ENGI 9874
Software Design and Specification

Assignment 2

Due Friday, October 26th, 2018 by 4:00 PM

Instructions

For each question you will be marked on programming style as well as correctness. To see my opinion about what constitutes good programming style see `http://www.engr.mun.ca/~theo/Courses/ds/pub/style.pdf`. In short:

- All .java files must be professionally commented; in particular, each file should contain a comment header that gives your name, student number, and mun email address. Each subroutine and class should have a comment at the start of it. I encourage you to use the "javadoc" conventions for comments.
- Code and comments must be consistently indented; tab stops should be set every 4 characters.
- Names must be chosen carefully and spelled correctly. (Use names starting with lower case letters for variables and methods; use names starting with upper case letters for classes and interfaces.)
- Use subroutines to avoid redundant coding.
- Keep control structures and data structures simple.

All classes must be tested by you prior to being submitted. You are welcome to share test code with each other.

The assignment is to be done alone. Each file should contain the following declaration in comments near the top. "This file was prepared by [your name here]. It was completed by me alone.". If you obtained help in doing the assignment, do not include this declaration, but rather an explanation of the nature of any help that you received in doing the assignment.

For each question submit a standard zip file containing the source code for all classes you wrote or modified.

This is an individual assignment. Complete it by yourself, alone.

Question 1: Observer and Command Patterns [50 marks]

(a) Create a Java interface for a mutable (changeable) sequence of integers called `IntSequence`. **[7 marks]**

It should be possible to:

- Find the current length of the sequence.
- Retrieve any sub-range of the sequence as an array of integers.
- Remove and sub-range of integers from the sequence.
- Insert an array of integers at any point in the sequence.
- Add an observer (see the Observer pattern).
- Remove an observer (see the Observer pattern).

(b) Create an implementation of the IntSequence interface that is observable. (This does not need to be an efficient implementation.) **[14 marks]**

(c) Create a set of JUnit tests that demonstrate that your class from (b) functions correctly. **[8 marks]**

(d) Use the Command pattern to make a class that implements the IntSequence interface but also supports undoing and redoing actions. You should resulse the class from part (b). **[14 marks]**

(e) Create a set of JUnit tests that demonstrate that your class from (d) functions correctly. You can reuse the tests from (c), but will also need additional tests to test undo/redo. **[7 marks]**

Question 2: Expressions—Composite and Abstract Factory Pattenrs [30 marks]

You will implement a set of immutable classes representing expressions in x, i.e., expressions with one free variable, x. Use at least one abstract class (with at least one abstract method). Your classes should be subtypes of the following interface `expr.Expression`.

```
interface Expression {
    double value(double x);
}
```

Your classes should be able to represent expressions such as $\sin(2x + \pi/2)$. Of course calling value(double) on an object representing this expression with an argument of say 0.39270 would give a value of 0.70711, which is $\sin(2 \times 0.39270 + \pi/2)$.

In addition to implementing the value(double) method, override the toString() method, which is inherited from java.lang.Object, so that your Expression objects can be converted to readable strings.

You should also write a class for producing objects that represent expressions. This class should be called `expr.ExpressionFactory`, have a no-argument constructor, and implement interface `expr.ExpressionFactoryI` provided.

So our example expression $\sin(2x + \pi/2)$ can be constructed, printed, and evaluated via the following code

```
ExpressionFactoryI f = new ExpressionFactory() ;

Expression a = f.multiply( f.constant(2.0), f.x() ) ;

Expression b = f.divide( f.constant( Math.PI), f.constant(2.0) ) ;

Expression c = f.sin( f.add( a, b ) ) ;

double x = 0.39270 ;

System.out.println( "The␣value␣of␣" +c+ "␣at␣" +x+ "␣is␣"\ +c.value(x) ) ;
```

The library class java.lang.Math will be helpful for doing the calculations.

Test your class with the supplied JUnit tests.

(Note on parentheses. In order that everything prints nicely, we use an explicit operator for parentheses. You may assume that the client that constructs expressions respects the conventional rules of parenthesization. For example, it would be an error to use an Expression returned by add as an argument of multiply. There is no need to check that these preconditions are respected, but you may if you wish.)

Question 3: Graphing [20 marks]

Create a class `expr.ChartData` with a 0 argument constructor and following methods:

- `public void setExpression( Expression a )`
- `public Expression getExpression( )`
- `public void setXRange( double xMin, double xMax )`
- `public double getXMin( )`
- `public double getXMax( )`
- `public void setYRange( double yMin, double yMax )`
- `public double getYMin( )`
- `public double getYMax( )`

The following class invariants should be respected:

```
getExpression() != null

Double.NEGATIVE_INFINITY < getXMin( )

getXMin( ) < getXMax( )

getXMax( ) <  Double.POSITIVE_INFINITY

Double.NEGATIVE_INFINITY\ < getYMin( )

getYMin( ) < getYMax( )

getYMax( ) <  Double.POSITIVE_INFINITY
```

The way you will use to ensure these invariants are respected is by using preconditions for the setter methods—i.e. we will put the responsibility on the objects' clients—and by using reasonable defaults in the constructor. These preconditions should be checked using class util.Assert, which I will supply. The class invariants should be checked by calling a private invariant method at the end of the constructor and of each mutator. The invariant method can use util.Assert to check the invariant.

Once you have implemented this class you should be able to test it with the supplied JUnit tests and run the supplied.

Submission: Zip (use zip format) up all files that you have modified or created and submit via D2L dropbox.