Homework #2: From Calculators to Calculus

Due Tuesday, January 26th at 11:59 p.m.

In this assignment you will write various classes that implement the behavior of a calculator. This assignment has three 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 clases representing arithmetic expressions, using the operations you wrote in Part 1. Part 3 combines ideas from functional and object-oriented programming. In that part 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.

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, and demonstrate an understanding of basic software design principles.
- Write unit tests and automate builds and tests using JUnit, Gradle and Travis-CI.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics.
- Demonstrate good Java coding and testing practices and style.

This document continues by describing the three 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 these seven operators, (2) provide a UML class diagram that partially describes your operator class hierarchy, including all interfaces and abstract classes and your concrete implementations of addition and absolute value, and (3) complete a short program that passes a list containing all of your operators to the constructor for a GUI calculator class, `edu.cmu.cs.cs214.hw2.gui.Calculator`. See the `edu.cs.cs214.hw2.gui.Main.main` method for details.

When you are done you should be able to call the launch method on the resulting calculator and make sure your operators work; fix them if they don't. We recommend (but do not require) that you write additional operators to see how easy it is to extend the functionality of the calculator. Note that the calculator operators embody the strategy pattern; each operator represents a strategy for calculation.

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 informally test your implementation 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 also provided a command line calculator to help you informally test your solution. Similar to the GUI calculator from Part 1, the command line calculator takes a set of `Operators`; in `edu.cs.cs214.hw2.parser.CommandLineParser.main` you should add your operators to the operator set. The parser also needs to be able to construct the `Expressions` you write; implement the three functions in `edu.cmu.cs.cs214.hw2.parser.ExpressionMaker` by calling your respective `Expression` constructors, and then run the CommandLineParser program.

The parser requires that spaces be placed between numbers and binary operators:

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

Enter an expression
1+(2*(3-4)/5)-2
Input format not accepted. Please try again.
```

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

This part of the assignment has three sub-parts. You will first 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. Finally, 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 expression, essentially a named box to represent a value much like a variable represents a value in algebra. Complete that skeleton 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 in that function 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, much like evaluating a function in Part 2. 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:

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

and 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.

To complete this part, write 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 zero the returned zero 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 your implementation

You must formally test your solution using JUnit tests. When possible you should check the correctness of normal cases as well as edge cases of your implementation. Ideally you should achieve nearly 100% line coverage of your code, excluding user interface code, any code provided by the course staff, and the test code itself. If you do not achieve 100% coverage please explain with a short comment in each uncovered area why you can't achieve 100% coverage.

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 to even write your unit tests before you write your code, as you design your software specification.

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
- Compatibility of your Part 1 solution with your UML class diagram: 10 points
- Compatibility of your solution with our informal specification: 30 points
- Unit testing, including adequate coverage and compliance with best practices: 30 points
- Documentation and style: 10 points

Additional hints:

- It is possible (though not required) to use a single Java enum type to represent all your binary operators, and another enum type to represent all of your unary operators. Java enum types make this assignment easier and your code cleaner.
- 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 may hand-write (and photograph or scan) your UML class diagram or use a drawing program to generate UML. We will suggest some UML editors on Piazza.
- You are not required to throw any exceptions on faulty input arguments for this assignment. Your implementation may behave arbitrarily on incorrect input. You do not have to write tests for handling incorrect inputs.