
SOFTWARE DEVELOPMENT II – MYE004 COURSE PROJECT DELIVERABLE

MINNESOTA INCOME TAX CALCULATOR

REFACTORED

OVERALL REPORT

FINAL VERSION (20/12/2020)

Todhri Angjelo 3090

Apostolis Kasselouris 2994

TABLE OF CONTENTS

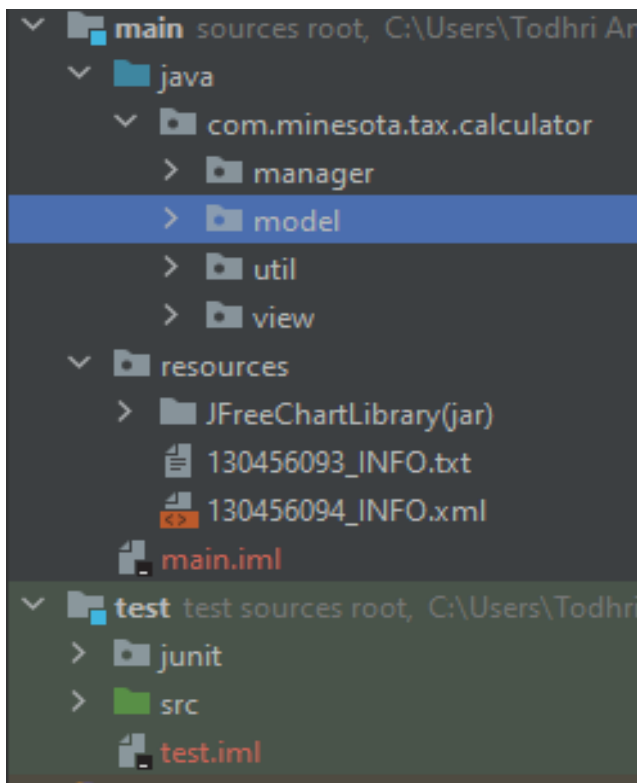
Introduction	3
Refactored Design	3
Architecture	3
Detailed Design (packaging/structure/responsibilities) – before/after	3
Implementation DETAILS (GCR cards)	5
Working METHODOLOGY AND PRACTICES	11
TODOS and future releases	14
Links and github	15

INTRODUCTION

We were given a legacy project to work on which was poorly designed and the architecture didn't allow the project to be easily maintained, scaled or modified. Based on specific guidelines the legacy code was redesigned and refactored based on community best practices and modern design patterns.

REFACTORED DESIGN

ARCHITECTURE



Here we can see our refactored packages structure.

We didn't stick to the old design. We introduced a **model package**, a **view package** and a **manager package**.

The model package holds all the application necessary model structures. They are simple and they hold no logic. The logic is transferred into the managers and the view classes.

The view classes are now responsible for the User Interface manipulation/display and the manager classes for serving data to the UI classes and managing the application memory and the cached taxpayers.

The resource package holds the inputFiles and the jFreeChart library's jar and the test module holds the junit library jar and the source code with the single AppTest java class.

DETAILED DESIGN (PACKAGING/STRUCTURE/RESPONSIBILITIES) – BEFORE/AFTER

The legacy code was refactored based on our experience in the Java community standards and our perspective for a well structured and easily extendable software.

PACAKAGING BEFORE	PACKAGING AFTER
<ul style="list-style-type: none"> - dataManagePackage (a database class and some models) - gui package (the UI classes) - inputFiles (the input txt files) - InputManagePackage (the files parsing logic) - OutputManagePackage (the charts generation and the .log file writing and taxpayers receipt update logic) 	<ul style="list-style-type: none"> ➤ Main <ul style="list-style-type: none"> ○ Java <ul style="list-style-type: none"> ▪ Manager package (holds the business logic executors) ▪ Model package (pure models with no logic/responsible for carrying / representing data structures) ▪ View package ○ Resources <ul style="list-style-type: none"> ▪ Input files and JFreeChartLibrary (chart) ➤ Test <ul style="list-style-type: none"> ○ Junit (jar) ○ Src (the test java class)

Previously:

There existed a **Database** class which was confusing because the application does not communicate with any sort of database but instead reads and writes file through Java IO.

There existed an **InputSystem** class which had the logic of reading the .txt or .xml files, parsing the data and loading the files into some static property at the Database class.

There existed an **OutputSystem** class which had the responsibility of manipulating the UI by creating and displaying the respective tax payers pie or bar charts and also saving the files statically hold in the Database class.

Previous packaging division was not very clear and it had the dependency of reading and understanding the code for explaining the philosophy and being able to maintain and extend the code. **Input and Output system are bad terms** for describing what the code needed to do at the respective package classes, **Database and DataManagePackage** was very misleading as long as there is no database in our application and a **gui package** existed for the view classes that manages our application User Interface.

After:

We decided to remove the **Database** class and introduce a new **FileManager** class which has the responsibility of reading and parsing the input files, loading and caching the tax payers from the input .txt or .xml files and altering tax payers' receipts or creating the .log taxpayers' files.

There was a really bad and not well-structured approach of designing the software because as we explained there is no database or any kind of storage. We just cache some tax payers in our application memory so we can display them as the application is running and write them to a file in order we keep track of the changes we made in previous application executions.

We introduced a new **ChartManager** class and transferred the file saving logic (from **OutputSystem**) in the previously mentioned **FileManager** class. The **ChartManager** still affects UI which is not a good practice here but it is mentioned in the later TODOs for our application development and future releases. Check TODO section

All source code for our application core functionalities and the GUI were added under a **src.main.java** module, test code was added under a new **test.src** module and resources (libraries and inputFiles) were added under previously mentioned **src.main module** in a new **resources** package

IMPLEMENTATION DETAILS (GCR CARDS)

- GCR cards which describe the protagonist classes and its core responsibilities and dependencies (collaborations)

Class Name: ChartsManager	
Responsibilities	Collaborations
This class is responsible for managing all the charts related actions requested from the UI and displaying the pie or bar charts respectively.	<p>FilesManager: to retrieve the cached taxpayer's info and display related information in a chart representation</p> <p>TaxPayer: model for holding the taxpayer information</p> <p>JFreeChartLibrary: collaboration with chart classes from external library to make proper displays</p> <p>TaxPayerUtils: collaboration with utility class which holds</p>

	<p>mechanism for amounts calculation</p> <p>Java AWT: collaboration with UI classes for displaying the charts when requested and creating them</p>
--	-----------------------------------------------------------------------------------------------------------------------------------------------------------

Class Name: FileManager	
Responsibilities	Collaborations
<p>Class responsible for the files management and parsing business logic. Also holds the list of taxPayers that are loaded and cached in the application.</p>	<p>Receipt/Taxpayer/FamilyStatusEnum: models for holding taxpayer, receipt and family status logic</p> <p>ApplicationConstans: helper class that holds application constants and final values</p> <p>Swing classes: dependency with Swing classes to show modals after successful operation of some actions</p> <p>Java Input/Output: for loading files and saving files at the explorer</p> <p>Java util Logger: for logging output at the commandl. System out is not a proper way of doing this anymore</p> <p>TaxPayerUtils: collaboration with utility class which holds mechanism for amounts calculation</p> <p>BasicTaxBuilder: utility helper class which calculates the basic tax of a taxpayer</p>

Class Name: Company	
Responsibilities	Collaborations
Model class which holds information for a company used at a taxpayer receipt	No collaborations. It's a model class. We aim to keep it stupid and simple.

Class Name: Receipt	
Responsibilities	Collaborations
Model class which holds information for a taxpayer receipt	No collaborations. It's a model class. We aim to keep it stupid and simple.

Class Name: FamilyStatusEnum	
Responsibilities	Collaborations
Enum class which holds constants for a taxpayers family status	Java Logger to log some warnings if casual fromValue() internal method fails to give results

Class Name: TaxPayer	
Responsibilities	Collaborations
Model class which holds information for an American taxpayer	Depends on FamilyStatusEnum also

Class Name: ApplicationConstants	
Responsibilities	Collaborations
Helper class that holds application constants and final values	No collaborations

Class Name: ApplicationConstants	
Responsibilities	Collaborations
Helper class that holds application constants and final values	No collaborations

Class Name: BasicTaxBuilder	
Responsibilities	Collaborations
This class holds the logic for the taxpayer's tax calculations based on his family status and income	Dependency with TaxCategory and FamilyStatusEnum models

Class Name: InsertNewReceiptJDialog	
Responsibilities	Collaborations
This class holds the logic for managing the UI for the insert of a new receipt	<ul style="list-style-type: none"> • Swing and AWT APIs • FileManager • TaxPayerUtils • Receipt model

Class Name: LoadedTaxpayersJDialog	
Responsibilities	Collaborations
This class holds the logic for managing the UI for loaded taxpayer	<ul style="list-style-type: none"> • Swing and AWT APIs • FileManager • ChartsManager

Class Name: MainJFrameWindow	
Responsibilities	Collaborations
This class holds the java main method and launches the application. Displays the initial modal and connects it with the rest of the modals which will allow the user to consume the rest of the applications functionalities	<ul style="list-style-type: none"> • Swing and AWT APIs • FileManager

Class Name: TaxpayerLoadDataJDialog	
Responsibilities	Collaborations
This class holds the UI logic for loading taxpayers data from an input .txt or .xml file into our application memory	<ul style="list-style-type: none"> • Swing and AWT APIs • Java Input/Output • FileManager

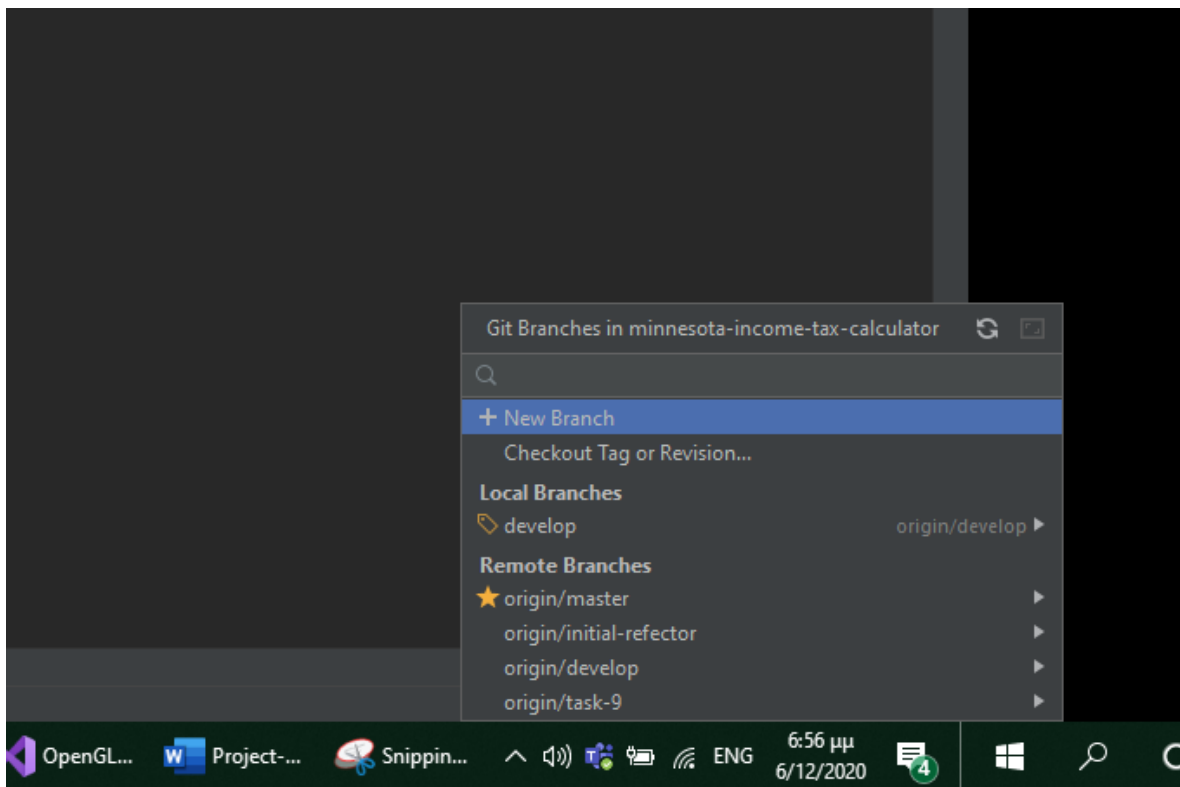
Class Name: TaxCategory	
Responsibilities	Collaborations
Model class which holds information for a tax category	Depends on FamilyStatusEnum

Class Name: TaxPayerUtils	
Responsibilities	Collaborations
This class holds the logic for some basic taxpayer functions, such as total receipts amounts calculations, taxPayer tax adjustment bases on the receipts he has collected	Dependency with Receipt and TaxPayer models

WORKING METHODOLOGY AND PRACTICES

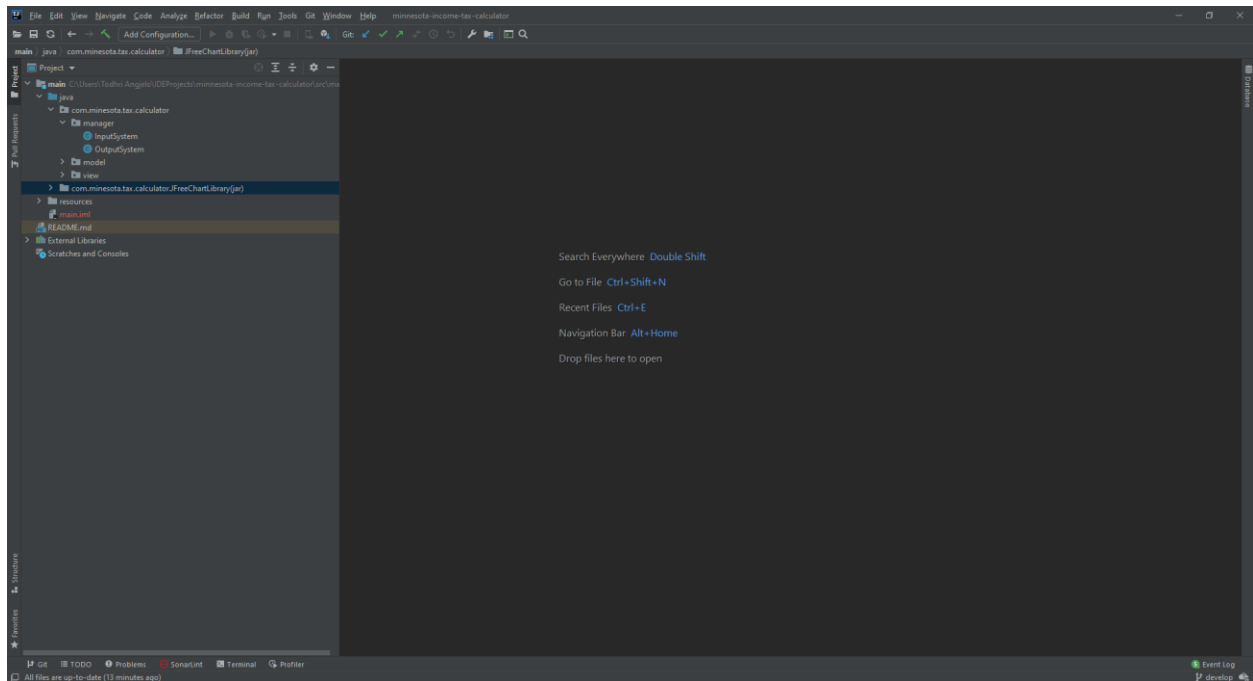
- **Github and code versioning.**

During the development and refactoring of the project github was used for code versioning and the tasks and backlog implementation. Pull requests were requested to develop each time a new feature was introduced and a new release was released each time a major pack of code changes was available for testing and consuming by the end user. (1 release only for the project delivery)



- **JetBrains IntelliJ IDEA was used for project development and code development.**

IntelliJ IDEA is the #1 IDE used and preferred by the majority of Java Developers. It is constantly updated and enhanced with new functionality that makes development easy and fun. Code completion, syntax highlighting, integrated commandline-less version control and code search among other goodies are one of the main factors that our project was developed fast, efficiently, easy and without errors.



(Zoom at the word to see picture details)

- TaxPayer class was refactored based on POJOs standards and best practices. An object class should be simple and carry no heavy logic for auto settings its fields on its own. Business logic should be in some util class or in some manager class. Pojos should by definition be simple and stupid. Only getters, setters and its variables.
- Javadocs were introduced for the key mechanisms (like tax calculations) and the complex business logic. Future developers (including the developer that wrote the javadocs) now do not need to rely on reading the code but can instead consume the method only by reading the high level description of it.

```

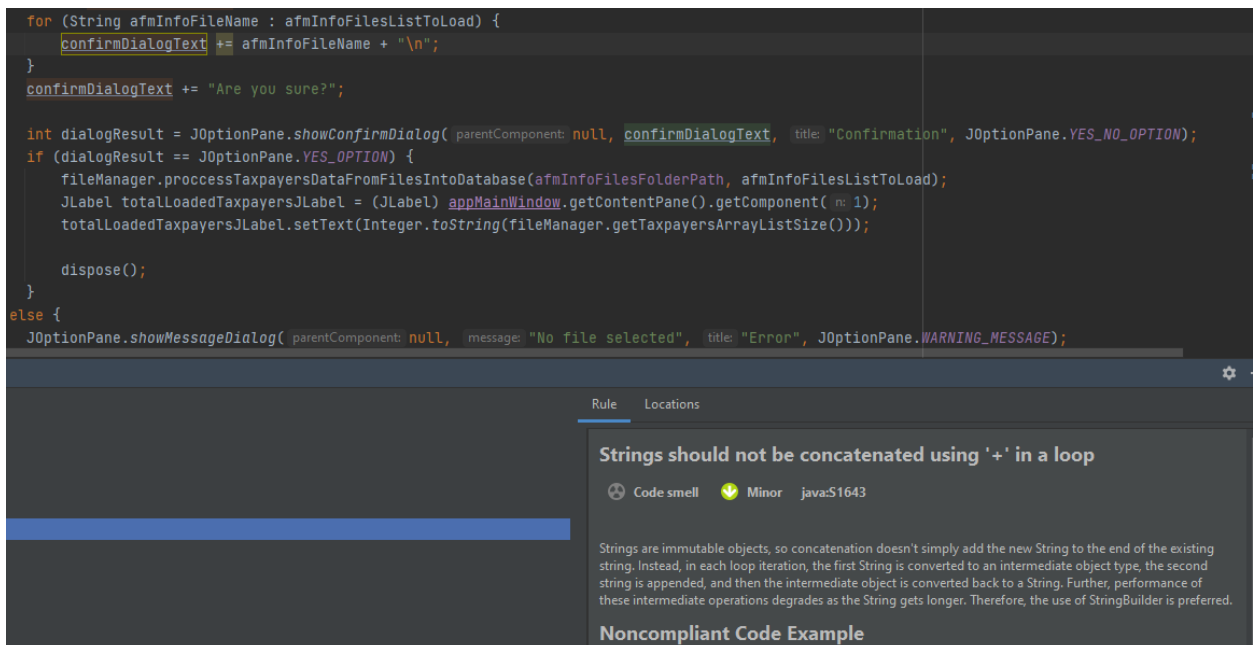
/**
 * Based on the receipts that a tax payer has turned in the system he gets
 * a tax increase or decrease accordingly
 *
 * @param taxpayer the taxpayer object to be adjusted
 */
public static void applyTaxpayerTaxAdjustments(TaxPayer taxpayer) {
    double receiptsTotal = getReceiptsTotalAmount(taxpayer.getReceipts());

    if ((receiptsTotal / taxpayer.getIncome()) < 0.2) {
        taxpayer.setTaxIncrease(taxpayer.getBasicTax() * 0.08);
    } else if ((receiptsTotal / taxpayer.getIncome()) < 0.4) {

```

(Zoom at the word to see picture details)

- SonarLint was used to prevent and refactor tricky error-prone code sections. This way we can use the IDE for both auto completion but also code proposals.



```

for (String afmInfoFileName : afmInfoFilesListToLoad) {
    confirmDialogText += afmInfoFileName + "\n";
}
confirmDialogText += "Are you sure?";

int dialogResult = JOptionPane.showConfirmDialog( parentComponent: null, confirmDialogText, title: "Confirmation", JOptionPane.YES_NO_OPTION);
if (dialogResult == JOptionPane.YES_OPTION) {
    fileManager.proccessTaxpayersDataFromFilesIntoDatabase(afmInfoFilesFolderPath, afmInfoFilesListToLoad);
    JLabel totalLoadedTaxpayersJLabel = (JLabel) appMainWindow.getContentPane().getComponent(0);
    totalLoadedTaxpayersJLabel.setText(Integer.toString(fileManager.getTaxpayersArrayListSize()));

    dispose();
}
else {
    JOptionPane.showMessageDialog( parentComponent: null, message: "No file selected", title: "Error", JOptionPane.WARNING_MESSAGE);
}

```

Strings should not be concatenated using '+' in a loop

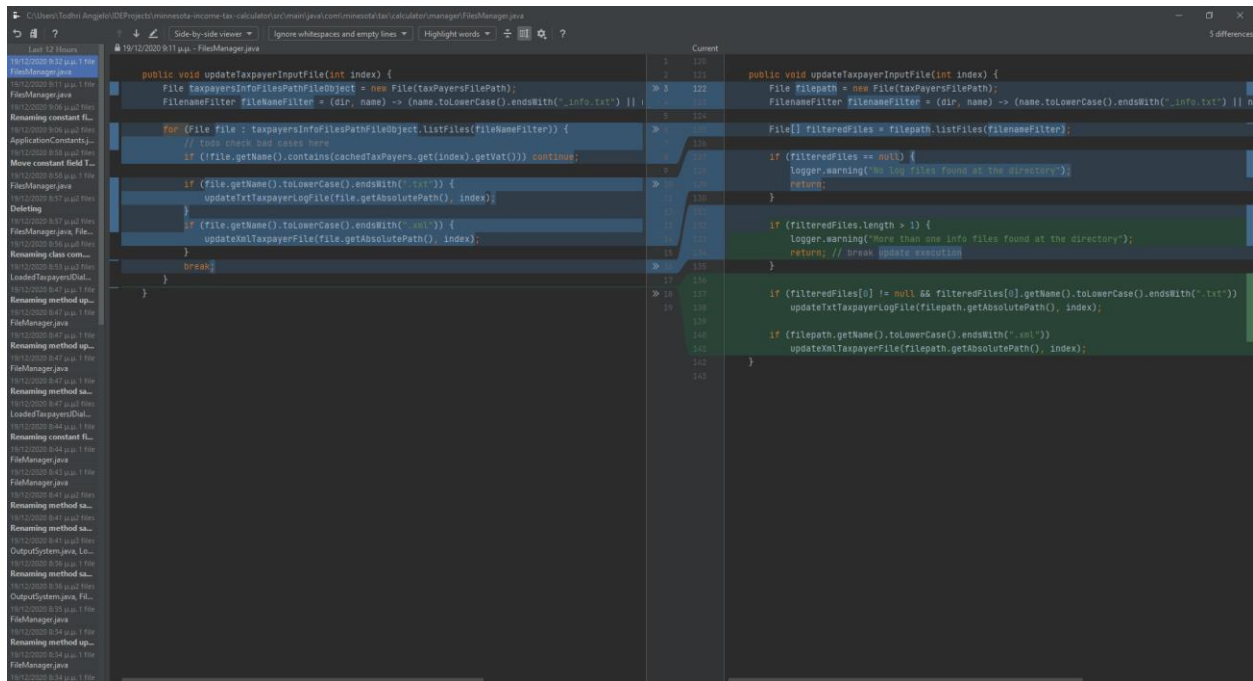
Code smell Minor java:S1643

Strings are immutable objects, so concatenation doesn't simply add the new String to the end of the existing string. Instead, in each loop iteration, the first String is converted to an intermediate object type, the second string is appended, and then the intermediate object is converted back to a String. Further, performance of these intermediate operations degrades as the String gets longer. Therefore, the use of StringBuilder is preferred.

Noncompliant Code Example

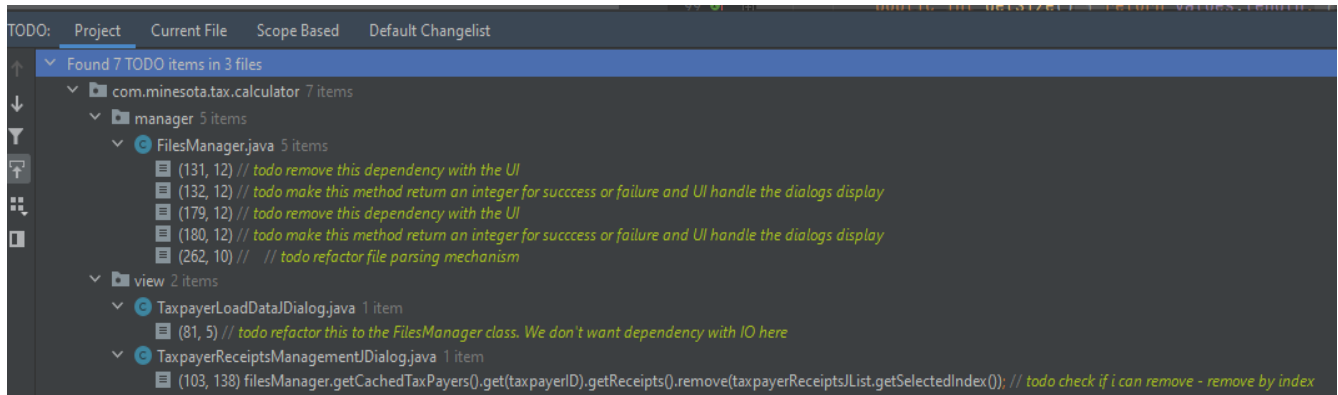
(Zoom at the word to see picture details)

- It's important for us as developers, that we don't program only the happy paths for our application and unfortunately the previous version of the application we were handed over had a lot of unhappy paths unhandled so we had to fix it. Using static code analysis inspection tools and our experience we fixed many cases like the following:



(Zoom at the word to see picture details)

TODOs AND FUTURE RELEASES



(Zoom at the word to see picture details)

To-dos and future plans are important for our application. The whole idea of the project is to re-create an application in a way that is easy to enhance and modify. Some basic ideas for future enhancement and modifications are the following:

- **Remove the dependency of FileManager to the User Interface classes.** We don't want the FileManager to alter the UI. Only the view classes should do this. The FileManager is called from the UI classes. The UI classes just need to know that the operation was successful. For example, if we have requested to create some files given some input, the UI just needs to know SUCCESS OR FAILURE in order to display the proper modal. FileManager methods should be refactored to return an integer which the UI classes will have to handle.
- The **parsing mechanism** which we implemented **was draft and still a little bit of complicated**. We should make the parsing mechanism way more abstract and exploit a ready method from some external library if that exists.
- There is **dependency at the View classes** (TaxPayerLoadDataJDialog) **with the Java IO library**. This is not a good technique as we want the UI classes to be responsible only for proper display and managing the AWT and swing modals. FileManager should be the one to have this logic that the UI classes will request for.
- **FileManager started as a helper class** in order we remove away Input and Output system. **It tends to be a GOD class** which holds a lot of logic and it could possibly be refactored into smaller and more readable units.
- External libraries like Junit and JFreeChart **should not be committed** into the code base and we shouldn't have to worry about adding each external dependency in our application separately **when a tool like Maven can do this for us**.
- Java is moving fast and beautiful things are coming in **later versions** (we used java 8) **and in Micromanagement frameworks like Spring Boot**. Things like dependency injections and inversion of control are super trendy nowadays and we would like our application to exploit those features that are offered without having to reinvent the wheel like we do in some cases inside our application (singleton and getInstance())

LINKS AND GITHUB

You can check here the source code, the video demo of the application and the readme.md

<https://github.com/todhriAngjelo/Minnesota-Income-Tax-Calculation-Project>

Guidelines regarding how to setup and launch the application or the Junit tests and also the pre-engineered source code can be located at github.

Guidelines: README.md

Pre-engineered version: master branch