold if code works correctly. (memo: add -ea to run configuration, VM parameter settings)
        5) If test cases fail use the debugger to track down the root cause if necessary.
        6) Check code coverage with Eclemma, look for uncovered lines of code. Refine your test code to do some serious white box testing.
        7) Once the algorithm seems to work within your Junit test environment, adjust the MazeApplication to execute it (with new parameter "Eller" just like "Prim"). Run the whole application with Eclemma to see if your new code is really executed.
        8) Manually test if the overall application works. Take your time to enjoy the success before you move on to the next task!
        9) Format your comments from the pseudo code programming process to work with Javadoc. Produce documentation with the help of Javadoc.
            1) In Eclipse: call menu-Project-Generate Javadoc, let generated files reside in directory Maze/doc
        10) Do not forget to commit your changes to the SVN repository in a regular manner!

    4) **Look back**:
        1) Look back and recognize what worked and what did not work in the way you organized your work. What was unproductive? What took a long time? How could you get to the final stage with less work? (work smarter not harder)

7) **Create a release2.0 in your SVN repository.** The repository should have 3 directories then, namely SlidingPuzzle, Maze and tags. The grader will look for tags/release2.0 (or higher, if there is a release2.1 he will consider that instead of 2.0)
    1) We will run and evaluate your tests as well as some additional ones of our own that will test MazeFactory.java, MazeBuilderEller.java.
    2) Our tests rely on the generation of a log file that tracks whenever you call Cells.deleteWall() so please do not manipulate the log file generation as this will create real problems for the grader.
    3) We will evaluate your tests to see if they perform a rigid test and if they achieve a high coverage on the tested classes. Make sure your code passes your own tests!
    4) We will also evaluate your code documentation with Javadoc for MazeBuilderEller.java, and its corresponding test cases.

Grading:

You will receive points for:

1) **Your SVN Repository**:
    1) your subversion repository is set up correctly and the grader can download your release2.0 from its tags folder.
    2) your subversion repository history shows at least 10 revisions (commits) together with meaningful comments. Please note the difference between revision and release! You will do many revisions but typically only one release per assignment.
2) **Your implementation: Eller**
    1) A class design with superclass MazeBuilder, subclasses MazeBuilderPrim (already there), and MazeBuilderEller (new) that overwrite appropriate methods from MazeBuilder.
    2) MazeBuilderEller implements Eller's algorithm to generate a maze (make sure your algorithm supports the generation of rooms).
    3) Algorithm is Eller's algorithm:
        1) We will evaluate this with a Junit test that relies on the log file produced with methods in Cells.java. We will log usage of the Cells.deleteWall method. You need not do anything with the logging mechanism. But you want to make sure that you use the Cells.deleteWall method when you take down walls in Eller's algorithm.
    4) Eller’s Algorithm is correct:
        1) We will use Junit tests to see if your implementation of the algorithm delivers a correct maze (there is exactly one exit, it is possible to reach the exit from each cell, etc). Correctness considerations become obsolete if your algorithm is actually Falstad's original algorithm or Prim's algorithm.
    5) Eller’s Algorithm is reasonably documented: We will run Javadoc on your MazeBuilderEller class and see if methods are reasonably documented.
    6) Test cases for MazeBuilderEller in file MazeBuilderEllerTest.java
        1) Junit tests contain non-trivial test code that covers MazeBuilder / MazeBuilderEller and contains assert statements in each test.
        2) Junit tests contain Javadoc comments that produce a reasonable HTML documentation such that each test comes with an explanation on what is tested etc.

3) Your implementation passes your own Junit test cases, there are no failures, no errors.
4) MazeBuilderEllerTest achieves a high coverage on the MazeBuilderEller.java class