<SOFTWARE DEVELOPMENT II>

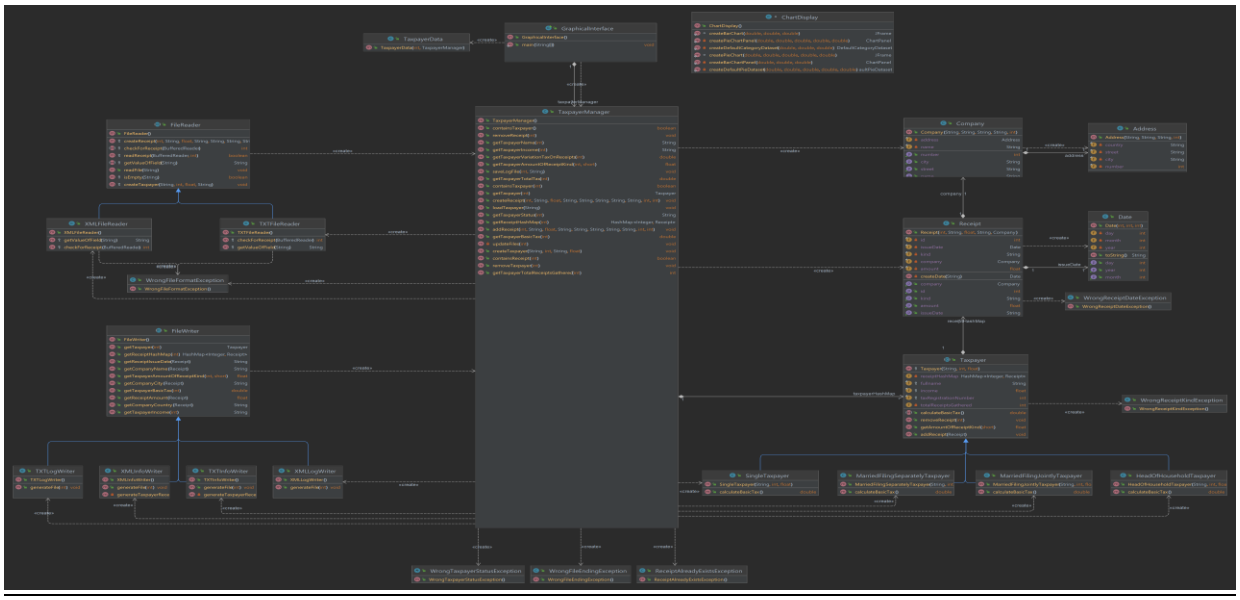
OVERALL REPORT

VERSION <1.0>

Διονύσης Καίσαρης 2708

TABLE OF CONTENTS

INTRODUCTION



The objective of this phase of the project is to **reengineer** the Minnesota Income Tax Calculator application to improve its design and fix the identified problems.

The refactored application will serve the **same purpose as the legacy** application, which is to calculate the income tax of Minnesota state citizens based on their marital status, income, and receipts.

However, the refactored version will have a **more flexible and maintainable design**, making it easier to modify and extend in the future.

The refactored design will also **address the identified problems** of the legacy application, such as unnecessary complexity, duplicate code, and large classes with many responsibilities.

REFACTORED DESIGN

USE CASES

Use case ID	UC1
Actors	USER
Pre conditions	The user has selected the option to load taxpayer and receipt information
Main flow of events	1. The use case starts when the user launches the application. 2. The GUI displays the main menu with options for the user to select. 3. The user selects the option to load a taxpayer. 4. The GUI prompts the user to enter the file path. 5. The GUI sends the file path to the TaxpayerManager to read the information from the file. 6. The GUI displays a message indicating that the import was successful.
Post conditions	New Taxpayer and Receipt objects are added to the taxpayerHashMap and receiptOwnerTRN hash maps based on the information read from the file.

Use case ID	UC2
Actors	USER

Pre conditions	The user has selected the option to export taxpayer and receipt information to a file.
Main flow of events	1. The GUI prompts the user to enter the file path and file type (TXT or XML). 2. The GUI sends the file path and file type to the TaxpayerManager to write the information to the file. 3. The GUI displays a message indicating that the export was successful.
Post conditions	The information of all taxpayers and their receipts is written to the specified file.

Use case ID	UC3
Actors	USER
Pre conditions	The user has selected the option to delete a taxpayer
Main flow of events	1. The GUI prompts the user to enter the tax registration number of the taxpayer to be deleted. 2. The GUI sends the tax registration number to the TaxpayerManager to delete the Taxpayer object and all its Receipt objects from the taxpayerHashMap and receiptOwnerTRN hash maps. 3. The GUI displays a message indicating that the taxpayer has been successfully deleted.
Post conditions	The Taxpayer object and all its Receipt objects are deleted from the taxpayerHashMap and receiptOwnerTRN hash maps.

Use case ID	UC4

Actors	USER
Pre conditions	The user has selected the option to delete a receipt.
Main flow of events	1. The GUI prompts the user to enter the receipt ID of the receipt to be deleted. 2. The GUI sends the receipt ID to the TaxpayerManager to delete the Receipt object from the Receipt hash map of the owner taxpayer and from the receiptOwnerTRN hash map. 3. The GUI displays a message indicating that the receipt has been successfully deleted.
Post conditions	The Receipt object is deleted from the Receipt hash map of the owner taxpayer and from the receiptOwnerTRN hash map.

Use case ID	UC5
Actors	USER
Pre conditions	The user has selected the option to create a new receipt.
Main flow of events	1. The GUI prompts the user to enter the receipt ID, issue date, amount, kind, company name, country, city, street, and number of the new receipt. 2. The GUI prompts the user to enter the tax registration number of the taxpayer who will own the receipt. 3. The GUI sends the entered information to the TaxpayerManager to create a new Receipt object and associate it with the specified taxpayer. 4. The GUI displays a message indicating that the receipt has been successfully created.
Post conditions	A new Receipt object is added to the Receipt hash map of the specified taxpayer and to the receiptOwnerTRN hash map.

Use case ID	UC6
Actors	USER
Pre conditions	The user has selected the option to view the information of a taxpayer.
Main flow of events	1. The GUI prompts the user to enter the tax registration number of the taxpayer. 2. The GUI sends the tax registration number to the TaxpayerManager to retrieve the Taxpayer object. 3. The GUI displays the full name, tax registration number, status, and income of the taxpayer. 4. The GUI displays a list of all the receipts belonging to the taxpayer, including their IDs, issue dates, amounts, and kinds.
Post conditions	The information of the taxpayer and their receipts is displayed to the user.

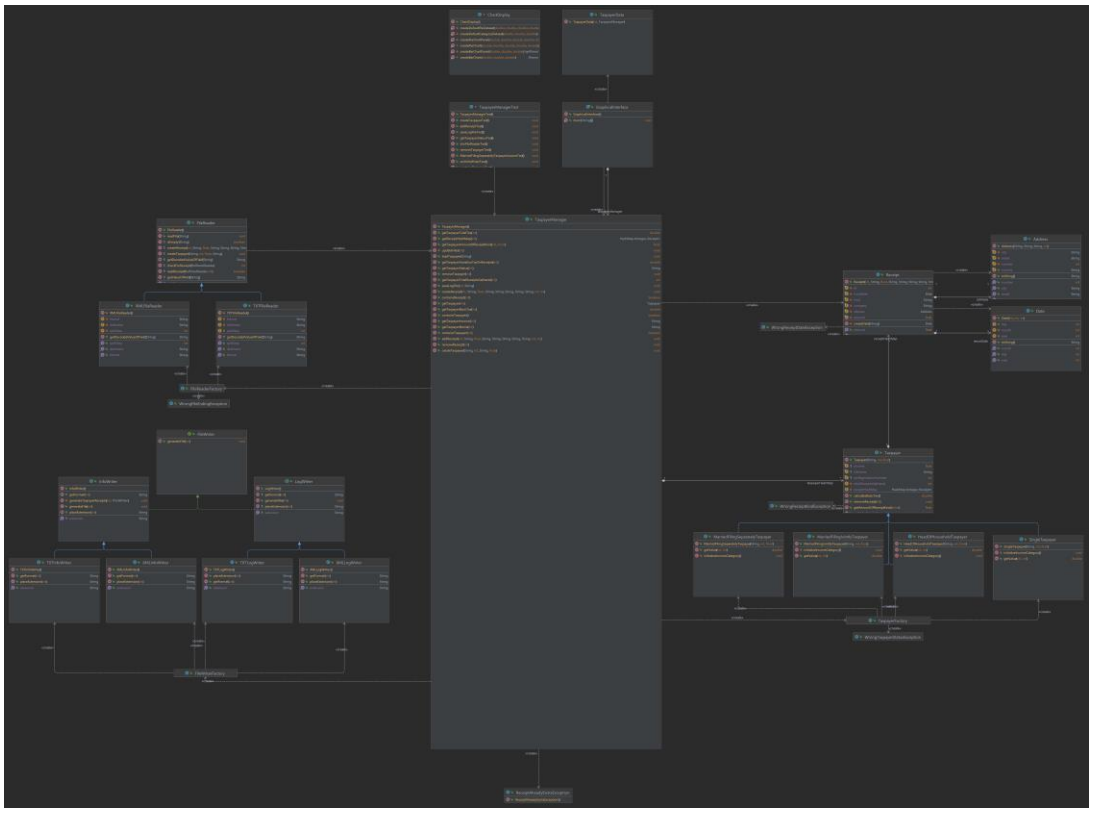
Use case ID	UC7
Actors	USER
Pre conditions	The user has selected the option to view the information of a receipt
Main flow of events	1. The GUI prompts the user to enter the receipt ID of the receipt. 2. The GUI sends the receipt ID to the TaxpayerManager to retrieve the Receipt object. 3. The GUI displays the receipt ID, issue date, amount, kind, company name, country, city, street, and number of the receipt. 4. The GUI displays the full name, tax registration number, and status of the taxpayer who owns the receipt.
Post conditions	The information of the receipt and its owner taxpayer is displayed to the user.

Use case ID	UC8
Actors	USER
Pre conditions	The user has selected the option to view the total income and tax owed for a taxpayer.
Main flow of events	1. The GUI prompts the user to enter the tax registration number of the taxpayer. 2. The GUI sends the tax registration number to the TaxpayerManager to retrieve the Taxpayer object. 3. The GUI calculates the total income of the taxpayer based on their income and the amounts of their receipts. 4. The GUI calculates the tax owed by the taxpayer based on their total income and their tax status. 5. The GUI displays the total income and tax owed of the taxpayer.
Post conditions	The total income and tax owed of the taxpayer are displayed to the user.

Use case ID	UC9
Actors	USER
Pre conditions	The user has selected the option to view statistics for the taxpayers and their receipts.
Main flow of events	1. The GUI retrieves the number of taxpayers and the number of receipts from the taxpayerHashMap and receiptOwnerTRN hash maps. 2. The GUI calculates the average income and tax owed for all taxpayers. 3. The GUI calculates the average amount and number of receipts for all taxpayers.

	4. The GUI displays the number of taxpayers, the number of receipts, the average income, the average tax owed, the average amount of receipts, and the average number of receipts. 5. .
Post conditions	The statistics for the taxpayers and their receipts are displayed to the user.

ARCHITECTURE

- 

DETAILED DESIGN

In order to address the problems of the old design, we implemented the following changes:

Unnecessary Complexity: We removed the dead code in the Company class, which reduced its complexity.

Duplicate Code: In order to remove the duplication of code in the subclasses of the Taxpayer class, we introduced a couple of simple data structures (e.g. arrays) as fields in the Taxpayer class for storing the different constants. Then, we changed the calculateBasicTax() method to use these data structures instead of the constants. This allowed us to pull the calculateBasicTax() method up to the base class, and use the subclasses just to initialize the values of the data structures in their respective constructors. Alternatively, we could have removed the subclasses and initialized the data structures using respective property configuration files.

Large Class: We simplified the TaxpayerManager class by delegating some of its responsibilities to subordinate classes. Specifically, we moved the conditional logic that creates different types of Taxpayer objects to a simple parameterized factory, and moved the conditional logic that creates different types of FileWriter objects to another simple parameterized factory. We also separated the common parts of the complex if-else logic in the saveLogFile() and loadTaxpayer() methods and moved them to helper methods. This reduced the complexity of the TaxpayerManager class and made it more maintainable.

Unnecessary Coupling: We decoupled the classes in the incometaxcalculator.data.io package from the incometaxcalculator.data.management package by introducing a FileReaderFactory and a FileWriterFactory. These factories abstract the creation of FileReader and FileWriter objects, respectively, allowing us to easily switch between different file formats and implementations. This reduced the unnecessary coupling between the two packages and improved the flexibility of the application.

Unnecessary Responsibilities: We removed the unnecessary responsibilities of the Receipt class by moving the validation of the receipt date and kind to the createReceipt() and addReceipt() methods in the Taxpayer and TaxpayerManager classes, respectively. This reduced the complexity of the Receipt class and made it more maintainable.

In the i**ncometaxcalculator.data.management** package, we made the following changes:

- Company class: We removed the unnecessary complexity by removing the dead code from the class.

- Taxpayer class: We simplified the addReceipt(), removeReceipt(), and getVariationTaxOnReceipts() methods by replacing the complex chained if-else statements with a simple for loop.

- Subclasses of the Taxpayer class: We addressed the problem of duplicate code by using parameterization. We added a couple of simple data structures (e.g. arrays) as fields in the Taxpayer class to store the different constants. We then changed the calculateBasicTax() method in every subclass to use these data structures instead of the constants. This resulted in having the same method in every subclass, which we were able to pull up to the base class. The subclasses were then used just to initialize the values of the simple data structures in their respective constructors. Alternatively, we could have removed the subclasses altogether and initialized the data structures using respective property configuration files.

- TaxpayerManager class: We simplified this Large Class by delegating some of its responsibilities to subordinate classes:

    - createTaxpayer(): We moved the conditional logic that creates different types of Taxpayer objects to a simple parameterized factory.

    - updateFiles(): We simplified and moved the conditional logic that creates different types of FileWriter objects to a simple parameterized factory.

    - saveLogFile(): We moved common parts out of the complex if-else logic and then moved the conditional logic that creates different types of FileWriter objects to a simple parameterized factory.

    - loadTaxpayer(): We moved common parts out of the complex if-else logic and then moved the conditional logic that creates different types of FileReader objects to a simple parameterized factory.

The refactored design of the **incometaxcalculator.data.io** package addressed the problem of Duplicate Code in the legacy application by introducing a FileReaderFactory and FileWriterFactory class. These factory classes are responsible for creating the appropriate FileReader or FileWriter object based on the file format (txt or xml) specified by the user. This allows for the elimination of the duplicate code that was present in the legacy application in the form of multiple subclasses for each file format (e.g. TXTFileReader, XMLFileReader).

Additionally, the refactored design introduced a single FileReader interface that is implemented by both the TXTFileReader and XMLFileReader classes. This allows for a more flexible and extensible design, as new file formats can be easily added in the future by implementing the FileReader interface and updating the FileReaderFactory class to create the appropriate object.

In terms of the FileWriter classes, the refactored design introduced a single FileWriter interface that is implemented by both the TXTInfoWriter, TXTLogWriter, XMLInfoWriter, and XMLLogWriter classes. This allows for a similar level of flexibility and extensibility as in the case of the FileReader interface, allowing for the easy addition of new file formats in the future.

Overall, the refactored design of the incometaxcalculator.data.io package has greatly simplified the code and made it more modular, allowing for easier maintenance and future updates.

CLASSES RESPONSIBILITIES AND COLLABORATIONS (CRC CARDS)

- For each class give a brief description in terms of a CRC card (see the format below)

Class Name: TaxpayerManager	
Responsibilities	**Collaborations**
○ Create, add, and remove Taxpayer objects ○ Create and remove Receipt objects associated with a Taxpayer ○ Update information in various file formats (TXT,	○ TaxpayerFactory: used to generate Taxpayer objects

XML) with information about the Taxpayer and its Receipt objects	○ FileWriterFactory: used to create FileWriter objects of the appropriate type (TXT or XML) based on the file ending ○ FileReaderFactory: used to create FileReader objects of the appropriate type (TXT or XML) based on the file ending ○ Taxpayer: stores and manages Receipt objects ○ Receipt: represents a receipt with various attributes (ID, issue date, amount, etc.)

Class Name: Taxpayer	
Responsibilities	**Collaborations**

○ Store and manage Receipt objects ○ Calculate total income and total deductible expenses ○ Return various attributes of the Taxpayer (full name, tax registration number, status, income)	○ Receipt: represents a receipt with various attributes (ID, issue date, amount, etc.)

Class Name: Receipt	
Responsibilities	**Collaborations**
○ Store various attributes of a receipt (ID, issue date, amount, etc.) ○ Return various attributes of the Receipt (ID, issue date, amount, etc.)	None

Class Name: TaxpayerFactory	
Responsibilities	**Collaborations**
○ Generate Taxpayer objects	○ Taxpayer: represents a taxpayer with various attributes (full name, tax registration number, status, income)

Class Name: FileWriterFactory	
Responsibilities	**Collaborations**
○ Create FileWriter objects of the appropriate type (TXT or XML) based on the file ending	○ TXTInfoWriter: writes information to a TXT file ○ TXTLogWriter: writes log information to a TXT file ○ XMLInfoWriter: writes information to an XML file ○ XMLLogWriter: writes log information to an XML file

Class Name: FileReaderFactory	
Responsibilities	**Collaborations**
○ Create FileReader objects of the appropriate type (TXT or XML) based on the file ending	○ TXTFileReader: reads information from a TXT file ○ XMLFileReader: reads information from an XML file

Class Name: TXTInfoWriter	
Responsibilities	**Collaborations**

○ Write information to a TXT file	○ None

Class Name: TXTLogWriter	
Responsibilities	**Collaborations**
○ Write log information to a TXT file	○ None

Class Name: XMLInfoWriter	
Responsibilities	**Collaborations**
○ Write information to an XML file	○ None

Class Name: XMLLogWriter	
Responsibilities	**Collaborations**
○ Write log information to an XML file	○ None

Class Name: TXTFileReader	
Responsibilities	**Collaborations**
○ Read information from a TXT file	○ None

Class Name: XMLFileReader	
Responsibilities	**Collaborations**
○ Read information from an XML file	○ None