# MINNESOTA INCOME TAX CALCULATION PROJECT

## PHASE 2 - REENGINEERING THE CODE

| | |
|---|---|
| | Date 10/2/2018 |

## 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**.

## PHASE 2 TASK LIST

### [BASIC REFACTORING]

Refactor the application so as to correct the style and size problems that you identified with CheckStyle during Phase 1.

**Attention:** Checkstyle reports many formatting problems. To fix them more easily you can use the Eclipse Source>Format facility.

### [REENGINEERING MORE GENERAL PROBLEMS]

Refactor the application so as to fix the following more general problems:

#### incometaxcalculator.data.management package:

1. **Date class**: the problem here is **Unnecessary Complexity**. Java provides its own Date class that can be used instead of this Date class.

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

3. **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.

4. **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 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.

5. **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 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.

   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**. The core algorithms of the classes methods are similar. Remove the code duplication. An idea here is to form template methods in the FileReader super class and abstract the parts of the code that are different with simple abstract methods implemented in the subclasses.

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; 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.

## [PREPARE REPORT]

**Extend the report of Phase 1 with the new design of the legacy application, along with a discussion that explains how you dealt with the all the problems (Project-Deliverable-Phase2.doc).**