The CSC461 Scala, Composite and Chain of Responsibility patterns

DUE: Wednesday, December 5th, at 8AM

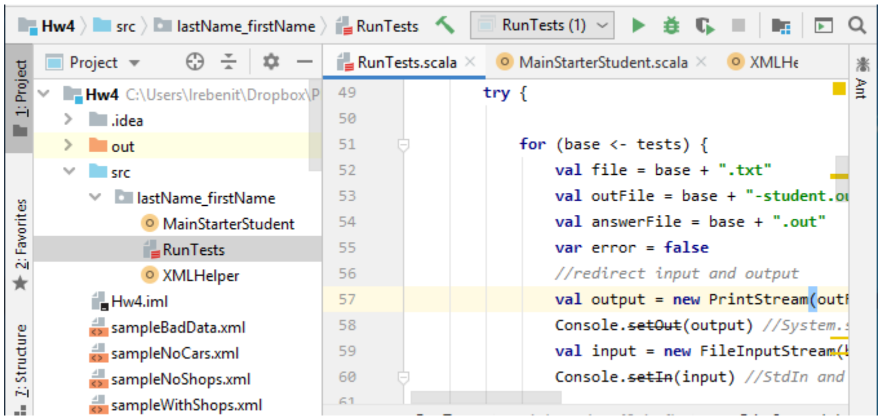
The purpose of this assignment is to give you practice with Scala, XML and the composite/RDP and chain of responsibility patterns.

Overview

You will be coding insurance estimation software. To keep the project size reasonable, there will only be a small subset of items considered. The first part of the assignment will be to load and write insurance information that is stored in an XML file. See section *XML Format* for format of the XML and the information that will be stored. The second part is to add/remove data from the insurance information. The third part will be performing a few pieces of insurance functionality. In summary,

1. Load XML insurance data
2. Write out the insurance data to an XML file
3. Make a string of all information, except car shops
4. Add insurance info
5. Delete a zip code
6. Find the first mechanic shop handling a specific code
7. Determine the value of all cars within a given zip code
8. Determine the monthly insurance rate for a customer

I will be posting a **RunTests.scala** and a **MainStarterStudent.scala** file for built in tests. **MainStarterStudent** purpose is to give a set entry point. It must remain the entry point **in the root of the lastName_firstName** folder. I do NOT have the test requiring a command line argument this time. They will run if you select **RunTests.scala** as the entry point, period. I cannot check the XML files with this, as the output is not formatted. Therefore, check your XML work by reformatting (ctrl+alt+L) and then do a diff check with IntelliJ's built-in diff checker. Below is the structure you should have when starting:



```
Hw4 C:\Users\lrebenit\Dropbox\P
  .idea
  out
  src
    lastName_firstName
      MainStarterStudent
      RunTests
      XMLHelper
  Hw4.iml
  sampleBadData.xml
  sampleNoCars.xml
  sampleNoShops.xml
  sampleWithShops.xml
```

```
        try {

            for (base <- tests) {
                val file = base + ".txt"
                val outFile = base + "-student.o
                val answerFile = base + ".out"
                var error = false
                //redirect input and output
                val output = new PrintStream(outF
                Console.~~setOut~~(output) //System.
                val input = new FileInputStream(
                Console.~~setIn~~(input) //StdIn and
```

Tip: review the collection functions. Your code will likely be 30% less in size if you use them.
Aside: you may ignore deprecations. I have 2 deprecated calls in RunTests. The short explanation is Scala is a young language. If you are curious about the *long* answer, please ask.

Required functionality

Initially, the insurance data will be empty. The menu (given) must be in the following format:

```
1) Load XML
2) Write XML
3) Display data
4) Add data
5) Remove zip code
6) Find service
7) Total value insured
8) Insurance for
0) Quit
```

You may assume all good console input this time, although the XML files may be faulty.

Load XML

To load an xml file, first ask for the file name, and then load it. The format must be

```
File name: <insertFileNameHere>
```

The menu should then reprint after loading. You do NOT need to worry about loading a second file.

This must be done with the RDP technique. Tag the start of the read RDP chain ONCE with GRADING: READ.

XML loading Errors

The XML file is not guaranteed to exist or to be an insurance file. The following are errors that may occur and how you must handle them:

- Calling display (choice 3) before any contents are loaded
  - Should repeat the menu with no warning
- The file cannot be opened
  - Output: `Could not open file: errorMessage`
  - You can get the errorMessage with `e.getMessage`
- The topmost tag is not <insurance>
  - Output: `Invalid XML file. Needs to be an insurance XML file`
- There may be additional tags throughout
  - These are to be ignored/skipped
- There may be additional text throughout
  - These are to be ignored/skipped
- There may be additional attributes throughout
  - These are to be ignored/skipped

Aside: To better meet functional paradigms, this would normally be moved into a constructor with a Node parameter. This way, the class could be immutable. However, this project is just as much about the composite and chain patterns. Therefore, this is being done the imperative way.

Write XML

To write an XML file, first ask for the file name, and then save the current insurance contents in the named file. The format must be

```
File name: <insertFileNameHere>
```

This must be done with the RDP technique. Tag the starting RDP write ONCE in your code with GRADING: WRITE.

Display Data

Display the contents of the file except for the car shops. A car uses a spacing of 15 characters for each value. For each nested level, preface another ---. For example,

```
Zip Code: 57701
---Name: Harold
------Cars:
---------Make: Taurus         Model: Ford            Year: 2015            Value: $1000
------Accidents:
---------Accident date: 12-16-16
```

If there are no cars/accidents, do not output “Cars”/“Accidents:”. Also, this MUST be called in your menu similarly to: `println(data.getInfo(…))`. This may NOT print inside the classes (which is actually code smell since it decreases testability). Some help on this:

- Use the collection map() function with `mkString(…)`
- You can multiply stings in Scala just like you can in Python. Therefore `“---” * 2` will output `“------”` You just need to know the level.

This must be done with the RDP technique. Tagthe starting RDP function for string creation with GRADING: PRINT.

If you are wondering why I’m not requiring the toString() function to be overridden, toString() typically works within a single class, and this is altered with the level. It *is possible* to handle it with toString(), but is some tricky code.

Add Info

When you add info, ask for the current item's information. If it is found, continue to its components. If it is NOT found, add it to the data, and reprint the menu. The XML output should reflect this change. For example, to add a new zip code with the assumption that no data has been loaded:

```
What zip code: 57703
Added zip code
```

Then if you display, it would show

```
Zip Code: 57703
```

Then to add a new owner to 57703 (this MUST be case **in**sensitive)

```
What zip code: 57703
What owner: Harold
Added owner
```

Then to add a new car and accident for Harold (this MUST be case **in**sensitive)

```
What zip code: 57703
What owner: harold
Add Car y/n? y
Make: Chevy
Model: Volt
Year: 2017
Value: 45000
Add Accident y/n? y
Date: 3/22/2018
```

This should be case insensitive to y, and ignores everything else. After all of this, displaying the data should output this:

```
Zip Code: 57703
---Name: Harold
------Cars:
---------Make: Chevy          Model: Volt          Year: 2017          Value: $45000
------Accidents:
---------Accident date: 3/22/2018
```

This must be done with the RDP technique. Tag ALL functions that are used for the RDP in your code with GRADING: ADD.

Remove Zip Code

When you remove a zip code, ask for the code, and then output that the code was removed. The format must be

```
What zip code: 57703
Removed 57703
```

The XML output should reflect this change.

If you are wondering why this does not have the level of detail "adding" does, it is to decrease the size of the project.

Find service

Find service should ask for the service code and then find the first car shop in the list that has that service. Then, it outputs the name of the car shop and the description (this is the nested text of a service tag). This MUST be done using the chain of responsibility pattern. This means using the **linked list** version with an embedded "next" reference. There should be NO for-each loop to traverse the carShops to accomplish this task. You may use the Scala LinkedBuffer data type rather than building up the list yourself, but you **must still use recursion**. Sample code is provided. The format must be

```
Code: 1
"Easy Fix" handles code: 1. Description: Oil change: $36
```

"Easy Fix" is the name of the shop. You must be able to handle not locating a service by outputting

```
Service not found
```

Note: you will have to load car shops in before this can work. The car shops are not printed to the screen.

This must be done with the "chain of responsibility" technique. Tag the start of the search with GRADING: FIND. Tag the jumping to the next item with GRADING: CHAIN.

Total value insured

This should first ask for the zip code to sum the car values. The value is formatted to two decimal places with commas. For example, for a zip code with nothing in it,

```
What zip code: 5
Value: $0.00
```

For a zip code with $5,000 total dollars,

```
What zip code: 57701
Value: $5,000.00
```

This MUST be done in parallel. You may convert the data to a parallel at any point you wish. Reminder: use Java's decimal formatter to format the output.

Tag your parallelization with GRADING: TOTAL.

Insurance For

To get the monthly payment for a person, first ask for their zip code, and then the name to find the owner (case insensitive). The monthly payment is calculated as the car's total value times 1% plus the car's total value times 1% times the number of accidents. In other words: carsTotal * 0.01 + carsTotal * 0.01 * accidentCount. The output format should be

```
What zip code: 57701
What owner: Harold
Monthly payment: $20.00
```

Searching for an owner MUST be done in parallel. You may convert the data to a parallel at any point. Reminder: use Java's decimal formatter.

Tag your parallelization with GRADING: INSURANCE.

XML Format

The information you will be working on includes the following:

- Zip codes, which have a code (integer)
- Owners, which have a name, and 0+ cars, and 0+ accidents
- A Car, which has make, model, year, and value (integer)
- An accident, which has a date (no required format)
- A CarShop, which has a name and services
- A service, which has a code and a description with its price

The outer most tag must be insurance. The remaining location of the data will be as follows:

```
<insurance>
    <zip code="57701">
        <owner name="Harold">
            <car make="Taurus" year="2015" value="1000" model="Ford"></car>
            <accident>
                <date>12-16-16</date>
            </accident>
        </owner>
        …more owners,,,
    </zip>
    …more zip codes…
    <carShop name="Easy Fix">
        <service code="1">Oil change: $36</service>
         …more services…
    </carShop>
    …more carShops…
</insurance>
```

There is NO guarantee of order of nested tags. You MUST process these in the order of the file using the technique shown in class. This is so that converting to streaming data would be readily doable in the "future."

Tip: Ctrl+Alt+L will reformat the XML in IntelliJ for you.

You MUST output with the zip codes before the car shops.

Additional restrictions

- You MUST put your code into a package named your lastName_firstName (lowercase with no prefixes or suffixes.
- While some ***minor*** differences in spacing and newlines are expected, the case, wording, error messages, and the output from displaying the content must be exact. Points will be docked otherwise.

Grading Tiers

These tiers start with the simplest tasks, and go to the most involved. Please refer to the rubric for the tiers. You must "reasonably" complete the lower tiers before gaining points for the later tiers. By "reasonably," I can reasonable assume you honestly thought it was complete.

Submission instructions

1. Check the coding conventions before submission.
2. If anything is NOT working from the following rubric, please put that in the bug section of your main file header (it helps with partial credit)
3. All of your Scala files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes)
4. Delete the out folders, then zip your **ENTIRE PROJECT** into ***ONE*** zip folder named your lastName_firstName (lowercase with no prefixes or suffixes). Make sure this matches your package name!

   If you are on Linux, make sure you see "hidden folders" and that you grab the ".idea" folder. That folder is what actually lists the files included in the project!

5. Submit to D2L. The dropbox will be under the associated topic's content page.
6. *Check* that your submission uploaded properly. No receipt, no submission.

You may upload as many times as you please, but only the last submission will be graded and stored

If you have any trouble with D2L, please email me (with your submission if necessary).

Rubric

Item	Points
1. Outer 2 most XML nodes (ignoring car shops) loaded (tier1.txt)	12
a. Able to load in a zip code and owners (2 each)	4
b. Able to write out a zip code and owners (2 each)	4
c. Able to handle a bad file (no file, or not an insurance file) (2 each)	4
2. XML nodes other than car shops loaded: (tier2.txt)	18
a. Able to load in other than car shops	4
b. Able to write out other than car shops	4
c. Nodes handled in the order given	4
d. Ignores additional tags, text, and attributes	2
e. RDP*	4
3. Able to display formatted information (tier3.txt)	23
a. All info there and returned as a string	2
b. Able to print out most data other than car shops loaded (unformatted)	2
c. Able to print the zip code and owners (unformatted)	3
d. Correctly indented (3 if some cases work)	6
e. Handles “no accident/car case” properly (2pt each)	4
f. Nothing printed when nothing is loaded	2
g. RDP*	4
4. Add info (tier4.txt)	16
a. Case insensitive	2
b. Able to add zip, owner, car, accident (2 each)	8
c. New data is appended when writing to XML	2
d. RDP*	4
5. Remove zip code (tier5.txt)	4
a. Able to remove zip code	2
b. Data is no longer there when writing to XML	2
6. Find service (tier6.txt)	13
a. Loaded and able to write out (2 each)	4
b. Service found and correct output	2
c. Service no found	2
d. Chain of responsibility used *	5
7. Total value insured (tier7.txt)	7
a. Correct with empty zip code	2
b. Correct with filled zip code	2
c. Parallelized *	3
8. Insurance for (tier8.txt)	7
a. Owner found (Case insensitive)	2
b. Correct insurance	2
c. Parallelized *	3
Total	100

* these have required tagging in the comments