CSE 331 Software Design and Implementation

Homework 3: ***Java and Coding to Specifications***
Due: Friday April 19 @ **11pm** (except Problem 0)

Handout H3

Quick links:

- Specifications

Contents:

As always, we recommend that you read the entire homework before you begin work. It's long but it's in reasonable size chunks.

Introduction

This homework focuses on reading and interpreting specifications, and reading and writing Java source code. Additionally, you will be given an introduction to using `checkRep` methods and testing strategies. You will implement three classes that will complete the implementation of a graphing polynomial calculator, and you will answer questions about both the code you are given and the code you are going to write.

Beware, this is your first real programming assignment for this class. In previous quarters students have often found themselves surprised by how much work this assignment involves, and end up using late days. Needless to say, start early.

To complete this homework, you will need to know:

a. Basic algebra (rational and polynomial arithmetic), review of polynomial arithmetic here
b. How to read and write basic Java code
   - code structure and layout (class and method definition, field and variable

declaration)
- method calls
- operators for:
    - object creation: `new`
    - field and method access: `.`
    - assignment: `=`
    - comparison: `==`, `!=`, `<`, `>`, `?`, `?`
    - arithmetic: `+`, `-`, `*`, `/`
- control structure: loops (`while` and `for`) and conditional branches (`if`, `else`)

c. How to read procedural specifications (requires, modifies, effects)

Problems

Problem 0: Polynomial arithmetic algorithm (7 points)

Problem 0 is due on Friday April 12 @ 11pm, separately from the remainder of the homework. Commit the solution to problem 0 to your repository in the file `hw2/answers/problem0_hw3.txt` before the deadline.

For this problem you will write pseudocode algorithms for arithmetic operations applied to single-variable polynomial equations. We provide an example for polynomial addition below.

```
$r = p + q$:
    set $r = q$ by making a term-by-term copy of all terms in $q$ to $r$
    foreach term, $t_p$, in $p$:
        if any term, $t_r$, in $r$ has the same degree as $t_p$,
            then replace $t_r$ in $r$ with the sum of $t_p$ and $t_r$
            else insert $t_p$ into $r$ as a new term
```

You may use ordinary arithmetic operations on individual terms of polynomial equations without defining them yourself. For the above example, the algorithm uses addition on the terms t_p and t_r. Furthermore, after defining an algorithm, you may use it to define other algorithms. For example, if helpful, you may use polynomial addition within your algorithms for subtraction, multiplication, and division. Be sure your types are correct: if addition is defined over terms, and is defined over polynomials, that does not mean you can add a term to a polynomial unless you have also defined that case.

Answer the following questions:

a. Write a pseudocode algorithm for subtraction.
b. Write a pseudocode algorithm for multiplication.
c. Give an inductive proof of correctness for your multiplication algorithm. Although we will be sticklers in your pseudocode that you do not use operations without defining them, you are permitted to do so (such as adding a term to a polynomial) in the proof.

Here is an example inductive proof of correctness for the addition algorithm:

There are two parts to the proof: partial correctness and termination. Termination is trivial, because the loop is a foreach loop; no proof is necessary. We now proceed to partial correctness.

We first slightly modify the loop to add a loop-index-like variable *d*, called a "ghost variable". A ghost variable is convenient for a proof but has no effect on the algorithm's operation, because no real variable is ever set based on the ghost variable's value. The algorithm is now:

```
r = p + q:
    set r = q by making a term-by-term copy of all terms in q to r
    set d = 0
    foreach term, t_p, in p in order of increasing degree:
        if any term, t_r, in r has the same degree as t_p,
            then replace t_r in r with the sum of t_p and t_r
            else insert t_p into r as a new term
        set d = degree(t_p) + 1
```

We claim the following loop invariant:
$p[d..] + r = p + q$
where the notation a[i..] means all the terms in polynomial a with degree i or higher.

We have three properties to prove:

1. **precondition => LI**

   The loop precondition (after the algorithm's initialization statements) is $r=q$ and $d=0$. The loop invariant holds.

2. **{ LI & t_p exists } body { LI }**

   For brevity, proofs over foreach loops dispense with repeating the loop guard "t_p exists". The proof obligation (the fact we wish to prove) can then be written as:
   $\{ p_{pre}[d_{pre}..] + r_{pre} = p_{pre} + q_{pre} \}$ **body** $\{ p_{post}[d_{post}..] + r_{post} = p_{post} + q_{post} \}$

   Since *p* and *q* are not modified by the loop body, this simplifies to

   $\{ p[d_{pre}..] + r_{pre} = p + q \}$ **body** $\{ p[d_{post}..] + r_{post} = p + q \}$

   Either forward or backward reasoning works. Let's try forward.

   then clause: $r_{post} = r_{pre} - t_r + (t_r + t_p) = r_{pre} + t_p$

   else clause: $r_{post} = r_{pre} + t_p$

   The overall loop body postcondition is $r_{post} = r_{pre} + t_p$. Since $p[d_{post}..] =$

$p[d_{pre}..]$ - t_p, the loop invariant holds after execution of the loop body.

3. **LI => postcondition**

   Since d-1 is the maximum degree in p, $p[d..]$ = "0" and the postcondition, $r = p + q$, holds.

Note: We have proved that the algorithm is correct. An equally important, and sometimes more useful, property to prove is that the representation invariant is maintained by execution of the implementation. This depends on the code itself (the representation and implementation), rather than on the pseudocode algorithm.

d. Write a pseudocode algorithm for division. The following is the definition of polynomial division as provided in the specification of `RatPoly`'s `div` method:

> Division of polynomials is defined as follows:
>
> Given two polynomials u and v, with v != "0", we can divide u by v to obtain a quotient polynomial q and a remainder polynomial r satisfying the condition u = "q * v + r", where the degree of r is strictly less than the degree of v, the degree of q is no greater than the degree of u, and r and q have no negative exponents.
>
> For the purposes of this class, the operation "u / v" returns q as defined above.
>
> The following are examples of div's behavior:
>
> - (x^3-2*x+3) / (3*x^2) = 1/3*x (with r = "-2*x+3").
> - (x^2+2*x+15) / (2*x^3) = 0 (with r = "x^2+2*x+15").
> - (x^3+x-1) / (x+1) = x^2-x+2 (with r = "-3").
>
> Note that this truncating behavior is similar to the behavior of integer division on computers.

Also, see the Hints section for a diagram illustrating polynomial division. For this question, you do not need to handle division by zero; however, you will need to do so for the Java programming exercise.

e. Illustrate your division algorithm running on these two examples:
   - (x^3-2*x+3) / (3*x^2) = 1/3*x
   - (x^3+x-1) / (x+1) = x^2-x+2

   Be sure to show the values of all variables in your pseudocode at the beginning of each loop iteration.

   Here is an example illustration of the addition algorithm running on (2x^2 + 5) + (3x^2 - 4x):

```
p = (2x^2 + 5)
q = (3x^2 - 4x)
r = copy of q = (3x^2 - 4x)
foreach term, $t_p$, in p
    Iteration 1: $t_p$ = 2x^2, r = (3x^2 - 4x), p = (2x^2 + 5), q = (3x^2 -
        4x)
            [if any term, $t_r$, in $r$ has the same degree as $t_p$] YES, $t_r$ = 3x^2
            [then replace $t_r$ in $r$ with the sum of $t_p$ and $t_r$] $t_p$ + $t_r$ = 5x^2, so
            now $r$ = (5x^2 - 4x)
            [else insert $t_p$ into $r$ as a new term]
        Iteration 2: $t_p$ = 5, $r$ = (5x^2 - 4x), $p$ = (2x^2 + 5), $q$ = (3x^2 - 4x)
            [if any term, $t_r$, in $r$ has the same degree as $t_p$] NO
            [then replace $t_r$ in $r$ with the sum of $t_p$ and $t_r$]
            [else insert $t_p$ into $r$ as a new term] $r$ = (5x^2 - 4x + 5)
    We are done! $r$ = (5x^2 - 4x + 5)
```

(Notice that the values of *p* and *q* did not change throughout the execution of the algorithm. Thus, this algorithm works when *p* and *q* are required to be immutable (unchanged). You will learn about immutable objects as you progress on this homework.)

Since your answer will be ASCII text, you won't be able to use italics and subscripts in your answer. You can dispense with italics, and you can represent "t_p" as "`t_p`", for example.

Problem 1: RatNum (26 points)

Problems 1-5 require some setup. Specifically, Update the `src` directory in your SVN repository. Then, work through the problems below, using the provided Specifications to help you fill in the missing methods.

For this first problem you don't have to write any code, but you do have to answer written questions. Read the specifications for **`RatNum`**, a class representing rational numbers. Then read over the provided implementation, RatNum.java.

You will likely want to look at the code in RatNumTest.java to see example usages of the RatNum class (albeit in the context of a test driver, rather than application code).

Answer the following questions, writing your answers in the file `hw3/answers/problem1.txt`. **Two or three sentences should be enough to answer each question. For full credit your answers should be short and to the point. Points will be deducted for answers that are excessively long or contain irrelevant information.**

a. What is the point of the one-line comments inside the `add`, `sub`, `mul`, and `div` methods?
b. `add`, `sub`, `mul`, and `div` all require that "arg != null". This is because all of the methods access fields of 'arg' without checking if 'arg' is null first. But the methods also access fields of 'this' without checking for null; why is "this != null" absent from the requires-clause for the methods?
c. `RatNum.div(RatNum)` checks whether its argument is NaN (not-a-number). `RatNum.add(RatNum)` and `RatNum.mul(RatNum)` do not do that. Explain.
d. Why is `RatNum.valueOf(String)` a static method? What alternative to static methods would allow one to accomplish the same goal of generating a RatNum from an input String?
e. Imagine that the representation invariant were weakened so that we did not require

that the `numer` and `denom` fields be stored in reduced form. This means that the method implementations could no longer assume this invariant held on entry to the method, but they also no longer would be required to enforce the invariant on exit. The new rep invariant would then be:

```
// Rep Invariant for every RatNum r: ( r.denom >= 0 )
```

List the method or constructor implementations that would have to change? For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given spec; in particular, `RatNum.toString()` needs to output fractions in reduced form.

f. `add`, `sub`, `mul`, and `div` all end with a statement of the form `return new RatNum ( numerExpr , denomExpr);`. Imagine an implementation of the same function except the last statement is:

```
this.numer = numerExpr;
this.denom = denomExpr;
return this;
```

For this question, pretend that the `this.numer` and `this.denom` fields are not declared as `final` so that these assignments compile properly. How would the above changes fail to meet the specifications of the function (Hint: take a look at the `@requires` and `@modifies` statements, or lack thereof.) and fail to meet the specifications of the RatNum class?

g. Calls to `checkRep` are supposed to catch violations in the classes' invariants. In general, it is recommended that one call `checkRep` at the beginning and end of every method. In the case of `RatNum`, why is it sufficient to call `checkRep` only at the end of the constructors? (Hint: could a method ever modify a `RatNum` such that it violates its representation invariant? Could a method change a `RatNum` at all? How are changes to instances of `RatNum` prevented?)

Problem 2: RatTerm (30 points)

Read over the specifications for the **`RatTerm`** class, making sure you understand the overview for RatTerm and the specifications for the given methods.

Read through the provided skeletal implementation of RatTerm.java , especially the comments describing how you are to use the provided fields to implement this class.

Fill in an implementation for the methods in the specification of `RatTerm`. You may define new private helper methods as you like. You may not add public methods; the external interface must remain the same.

For all of this assignment, if you define new methods, you must specify them completely. You can consider the specifications of existing methods (where you fill in the body) to be adequate. You should comment any code you write, as needed; please do not over-comment.

We have provided a `checkRep()` method in RatTerm that tests whether or not a RatTerm instance violates the representation invariants. We highly recommend you use `checkRep()` where appropriate in the code you write. Think about the issues discussed in the last question of problem 1 when deciding where `checkRep` should be called.

We have provided a fairly rigorous test suite in RatTermTest.java. You can run the given test suite with JUnit to evaluate your progress and the correctness of your code.

Answer the following questions, writing your answers in the file `hw3/answers/problem2.txt`. **Remember, keep your answers to 2-3 sentences.**

a. Where did you include calls to `checkRep` (at the beginning of methods, the end of methods, the beginning of constructors, the end of constructors, some combination)? Why?
b. Imagine that the representation invariant was weakened so that we did not require `RatTerm`s with zero coefficients to have zero exponents. This means that the method implementations could no longer assume this invariant held on entry to the method, but they also no longer were required to enforce the invariant on exit. Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given spec; in particular, `RatTerm.toString()` still cannot produce a term with a zero coefficient (excluding `0`).
c. In the case of the zero `RatTerm`, we require all instances to have the same exponent (`0`). No such restriction was placed on `NaN` `RatTerm`'s. Imagine that such a restriction was enforced by changing the representation invariant to include the requirement:

   ```
   coeff.isNaN() ==> expt = 0.
   ```

   This means that the method implementations could assume this invariant held on entry to the method, but they would also be required to enforce the invariant on exit. Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given spec (except for the part where terms like `NaN*x^74` are explicitly allowed).

   Which set of `RatTerm` invariants (coeff.isNaN() ==> expt = 0; coeff.equals(RatNum.ZERO) ==> expt = 0; both; neither) do you prefer? Why?

Problem 3: RatPoly (45 points)

Following the same procedure given in Problem 2, read over the specifications for the **`RatPoly`** class and its methods and fill in the blanks for RatPoly.java. The same rules apply here (you may add private helper methods as you like). Since this problem depends on problem 2, you should not begin it until you have completed problem 2 (and the `hw3.test.RatTermTest` test suite runs without any errors).

You may also want to take a look at the specifications for the **`java.util.List`** class, especially the `get()`, `add()`, `set()`, and `size()` methods.

You are welcome to do what you like with the private helper methods that we give you in `RatPoly` (`scaleCoeff`) and the like; you may implement them exactly as given, implement variants with different specifications, or even delete them; and you may add your own private helper functions. However, you must make sure that every private helper function in the final version of the class has an accurate specification and is not still an unimplemented skeleton.

Make sure your code passes all the tests in RatPolyTest.java.

Problem 4: RatPolyStack (20 points)

Following the same procedure given in Problem 2, read over the specifications for the RatPolyStack class and its methods and fill in the blanks for RatPolyStack.java. The same rules apply here (you may add private helper methods as you like). Since this problem depends on problems 2 and 3, you should not begin it until you have completed problems 2 and 3 (and the `hw3.test.RatTermTest` and `hw3.test.RatPolyTest` test suites run without any errors).

Make sure your code passes all the tests in RatPolyStackTest.java.

Problem 5: CalculatorFrame (1 point)

Now that RatPoly and RatPolyStack are finished, you can run the calculator application. This allows you to input polynomials and perform arithmetic operations on them, through a point-and-click user interface. The calculator graphs the resulting polynomials as well.

When you run `hw3.CalculatorFrame`, a window will pop up with a stack on the left, a graph display on the right, a text area to input polynomials into, and a set of buttons along the bottom. Click the buttons to input polynomials and to perform manipulations of the polynomials on the stack. The graph display will update on the fly, graphing the top four elements of the stack.

Submit your four favorite polynomial equations, in the `RatPoly.toString` format, in the file `hw3/answers/problem5.txt`.

Problem 6: Invariant detection (4 points)

Daikon is a software engineering tool that reports properties about programs similar to representation invariants and method pre- and postconditions. Please read the introduction to Daikon.

For this problem, you will generate and analyze the results of running Daikon on the `RatNum` class using two different test suites: `RatNumTest`, the original test suite, and `RatNumSmallTest`, a reduced test suite that does not provide as much coverage.

To ease your work, we have provided you with `ant` targets to run Daikon with the correct arguments for this particular problem. Like most of the other targets (e.g. "build"), these targets can be run from either Eclipse or the command line.

First, run Daikon from your hw3 directory using the original RatNum test suite (`RatNumTest`):

```
ant daikon-RatNumTest
```

This command will place Daikon's results in `daikon-RatNumTest.inv.txt` in your `hw3` directory. If you are using Eclipse, you will need to refresh the `hw3` directory to see the results file in the **Package Browser**.

Next, run Daikon on the reduced RatNum test suite (`RatNumSmallTest`):

```
ant daikon-RatNumSmallTest
```

This command will place Daikon's results in `daikon-RatNumSmallTest.inv.txt` in the `hw3` directory. If you are using Eclipse, you will need to refresh the `hw3` directory to see the results file in the **Package Browser**.

Please answer the following questions in the file `hw3/answers/problem6.txt`:

a. Examine the representation invariants Daikon generates for `RatNum` using the original test suite. These can be found at the top of the `daikon-RatNumTest.inv.txt` file under the heading `hw3.RatNum:::OBJECT`. Does Daikon generate the same invariants that are recorded in the representation invariant of `RatNum`? How do they differ?
b. Compare the invariants generated by Daikon at the entrance of the `RatNum.sub` method (found under the heading `hw3.RatNum.sub(hw3.RatNum):::ENTER`), using the two different test suites. How are the invariants generated by the two test suites different? What do the differences tell you about the test cases that are missing from the reduced test suite for `RatNum.sub`? That is, what case is the reduced test suite not testing?

Collaboration (1 point)

Please answer the following questions in a file named `collaboration.txt` in your `hw3/answers/` directory.

The standard collaboration policy applies to this homework.

State whether or not you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

Reflection (1 point)

Please answer the following questions in a file named `reflection.txt` in your `hw3/answers/` directory. Answer briefly, but in enough detail to help you improve your own practice via introspection and to enable the course staff to improve CSE 331 in the future.

a. In retrospect, what could you have done better to reduce the time you spent solving this homework?
b. What could the CSE 331 staff have done better to improve your learning experience in this homework?
c. What do you know now that you wish you had known before beginning the homework?

Time Spent

Tell us how long you spent on this homework via this catalyst survey: https://catalyst.uw.edu/webq/survey/mernst/198073.

Returnin (25 points)

After you receive your corrected homework back from your TA, you will have until **Tuesday, May 8 @ 11pm** before you have to resubmit it electronically by committing the homework to your repository and then running returnin331. Returnin will be enabled on **Tuesday, May 1 @ 11pm** for this homework.

You will only receive credit if you pass all the tests, and you have a fixed number of trials without penalty. See the returnin331 documentation for details.

Provided classes

The following classes are all provided for you. Take a look at the Javadoc specifications for more information on how to use them.

With source code provided:

- hw3/RatNum.java
- hw3/PolyGraph.java
- hw3/CalculatorFrame.java
- hw3/test/RatNumTest.java
- hw3/test/RatNumSmallTest.java
- hw3/test/RatTermTest.java
- hw3/test/RatPolyTest.java
- hw3/test/RatPolyStackTest.java
- hw3/test/ImplementationTests.java
- hw3/test/SpecificationTests.java

Hints

You are free to work from home, but keep in mind that your final code must run correctly on the lab machines (we'll test it on attu). Once everything is in order, make sure you've committed your changes. You should also run `ant validate` on attu as a double-check.

Your answers to problem 0 should have gone into **Homework 2** subdirectory as `hw2/answers/problem0_hw3.`**`txt`**.

By the end of the homework, you should have the following files committed to SVN in your hw3 directory:

- `answers/problem1.txt`
- `RatTerm.java`
- `answers/problem2.txt`
- `RatPoly.java`
- `RatPolyStack.java`
- `answers/problem5.txt`
- `answers/problem6.txt`
- `answers/collaboration.txt`
- `answers/reflection.txt`

Don't forget to use the forum.

If you're having trouble running JUnit tests in Eclipse, try running them from the command line. Instructions are in the Editing and Compiling handout.

All of the unfinished methods in RatTerm, RatPoly and RatPolyStack throw **`RuntimeException`**s. When you implement a method, you should be sure to remove the `throw new RuntimeException();` statement. For those of you who use Eclipse, we have also added a `TODO:` comment in each of these methods. The "Tasks" window will give you a list of all `TODO:` comments, which will help you find and complete these methods.

Think before you code! The polynomial arithmetic functions are not difficult, but if you begin implementation without a specific plan, it is easy to get yourself into a terrible mess.

The most important method in your `RatPoly` class will probably be `sortedInsert`. Take special care with this method.

The provided test suites in this homework are the same ones we will use to grade your implementation; in later homeworks the staff will not provide such a thorough set of test cases to run on your implementations, but for this homework you can consider the provided set of tests to be rigorous enough that **you do not need to write your own tests**.

Division of polynomials over the rationals is similar to the long division that one learns in grade school. We draw an example of it here:

$$\frac{x^8+x^6+10x^4+10x^3+8x^2+2x+8}{3x^6+5x^4+9x^2+4x+8} \Rightarrow$$

$$\begin{array}{r|ccccccccc}
 & & & & & & & \frac{1}{3} & 0 & -\frac{2}{9} \\ \hline
3\ 0\ 5\ 0\ 9\ 4\ 8 & 1 & 0 & 1 & 0 & 10 & 10 & 8 & 2 & 8 \\
 & 1 & 0 & \frac{5}{3} & 0 & 3 & \frac{4}{3} & \frac{8}{3} & & \\ \cline{4-10}
 & & & -\frac{2}{3} & 0 & 7 & \frac{26}{3} & \frac{16}{3} & 2 & 8 \\
 & & & -\frac{2}{3} & 0 & -\frac{10}{9} & 0 & -2 & -\frac{8}{9} & -\frac{16}{9} \\ \cline{6-10}
 & & & & & \frac{73}{9} & \frac{26}{3} & \frac{22}{3} & \frac{26}{9} & \frac{88}{9}
\end{array}$$

$$\Rightarrow$$

$$x^8+x^6+10x^4+10x^3+8x^2+2x+8 =$$
$$\left(\frac{1}{3}x^2-\frac{2}{9}\right)(3x^6+5x^4+9x^2+4x+8)+\left(\frac{73}{9}x^4+\frac{26}{3}x^3+\frac{22}{3}x^2+\frac{26}{9}x+\frac{88}{9}\right)$$

Errata

In part c of Problem 0, in the example proof of addition, the description of step 1 (precondition => LI) now reads "The loop precondition ... is r=q and d=0" whereas it was originally "r=q and d=-1." This does not change the correctness of the proof but is more consistent (see "HW3: Problem0c" on the message board for further discussion).

Q & A

None yet.

For problems or questions regarding this page, contact: cse331-staff@cs.washington.edu.