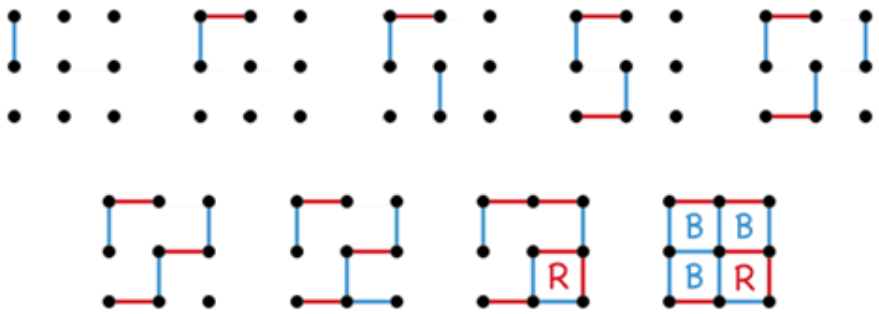
CSCI-605 Advanced Object-Oriented Programming Concepts

Homework 2: Dots and Boxes



1 Introduction

In this homework, you will be implementing a program for representing and playing the game of Dots and Boxes.

1.1 Goals

Students will gain experience with the fundamentals of object oriented programming in relation with basic class concepts such as:

- Classes vs Objects
- State vs Behavior
- Encapsulation
- Constructors and Instantiation
- Accessor and Mutators
- Overriding methods in `Object`, i.e `equals` and `toString`
- UML Class Diagrams
- Test Driven Development using JUnit

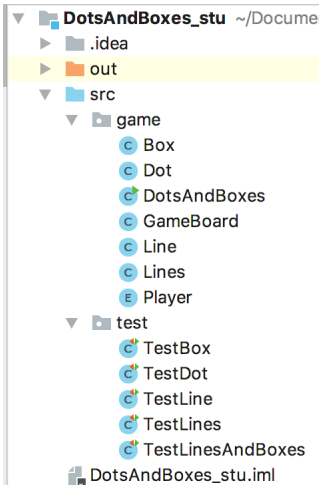
1.2 Provided Files

1. The **class_diagram.png** illustrates the program's design.
2. The **sample** folder contains sample runs.
3. The **tests** folder contains a set of tests for your program.
4. The Javadoc documentation for the classes in the `game` package here.

2 Implementation

2.1 Project Structure

Your project should be structured as follows:



2.2 Design

Although there are several different ways that these classes could be written, for this homework, we are requiring you to closely follow the supplied design (see provided files section). For more information about these classes, their methods and fields, see the javadoc provided.

To understand the UML class diagram, read here.

2.3 Command line arguments

The program is run on the command line as:

```
$ java DotsAndBoxes rows columns
```

If the number of arguments are correct, they are guaranteed to be integers greater than zero. If the number of arguments is incorrect, display a usage message and exit the program, i.e.:

```
Usage: java DotsAndBoxes rows columns
```

2.4 Program Operation

All input comes from standard input. Use the java.util.Scanner class to read the user's input.

The current state of the board and game statistics are always displayed in the standard input. The user is prompted for input with `"> "`.

When prompted, the user is expected to enter 4 numbers to claim a line - the starting row and column (0 based) and the ending row and column. To simplify things, the line must be specified from left to right (horizontal) and top to bottom (vertical).

If the user enters an invalid line to claim (the coordinates are invalid, or the line has already been claimed), you should display an `Invalid line!` error message and then re-prompt to let the same player make a move. If the user enters a "q", the program should terminate gracefully with no further output.

If the game is played to completion, the result of the game should be displayed. There are one of three possible outcomes, where # is the number of boxes claimed by the respective player:

```
RED wins # to #!
BLUE wins # to #!
TIE game # to #!
```

2.5 Sample Runs

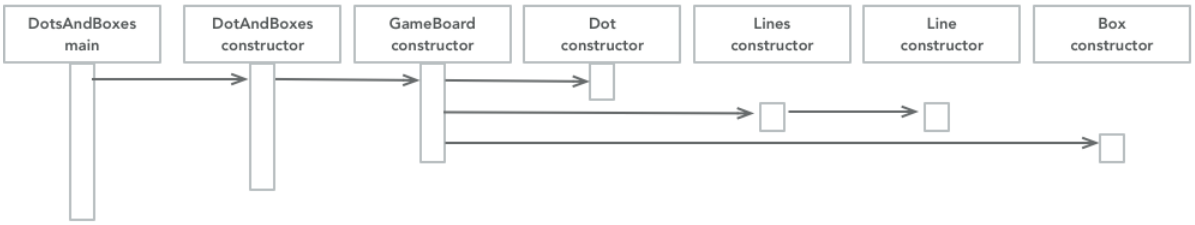
You are provided with three sample runs:

1. Full run (blue wins): A complete game on a 2 x 3 board (sample-input-1.txt and sample-output-1.txt).
2. Full run (tie): A complete game on a 2 x 2 board (sample-input-3.txt and sample-output-3.txt).
3. Errors: Samples of the various errors your program should be able to handle. The board is 2 x 3 (sample-input-2.txt and sample-output-2.txt).

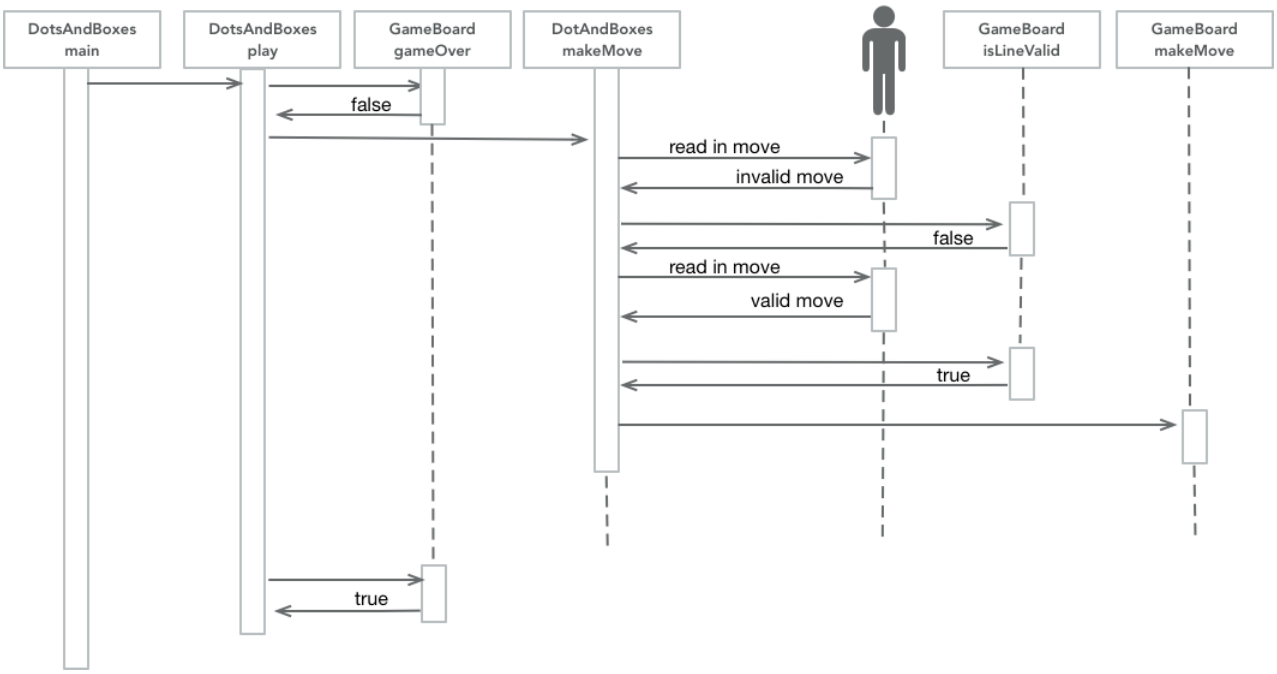
2.6 Sequence Diagrams

These sequence diagrams show how the classes and methods flow for object creation and the high level game play.

Object Construction Sequence



High Level Gameplay Sequence



3 Testing

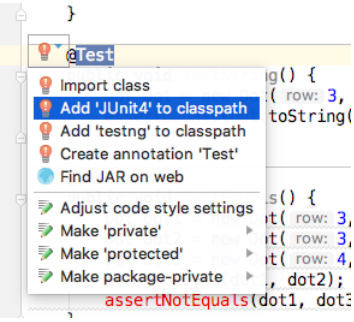
Because we are providing a complete JUnit set of tests for your homework, it is suggested you use the Bottom-up approach for developing. This means you start with the most basic classes that have no other dependencies, and then work your way up. Each time you develop a new class you should run the test cases and make sure that the module you just developed, plus the previous ones, all work correctly. This software development process is referred to as Test Driven Development.

3.1 JUnit

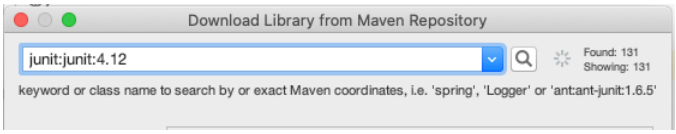
We are provided you with a tests folder that has 5 unit tests files for the core classes, `TestBox`, `TestDot`, `TestLine`, `TestLines`, and `TestLinesAndBoxes`. Import all those files in a `test` package in your project.

JUnit comes with IntelliJ but it is not in the path. Once you have stubbed out all the classes in your `game` package you can set up JUnit.

To set up JUnit, go to the TestDot source code. Highlight over a @Test annotation, hit the ALT + ENTER keys and select Add 'JUnit4' to the classpath.



Select that and a dialog should come up to download it - click the OK button.

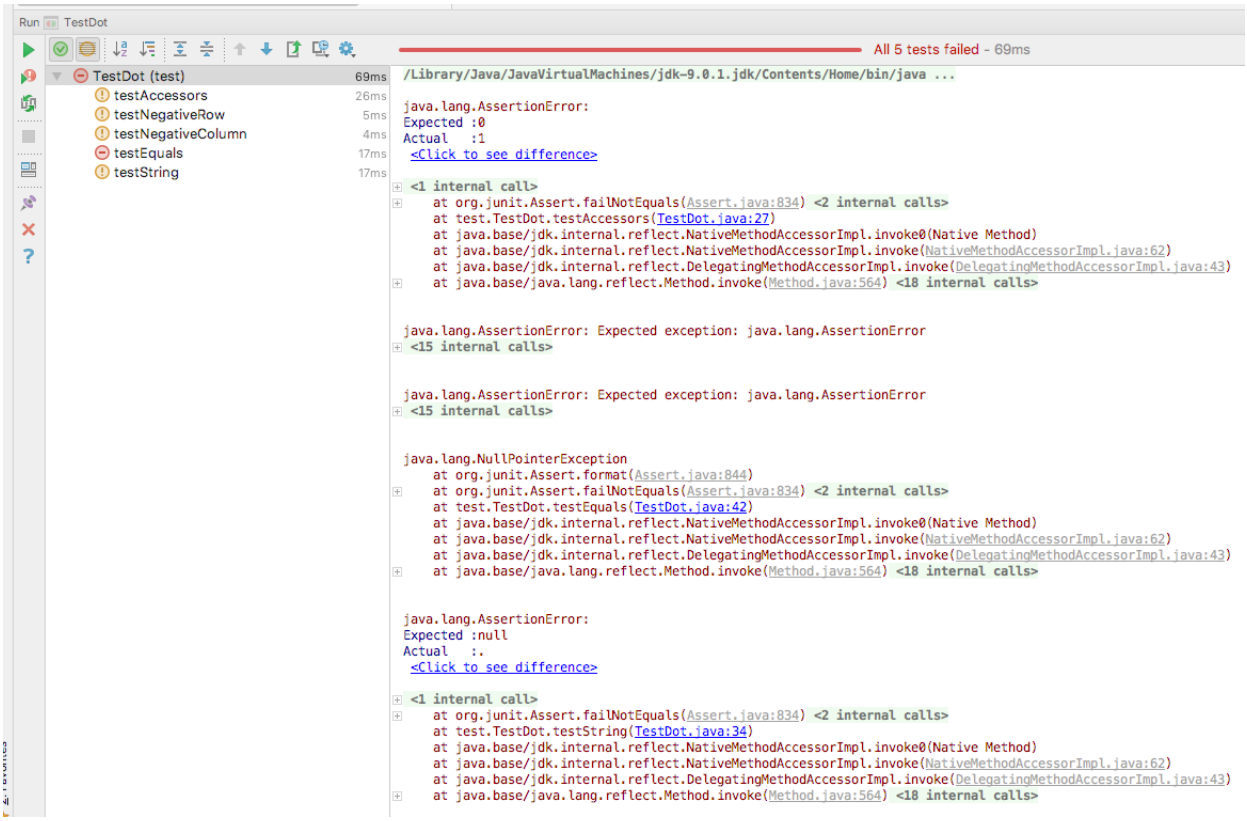


At this point there should be no errors related to JUnit. You may have still have errors if you haven't implemented the `Dot` class yet.

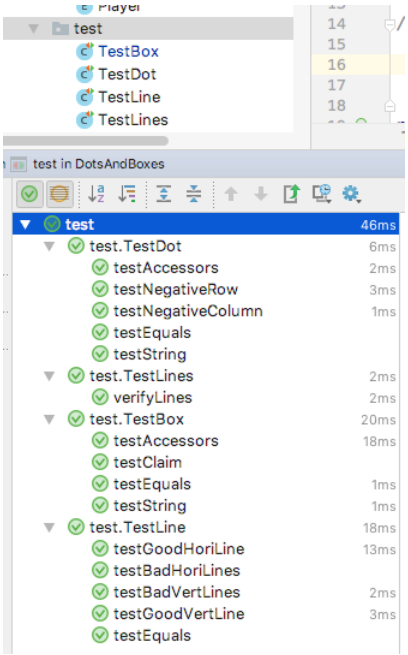
3.1.1 Run Tests

You can run all the tests at once by right clicking on the `test` package, or you can run each test case individually by right clicking on it.

When a test fails it shows you what it Expected, and the Actual which is what your program produced. These two things must match exactly in order to pass the test. If strings are being compared they must match character by character exactly.



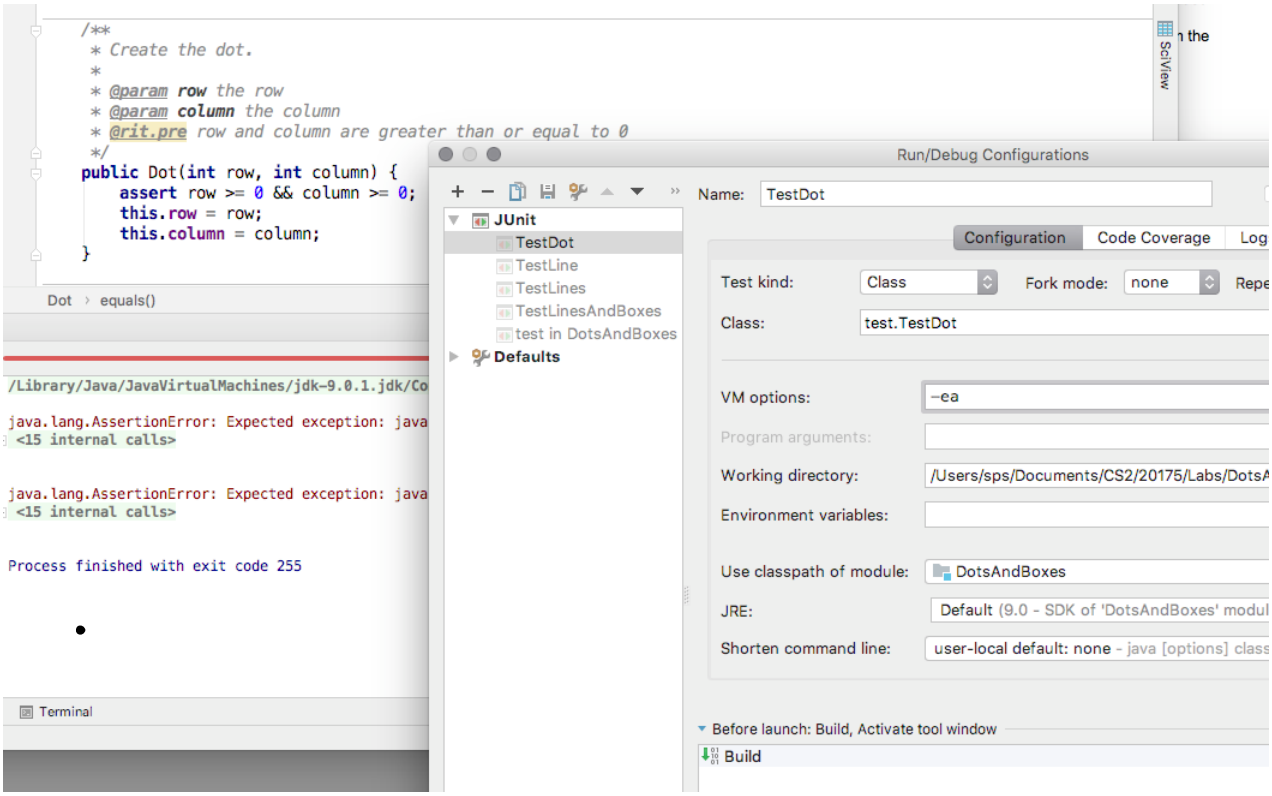
Your goal is to make sure you pass all the tests, e.g.



3.2 Assertions

Some of the tests expect an assertion exception to occur - for example if trying to make a Dot with a negative row. In Java, assert is a keyword that takes a boolean expression.

If the expression evaluates to true, execution continues on the next statement as normal. If the expression is false, execution halts and an AssertionException is thrown. By default assertions are disabled by the virtual machine. To enable, go to the run configuration for the test and add -ea to the VM Arguments.



4 Submission

You will need to submit all of your code to the MyCourses assignment before the due date. You must submit your src folder as a ZIP archive named "hw2.zip" (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this homework).

5 Grading

The grade breakdown for this homework is as follows:

- Design: 15%.
- Functionality: 25%
- Testing: 50%
- Code Style & Documentation: 10%