MINESSOTA INCOME TAX CALCULATOR

OVERALL REPORT

VERSION <1.0>

ΣΤΕΡΓΙΟΥ ΒΑΣΙΛΕΙΟΣ, AM: 4300

TABLE OF CONTENTS

Introduction 3

Refactored Design 4

Use Cases 4

Architecture 9

Detailed Design 10

Classes Responcibilities and Collaborations (CRC CARDS) 17

Bug Handling 29

INTRODUCTION

The main focus of the present project is the refactoring of the given source code, so as to make the code easier to understand, remove its unnecessary parts and duplicates and make it, as much as possible easier to expands its functionalities in the future. Moreover, the Graphical User Interface (GUI),that interacts with the back-end code, was also remodeled, in order to make it easier for the user to understand it and use it as well. To confirm the valid functionality of the code during the refactoring process, some JUnit tests were created to test its functionalities. During the testing process, it was confirmed that the original code, at the deletion of a receipt from a user, can delete a receipt from another user, if at least two users with a common receipt ID. To provide a solution in this malfunction of the original code, the back-end code, specifically in TaxpayerManager class, was slightly modified and tested using the JUnit tests and, as a result, the problem was fixed. The addition and deletion of a receipt, and the saving into a .txt or .xml file, are maintained from the original code and still working on the refactored one.

REFACTORED DESIGN

USE CASES

Use case ID	**Load Taxpayer**
Actors	The user of the application
Pre conditions	The application must be running
Main flow of events	1. The use case starts when the user clicks on the "Load Taxpayer" button 2. The user can browse and select the file he desires
Alternative flow 1	If the user clicks the " Load Taxpayer " button and does not load any taxpayer, a panel appears that informs the user that he did not load a user
Alternative flow 2	If the user attempts to load a taxpayer that is not in the "(taxpayer_number)_INFO.(txt or xml)" format, a panel appears that informs the user that the taxpayer file format is incorrect
Alternative flow 3	1. If the user attempts to load a taxpayer in the "(taxpayer_number)_INFO.(txt or xml)" format but the taxpayer number is not a 9-digit number, a panel appears that informs the user that the taxpayer file format is incorrect 2. The user can browse again to select a taxpayer file 2.1. If the user does not load any taxpayer, a panel appears that informs the user that he did not load a user and the browsing terminates 2.2. If the user attempts to load a taxpayer that is not in the "(taxpayer_number)_INFO.(txt or xml)" format, a panel appears that informs the user that the taxpayer file format is incorrect and the browsing

	terminates 2.3. If the user attempts to loads a taxpayer as described in case 1, the system follows the steps as described in case 2, until he loads a valid taxpayer or not load a taxpayer.
Post conditions	The taxpayer is loaded in the list of taxpayers

Use case ID	**Select Taxpayer**
Actors	The user of the application
Pre conditions	The application must be running and a taxpayer must already be selected from the taxpayers list
Main flow of events	The use case starts when the user clicks on the "Select Taxpayer " button
Alternative flow 1	If the user clicks the "Select Taxpayer " button and a taxpayer has not been selected from the taxpayers list, a panel appears that informs the user that a taxpayer has not been selected from the taxpayer list
Post conditions	The system projects a form with information regarding the user and his receipts

Use case ID	**Delete Taxpayer**
Actors	The user of the application
Pre conditions	The application must be running and a taxpayer must already be selected from the taxpayes list
Main flow of events	The use case starts when the user clicks on the “Delete Taxpayer ” button.
Alternative flow 1	If the user clicks the “Select Taxpayer ” button and a taxpayer has not been selected from the taxpayers list, a panel appears that informs the user that a taxpayer has not been selected from the taxpayer list
Post conditions	The taxpayer is deleted from the taxpayers list

Use case ID	**Add Receipt**
Actors	The user of the application
Pre conditions	The application must be running and a taxpayer must already be selected from the taxpayers list when the “Select Taxpayer ” button is clicked
Main flow of events	1. The use case starts when the user clicks on the “Add Receipt ” button from the created form that appears once the “Select Taxpayer ” button is clicked 2. A form appears, so that the user can provide the appropriate info regarding the receipt, and choose the receipt kind from the given categories as well
Alternative flow 1	If the user attempts to add a receipt that its receipt ID is lower than 0, a panel appears that informs the user that the receipt is invalid

Alternative flow 2	If the user attempts to add a receipt that the date is not in the " day/month/year " format , a panel appears that informs the user that the receipt is invalid
Alternative flow 3	If the user attempts to add a receipt that the amount is not a non-numeric value, a panel appears that informs the user that the receipt is invalid
Post conditions	1. The receipt is added to the receipts list 2. The updated file is saved in a directory that the user browses and in the file format that the user has chosen.

Use case ID	**Delete Receipt**
Actors	The user of the application
Pre conditions	1. The application must be running and a taxpayer must already be selected from the taxpayers list when the "Select Taxpayer " button is clicked 2. A receipt must be already selected from the receipts list
Main flow of events	The use case starts when the user clicks on the "Delete Receipt " button from the created form that appears once the "Select Taxpayer " button is clicked
Alternative flow 1	If the user attempts to delete a receipt and no receipt is selected from the receipts list, a panel appears that informs the user that no receipt was selected to be deleted
Post conditions	1. The receipt is removed from the receipts list 2. The updated file is saved in a directory that the user browses and in the file format that the user has chosen.

Use case ID	**View Report**
Actors	The user of the application
Pre conditions	The application must be running and a taxpayer must already be selected from the taxpayers list when the “Select Taxpayer ” button is clicked
Main flow of events	1. The use case starts when the user clicks on the “View Report ” button from the created form that appears once the “Select Taxpayer ” button is clicked 2. A pie chart appears that presents the distribution of the amount of on receipts of each receipt kind 3. A bar chart appears that presents a tax analysis, depending on the taxpayer’s info

Use case ID	**Save Data**
Actors	The user of the application
Pre conditions	The application must be running and a taxpayer must already be selected from the taxpayers list when the “Select Taxpayer ” button is clicked
Main flow of events	1. The use case starts when the user clicks on the “Save Data ” button from the created form that appears once the “Select Taxpayer ” button is clicked 2. A from appears, that the user can choose the type of file (txt or xml) that the log data will be saved, which are the data that are presented when the “View Report” button is clicked
Post conditions	The log data are saved in a file, whose file format is chosen by the user

ARCHITECTURE

In the image bellow, the architecture of the application is presented in the form of a package diagram. Each package contains classes, in which the responsibilities are distributed depending on the desired functionalities of the application. The refactoring tasks that are implemented will be explained after a short explanation of the packages that the responsibilities of the application are distributed to.

The package that is responsible for the interactions between the front-end and the back-end parts of the code, is the **incometaxcalculator.data.gui** package. The GUI part of the application consists of the main GUI of the application, implemented in **GraphicalInterface** class. The form that is created once a user is selected, when the " **Select User** " button is clicked is implemented by the **TaxpayerData** class. Finally, the pie chart and the bar chart that are exported once the " **View Report** " button is clicked are iclued in the responsibilities of the " **Chart Display** " class. A part of the code of the three classes mentioned above has been remodeled, so that the GUI is easier to understand and to be used.

The package responsible for handling the importing and exporting the data of the application is the **incometaxcalculator.data.io** package. Those responsibilities are distributed in **TXTFileReader** and **XMLFileReader** classes, whose role is to handle the importing of the files that contain the taxpayer data in the application, the **TXTInfoWriter** and **XMLInfoWriter** classes that handle the saving of the changes in the files and the **TXTLogWriter** and **XMLLogWriter** classes, whose role is to export a txt or xml file, depending on the user's choice, that the log data are saved.

The package responsible for the management of the data, once they are imported to the application, is the **incometaxcalculator.data.management** package. The calculation of the tax and the distribution on the amount spent on receipts are some basic responsibilities of the **Taxpayer** class. The main handler of the taxpayers in the application is the **TaxpayerManager** class, whose responsibilities are the loading and deletion of a taxpayer to the application, and updating the files of the taxpayers as well.

The package responsible for handling exceptions that may occur, due to wrong data input from the side of the user is the the **incometaxcalculator.exceptions** package. The exceptions that may occur during the use of the application include importing a file that is not in the required format, attempting to add a receipt whose date or receipt ID are not the correct ones, or attempting to load a taxpayer that is already loaded in the taxpayer list.

Finally, The package responsible for running some tests to check the expected functionality of the application is the **incometaxcalculator.tests** package. The tests implemented are focused on testing the expected functionality of the use cases, which in fact are the actions that the application executes when one of the buttons is clicked.

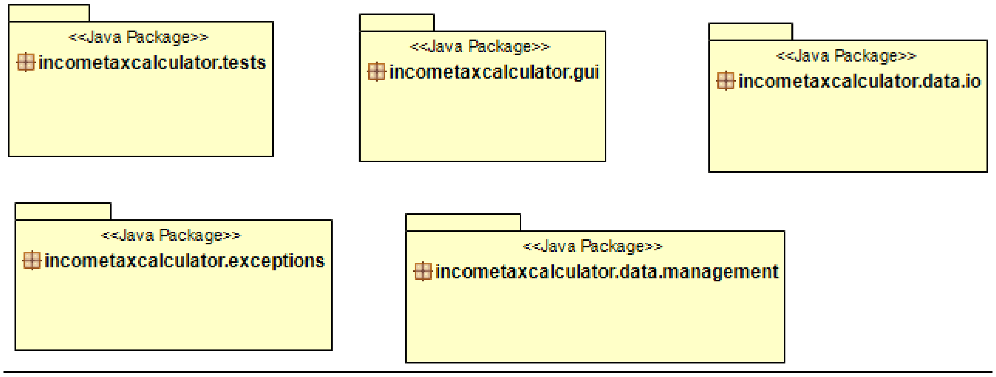
<<Java Package>>
incometaxcalculator.tests
<<Java Package>>
incometaxcalculator.gui
<<Java Package>>
incometaxcalculator.data.io
<<Java Package>>
incometaxcalculator.exceptions
<<Java Package>>
incometaxcalculator.data.management

Figure 1 The package diagram of the application

DETAILED DESIGN

In this section, the required refactoring tasks of the project will be presented, along with the appropriate measures taken in order to achieve a successful code refactoring, while maintaining the functionalities implemented in the original code as well.

Before explaining the refactoring done in the original code, the UML class diagrams are presented on each of its packages, so that the correlations and dependencies between the classes are clear after the reengineering of the original code.

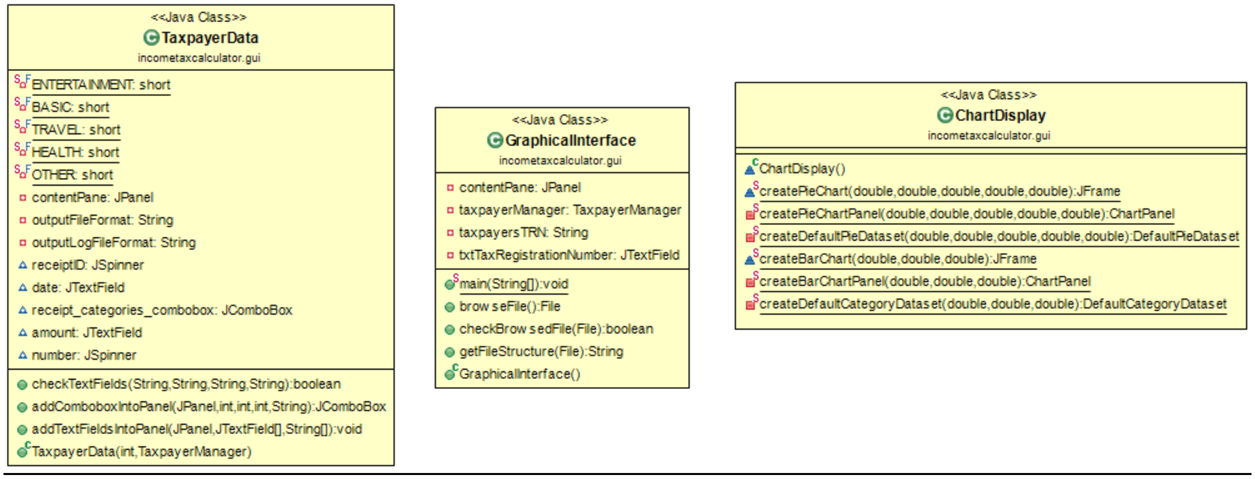
<<Java Class>>
TaxpayerData
incometaxcalculator.gui
ENTERTAINMENT: short
BASIC: short
TRAVEL: short
HEALTH: short
OTHER: short
contentPane: JPanel
outputFileFormat: String
outputLogFileFormat: String
receiptID: JSpinner
date: JTextField
receipt_categories_combobox: JComboBox
amount: JTextField
number: JSpinner
checkTextFields(String,String,String,String):boolean
addComboboxIntoPanel(JPanel,int,int,int,String):JComboBox
addTextFieldsIntoPanel(JPanel,JTextField[],String[]):void
TaxpayerData(int,TaxpayerManager)
<<Java Class>>
GraphicalInterface
incometaxcalculator.gui
contentPane: JPanel
taxpayerManager: TaxpayerManager
taxpayersTRN: String
txtTaxRegistrationNumber: JTextField
main(String[]):void
browseFile():File
checkBrowsedFile(File):boolean
getFileStructure(File):String
GraphicalInterface()
<<Java Class>>
ChartDisplay
incometaxcalculator.gui
ChartDisplay()
createPieChart(double,double,double,double,double):JFrame
createPieChartPanel(double,double,double,double,double):ChartPanel
createDefaultPieDataset(double,double,double,double,double):DefaultPieDataset
createBarChart(double,double,double):JFrame
createBarChartPanel(double,double,double):ChartPanel
createDefaultCategoryDataset(double,double,double):DefaultCategoryDataset

Figure 2 UML class diagram of the "incometaxcalculator.gui" package

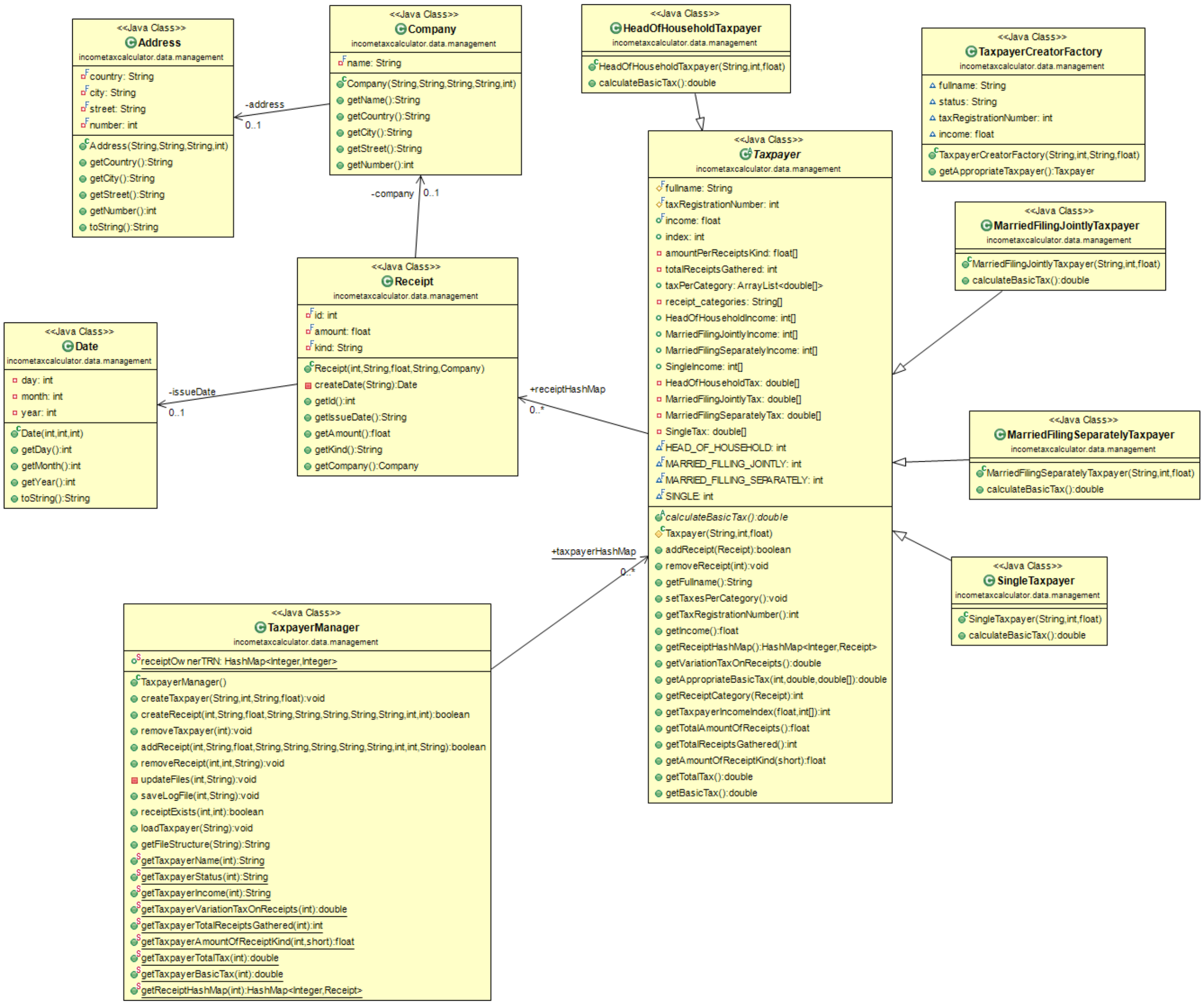
<<Java Class>>
Address
incometaxcalculator.data.management
country: String
city: String
street: String
number: int
Address(String,String,String,int)
getCountry():String
getCity():String
getStreet():String
getNumber():int
toString():String
-address
0..1
<<Java Class>>
Company
incometaxcalculator.data.management
name: String
Company(String,String,String,String,int)
getName():String
getCountry():String
getCity():String
getStreet():String
getNumber():int
-company
0..1
<<Java Class>>
HeadOfHouseholdTaxpayer
incometaxcalculator.data.management
HeadOfHouseholdTaxpayer(String,int,float)
calculateBasicTax():double
<<Java Class>>
TaxpayerCreatorFactory
incometaxcalculator.data.management
fullname: String
status: String
taxRegistrationNumber: int
income: float
TaxpayerCreatorFactory(String,int,String,float)
getAppropriateTaxpayer():Taxpayer
<<Java Class>>
Taxpayer
incometaxcalculator.data.management
fullname: String
taxRegistrationNumber: int
income: float
index: int
amountPerReceiptsKind: float[]
totalReceiptsGathered: int
taxPerCategory: ArrayList<double[]>
receipt_categories: String[]
HeadOfHouseholdIncome: int[]
MarriedFilingJointlyIncome: int[]
MarriedFilingSeparatelyIncome: int[]
SingleIncome: int[]
HeadOfHouseholdTax: double[]
MarriedFilingJointlyTax: double[]
MarriedFilingSeparatelyTax: double[]
SingleTax: double[]
HEAD_OF_HOUSEHOLD: int
MARRIED_FILLING_JOINTLY: int
MARRIED_FILLING_SEPARATELY: int
SINGLE: int
calculateBasicTax():double
Taxpayer(String,int,float)
addReceipt(Receipt):boolean
removeReceipt(int):void
getFullname():String
setTaxesPerCategory():void
getTaxRegistrationNumber():int
getIncome():float
getReceiptHashMap():HashMap<Integer,Receipt>
getVariationTaxOnReceipts():double
getAppropriateBasicTax(int,double,double[]):double
getReceiptCategory(Receipt):int
getTaxpayerIncomeIndex(float,int[]):int
getTotalAmountOfReceipts():float
getTotalReceiptsGathered():int
getAmountOfReceiptKind(short):float
getTotalTax():double
getBasicTax():double
<<Java Class>>
MarriedFilingJointlyTaxpayer
incometaxcalculator.data.management
MarriedFilingJointlyTaxpayer(String,int,float)
calculateBasicTax():double
<<Java Class>>
MarriedFilingSeparatelyTaxpayer
incometaxcalculator.data.management
MarriedFilingSeparatelyTaxpayer(String,int,float)
calculateBasicTax():double
<<Java Class>>
SingleTaxpayer
incometaxcalculator.data.management
SingleTaxpayer(String,int,float)
calculateBasicTax():double
<<Java Class>>
Date
incometaxcalculator.data.management
day: int
month: int
year: int
Date(int,int,int)
getDay():int
getMonth():int
getYear():int
toString():String
-issueDate
0..1
<<Java Class>>
Receipt
incometaxcalculator.data.management
id: int
amount: float
kind: String
Receipt(int,String,float,String,Company)
createDate(String):Date
getId():int
getIssueDate():String
getAmount():float
getKind():String
getCompany():Company
+receiptHashMap
0..*
+taxpayerHashMap
0..*
<<Java Class>>
TaxpayerManager
incometaxcalculator.data.management
receiptOwnerTRN: HashMap<Integer,Integer>
TaxpayerManager()
createTaxpayer(String,int,String,float):void
createReceipt(int,String,float,String,String,String,String,String,int,int):boolean
removeTaxpayer(int):void
addReceipt(int,String,float,String,String,String,String,String,int,int,String):boolean
removeReceipt(int,int,String):void
updateFiles(int,String):void
saveLogFile(int,String):void
receiptExists(int,int):boolean
loadTaxpayer(String):void
getFileStructure(String):String
getTaxpayerName(int):String
getTaxpayerStatus(int):String
getTaxpayerIncome(int):String
getTaxpayerVariationTaxOnReceipts(int):double
getTaxpayerTotalReceiptsGathered(int):int
getTaxpayerAmountOfReceiptKind(int,short):float
getTaxpayerTotalTax(int):double
getTaxpayerBasicTax(int):double
getReceiptHashMap(int):HashMap<Integer,Receipt>

Figure 3 UML class diagram of the "incometaxcalculator.data.management" package

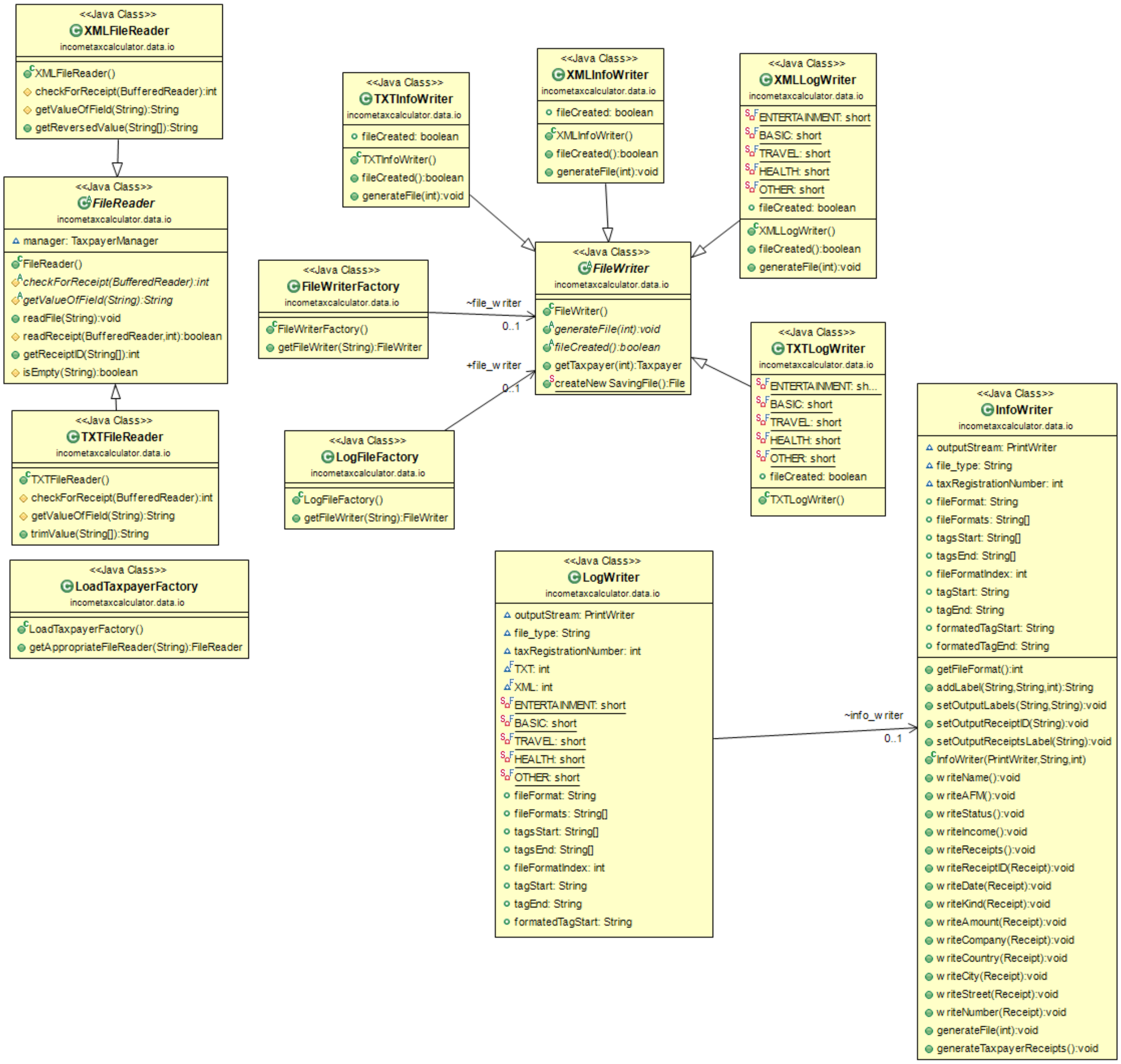

Figure 4 UML class diagram of the "incometaxcalculator.data.io" package

Having presented the necessary UML class diagrams that depict the way the classes are correlated to each other and their dependencies, the required refactoring tasks implemented will be explained bellow on the specified packages and the classes that they were implemented ,as for the way they are implemented and which designing patterns where used for their implementation as well.

Incometaxcalculator.data.management package

1. **Company** class: The **getAddress()** method was not used in the other classes of the code, so it was removed from the class.

2. **Taxpayer** class: The chained if-else statements used to getting the receipt category was reduced to a simple for-loop, where the categories are stored in a static array that contains the receipt categories. This code is implemented by the **getReceiptCategory()** method, which was added for the purpose of refactoring the **addReceipt(), removeReceipt()** and **getVariationTaxOnReceipts()** methods. Additionally in latter method, the income percentages that are used in the chained if-statement are now stored in an array and the and the final tax is calculated by **getAppropriateBasicTax()** method, implemented for the purpose of the refactoring.

3. **Subclasses of Taxpayer** class**:** The problem in the Taxpayer classes of the Taxpayer class is that the method **calculateBasicTax()** is implemented in the same way, but using different constants on each class. In order to avoid code duplication using constant parameters, there are four initialized arrays in the Taxpayer class, as there are 4 taxpayer categories. Each category and the appropriate tax variation on each category are initialized in the newly implemented **setTaxesPerCategory()** method. Thus, the four new arrays and the ArrayList initialized on the Taxpayer class are used accordingly on each of the subclasses of the Taxpayer class.

4. **TaxpayerManager** class:

In order to remove some of the unnecessary code of the class, the **containsTaxpayer()** and **containsReceipt()** methods have been removed from the **TaxpayerManager** class, as their functionalities can be directly called by the TaxpayerManager object in the main GUI code. Moreover, the factory pattern has been used in the following functions of the class, simplifying the conditional logic using the necessary parameters in the factories.

a. **createTaxpayer()** method**:** The condional logic that creates different kinds of Taxpayer is moved in **TaxpayerCreatorFactory** class, which uses the **getAppropriateTaxpayer()** method to get the respective taxpayer class, according to the status field as previously used in the original code.

b. **updateFiles()** method**:** The condional logic that creates a new txt or xml file, depending on the user's preference of saving the changes a file is moved into **FileWriterFactory** class of **Incometaxcalculator.data.io** package, which uses the **getFileWriter()** method, which return the appropriate subclass of the **FileWriter** class, which is either the **TXTInfoWriter** or the **XMLInfoWriter** class, depending on the file format the user chooses.

c. **saveLogFile()** method**:** The condional logic that creates a new txt or xml log file, depending on the user's preference of saving the changes a file is moved into **LogFileFactory** class of **Incometaxcalculator.data.io** package, which uses the **getFileWriter()** method, which return the appropriate subclass of the **FileWriter** class, which is either the **TXTLogWriter** or the **XMLLogWriter** class, depending on the file format the user chooses for the output log file.

d. **loadTaxpayer() method:** The condional logic that opens a txt or xml file to read its data, depending on the file type that the user chooses by browsing, is moved into **LoadTaxpayerFactory** class of **Incometaxcalculator.data.io** package, which uses the **getAppropriateFileReader()** method, which return the appropriate subclass of the **FileReader** class, which is either the **TXT FileReader** or the **XML FileReader** class, depending on the file format of the file the user has imported to the application.

Incometaxcalculator.data.io package

1. **TXTFileReader, XMLFileReader** classes**:** The main refactoring performed in these two classes is the implementation of the **getReceiptId()** method in the **FileReader** class. The extraction of a receipt ID differs in the txt and xml files, but since both cases need the receipt ID extraction, a common parametrized function has been implemented, in order to avoid code duplication. The implemented function is inherited by the **TXTFileReader** and **XMLFileReader** classes. The non-common functionalities have been implemented in individual functions in each of the two subclasses, so that the code is easier to understand.

2. **FileWriter** class**:** The problem with this class was the extended responsibilities that it originally had and the **Middle Man** problem. Some of the implemented functions were removed from the code, as they can be directly called from the TaxpayerManager class that they were originally implemented. The newly created **InfoWriter** and **LogWriter** classes, and the main GUI as well, are examples of code that they were originally using those functions. The direct calls from the **TaxpayerManager** class not only has reduced the size of the **FileWriter** class, but also reduced the dependencies of the other classes to this class.

3. **TXTInfoWriter, XMLInfoWriter classes:** The main problem with the two mentioned classes is the code duplication. The algorithms that are used from writing in a txt or xml file differ in some tags, depending on the file. To avoid the code duplication, the **InfoWriter** class was created. Its responsibilities is the implementation of simple writing functions, depending on the file format of the output file. In this way, the appropriate writing responsibilities of the two subclasses are contained in a super class and the writing into file is parametrized, depending on the file format of the output file.

4. **TXTLogWriter, XMLLogWriter classes:** The main problem with the two mentioned classes is the code duplication. The algorithms that are used from writing in a txt or xml log file differ in some tags, depending on the file. To avoid the code duplication, the **LogWriter** class was created. Its responsibilities is the implementation of simple writing functions, depending on the file format of the output log file. In this way, the appropriate writing responsibilities of the two subclasses are contained in a super class and the writing into file is parametrized, depending on the file format of the output file.

CLASSES RESPONSIBILITIES AND COLLABORATIONS (CRC CARDS)

Class Name: FileReader	
Responsibilities	**Collaborations**
○ This class is responsible for reading the data from the file that is imported to the application	▪ XMLFileReader and TXTFileReader classes has an **Inheritance** relationship with FileReader class ▪ They inherit its file reading methods

Class Name: TXTFileReader	
Responsibilities	**Collaborations**
○ This class is responsible for reading the data from a TXT file that is imported to the application	▪ This class has an Inheritance relationship with FileReader class ▪ It inherit its file reading methods

Class Name: XMLFileReader	
Responsibilities	**Collaborations**
○ This class is responsible for reading the data from a XML file that is imported to the application	▪ This class has an Inheritance relationship with FileReader class ▪ It inherit its file reading methods

Class Name: LoadTaxpayerFactory	
Responsibilities	**Collaborations**
○ This class is responsible for creating the appropriate subclass of the FileReader class ○ It serves as parametrized factory with the file format as its parameter	▪ This class does not share a relationship with other classes or with the subclasses it creates

Class Name: FileWriter	
Responsibilities	**Collaborations**
○ This class contains the abstract methods that its subclasses implement, in order to write the data into a file	▪ XMLFileWriter, TXTFileWriter, XMLLogWriter, TXTLogWriter classes have an Inheritance relationship with this class

Class Name: TXTInfoWriter	
Responsibilities	**Collaborations**
○ This class contains functionalities that serve the saving of the changes in a TXT file	▪ This class has an Inheritance relationship with the FileWriter class ▪ It inherits and implements the abstract generateFile() method

Class Name: FileWriter	
Responsibilities	**Collaborations**
○ This class contains the abstract methods that its subclasses implement, in order to write the data into a file	▪ XMLFileWriter, TXTFileWriter, XMLLogWriter, TXTLogWriter classes have an Inheritance relationship with this class

Class Name: XMLInfoWriter	
Responsibilities	**Collaborations**
○ This class contains functionalities that serve the saving of the changes in an XML file	▪ This class has an Inheritance relationship with the FileWriter class ▪ It inherits and implements the abstract generateFile() method

Class Name: TXTLogWriter	
Responsibilities	**Collaborations**
○ This class contains functionalities that serve the saving of the log data in a TXT file	▪ This class has an Inheritance relationship with the FileWriter class ▪ It inherits and implements the abstract generateFile() method

Class Name: XMLLogWriter	
Responsibilities	**Collaborations**
○ This class contains the abstract methods that its subclasses implement, in order to write the data into a file	▪ This class has an Inheritance relationship with the FileWriter class ▪ It inherits and implements the abstract generateFile() method this class

Class Name: FileWriterFactory	
Responsibilities	**Collaborations**
○ This class is responsible for creating the appropriate InfoWriter subclass of the FileWriter class ○ It serves as parametrized factory with the output file format as its parameter	▪ This class creates a FileWriter object,as it returns a TXTInfoWriter or an XMLInfoWriter, depending on the parameter of the getFileWriter() method

Class Name: LogFileFactory	
Responsibilities	**Collaborations**
○ This class is responsible for creating the appropriate LogWriter subclass of the FileWriter class ○ It serves as parametrized factory with the output file format as its parameter	▪ This class creates a FileWriter object,as it returns a TXTLogWriter or an XMLLogWriter, depending on the parameter of the getFileWriter() method

Class Name: InfoWriter	
Responsibilities	**Collaborations**
○ This class contains the writing methods of TXTInfoWriter and XMLInfoWriter classes	▪ This class has no relationships with the other classes ▪ TXTInfoWriter and XMLInfoWriter classes collaborate with this class, as they use its writing methods

Class Name: LogWriter	
Responsibilities	**Collaborations**
○ This class contains the writing methods of TXTLogWriter and XMLLogWriter classes	▪ This class has no relationships with the other classes ▪ TXTLogWriter and XMLLogWriter classes collaborate with this class, as they use its writing methods ▪ This class also collaborates with the InfoWriter method, as it uses as some of its writing methods

Class Name: Address	
Responsibilities	**Collaborations**
○ This class contains the info associated with the address of a taxpayer	▪ This class has a Composite relationship with the Company class ▪ The Company class has an Address object as one of its fields

Class Name: Company	
Responsibilities	**Collaborations**
○ This class contains the info associated with the company of a taxpayer	▪ This class has a Composite relationship with the Address and Receipt classes ▪ It has an Address object as one of its fields ▪ It has a Receipt object as one of its fields

Class Name: Receipt	
Responsibilities	**Collaborations**
○ This class contains the info associated with a receipt that a taxpayer possesses	▪ This class has a Composite relationship with the Company and Date classes ▪ It has a Company object as one of its fields ▪ It has a Date object as one of its fields

Class Name: Company	
Responsibilities	**Collaborations**
○ This class contains the info associated with the creation date of a receiptS	▪ This class has a Composite relationship with the Receipt class ▪ This class is a field of the Receipt class

Class Name: Taxpayer	
Responsibilities	**Collaborations**
○ This class contains the info associated with a taxpayer	▪ This class has a Composite relationship the Receipt class and an inheritance relationship with its subclasses ▪ Its subclasses inherit and implement the calculateBasicTax() abstract method ▪ This class contains Receipt objects in the form of a Hashmap

Class Name: HeadOfHouseHoldTaxpayer	
Responsibilities	**Collaborations**
○ This class handles the data of a taxpayer that is in the Head of Household taxpayer category	▪ This class has an Inheritance relationship with the Taxpayer class ▪ This class is a subclass of the Taxpayer class ▪ It inherits and implements the calculateBasicTax() abstract method

Class Name: MarriedFillingJointlyTaxpayer	
Responsibilities	**Collaborations**
○ This class handles the data of a taxpayer that is in the Married Filling Jointly taxpayer category	▪ This class has an Inheritance relationship with the Taxpayer class ▪ This class is a subclass of the Taxpayer class ▪ It inherits and implements the calculateBasicTax() abstract method

Class Name: MarriedFillingSeparatelyTaxpayer	
Responsibilities	**Collaborations**
○ This class handles the data of a taxpayer that is in the Married Filling Separately taxpayer category	▪ This class has an Inheritance relationship with the Taxpayer class ▪ This class is a subclass of the Taxpayer class ▪ It inherits and implements the calculateBasicTax() abstract method

Class Name: SingleTaxpayer	
Responsibilities	**Collaborations**
○ This class handles the data of a taxpayer that is in the Single taxpayer category	▪ This class has an Inheritance relationship with the Taxpayer class ▪ This class is a subclass of the Taxpayer class ▪ It inherits and implements the calculateBasicTax() abstract method

Class Name: TaxpayerManager	
Responsibilities	**Collaborations**
○ This class is responsible for the management of the taxpayers that the application handles	▪ This class has a Composite relationship with the Taxpayer class ▪ The Taxpayer objects that this class consists of, are in the form of a Hashmap

Class Name: TaxpayerCreatorFactory	
Responsibilities	**Collaborations**
○ This class is responsible for the creation of the appropriate subclass of the Taxpayer class ○ The respective class that the factory creates depends on its input parameter	▪ This class has no relationship with the rest of the classes

BUG HANDLING

During the refactoring process of the legacy code, two main bug were fixed, regarding the deletion of a receipt in the application, and the validation of the data that the user inserts, when a new receipt is attempted to be loaded in the system. The tracking of these bugs was achieved through the testing process. In the bullets bellow, a brief explanation of the bug handling is provided on how those bugs were handled.

- **Deletion of a receipt**: In the legacy code, the **removeReceipt()** method of the **TaxpayerManager** class was originally using the receiptID field and **receiptOwnerTRN** hashmap, in order to get the owner of the receipt. However, this poses a problem when there are two or more users loaded at the same time that possess a receipt with a common receipt ID. The problem was fixed using the **taxRegistrationNumber** field, in order to remove the receipt from the specified taxpayer .

- **Invalid amount and empty text fields**: Through the testing process, a main lack of functionality was found when the user attempt to enter a non-numeric amount when filling the receipt form and/or has empty txt fields, when he attempts to enter a receipt. In order to check if the inserted data are valid, **checkTextFields()** method was implemented, and a try-catch statement for the validity of the amount.