MINESSOTA INCOME TAX CALCULATOR

REENGINEERING THE LEGACY CODE

GOAL OF THE PROJECT

The goal of this project is to reengineer a Java application. At a glance, the application 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 by 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**.

PREPARATION

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 (MinnesotaIncomeTaxCalculation-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 provided as an eclipse project (2023-IncomeTaxCalculatorProject folder). Setup a running version of the project and try to use its basic functionalities.

3. Build confidence:

    a. Read all the source code once and try to understand the legacy architecture, the role/responsibilities of each class, and so on.

    b. Specify the use cases as they are implemented in the legacy applications.

    c. Capture the legacy architecture in terms of a UML package diagram.

    d. Specify the detailed design in terms of UML class diagrams.

4. **Implement tests for the use cases of the application.**

	Date 9/29/2022

REFACTORING

Refactor the application so as to fix the following problems:

incometaxcalculator.data.management package:

1. **Company class**: the problem here is **Unnecessary Complexity**. The Company class contains dead code that can be removed.

2. **Taxpayer class**: addReceipt(), removeReceipt(), getVariationTaxOnReceipts() have some complex conditional logic in the form of chained if-else statements. Simplify these methods by changing the algorithm A simple idea is to use a for loop instead of the chained if-else statements.

3. **Subclasses of the Taxpayer class**: the problem here is **Duplicate Code**. Observe that the calculateBasicTax() method in every subclass is similar. All the methods perform the same steps which differ only in some constant values. Remove the code duplication using parameterization. An idea here is to use a couple of simple data structures (e.g. arrays) as fields in the Taxpayer class for storing the different constants. Then, change the calculateBasicTax() body to use these data structures instead of the constants. This will result in having the same method in every subclass, which can be pulled up to the base class. The subclasses can be used just to initialize the values of the simple data structures in the respective constructors. Alternatively, the subclasses could be removed; in this case the initialization of the data structures could be done using respective property configuration files.

4. **TaxpayerManager class:** This is a **Large Class** with many responsibilities. Simplify this class by delegating some responsibilities to subordinate classes:

    a. createTaxpayer(): you can move the conditional logic that creates different types of Taxpayer objects to a simple parameterized factory.

    b. updateFiles(): you can simplify and move the conditional logic that creates different types of FileWriter objects to a simple parameterized factory.

    c. saveLogFile(): you can move common parts out of the complex if-else logic; then you can move the conditional logic that creates different types of FileWriter objects to a simple parameterized factory.

    d. loadTaxpayer(): you can move common parts out of the complex if-else logic; then you can move the conditional logic that creates different types of FileReader objects to a simple parameterized factory.

incometaxcalculator.data.io package:

1. **TXTFileReader, XMLFileReader classes:** The problem here is **Duplicate Code**. Some classes methods are similar. An idea here is to extract the parts of the code that are different in simple methods implemented in the subclasses. This extraction will make the similar methods identical. Then you can pull up the identical methods in the base FileReader class. Finally you have to define abstract methods in the base class that correspond to the simple methods that have been extracted in the first step.

2. **FileWriter class:** One problem with this class is that it is a **Middle Man** for TaxpayerManager; it has many methods that simply delegate calls to respective methods of TaxpayerManager Another problem is **Refuse Bequest**, i.e. some methods (e.g., getReceipt*() methods, getCompany*() methods) are used only by some of the subclasses. An idea here is to simplify: remove the delegating methods and call directly the TaxpayerManager methods where this is needed; push down methods to the classes that need them; make FileWriter an interface instead of an abstract class.

3. **TXTInfoWriter and XMLInfoWriter classes:** The problem here is **Duplicate Code**. The core algorithms of the classes methods are similar. They only differ in constant string tags that are written along with the basic information. Remove the code duplication. An idea here is to form template methods in an abstract InfoWriter super class (that implements the FileWriter interface) and abstract the parts of the code that are different with simple abstract methods implemented in the subclasses. The different implementations of the simple abstract methods would return different string constants.

4. **TXTLogWriter and XMLLogWriter classes:** Again the problem here is **Duplicate Code**. The core algorithms of the classes methods are similar. They only differ in constant string tags that are written along with the basic information. The idea to remove duplication again is again to form template methods in an abstract LogWriter super class (that implements the FileWriter interface) and abstract the parts of the code that are different with simple abstract methods implemented in the subclasses. The different implementations of the simple abstract methods would return different string constants.

IMPROVEMENT TASK

The GUI of the application has several issues that make it hard to use. For example, the file selections cannot be done by browsing. the selection of Taxpayers requires typing his/her AFM, instead of just clicking on the list of available taxpayers. Replace the GUI of the application with a new one that is easier to learn and use.

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 **DEMO video**, about 15' minutes, using a screen capture tool like ActivePresenter for example. In the demo you should illustrate that the use cases 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, youtube, etc). Upload the demo video. In the project folder that you turnin include a txt file named **DEMO-LINK.txt** that contains the link that points to 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.