Homework #2: From Calculators to Calculus

Due Thursday, February 2nd at 11:59 p.m.

In this assignment you will first write and test various classes that implement the behavior of a calculator and then finish a small unit testing exercise designed to make you better at JUnit testing. This assignment has four parts. In Part 1 you will write objects that implement common calculator operations such as addition and subtraction. In Part 2 you will write a family of classes representing arithmetic expressions, using the operations you wrote in Part 1. Part 3 combines ideas from functional and object-oriented programming, where you will write a class representing a variable whose value can be changed, so that expressions can represent algebraic functions. You will then write a class to represent the derivative of an arbitrary function, and write a short program that uses your derivative to find zeros of an arbitrary function using Newton's method. Finally, Part 4 is an independent task in which you will implement and submit unit tests for a short `AvlTreeSet` class.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism, information hiding, and writing contracts, including an appropriate use of Java interfaces.
- Interpret, design, and implement software based on informal specifications.
- Write unit tests and automate builds and tests using JUnit, Gradle and Travis-CI.
- Gain experience writing tests based on a functional specification.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics.
- Learn good testing practices and style.

This document continues by describing the four parts of the assignment and testing requirements in more detail.

Part 1: Using polymorphism to implement calculator operations

In `edu.cmu.cs.cs214.hw2.operator`, we have provided an `Operator` interface to represent an arithmetic operator and sub-interfaces to represent binary operators (such as addition, subtraction, multiplication, division, and exponentiation) and unary operators (such as negation and absolute value). To complete this part you must (1) provide concrete implementations for at least these seven operators, and (2) complete the program in `edu.cmu.cs.cs214.hw2.guicalc.Main`, which simply passes your operators to the `GuiCalculator` constructor and starts the calculator.

Run `edu.cmu.cs.cs214.hw2.guicalc.Main` and play with the GUI calculator to make sure your operators work; fix them if they don't. We recommend (but don't require) that you write some more operators (e.g., sin, cos, tan, log) to see how easy it is to extend the power of the calculator. You provide the operators and `GuiCalculator` does the rest.

Part 2: Implementing calculator expressions

In `edu.cmu.cs.cs214.hw2.expression`, we have provided an `Expression` interface to represent arithmetic expressions. In this part your task is to implement classes to represent numbers, unary operator expressions, and binary operator expressions. You can test your implementations by manually constructing familiar arithmetic expressions (such as $\sqrt{3*3+4*4}$) and checking that they evaluate to the correct value.

The TAs have provided a terminal-based calculator to help you test your solution. It has a parser that takes arithmetic expressions in string form and translates them into expression trees using your `Expression` implementations. To give the parser access to your implementations, you need to implement the `edu.cmu.cs.cs214.hw2.termcalc.ExpressionMaker` interface, which has methods returning instances of each the eight expression types you were asked to implement. Once you've done this, edit `edu.cmu.cs.cs214.hw2.termcalc.Main` to pass an instance of your `ExpressionMaker` implementation to the constructor of the `TerminalCalculator`. Then run `Main`, and the program will read arithmetic expressions from the keyboard, parse them, evaluate them, and print the results. In other words, it will turn your expression implementations into a full-fledged text-based calculator that understands operator precedence. Here's how it looks in practice:

```
Enter an expression: 1 + 2*(3-4)/5 - 2
Result: -1.4

Enter an expression: (3*3 + 4*4)^.5
Result: 5.0
```

Part 3: Functional programming: Derivatives and Newton's method

Now, you will extend your work from Part 2 to support expressions containing named variables (e.g. $x * x$), and then use that solution to write an expression that numerically evaluates the derivative of another function. Afterward, you will use your derivative expression in an implementation of Newton's method, which is a numerical algorithm to find the zeros of a function.

Expressions with named variables

In `edu.cmu.cs.cs214.hw2.expression.VariableExpression` we have provided a skeletal implementation of an `Expression` representing a variable, essentially a named box to represent a value much like a variable in algebra. Complete that skeletal implementation and test it by creating a variable named x and an expression representing $x * x - 2$. You should not need any new constructors to do this; your `VariableExpression` and solution from Part 2 should suffice. Set x to some value and verify that the overall expression evaluates to the correct numerical value. If you'd like, you can write a little program to generate a table of values for a function by repeatedly setting x and evaluating the function. Also you can (but are not required to) play with functions of multiples variables (e.g., a$x^2 + bx + c$).

An expression to compute the derivative

In calculus, the *derivative* of a function $f(x)$ is another function $f'(x)$ whose value at each point x is the slope of $f(x)$ at x. If you already know calculus: Great! If not, for this assignment all you need to know is that the derivative of a function can be approximated with respect to a variable x as:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

where Δx (pronounced "delta x") is some arbitrary small value.

For this sub-part, write an implementation of the `Expression` interface whose instances represent the derivative of some specified function. The constructor for your implementation should resemble:

```
/**
 * Creates an expression representing the derivative of the specified
 * function with respect to the specified variable.
 *
 * @param fn the function whose derivative this expression represents
 * @param independentVar the variable with respect to which we're
 *   differentiating
 */
public DerivativeExpression(Expression fn,
                            VariableExpression indepedentVar) {...}
```

The `DerivativeExpression`'s `eval` method should return the approximate derivative of `fn` at whatever value `independentVar` is set to. To approximate derivatives for this assignment set Δx to be a private constant value `DELTA_X = 1e-9` (i.e., 10^{-9}). Test your implementation by evaluating the derivative of $x * x - 2$ at various points; the result should be approximately equal to $2 * x$. Similarly, the derivative of $\sin(x)$ should be approximately equal to $\cos(x)$.

Newton's method

Finally, use your `DerivativeExpression` to compute the zeros of a function using Newton's method. The *zeros* of a function $f(x)$ are the values of x at which $f(x)$ evaluates to zero. For example, the zeros of $x^2 - 3x + 2$ are 1 and 2.

Newton's method is a numerical algorithm that iteratively improves a coarse approximation of a zero until the approximation is sufficiently accurate. See the Newton's method Wikipedia article for details. Given an initial estimate x_0 of a zero, Newton's method computes an improved estimate x_1 as:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The same formula is used to compute each successive estimate:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The process can be repeated until the estimate is sufficiently close to an actual zero. Newton's method fails for some functions $f(x)$, but is known to converge for many functions.

In `edu.cmu.cs.cs214.hw2.ZeroFinder`, we have created a `ZeroFinder` class for you. Compete the class by writing a method to compute a zero of an arbitrary function given an initial rough approximation and a target accuracy:

```
/**
 * Returns a zero of the specified function using Newton's method with
 * approxZero as the initial estimate.
 *
 * @param fn the function whose zero is to be found
 * @param x the independent variable of the function
 * @param approxZero initial approximation for the zero of the function
 * @param tolerance how close to zero f(the returned value) has to be
 * @return a value x for which f(x) is "close to zero" (<= tolerance)
 */
public static double zero(Expression fn, VariableExpression x,
                          double approxZero, double tolerance) { ... }
```

If you have successfully completed Parts 1 and 2 and your `DerivativeExpression`, the body of this method should be relatively short and easy. Test this zero-finding method on a few functions, such as $x * x - 2$ with initial estimate 1.

Testing

For this assignment, we have strict testing requirements. You must write tests for all your solutions of Parts 1, 2, and 3 and additionally test the implementation of `AvlTreeSet.java` that we provide.

Ideally you should achieve nearly 100% line coverage, excluding user interface code, the test code itself, and any code provided by the course staff (except `AvlTreeSet.java`). If you do not achieve 100% line coverage please explain with a short comment in each uncovered area why you can't achieve 100% line coverage.

Test your own implementation

Test all code you have written for this assignment, aiming for 100% line coverage. When possible you should check the correctness of normal cases as well as edge cases of your implementation.

We recommend that you start writing tests for your solution as you complete your solution; do not delay writing unit tests until after your implementation is complete. It is far easier to test (and find any bugs in) early parts of your implementation before those parts are used. A common strategy is to write your unit tests as you write your code, or even to write your unit tests before you write your code, as you design your software specification.

If you discover bugs in your own implementation, it is good practice to write a bug report and fix the bug with a separate commit, as described for AVL trees below.

Unit test `AvlTreeSet.java`

AVL trees are self-balancing binary search trees that you can learn more about on Wikipedia: `http://en.wikipedia.org/wiki/AVL_tree`. We have provided a (potentially buggy) implementation of AVL trees in `AvlTreeSet.java`, and its specification is included in the source code as Javadoc comments.

Your task is to write unit tests to achieve 100% line coverage of the `AvlTreeSet` class.

Report an issue for each bug you find in `AvlTreeSet` on the GitHub issue tracker for your repository, which can be found at the following location: `https://github.com/CMU-15-214/ANDREWID/issues`. Produce a bug report each for each bug you find in the

AVL implementation. Your report should include at least the following information: the context of the bug (i.e., where in the code the bug occurs), the error that occurs as a result of the bug, and (optionally) a suggested fix for the bug.

Fix each issue you find in an independent commit. Write meaningful commit messages and close the bug report. You can close an issue by adding '`fixes #xxx`' to your commit message where `#xxx` refers to the issue number or by referring to the commit from the message with which you close the bug. You can read more about fixing issues with commits on the Github Blog. If you do not close the issue with a commit, please close the issue manually (on the GitHub issue tracker page) and refer to the commit on the issue page.

Evaluation

Overall this homework is worth 100 points. We will grade your work approximately as follows:

- Correctly applying the concepts of polymorphism and information hiding: 20 points
- Java best practices and compatibility with our informal specification: 40 points
- Unit testing, including coverage, reporting and fixing issues, and compliance with good testing practices: 30 points
- Documentation and style: 10 points

Additional hints:

- The TAs have graciously provided a GUI calculator that will "bring your operations to life on the screen." You may use this GUI calculator for informal testing, but the GUI should not supplant a formal testing process. You do not need to test our GUI code for Part 4.
- We will assess your line coverage with the Jacoco reports that can be generated with `gradle jacocoTestReport`. The reports can be found in `build/reports/coverage`. You might want to use the IDE integration of coverage while you write your tests.