MINNESOTA INCOME TAX CALCULATION PROJECT

REENGINEERING THE LEGACY CODE

	Date 10/19/2020

GOAL OF THE PROJECT

The goal of this project is to reengineer a legacy Java application. At a glance, serves for the income tax calculation of the Minnesota state citizens. The tax calculation accounts for the marital status of a given citizen, his income, and the amount of money that he has spend, as witnessed by a set of receipts declared along with the income. The legacy application takes as input txt or xml files that contain the necessary data for each citizen. The tax calculation is based on a complex algorithm provided b the Minnesota state. The application further produces graphical representations of the data in terms of bar and pie charts. Finally. the application produces respective output reports in txt or xml.

PHASE 1 TASK LIST

1. **[Skim the documentation] The legacy application has been developed based on a more detailed requirements specification that is available along with the application source code (MinessotaIncomeTaxCalculation-Requirements.pdf). In a first step, study the documentation to get more information concerning the application's architecture and use cases.**

2. **[Do a mock installation] The application source code is provide as an eclipse project (Minnesota Income Tax Calculation Project folder). Few test input files are also available (InputFiles folder). Setup a running version of the project and test it based on the given input files.**

3. **[Build confidence] Read all the source code once and try to understand the legacy architecture, the role/responsibilities of each class, and so on.**

4. **[Write automated JUnit tests for the classes that will be refactored to understand and enable evolution] Prepare more test cases that will allow you to understand and test the inner workings of the classes. More specifically:**

   - Write JUnit tests for the public methods of the Database class.
   - Write JUnit tests for the public methods of the Taxpayer class.
   - Write JUnit tests for the public methods of the InputSystem class.

- **HINT** to test a method that reads something from a file you should verify that the data that are read are actually loaded in the main memory objects. So the expected result is that the state of the main memory objects matches the data read from the file.

- Write JUnit tests for the public methods of the OutputSystem class.

  - **HINT** to test a method that writes something to a file you should verify that the data that are written are actually the ones that are stored in the main memory objects. So the expected result is that the contents of the file match the state of the main memory objects written to the file.

  - **HINT** to test a method that creates a chart you should verify that the contents of the chart match the state of the main memory objects depicted to the chart. To this end you can use the getter methods of the OutputSystem class to gain access to the charts and the datasets of the charts.

5. **[Capture the design] Specify the legacy architecture in terms of a UML package diagram. Specify the detailed design in terms of UML class diagrams. Prepare CRC cards that describe the responsibilities and collaborations of each class.**

PHASE 2 TASK LIST

6. **The Taxpayer class has a lot of code duplication (https://refactoring.guru/smells/duplicate-code).**

   - Observe that all the calculateTax*() methods are very similar; difference is only in certain constants. Remove this sort of duplication. The goal is to make the similar methods one single method that is called multiple times, one for each different family status. The method should be parameterized (https://refactoring.guru/parameterize-method) with an array that contains the income limits and an array that contains the income tax rates that correspond to a particular family status. Then, you can call the method using different arrays for the different family statuses.

   - Observe that the get*ReceiptsTotalAmount() methods are also very similar; Remove this sort of duplication too using method parameterization (https://refactoring.guru/parameterize-method).

7. **The Taxpayer class suffers from primitive obsession (https://refactoring.guru/smells/primitive-obsession). A lot of constants for the income limits and and income tax rates, are used to calculate the tax of the Taxpayer. The values of these constants depend on the value of the family status attribute. Essentially the family status acts as a typecode for different kinds of Taxpayers.**

- **HINT** To deal with this problem you can Replace Typecode with Class (https://refactoring.guru/replace-type-code-with-class). Specifically instead of having a String familyStatus attribute in the Taxpayer class you can create a new class FamilyStatus with class attributes the income limits and the income tax rates. The FamilyStatus class itself can further have 4 different static final fields that are objects of FamilyStatus, one for each different status (SINGLE, HEAD_OF_HOUSEHOLD, MARRIED_FILLING_JOINTLY, MARRIED_FILLING_SEPARATELY), and a static method that returns the right object based on the value of a family status string parameter. You can use this static method when you create a new Taxpayer object, to set the value of the Taxpayer object family status field. You can instantiate the 4 different static fields of the FamilyStatus class at the bootstrap of the application using a properties files that contains the constants.

8. **The different subclasses of Receipt are trivial (https://refactoring.guru/smells/lazy-class); Get rid of these classes and simplify the design respectively (https://refactoring.guru/inline-class).**

9. **The Database, the InputSystem and the OutputSystem classes do not really follow the object oriented style. The main problem is that all of them just define a bunch of static methods that operate on static class attributes. All these static methods and attributes prevent further refactoring that will allow to separate the class responsibilities into smaller classes and get rid of code duplication by extracting abstract classes and template methods (see next).**

    - **HINT** Turn the static methods and attributes into normal instance method and attributes. Replace the calls to the static methods to normal object calls. Normally, the application needs a single Database, InputSystem and OutputSystem object. To implement this constraint and to provide a global point of access to these objects you can use the Singleton pattern (https://refactoring.guru/design-patterns/singleton) for each one of the three classes. The static methods and attributes involved in the Singleton pattern make sense and do not compromise the object-oriented style.

10. **The InputSystem class has too many responsibilities. Observe that it is used to parse two different formats (XML and TXT).**

    - **HINT** Split the parsing in different classes; try to avoid code duplication in these classes. To achieve this you can use the Form Template Method refactoring (https://refactoring.guru/form-template-

method). In a next step you can try to further minimize the code duplication and derive a single class for parsing, which is parameterized the with the appropriate tags (e.g. "Name:" vs "<Name>", "</Name>") that are needed for the parsing of the different formats.

11. **The OutputSystem class has too many responsibilities. Observe that it is used to generate reports in two different formats. Moreover, it creates the graphical representation of the data (pie charts, bar charts and so on).**

    - **HINT** Split these responsibilities in 3 different classes. For the generation of the reports try to avoid code duplication by parameterizing the code with the appropriate tags (e.g. "Name:" vs "<Name>", "</Name>") that are needed for the different formats.

12. **Prepare deliverables:**

    - A **report** based on the given template (Project-Deliverable-Template.doc). For the delivery of the report include a pdf of the report in the project folder.

    - A small **DEMO video**, about 15' minutes, using a screen capture tool like ActivePresenter for example. In the demo you should illustrate that the user stories of the application are still working after the refactoring. During the demo further explain in the code how you dealt with the reengineering problems of the application. Moreover you should state which of the problems you DID NOT handle. Finally, demonstrate the execution of the JUnit tests that you prepared for the project.

        - For the delivery of the demo make an account - if you do not have one - on GitHub (or Google drive). Put the demo video on GitHub. In the project folder that you turnin include a txt file named **DEMO-LINK.txt** that contains the link that points to the GitHub folder that contains the video.

    - Turn in the **refactored project** and the other deliverables using **turnin deliverables@mye004 <your-project>.zip**, where your-project is a zip file of your Eclipse refactored project.