SALES & COMMISSIONS MANAGEMENT REENGINEERING THE LEGACY CODE

OVERALL REPORT

FINAL VERSION

Ioannis Mpouzas AM:5025

Dimitrios Tzalokostas AM:4994

TABLE OF CONTENTS

Introduction	4
Refactored Design	4
Use Cases	4
Architecture	9
Detailed Design	10
Classes Responcibilities and Collaborations (CRC CARDS)	50

INTRODUCTION

The objective of the project was to reengineer a Java application. The general purpose of the application was to manage the sales made and the commissions attributed to the sales representatives of a small clothing company. Generally, the application accepts as input txt or xml files that contain receipts information about the sales made by a sales representative. The user of the application can add more receipts manually using the graphical user interface of the application. Based on the receipts information the application calculates the commission to be paid to the sales representative by the clothing company. The application generates respective reports in txt or xml format. After we refactored the code we added the functionality to provide input to the application via an html file and to store a report for a sales representative in an html file.

REFACTORED DESIGN

USE CASES

Use case ID	Load Sales Receipts Informations
Actors	User
Pre conditions	Existing file that is in .txt or .xml or .html format and has the name <AFM>_RECEIPTS
Main flow of events	1. The use case starts when the user selects the file that contains the sales receipts that a salesman made 2. The application loads the data from the receipts file
Alternative flow 1	In case that the user does not select file the application pops up an error message that says that the user does not have select a specific file and he tries again

Alternative flow 2	In case that the user selects to load a file that has the same salesman name with a name in the salesman list the application pops up a message that says that the specific salesman already exists and then he tries to load a file with different salesman name
Post conditions	The user can proceed to select the name of a salesman from a list of salesmans

Use case ID	Salesman Selection
Actors	User
Pre conditions	The correct load of the elements of the file
Main flow of events	1. The use case starts when the user selects the name of the salesman that he wants 2. The application loads the personal informations of the selected salesman and present them
Alternative flow 1	The application pops up an error message when the user has not selected a salesman from the list that says that the user has not select a salesman and then he selects a salesman name in order to continue
Post conditions	The user can proceed to take a general report

Use case ID	Display General Report
Actors	User
Pre conditions	Salesman selection
Main flow of events	1. The use case starts when the user selects the informations of the receipts that he want to display such as: 1.1.1. Total sales value 1.1.2. The number of sales for every different kind of clothing 1.1.3. The number of sales for a specific kind of clothing that the user selects 1.1.4. The commission of the salesman based on the sales that he made
Alternative flow 1	If the user does not select any field then the application will display no values in the fields that can display
Post conditions	The application will display the informations that the user selected

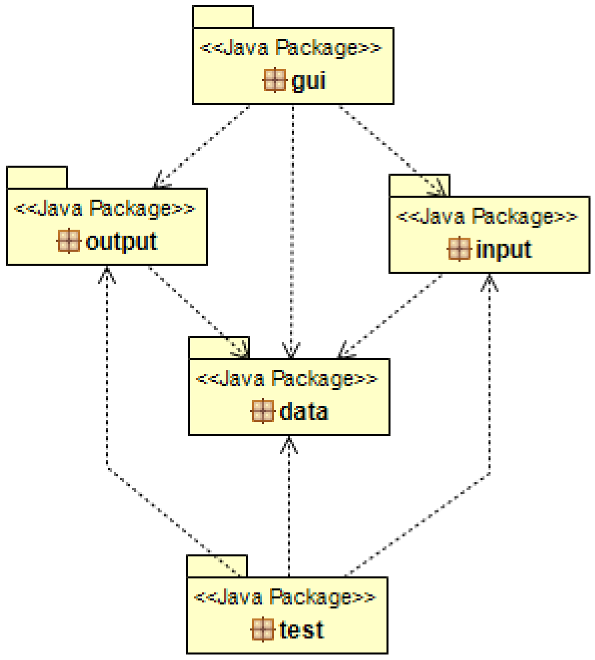
Use case ID	Save General Report
Actors	User
Pre conditions	Salesman selection
Main flow of events	1. The use case starts when the user selects to save the general report. 2. The user will name the file: <NAME FILE>_SALES 3. The user will save in the form of .txt , .xml or .html
Alternative flow 1	If the user does not provide the file with a name he will not be able to save the report
Post conditions	The application pops up a message that says that the report was successfully saved

Use case ID	Add New Receipts
Actors	User
Pre conditions	The new receipts are not included in the <AFM>_RECEIPTS file
Main flow of events	1. The use case starts when the user presses the button to add a new receipt 2. The user fills in all the necessary fields correctly 3. Then the user presses the add button
Alternative flow 1	If the user does not fill in a field the application pops up a message that says that the user should fill in all the fields so he tries again
Alternative flow 2	If the user does not fill in correctly a field, the number of receipts that have been added does not increases so he should try again to add a receipt
Post conditions	The new receipt is successfully added in the file <AFM>_RECEIPTS

Use case ID	Modify\Update Files's Informations
Actors	User
Pre conditions	Existing file <AFM>_RECEIPTS
Main flow of events	1. The use case starts when the user adds a new receipt 2. The new data modifies\updates the already existing data depending on what the user has added
Post conditions	The file is updated and the data are modified

ARCHITECTURE

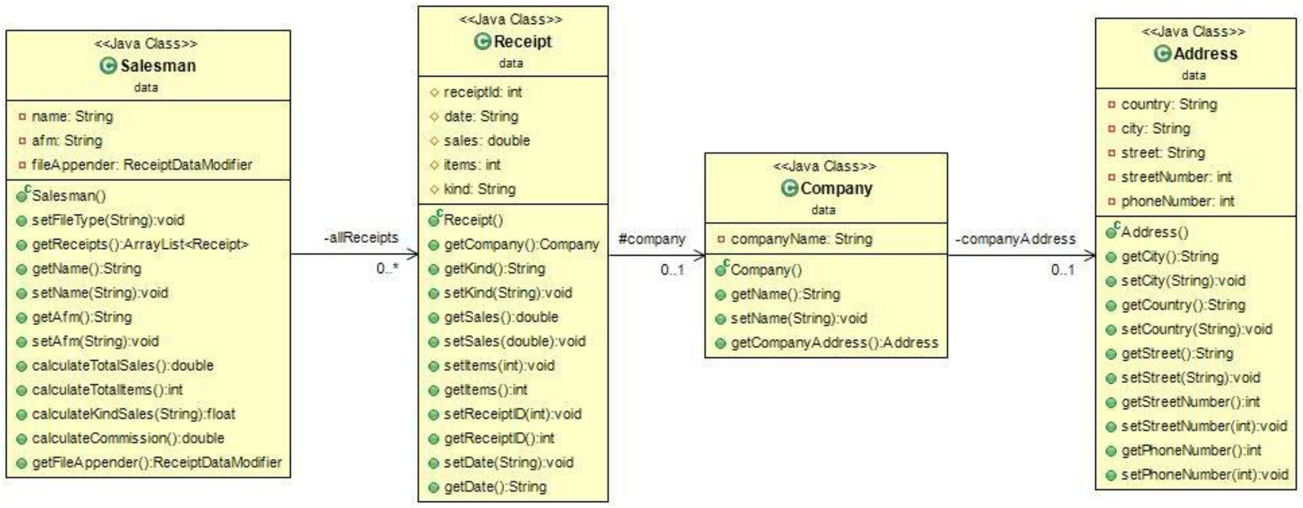
- First provide a overall UML package diagram that shows the architecture of the **refactored** application



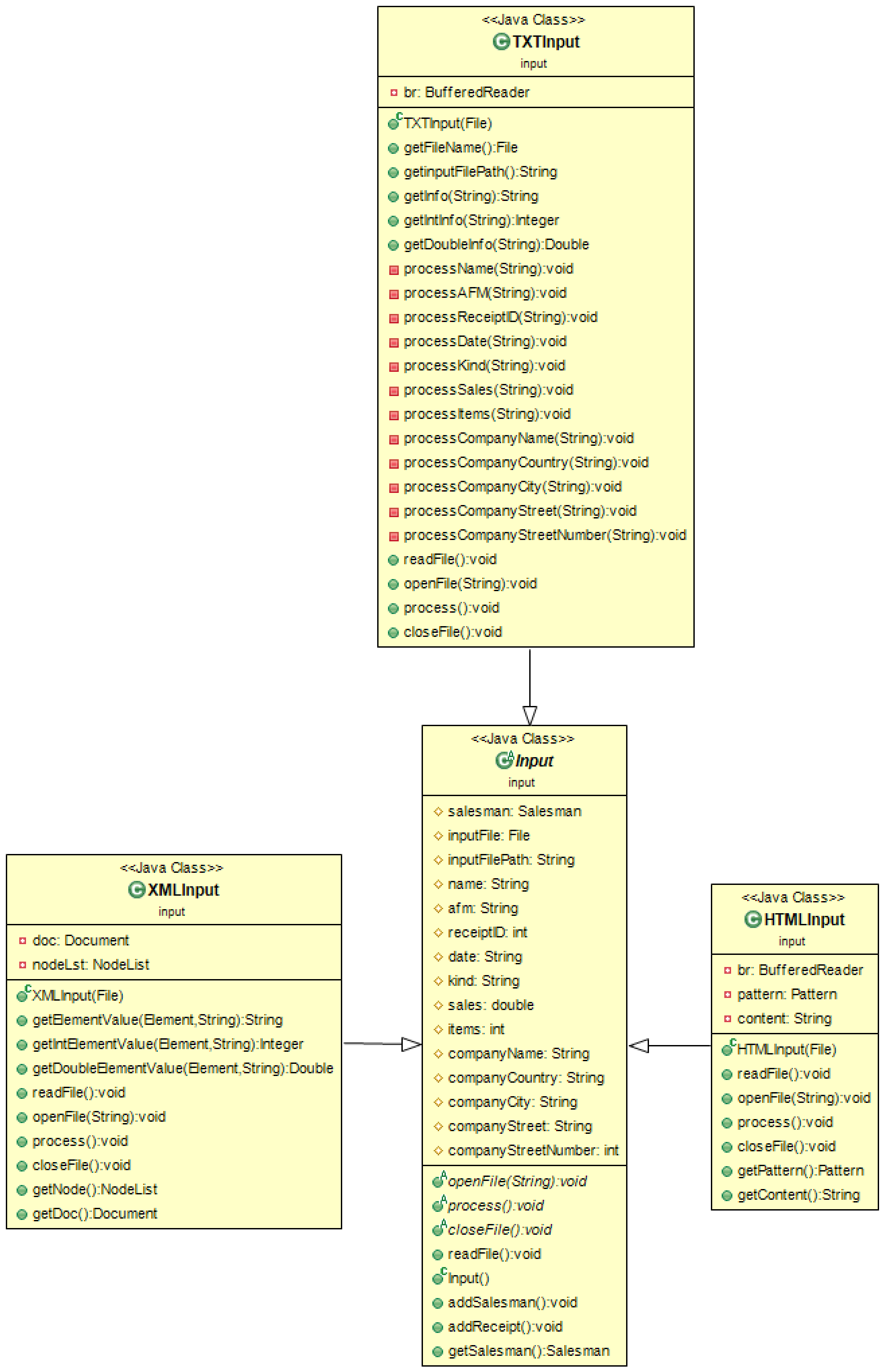
DETAILED DESIGN

- Then, provide the UML class diagrams for the classes of each package of the **refactored** application.

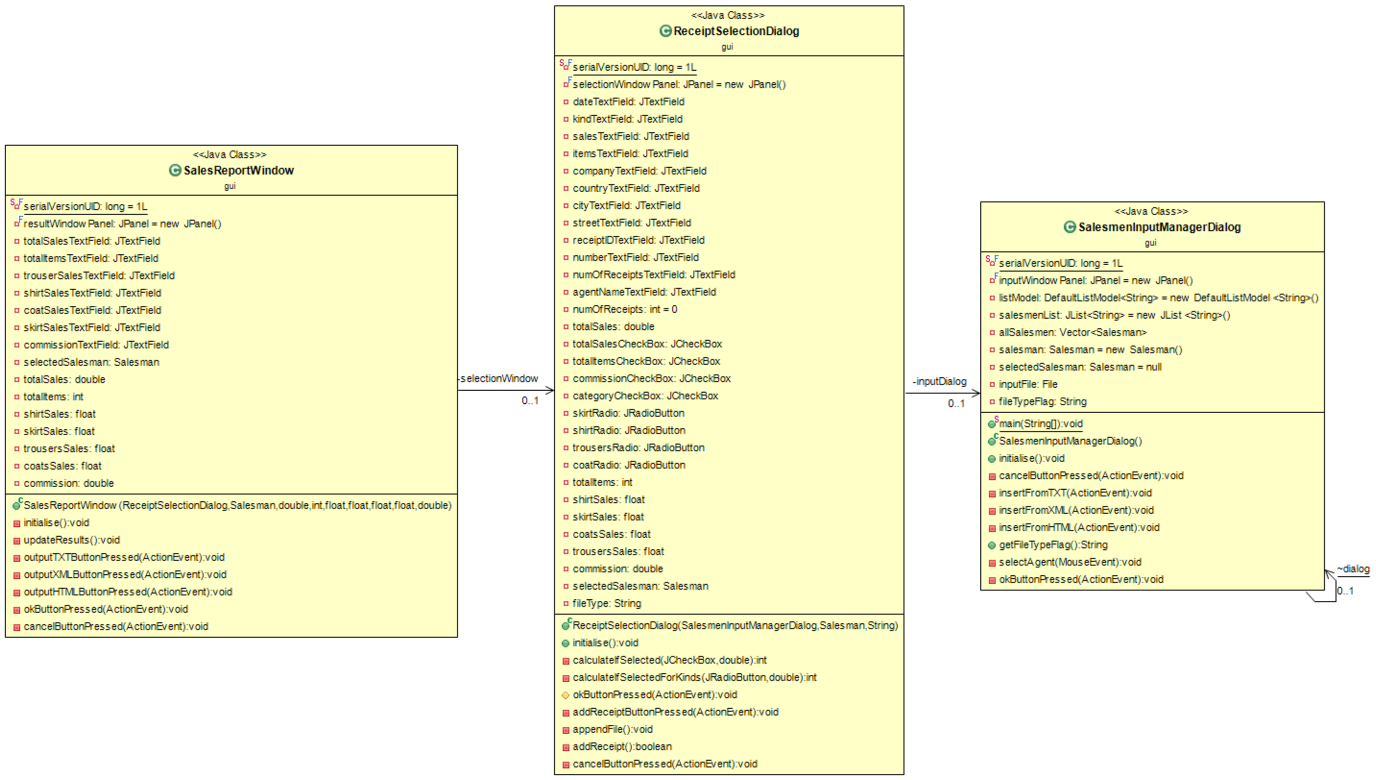
- **Data package**



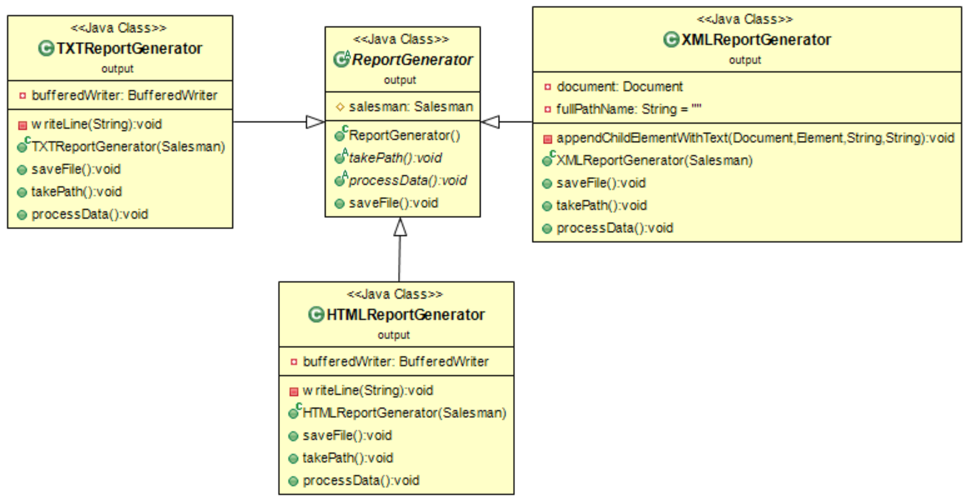
Input package

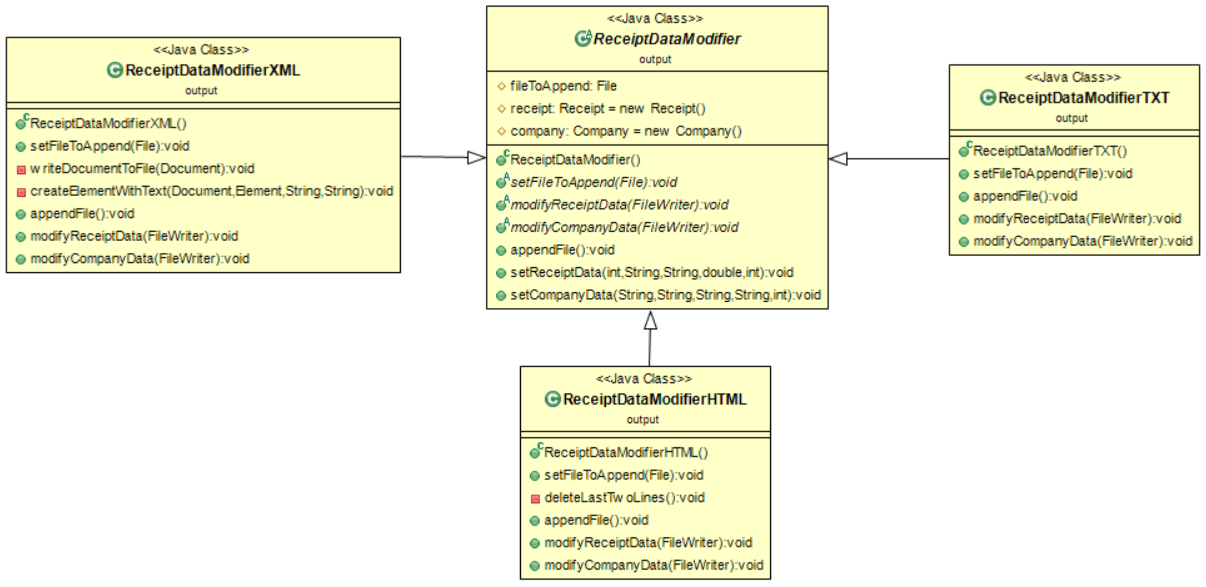


Gui package


<<Java Class>>
SalesReportWindow
gui
serialVersionUID: long = 1L
resultWindowPanel: JPanel = new JPanel()
totalSalesTextField: JTextField
totalItemsTextField: JTextField
trouserSalesTextField: JTextField
shirtSalesTextField: JTextField
coatSalesTextField: JTextField
skirtSalesTextField: JTextField
commissionTextField: JTextField
selectedSalesman: Salesman
totalSales: double
totalItems: int
shirtSales: float
skirtSales: float
trousersSales: float
coatsSales: float
commission: double
SalesReportWindow(ReceiptSelectionDialog,Salesman,double,int,float,float,float,float,double)
initialise():void
updateResults():void
outputTXTButtonPressed(ActionEvent):void
outputXMLButtonPressed(ActionEvent):void
outputHTMLButtonPressed(ActionEvent):void
okButtonPressed(ActionEvent):void
cancelButtonPressed(ActionEvent):void
-selectionWindow
0..1
<<Java Class>>
ReceiptSelectionDialog
gui
serialVersionUID: long = 1L
selectionWindowPanel: JPanel = new JPanel()
dateTextField: JTextField
kindTextField: JTextField
salesTextField: JTextField
itemsTextField: JTextField
companyTextField: JTextField
countryTextField: JTextField
cityTextField: JTextField
streetTextField: JTextField
receiptIDTextField: JTextField
numberTextField: JTextField
numOfReceiptsTextField: JTextField
agentNameTextField: JTextField
numOfReceipts: int = 0
totalSales: double
totalSalesCheckBox: JCheckBox
totalItemsCheckBox: JCheckBox
commissionCheckBox: JCheckBox
categoryCheckBox: JCheckBox
skirtRadio: JRadioButton
shirtRadio: JRadioButton
trousersRadio: JRadioButton
coatRadio: JRadioButton
totalItems: int
shirtSales: float
skirtSales: float
coatsSales: float
trousersSales: float
commission: double
selectedSalesman: Salesman
fileType: String
ReceiptSelectionDialog(SalesmenInputManagerDialog,Salesman,String)
initialise():void
calculateIfSelected(JCheckBox,double):int
calculateIfSelectedForKinds(JRadioButton,double):int
okButtonPressed(ActionEvent):void
addReceiptButtonPressed(ActionEvent):void
appendFile():void
addReceipt():boolean
cancelButtonPressed(ActionEvent):void
-inputDialog
0..1
<<Java Class>>
SalesmenInputManagerDialog
gui
serialVersionUID: long = 1L
inputWindowPanel: JPanel = new JPanel()
listModel: DefaultListModel<String> = new DefaultListModel<String>()
salesmenList: JList<String> = new JList<String>()
allSalesmen: Vector<Salesman>
salesman: Salesman = new Salesman()
selectedSalesman: Salesman = null
inputFile: File
fileTypeFlag: String
main(String[]):void
SalesmenInputManagerDialog()
initialise():void
cancelButtonPressed(ActionEvent):void
insertFromTXT(ActionEvent):void
insertFromXML(ActionEvent):void
insertFromHTML(ActionEvent):void
getFileTypeFlag():String
selectAgent(MouseEvent):void
okButtonPressed(ActionEvent):void
~dialog
0..1

Output package





<<Java Class>>
ReceiptDataModifierXML
output
ReceiptDataModifierXML()
setFileToAppend(File):void
writeDocumentToFile(Document):void
createElementWithText(Document,Element,String,String):void
appendFile():void
modifyReceiptData(FileWriter):void
modifyCompanyData(FileWriter):void
<<Java Class>>
ReceiptDataModifier
output
fileToAppend: File
receipt: Receipt = new Receipt()
company: Company = new Company()
ReceiptDataModifier()
setFileToAppend(File):void
modifyReceiptData(FileWriter):void
modifyCompanyData(FileWriter):void
appendFile():void
setReceiptData(int,String,String,double,int):void
setCompanyData(String,String,String,String,int):void
<<Java Class>>
ReceiptDataModifierTXT
output
ReceiptDataModifierTXT()
setFileToAppend(File):void
appendFile():void
modifyReceiptData(FileWriter):void
modifyCompanyData(FileWriter):void
<<Java Class>>
ReceiptDataModifierHTML
output
ReceiptDataModifierHTML()
setFileToAppend(File):void
deleteLastTwoLines():void
appendFile():void
modifyReceiptData(FileWriter):void
modifyCompanyData(FileWriter):void

HOW WE ADDRESSED THE DIFFERENT PROBLEMS OF THE OLD DESIGN

1. *Data Package*

➔ **Address.java**

The class a includes getter and setter methods for each of these properties. These methods are used to retrieve (get) and modify (set) the values of the properties. For example, the `getCity` method returns the value of the `city` property, and the `setCity` method sets a new value for the `city` property. These methods provide a way to access and modify the properties of an `Address` object in a controlled manner.

Here's a brief explanation of how getter and setter methods work:

- Getter: This is a method that returns the value of a specific property. For example, `getCity()` returns the `city` property of the `Address` object.

- Setter: This is a method that sets the value of a specific property. For example, `setCity(String city)` sets the value of the `city` property of the `Address` object.

➔ **Company.java**

The provided code defines a Java class named `Company`. This class is used to represent a company with the following properties:

- `companyName`: A `String` that represents the name of the company.

- `companyAddress`: An `Address` object that represents the address of the company.

The class includes a constructor, getter, and setter methods for these properties:

- **Constructor**: The `Company()` constructor initializes the `companyAddress` property with a new `Address` object. This means that when a new `Company` object is created, it will automatically have an `Address` object associated with it.

- **Getter and Setter for companyName**: The `getName()` method returns the value of the `companyName` property, and the `setName(String name)` method sets a new value for the `companyName` property.

- **Getter for companyAddress:** The `getCompanyAddress()` method returns the `Address` object associated with the `Company` object. There is no setter method for `companyAddress`, which means the address of the company can only be modified through the `Address` object itself (using its own setter methods), not directly from the `Company` object.

➔ **Receipt.java**

This class is used to represent a receipt with the following properties. Also, This class provides a way to create and manipulate `Receipt` objects, which can be used to represent receipts in a larger application.

- `receiptId`: An `int` that represents the ID of the receipt.

- `date`: A `String` that represents the date of the receipt.

- `sales`: A `double` that represents the sales amount of the receipt.

- `items`: An `int` that represents the number of items on the receipt.

- `company`: A `Company` object that represents the company associated with the receipt.

- `kind`: A `String` that represents the kind of receipt.

The class includes a constructor, getter, and setter methods for these properties:

- **Constructor**: The `Receipt()` constructor initializes the `kind` property with the string "No specific kind" and the `company` property with a new `Company` object. This means that when a new `Receipt` object is created, it will automatically have a kind and a `Company` object associated with it.

- **Getter and Setter for each property**: For each property, there is a getter method that returns the value of the property, and a setter method that sets a new value for the property. For example, the `getSales()` method returns the value of the `sales` property, and the `setSales(double sales)` method sets a new value for the `sales` property.

➔ **Salesman.java**

This class is used to represent a salesman with the following properties:

- `name`: A `String` that represents the name of the salesman.

- `afm`: A `String` that represents the AFM (a unique identifier) of the salesman.

- `allReceipts`: An `ArrayList` of `Receipt` objects that represents all the receipts associated with the salesman.

- `fileAppender`: A `ReceiptDataModifier` object that is used to modify the data of the receipts.

The class includes a constructor, getter, and setter methods for these properties, as well as several methods to calculate various statistics:

- **Constructor**: The `Salesman()` constructor initializes the `allReceipts` property with a new `ArrayList` of `Receipt` objects. This means that when a new `Salesman` object is created, it will automatically have an empty list of receipts.

- **Getter and Setter for each property**: For each property, there is a getter method that returns the value of the property, and a setter method that sets a new value for the property. For example, the `getName()` method returns the value of the `name` property, and the `setName(String name)` method sets a new value for the `name` property.

- **setFileType(String filetype):** This method sets the `fileAppender` property based on the provided file type. If the file type is "TXT", it creates a new `ReceiptDataModifierTXT` object. If the file type is "HTML", it creates a new `ReceiptDataModifierHTML` object. For any other file type, it creates a new `ReceiptDataModifierXML` object.

- calculateTotalSales(): This method calculates the total sales of all the receipts by summing up the sales of each receipt.

- calculateTotalItems(): This method calculates the total number of items of all the receipts by summing up the items of each receipt.

- **calculateKindSales(String s):** This method calculates the total sales of a specific kind of receipt. It sums up the items of each receipt that has the same kind as the provided string.

- calculateCommission(): This method calculates the commission of the salesman based on the total sales. The commission is calculated differently depending on the total sales amount.

2.Input Package

➔ **Input.java**

The `Input.java` file is an abstract class that represents a generic input source for a sales system. It has several protected fields that store information about a salesman and a receipt, as well as a `File` object that represents the input file.

The methods that implemented in this class are:

- **`openFile(String inputFilePath)`:** This is an abstract method that should be implemented by subclasses to open the input file. The `inputFilePath` parameter specifies the path to the input file.

- **`process()`:** This is another abstract method that should be implemented by subclasses to process the input file. This could involve reading data from the file and storing it in the fields of the `Input` object.

- **`closeFile()`:** This is the third abstract method that should be implemented by subclasses to close the input file after it has been processed.

- **`readFile()`:** This method calls the `openFile`, `process`, and `closeFile` methods in order. It also handles any exceptions that are thrown by these methods by printing the stack trace of the exception.

The `Input` class is abstract because it provides a general structure for reading and processing an input file, but it doesn't specify how these tasks should be performed. This allows you to create subclasses that implement these tasks in different ways.

➔ **TXTInput.java**

The `TXTInput.java` file is a subclass of the `Input` class that reads data from a text file. It uses a `BufferedReader` to read the file line by line.

The methods that implemented in this class are:

- `TXTInput(File recieptFileTXT)`: This is the constructor. It takes a `File` object as a parameter, which represents the text file to read. It stores this file in the `inputFile` field and sets `inputFilePath` to the absolute path of the file.

- **`getFileName()`:** This method returns the `File` object that was passed to the constructor.

- **`getinputFilePath()`:** This method returns the absolute path of the file.

- `getInfo(String line)`: This method takes a line of text as a parameter and returns the part of the line that comes after the first colon (`:`). It trims any leading or trailing whitespace from this part of the line.

- `getIntInfo(String line)`: This method is similar to `getInfo`, but it converts the part of the line after the first colon to an `Integer`.

- `getDoubleInfo(String line)`: This method is similar to `getInfo`, but it converts the part of the line after the first colon to a `Double`.

- `processName(String line)`: This method is not fully shown in the provided code, but it appears to process a line of text that starts with "Name:". The implementation of this method would depend on how you want to process the name.

The `TXTInput` class is designed to read a text file where each line has a key and a value separated by a colon. For example, a line might look like this: "Name: Apostolos Zarras". The `getInfo()`, `getIntInfo()`, and `getDoubleInfo()` methods can be used to extract the value from such a line.

➔ XMLInput.java

The `XMLInput.java` file is a subclass of the `Input` class that reads data from an XML file. It uses the `javax.xml.parsers.DocumentBuilder` to parse the XML file into a `Document` object, which can then be traversed to extract the data.

The methods that implemented in this class are:

- `XMLInput(File receiptFileXML)`: This is the constructor. It takes a `File` object as a parameter, which represents the XML file to read. It stores this file in the `inputFile` field and sets `inputFilePath` to the absolute path of the file.

- `getElementValue(Element element, String tagName)`: This method takes an `Element` object and a tag name as parameters. It finds the first element with the given tag name that is a child of the given element, and returns its text content. If the element or the text content does not exist, it will throw a `NullPointerException`.

- `getIntElementValue(Element element, String tagName)`: This method is similar to `getElementValue`, but it converts the text content to an `Integer`.

- `getDoubleElementValue(Element element, String tagName)`: This method is similar to `getElementValue`, but it converts the text content to a `Double`.

- `readFile()`: This method calls the `openFile`, `process`, and `closeFile` methods in order. It also handles any exceptions that are thrown by these methods by printing the stack trace of the exception.

The `XMLInput` class is designed to read an XML file where each piece of data is enclosed in a pair of tags. For example, a name might be enclosed in `<Name>` and `</Name>` tags. The `getElementValue`, `getIntElementValue`, and `getDoubleElementValue` methods can be used to extract the data from such tags.

Please note that the `openFile`, `process`, and `closeFile` methods are not shown in the provided code. They should be implemented to open the XML file, parse it into a `Document` object, traverse the document to extract the data, and close the file, respectively.

➔ **HTMLInput.java**

The `HTMLInput.java` file is a subclass of the `Input` class that reads data from an HTML file. It uses a `BufferedReader` to read the file line by line.

The methods that implemented in this class are:

- `HTMLInput(File receiptFileHTML)`: This is the constructor. It takes a `File` object as a parameter, which represents the HTML file to read. It stores this file in the `inputFile` field and sets `inputFilePath` to the absolute path of the file.

- `readFile()`: This method calls the `openFile`, `process`, and `closeFile` methods in order. It also handles any exceptions that are thrown by these methods by printing the stack trace of the exception.

- `openFile(String inputFilePath)`: This method creates a `BufferedReader` to read from the file at the given path. If an `IllegalArgumentException` is thrown (which might happen if the file is not valid), it shows a dialog with an error message.

- `process()`: This method is not fully shown in the provided code, but it should read and process the lines of the HTML file. This could involve using a `Pattern` and `Matcher` to extract data from the HTML tags.

The `HTMLInput` class is designed to read an HTML file and extract data from it. The exact way the data is extracted would depend on the format of your HTML file. For example, you might use regular expressions to find specific HTML tags and extract the text between them.

Please note that the `closeFile` method is not shown in the provided code. It should be implemented to close the `BufferedReader` after the file has been processed.

3.GUI Package

➔ **SalesmenInputManagerDialog.java**

SalesmenInputManagerDialog.java is the previous InputWindow.java and the class that provides the main method of the whole project. Also,this Java code is part of a GUI application that allows users to select and read files of different types (TXT, XML, HTML), and perform operations on the data read from these files. Here's a breakdown of the methods:

-initialize() method

The initialize() method appears to be setting up a Java Swing GUI for a dialog. The method that we added are:

- \`buttonTXTInput.addActionListener(new ActionListener() {...});\`, \`buttonXMLInput.addActionListener(new ActionListener() {...});\`, and \`buttonHTMLInput.addActionListener(new ActionListener() {...});\` - These lines add action listeners to the three input buttons. When each button is clicked, a corresponding method is called to insert data from a TXT, XML, or HTML file, respectively.

- **\`cancelButtonPressed(ActionEvent e)\`:** This method is called when the "Cancel" button is pressed. It exits the application.

- **\`insertFromTXT(ActionEvent evt)\`, \`insertFromXML(ActionEvent evt2)\`, \`insertFromHTML(ActionEvent evt3)**\`: These methods are called when the corresponding button is pressed. They open a file chooser dialog, read the selected file, create a \`Salesman\` object from the file, and add it to the \`allSalesmen\` list. If a salesman with the same name already exists in the list, a message is displayed. Otherwise, the salesman's name is added to the \`listModel\` and the \`fileTypeFlag\` is set to the corresponding file type.

- **\`getFileTypeFlag()\`:** This method returns the current value of \`fileTypeFlag\`.

- `selectAgent(MouseEvent e)`: This method is called when a mouse event occurs on `salesmenList`. It sets `selectedSalesman` to the `Salesman` object in `allSalesmen` that has the same name as the selected value in `salesmenList`.

- `okButtonPressed(ActionEvent evt)`: This method is called when the "OK" button is pressed. If no salesman is selected, a message is displayed. Otherwise, a new `ReceiptSelectionDialog` is created and displayed, and the current dialog is hidden.

➔ **ReceiptSelectionDialog.java**

The `ReceiptSelectionDialog` is the previous SelectionWindow.java and is a Java Swing dialog that allows users to select and view details of receipts. Here's a breakdown of the class members:

The constructor for the `ReceiptSelectionDialog` class takes three parameters: a `SalesmenInputManagerDialog` object, a `Salesman` object, and a `String` representing a file type flag. Here's what each line does:

- `inputDialog= dialog;` - This line assigns the `SalesmenInputManagerDialog` passed into the constructor to the `inputDialog` field of the `ReceiptSelectionDialog` object. This could be used later to interact with the `SalesmenInputManagerDialog` object.

- `selectedSalesman = salesman;` - This line assigns the `Salesman` passed into the constructor to the `selectedSalesman` field of the `ReceiptSelectionDialog` object. This `Salesman` object represents the currently selected salesman.

- `fileType = fileTypeFlag;` - This line assigns the `fileTypeFlag` passed into the constructor to the `fileType` field of the `ReceiptSelectionDialog` object. This flag could be used to determine the type of file to be processed.

- `initialise();` - This line calls the `initialise` method of the `ReceiptSelectionDialog` object. This method is likely responsible for setting up the GUI components of the dialog.

-The other methods are

- **`calculateIfSelected()` and `calculateIfSelectedForKinds()**`: These methods check if a checkbox or radio button is selected. If it is, they convert a double value to an integer and return it. If not, they return -1.

- **`okButtonPressed()`:** This method is called when the OK button is pressed. It calculates various sales metrics using the `calculateIfSelected` and `calculateIfSelectedForKinds` methods, then creates a new `SalesReportWindow` and makes it visible.

- **`addReceiptButtonPressed()**`: This method is called when the Add Receipt button is pressed. It checks if all text fields are filled in. If they are, it calls `addReceipt` to create a new receipt and `appendFile` to write the receipt data to a file. It then clears all text fields.

- **`appendFile()**`: This method writes the receipt data to a file using a `FileAppender` object associated with the selected salesman.

- **`addReceipt**()`: This method creates a new `Receipt` object and sets its fields using the text from various text fields. It then adds the receipt to the selected salesman's list of receipts and increments the number of receipts.

- **`cancelButtonPressed()`**: This method is called when the Cancel button is pressed. It disposes of the current dialog and makes the `inputDialog` visible again.

➔ **SalesReportWindow.java**

This is the previous ResultWindow.java and is used for creating dialog windows.

This is a Java class named `SalesReportWindow` that extends `JDialog`, which is a part of Java Swing library used for creating dialog windows.

The class has a **constructor** that takes several parameters:

- `sw`: An instance of `ReceiptSelectionDialog` to initialize `selectionWindow`.

- `salesman`: An instance of `Salesman` to initialize `selectedSalesman`.

- `tSales`, `tItems`, `shirtS`, `skirtS`, `trousersS`, `coatsS`, `com`: These are values to initialize `totalSales`, `totalItems`, `shirtSales`, `skirtSales`, `trousersSales`, `coatsSales`, `commission` respectively.

After initializing these fields, the constructor calls the `initialise` method.

The other methods in this class are:

- **`updateResults()`:** This method updates the text fields in the GUI with the current sales data. If the value of a particular sales data is negative, it disables the corresponding text field.

- `**outputTXTButtonPressed(ActionEvent evt)`:** This method is triggered when the "Output TXT" button is pressed. It creates a new `TXTReportGenerator` object, passing the `selectedSalesman` to the constructor. It then calls the `saveFile` method on the `TXTReportGenerator` object to save the report as a TXT file. After the file is saved, a message dialog is shown to the user indicating that the operation was successful.

- `**outputXMLButtonPressed(ActionEvent evt)`:** Similar to the previous method, but it creates an `XMLReportGenerator` object and saves the report as an XML file.

- `**outputHTMLButtonPressed(ActionEvent evt)`:** Similar to the previous methods, but it creates an `HTMLReportGenerator` object and saves the report as an HTML file.

- `**okButtonPressed(ActionEvent evt)`:** This method is triggered when the "OK" button is pressed. It simply exits the application using `System.exit(0)`.

- `**cancelButtonPressed(ActionEvent evt)`:** This method is triggered when the "Cancel" button is pressed. It makes the `selectionWindow` visible and disposes of the current window.

4.Output package

➔ **ReceiptDataModifier.java**

The `ReceiptDataModifier` is an abstract Java class that provides a structure for modifying receipt and company data and writing it to a file. Here's a breakdown of its components:

- `**fileToAppend**`: A `File` object that represents the file to which data will be appended.

- `**receipt**`: An instance of the `Receipt` class, which presumably represents a receipt.

- `**company`:** An instance of the `Company` class, which presumably represents a company.

The class has three abstract methods:

- **`setFileToAppend(File fileToAppend)`**: This method is intended to set the `fileToAppend` field. The implementation is left to the subclasses.

- `**modifyReceiptData(FileWriter fileWriter)`:** This method is intended to modify receipt data. The implementation is left to the subclasses.

- `**modifyCompanyData(FileWriter fileWriter)`:** This method is intended to modify company data. The implementation is left to the subclasses.

The class also has several concrete methods:

- `**appendFile()**`: This method opens the `fileToAppend` in append mode and calls `modifyReceiptData` and `modifyCompanyData` to write data to the file. If an `IOException` occurs, it prints the stack trace.

- `**setReceiptData(int id , String date , String kind , double sales , int items)`:** This method sets the data of the `receipt` object.

- `**setCompanyData(String companyName , String country , String city , String street , int streetNumber)**`: This method sets the data of the `company` object.

➔ **ReceiptDataModifierTXT.java**

This Java class, `ReceiptDataModifierTXT`, extends the `ReceiptDataModifier` class and overrides some of its methods to provide functionality specific to handling TXT files.

- `**setFileToAppend(File fileToAppend)`:** This method overrides the one in the superclass. It sets the `fileToAppend` field to the provided `File` object.

- `**appendFile()`:** This method opens the `fileToAppend` in append mode (as indicated by the `true` parameter in the `FileWriter` constructor). It then calls `modifyReceiptData(fileWriter)` and `modifyCompanyData(fileWriter)` to write data to the file. If an `IOException` occurs, it prints the stack trace.

- `modifyReceiptData(FileWriter fileWriter)`: This method overrides the one in the superclass. It writes the receipt data to the provided `FileWriter` in a specific format. If an `IOException` occurs, it prints the stack trace.

- `**modifyCompanyData(FileWriter fileWriter)**`: This method overrides the one in the superclass. It writes the company data to the provided `FileWriter` in a specific format. If an `IOException` occurs, it prints the stack trace.

The `appendFile` method prints "Mpike sto TXT" to the console when the extra data have been loaded to the .txt file.

➔ **ReceiptDataModifierXML.java**

This Java class, `ReceiptDataModifierXML`, extends the `ReceiptDataModifier` class and overrides some of its methods to provide functionality specific to handling XML files.

- **`setFileToAppend(File fileToAppend)`:** This method overrides the one in the superclass. It sets the `fileToAppend` field to the provided `File` object.

- **`writeDocumentToFile(Document doc)`:** This method uses a `Transformer` to write the provided `Document` to the `fileToAppend`. It sets the output properties to indent the XML for readability.

- `createElementWithText(Document doc, Element parent, String tagName, String textContent)`: This helper method creates a new `Element` with the provided `tagName` and `textContent`, and appends it to the `parent` element.

- `appendFile()`: This method opens the `fileToAppend` in append mode (as indicated by the `true` parameter in the `FileWriter` constructor). It then calls `modifyReceiptData(fileWriter)` and `modifyCompanyData(fileWriter)` to write data to the file. If an `Exception` occurs, it prints the stack trace.

- `**modifyReceiptData(FileWriter fileWriter**)`: This method overrides the one in the superclass. It creates a `Document` from the `fileToAppend`, creates a new "Receipt" element, and appends it to the document's root element. It then creates and appends several child elements to the "Receipt" element, and writes the `Document` to the file. If an `IOException` occurs, it prints the stack trace.

- `modifyCompanyData(FileWriter fileWriter)`: This method overrides the one in the superclass. It creates a `Document` from the `fileToAppend`, creates a new "Company" element, and appends it to the document's root element. It then creates and appends several child elements to the "Company" element, and writes the `Document` to the file. If an `IOException` occurs, it prints the stack trace.

The `appendFile` method prints "Mpike sto XML" to the console, when the extra data have been loaded to the .xml file.

➔ **ReceiptDataModifierHTML.java**

This Java class, `ReceiptDataModifierHTML`, extends the `ReceiptDataModifier` class and overrides some of its methods to provide functionality specific to handling HTML files.

- `setFileToAppend(File fileToAppend)`: This method overrides the one in the superclass. It sets the `fileToAppend` field to the provided `File` object.

- `**deleteLastTwoLines()**`: This method reads all lines from the `fileToAppend` into a list, removes the last two lines from the list, and then writes the list back to the file, effectively deleting the last two lines from the file.

- `appendFile()`: This method opens the `fileToAppend` in append mode (as indicated by the `true` parameter in the `FileWriter` constructor). It then calls `deleteLastTwoLines()`, `modifyReceiptData(fileWriter)`, and `modifyCompanyData(fileWriter)` to modify the file. If an `IOException` occurs, it prints the stack trace.

- `**modifyReceiptData(FileWriter fileWriter)`:** This method overrides the one in the superclass. It writes the receipt data to the provided `FileWriter` in a specific HTML format. If an `IOException` occurs, it prints the stack trace.

- **`modifyCompanyData(FileWriter fileWriter)`**: This method overrides the one in the superclass. It writes the company data to the provided `FileWriter` in a specific HTML format, and then writes the closing `</body>` and `</html>` tags. If an `IOException` occurs, it prints the stack trace.

The `appendFile` method prints "Mpike sto HTML" , when the extra data have been loaded to the .html file.

➔ **ReportGenerator.java**

This Java class, `ReportGenerator`, is an abstract class that provides a template for generating reports.

- `salesman`: This protected field is a `Salesman` object. Being protected, it can be accessed by this class, its subclasses, and classes in the same package.

- **`takePath()`**: This is an abstract method that doesn't have an implementation in this class. Subclasses of `ReportGenerator` are expected to provide their own implementation of this method. It's designed to throw an `Exception`, so any implementation should handle or declare this exception.

- `processData()`: This is another abstract method that doesn't have an implementation in this class. Subclasses of `ReportGenerator` are expected to provide their own implementation of this method. It's also designed to throw an `Exception`, so any implementation should handle or declare this exception.

- **`saveFile()`:** This method calls **the `takePath()` and `processData()`** methods inside a try block. If either method throws an `Exception`, the catch block catches it and prints the stack trace. This method provides a template for saving a file, but the specifics of taking the path and processing the data are left to the subclasses to implement.

➔ **TXTReportGenerator.java**

This Java class, `TXTReportGenerator`, extends the `ReportGenerator` abstract class and provides a concrete implementation for generating a report in a text file.

- `**bufferedWriter**`: This private field is a `BufferedWriter` object used for writing text to a character-output stream.

- `**writeLine(String line)`:** This private method writes a line of text to the `bufferedWriter` and then inserts a newline. If an `IOException` occurs, it displays an error message using `JOptionPane`.

- `TXTReportGenerator(Salesman a)`: This is the constructor of the class. It initializes the `salesman` field with the provided `Salesman` object.

- `saveFile()`: This method overrides the one in the superclass. It calls `takePath()` and `processData()` methods inside a try block. If either method throws an `Exception`, the catch block catches it and prints the stack trace.

- `**takePath()**`: This method overrides the one in the superclass. It opens a `JFileChooser` for the user to select a file. It sets a filter to only show text files. If the user approves, it gets the selected file's path, appends "_SALES.txt" to the path if it doesn't already end with ".txt", and initializes `bufferedWriter` with a `FileWriter` for the file at the path.

- `**processData()`:** This method overrides the one in the superclass. It writes the salesman's data to the `bufferedWriter` in a specific format, and then closes the `bufferedWriter`. If an `Exception` occurs, it's thrown to be handled by the caller.

➔ **XMLReportGenerator.java**

This Java class, `XMLReportGenerator`, extends the `ReportGenerator` abstract class and provides a concrete implementation for generating a report in an XML file.

- **`document`:** This private field is a `Document` object used for building an XML document.

- `fullPathName`: This private field is a `String` that stores the full path of the file where the XML document will be saved.

- **`appendChildElementWithText(Document document, Element parent, String tagName, String textContent)`:** This private method creates a new `Element` with the given `tagName` and `textContent`, and appends it as a child to the `parent` element.

- **`XMLReportGenerator(Salesman a)`:** This is the constructor of the class. It initializes the `salesman` field with the provided `Salesman` object.

- `saveFile()`: This method overrides the one in the superclass. It **calls `takePath()` and `processData()`** methods inside a try block. If either method throws an `Exception`, the catch block catches it and prints the stack trace.

- **`takePath()`**: This method overrides the one in the superclass. It opens a `JFileChooser` for the user to select a file. It sets a filter to only show XML files. If the user approves, it gets the selected file's path, appends "_SALES.xml" to the path if it doesn't already end with ".xml", and initializes `document` with a new `Document`.

- `processData()`: This method overrides the one in the superclass. It creates an XML document with the salesman's data, and then saves the document to the file at `fullPathName`. If an `Exception` occurs, it's caught and the stack trace is printed.

➔ **HTMLReportGenerator.java**

This Java class, `HTMLReportGenerator`, extends the `ReportGenerator` abstract class and provides a concrete implementation for generating a report in an HTML file.

- `bufferedWriter`: This private field is a `BufferedWriter` object used for writing text to a character-output stream.

- `writeLine(String line)`: This private method writes a line of text to the `bufferedWriter` and then inserts a newline. If an `IOException` occurs, it displays an error message using `JOptionPane`.

- `HTMLReportGenerator(Salesman a)`: This is the constructor of the class. It initializes the `salesman` field with the provided `Salesman` object.

- **`saveFile()`**: This method overrides the one in the superclass. It calls `takePath()` and `processData()` methods inside a try block. If either method throws an `Exception`, the catch block catches it and prints the stack trace.

- **`takePath()`:** This method overrides the one in the superclass. It opens a `JFileChooser` for the user to select a file. It sets a filter to only show HTML files. If the user approves, it gets the selected file's path, appends "_SALES.html" to the path if it doesn't already end with ".html", and initializes `bufferedWriter` with a `FileWriter` for the file at the path.

- `processData()`: This method overrides the one in the superclass. It writes the salesman's data to the `bufferedWriter` in HTML format, and then closes the `bufferedWriter`. If an `Exception` occurs, it's thrown to be handled by the caller.

***5.**<u>TEST PACKAGE</u>*

THIS IS THE PACKAGE THAT CONTAINS ALL THE TEST CASES OF THE PROJECT STARTING WITH:

➔ **DataTest.java**

This Java class, `DataTest`, contains a single test method `testData()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testData()` method does the following:

- Creates a `Salesman` object `s`.

- Retrieves the list of `Receipt` objects from `s` and asserts that this list is not null.

- Asserts that the name of `s` is not "AP ZARRAS".

- Asserts that the AFM (Tax Identification Number) of `s` is not null.

- Asserts that the total number of items sold by `s` is not null.

- Checks the sales of different kinds of items ("Shirts", "Trousers", "Coats", "Skirts") by `s` and asserts that these are not null.

- Checks if the `kind` is one of the valid kinds ("Shirts", "Coats", "Skirts", "Trousers"). If not, it fails the test with a message "Invalid kind provided for testing".

- Asserts that the total number of items sold by `s` is not null.

- Asserts that the commission of `s` is not null.

- Asserts that the `FileAppender` of `s` is not null.

- If all assertions pass, it prints "All data have been passed!!".

- If any exception occurs during the test, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `Salesman` class and its methods. It checks that the methods return valid, non-null results and that the `Salesman` object behaves as expected.

➔ **TXTTest.java**

This Java class, `TXTTest`, contains a single test method `testProcessMethodTXT()`. It uses the JUnit testing framework, as indicated by the `@org.junit.Test` annotation.

The `testProcessMethodTXT()` method does the following:

- Creates a `File` object that points to a test file located at ".\\src\\test\\test-case-1-TXT.txt".

- Prints a message to the console indicating that the file has been loaded.

- Prints the absolute path of the file to the console.

- Creates a `TXTInput` object, `txtInput`, passing the `File` object to its constructor. This presumably loads the file for processing.

- Creates a `DataTest` object, `dt`, and calls its `testData()` method. This seems to be a separate set of tests that are run as part of this test method.

- Calls the `readFile()` method on `txtInput`, which presumably reads and processes the file.

- Asserts that various properties of the `Salesman` object within `txtInput`, as well as various pieces of information retrieved from `txtInput` using the `getInfo()` method, are not null. These properties and pieces of information include the salesman's name and AFM, the receipt ID, date, kind, sales, items, company, country, city, street, and number.

- If any exception occurs during the test, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `TXTInput` class and its methods. It checks that the methods return valid, non-null results and that the `TXTInput` object behaves as expected when processing a file.

➔ **XMLTest.java**

This Java class, `XMLtest`, contains a single test method `testProcessMethodXML()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testProcessMethodXML()` method does the following:

- Creates a `File` object that points to a test file located at ".\\src\\test\\test-case-2-XML.xml".

- Prints a message to the console indicating that the file has been loaded.

- Prints the absolute path of the file to the console.

- Creates an `XMLInput` object, `xmlInput`, passing the `File` object to its constructor. This presumably loads the file for processing.

- Calls the `readFile()` method on `xmlInput`, which presumably reads and processes the XML file.

- Retrieves a `NodeList` from `xmlInput` and gets the first `Element` from the list, `agentElement`.

- Asserts that the "Name" and "AFM" values of `agentElement` are not null.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

- Creates a `DataTest` object, `dt`, and calls its `testData()` method. This seems to be a separate set of tests that are run as part of this test method.

- Iterates over the `NodeList` and for each `Element`, `receiptElement`, in the list, asserts that various properties are not null. These properties include "ReceiptID", "Date", "Kind", "Sales", "Items", "Company", "Country", "City", "Street", and "Number".

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `XMLInput` class and its methods. It checks that the methods return valid, non-null results and that the `XMLInput` object behaves as expected when processing an XML file.

➔ **HTMLTest.java**

This Java class contains a single test method `testProcessMethodHTML()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testProcessMethodHTML()` method does the following:

- Creates a `File` object that points to a test file located at ".\\src\\test\\test-case-3-HTML.html".

- Prints a message to the console indicating that the file has been loaded.

- Creates an `HTMLInput` object, `htmltest`, passing the `File` object to its constructor. This presumably loads the file for processing.

- Prints the absolute path of the file to the console.

- Calls the `readFile()` method on `htmltest`, which presumably reads and processes the HTML file.

- Retrieves a `Pattern` from `htmltest` and creates a `Matcher` object, `matcherSalesManInfo`, that matches this pattern against the content of the HTML file.

- Asserts that `matcherSalesManInfo` finds a match in the content and that the first two groups in the match are not null.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

- Creates another `Matcher` object, `matcher`, that matches the same pattern against the content of the HTML file.

- Creates a `DataTest` object, `dt`, and calls its `testData()` method. This seems to be a separate set of tests that are run as part of this test method.

- Asserts that `matcher` finds a match in the content and that various groups in the match are not null. These groups are converted to integers or doubles as necessary.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `HTMLInput` class and its methods. It checks that the methods return valid, non-null results and that the `HTMLInput` object behaves as expected when processing an HTML file.

➔ **ReceiptDataModifierTXTTest.java**

This Java class, `ReceiptDataModifierTXTTest`, contains a single test method `testReceiptTXT()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testReceiptTXT()` method does the following:

- Sets a `File` object to be appended by the `ReceiptDataModifierTXT` object.

- Calls the `appendFile()` method on `ReceiptDataModifierTXT`, which presumably appends some data to the file.

- Reads all bytes from the file into a `String` object, `content`.

- Asserts that `content` is not null, ensuring that the file was read successfully and contains data.

- Creates a `FileWriter` object, `fileWriter`, for the file.

- Calls the `modifyReceiptData()` and `modifyCompanyData()` methods on `ReceiptDataModifierTXT`, passing `fileWriter` to both. These methods presumably modify the data in the file.

- Flushes and closes `fileWriter`.

- Reads all bytes from the file into `content` again.

- Asserts that `content` is not null, ensuring that the file was read successfully and contains data after the modifications.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `ReceiptDataModifierTXT` class and its methods. It checks that the methods perform their intended operations without errors and that the file contains data after these operations.

➔ **ReceiptDataModifierXMLTest.java**

This Java class, `ReceiptDataModifierXMLTest`, contains two test methods: `testModifyReceiptData()` and `testModifyCompanyData()`. It uses the JUnit testing framework, as indicated by the `@Test` annotations.

The `testModifyReceiptData()` method does the following:

- Sets a `File` object to be appended by the `ReceiptDataModifierXML` object.

- Creates a `FileWriter` object, `fileWriter`, for the file in a try-with-resources statement, which ensures that `fileWriter` is closed at the end of the statement.

- Calls the `modifyReceiptData()` method on `ReceiptDataModifierXML`, passing `fileWriter`. This method presumably modifies the receipt data in the file.

- If any exception occurs during these operations, it fails the test with a message "Exception should not be thrown: " followed by the exception message.

- Reads all bytes from the file into a `String` object, `content`.

- Asserts that `content` contains the string "<Receipt>", ensuring that the receipt data was modified successfully.

The `testModifyCompanyData()` method does the following:

- Sets a `File` object to be appended by the `ReceiptDataModifierXML` object.

- Creates a `FileWriter` object, `fileWriter`, for the file in a try-with-resources statement.

- Calls the `modifyCompanyData()` method on `ReceiptDataModifierXML`, passing `fileWriter`. This method presumably modifies the company data in the file.

- If any exception occurs during these operations, it fails the test with a message "Exception should not be thrown: " followed by the exception message.

- Reads all bytes from the file into a `String` object, `content`.

- Asserts that `content` contains the string "<Company>", ensuring that the company data was modified successfully.

These test methods are designed to validate the functionality of the `ReceiptDataModifierXML` class and its methods. They check that the methods perform their intended operations without errors and that the file contains the expected data after these operations.

➔ **ReceiptDataModifierHTMLTest.java**

This Java class, `ReceiptDataModifierHTMLTest`, contains two test methods: `testModifyReceiptData()` and `testModifyCompanyData()`. It uses the JUnit testing framework, as indicated by the `@Test` annotations.

The `testModifyReceiptData()` method does the following:

- Sets a `File` object to be appended by the `ReceiptDataModifierHTML` object.

- Creates a `FileWriter` object, `fileWriter`, for the file in a try-with-resources statement, which ensures that `fileWriter` is closed at the end of the statement.

- Calls the `modifyReceiptData()` method on `ReceiptDataModifierHTML`, passing `fileWriter`. This method presumably modifies the receipt data in the HTML file.

- If any exception occurs during these operations, it fails the test with a message "Exception should not be thrown: " followed by the exception message.

- Reads all bytes from the file into a `String` object, `content`.

- Asserts that `content` contains the string "<!-- New Receipt -->", ensuring that the receipt data was modified successfully.

The `testModifyCompanyData()` method does the following:

- Sets a `File` object to be appended by the `ReceiptDataModifierHTML` object.

- Creates a `FileWriter` object, `fileWriter`, for the file in a try-with-resources statement.

- Calls the `modifyCompanyData()` method on `ReceiptDataModifierHTML`, passing `fileWriter`. This method presumably modifies the company data in the HTML file.

- If any exception occurs during these operations, it fails the test with a message "Exception should not be thrown: " followed by the exception message.

- Reads all bytes from the file into a `String` object, `content`.

- Asserts that `content` contains the string "<p>Company:", ensuring that the company data was modified successfully.

These test methods are designed to validate the functionality of the `ReceiptDataModifierHTML` class and its methods. They check that the methods perform their intended operations without errors and that the file contains the expected data after these operations.

➔ **TXTReportGeneratorTest.java**

This Java class, `TXTReportGeneratorTest`, contains a single test method `testProcessData()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testProcessData()` method does the following:

- Creates a `StringWriter` and a `BufferedWriter` that writes to the `StringWriter`.

- Uses reflection to access the private `bufferedWriter` field in the `TXTReportGenerator` object and sets it to the `BufferedWriter` created earlier. This allows the test to capture the output of `TXTReportGenerator` in the `StringWriter`.

- Checks that the file ".\\src\\test\\test-case-1-TXT.txt" exists.

- Calls the `processData()` method on `TXTReportGenerator`, which presumably processes some data and writes the results to its `bufferedWriter`.

- Converts the contents of the `StringWriter` to a `String`.

- Asserts that this string contains various pieces of information about the `Salesman` object, such as its name, AFM, total sales, sales of different kinds of items, and commission. This ensures that `TXTReportGenerator` processed the data correctly and wrote the expected output.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `TXTReportGenerator` class and its `processData()` method. It checks that the method performs its intended operations without errors and that it produces the expected output.

➔ **XMLReportGeneratorTest.java**

This Java class, `XMLReportGeneratorTest`, contains a test method `testProcessData()` and a helper method `createDocument()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testProcessData()` method does the following:

- Creates a `Salesman` object and an `XMLReportGenerator` object that uses this `Salesman`.

- Creates a new `Document` object using a `DocumentBuilderFactory` and a `DocumentBuilder`.

- Calls the `processData()` method on `XMLReportGenerator`, which presumably processes some data and writes the results to an XML document.

- Uses reflection to access the private `document` field in the `XMLReportGenerator` object and assigns it to the `document` field in the test class.

- Gets the root element of the document and asserts that it contains elements with the names "Name", "AFM", "TotalSales", "TrouserSales", "SkirtsSales", "ShirtsSales", "CoatsSales", and "Commission". This ensures that `XMLReportGenerator` processed the data correctly and wrote the expected output to the document.

The `createDocument()` method creates a new `Document` object using a `DocumentBuilderFactory` and a `DocumentBuilder`, and assigns it to the `document` field in the test class. This method is not called in the provided code, but it could be used to set up the `document` before each test.

This test method is designed to validate the functionality of the `XMLReportGenerator` class and its `processData()` method. It checks that the method performs its intended operations without errors and that it produces the expected output.

➔ **HTMLReportGenerator.java**

This Java class, `HTMLReportGeneratorTest`, contains a single test method `testProcessData()`. It uses the JUnit testing framework, as indicated by the `@Test` annotation.

The `testProcessData()` method does the following:

- Creates a `Salesman` object and a `HTMLReportGenerator` object that uses this `Salesman`.

- Creates a temporary file with a ".html" extension.

- Uses reflection to access the private `bufferedWriter` field in the `HTMLReportGenerator` object and sets it to a `BufferedWriter` that writes to the temporary file.

- Creates a `StringWriter` and a `BufferedWriter` that writes to the `StringWriter`, and sets the `bufferedWriter` field in `HTMLReportGenerator` to this `BufferedWriter`. This allows the test to capture the output of `HTMLReportGenerator` in the `StringWriter`.

- Calls the `processData()` method on `HTMLReportGenerator`, which presumably processes some data and writes the results to an HTML file.

- Converts the contents of the `StringWriter` to a `String`.

- Asserts that this string contains various pieces of HTML code and information about the `Salesman` object, such as its name, AFM, total sales, sales of different kinds of items, and commission. This ensures that `HTMLReportGenerator` processed the data correctly and wrote the expected output.

- If any exception occurs during these operations, it fails the test with a message "Exception occurred: " followed by the exception message.

This test method is designed to validate the functionality of the `HTMLReportGenerator` class and its `processData()` method. It checks that the method performs its intended operations without errors and that it produces the expected output.

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: Address from Address.java	
Responsibilities	**Collaborations**
- **Encapsulation**: It	

encapsulates the address details such as country, city, street, street number, and phone number. It provides getters and setters for these fields, ensuring that the fields can be safely accessed and modified. - **Data Storage**: It stores the data of an address. This data can be used by other classes that need to work with address information.	- **User or Customer Class**: A class representing a user or customer might have an `Address` object as a field, representing the user's or customer's address. - **Order or Shipment Class**: A class representing an order or shipment might also have an `Address` object, representing the delivery address for the order or shipment.

Class Name: Company from Company.java	
Responsibilities	**Collaborations**
- **Encapsulation**: It	- **Address Class**: A `Company` object has an `Address` object as a field, representing the company's address. The `Company` class uses

encapsulates the company details such as company name and company address. It provides getters and setters for these fields, ensuring that the fields can be safely accessed and modified. - **Data Storage**: It stores the data of a company. This data can be used by other classes that need to work with company information.	the `Address` class to store and retrieve the company's address information.

Class Name: Receipt from Receipt.java	
Responsibilities	**Collaborations**
- **Data Encapsulation**: It encapsulates the receipt	**- Company Class**: A `Receipt` object has a `Company` object as a field, representing the company associated with the receipt. The `Receipt` class

details such as receipt ID, date, sales, items, company, and kind. It provides getters and setters for these fields, ensuring that the fields can be safely accessed and modified. - **Data Storage**: It stores the data of a receipt. This data can be used by other classes that need to work with receipt information.	uses the `Company` class to store and retrieve the company's information**.**

Class Name: Salesman from Salesman.java	
Responsibilities	**Collaborations**
- Data Encapsulation: It encapsulates the	

salesman details such as name, afm (tax identification number), and allReceipts (a list of receipts associated with the salesman). It provides getters and setters for these fields, ensuring that the fields can be safely accessed and modified. **- Data Storage:** It stores the data of a salesman. This data can be used by other classes that need to work with salesman information. **- File Type Management**: It manages the file type for receipt data modification. Depending on the file type, it creates an instance of \`ReceiptDataModifierTXT\`, \`ReceiptDataModifierHTML\`, or \`ReceiptDataModifierXML\`.	**- Receipt Class:** A \`Salesman\` object has a list of \`Receipt\` objects as a field, representing the receipts associated with the salesman. The \`Salesman\` class uses the \`Receipt\` class to store and retrieve receipt information. **- ReceiptDataModifier Class:** A \`Salesman\` object has a \`ReceiptDataModifier\` object as a field, representing the file appender for receipt data. The \`Salesman\` class uses the \`ReceiptDataModifier\` class (and its subclasses) to modify receipt data based on the file type.

Class Name: SalesmenInputManagerDialog from SalesmenInputManagerDialog.java	
Responsibilities	**Collaborations**
- User Interaction: It interacts with the	

user, allowing them to input information related to salesmen. **- Data Validation**: It validates the user's input to ensure it's in the correct format and meets any necessary criteria. **- Data Transmission:** It transmits the user's input to other parts of the system that need this information.	**- Salesman Class**: The `SalesmenInputManagerDialog` class likely interacts with the `Salesman` class, creating new instances or updating existing ones based on the user's input. **- GUI Classes:** The `SalesmenInputManagerDialog` class likely interacts with other GUI classes in the system, updating them based on the user's input or triggering them to display certain information.

Class Name: ReceiptSelectionDialog from ReceiptSelectionDialog.java	
Responsibilities	**Collaborations**

- User Interaction: It interacts with the user, allowing them to select a receipt from a list or some other form of collection. **- **Data Presentation**:** It presents the available receipts to the user in a user-friendly format. - ****Data Transmission****: It transmits the user's selection to other parts of the system that need this information.	**- Receipt Class:** The \`ReceiptSelectionDialog\` class likely interacts with the \`Receipt\` class, retrieving instances to display to the user and transmitting the user's selection to other parts of the system. **- GUI Classes:** The \`ReceiptSelectionDialog\` class likely interacts with other GUI classes in the system, updating them based on the user's selection or triggering them to display certain information**.**

Class Name: SalesReportWindow from SalesReportWindow.java	
Responsibilities	**Collaborations**

- User Interface Management: It manages the user interface for the sales report window. This includes creating and positioning text fields and buttons, and adding them to the window. **- Data Presentation:** It presents sales data to the user in a user-friendly format. This data includes total sales, total items, and sales of different types of items.	**- JTextField Class:** The \`SalesReportWindow\` class uses \`JTextField\` objects to display sales data. It sets the text of these fields to represent the data, and adds them to the window. **- JButton Class:** The \`SalesReportWindow\` class uses a \`JButton\` object to allow the user to close the window. It adds an action listener to the button to handle the click event.

Class Name: ReceiptDataModifier from ReceiptDataModifier.java	
Responsibilities	**Collaborations**

- **Data Modification**: It provides abstract methods for modifying receipt and company data. These methods must be implemented by any concrete subclass. - **File Appending:** It provides a method for appending data to a file. This method uses the abstract methods to modify the data before it is written to the file. - **Data Setting**: It provides methods for setting receipt and company data. These methods directly modify the fields of a `Receipt` and `Company` object.	- **Receipt Class**: The `ReceiptDataModifier` class uses a `Receipt` object to store and modify receipt data. It directly modifies the fields of the `Receipt` object. - **Company Class:** The `ReceiptDataModifier` class uses a `Company` object to store and modify company data. It directly modifies the fields of the `Company` object.

Class Name: ReceiptDataModifierTXT from ReceiptDataModifierTXT.java	
Responsibilities	**Collaborations**

- Data Modification: It implements the abstract methods of `ReceiptDataModifier` to modify receipt and company data in a way that is suitable for TXT files. This could involve formatting the data as a string. **- File Appending:** It inherits the method for appending data to a file from `ReceiptDataModifier`. This method uses the implemented abstract methods to modify the data before it is written to the file.	**- Receipt Class**: The `ReceiptDataModifierTXT` class uses a `Receipt` object to store and modify receipt data. It directly modifies the fields of the `Receipt` object. **- Company Class:** The `ReceiptDataModifierTXT` class uses a `Company` object to store and modify company data. It directly modifies the fields of the `Company` object.

Class Name: ReceiptDataModifierXML from ReceiptDataModifierXML.java	
Responsibilities	**Collaborations**

- Data Modification: It implements the abstract methods of `ReceiptDataModifier` to modify receipt and company data in a way that is suitable for XML files. This could involve creating XML elements and attributes, setting their values, etc. **- File Appending:** It inherits the method for appending data to a file from `ReceiptDataModifier`. This method uses the implemented abstract methods to modify the data before it is written to the file. **- XML Document Writing:** It provides a method for writing an XML document to a file. This method uses the `Transformer` class to convert the XML document to a stream of bytes that can be written to the file. **- XML Element Creation:** It provides a method for creating an XML element with text content. This method uses the `Document` class to create the	**- Receipt Class:** The `ReceiptDataModifierXML` class uses a `Receipt` object to store and modify receipt data. It directly modifies the fields of the `Receipt` object. **- Company Class**: The `ReceiptDataModifierXML` class uses a `Company` object to store and modify company data. It directly modifies the fields of the `Company` object.

element and the text node, and adds them to the parent element.	

Class Name: ReceiptDataModifierHTML from ReceiptDataModifier.java	
Responsibilities	**Collaborations**
The `ReceiptDataModifierHTML` class in Java, based on the provided code snippet, appears to be a concrete subclass of `ReceiptDataModifier` that modifies receipt data for HTML files. It has a collaboration with the `Receipt` and `Company` classes. Here are the responsibilities of the `ReceiptDataModifierHTML` class: - **Data Modification**: It implements the abstract methods of `ReceiptDataModifier` to modify receipt and company data in a way that is suitable for HTML files. This could involve creating HTML elements and setting their values, etc. - **File Appending**: It inherits the method for appending data to a file from `ReceiptDataModifier`. This method uses the implemented abstract methods to modify the data before it is written to the file. - **HTML Document Writing**: It provides a method for writing an HTML document to a file. This method uses the `FileWriter` class to write the HTML document to the file. - **HTML Document Modification**: It provides a method for deleting the last two lines of an HTML file. This could be used to remove the closing tags of an HTML document before appending more data.	**- Receipt Class**: The `ReceiptDataModifierHTML` class uses a `Receipt` object to store and modify receipt data. It directly modifies the fields of the `Receipt` object. **- Company Class:** The `ReceiptDataModifierHTML` class uses a `Company` object to store and modify company data. It directly modifies the fields of the `Company` object.

Class Name: ReportGenerator from ReportGenerator.java	
Responsibilities	**Collaborations**
- Data Gathering: It gathers the necessary data for the report from various sources, which could include databases, files, or other classes in the system. **- Data Formatting:** It formats the gathered data into a report. This could involve organizing the data into sections, adding headers and footers, etc. **- Report Output**: It outputs the generated report. This could involve writing the report to a file, sending it over a network, or simply returning it to another part of the system.	**- Data Source Classes:** The `ReportGenerator` class likely interacts with other classes in the system that provide the data for the report. These could be classes that represent entities in the system, classes that manage databases, etc. **- File or Network Classes**: If the `ReportGenerator` class writes the report to a file or sends it over a network, it would interact with classes that provide these functionalities.

Class Name: TXTReportGenerator from TXTReportGenerator.java	
Responsibilities	**Collaborations**
- File Selection: It provides a method for the user to select a file where the report will be saved. This method uses a `JFileChooser` to display a file selection dialog. **- File Writing:** It provides a method for writing the sales data to the selected file. This method uses a `BufferedWriter` to write the data to the file. **- Data Formatting**: It formats the sales data into a report. This involves calling methods on a `Salesman` object to get the data, and writing it to the file in a user-friendly format.	**- JFileChooser Class:** The `TXTReportGenerator` class uses a `JFileChooser` object to display a file selection dialog. It sets the file selection mode and file filter of the `JFileChooser`, and gets the selected file. **- FileNameExtensionFilter Class:** The `TXTReportGenerator` class uses a `FileNameExtensionFilter` object to filter the files that can be selected in the `JFileChooser`. It sets the `FileNameExtensionFilter` to only allow TXT files.

	- File Class: The `TXTReportGenerator` class uses a `File` object to represent the selected file. It gets the absolute path of the `File`, and checks and modifies the file extension. **- FileWriter and BufferedWriter Classes:** The `TXTReportGenerator` class uses a `FileWriter` and `BufferedWriter` to write the sales data to the selected file. **- JOptionPane Class**: The `TXTReportGenerator` class uses a `JOptionPane` to display a message dialog when an exception is caught**.** **- Salesman Class**: The `TXTReportGenerator` class uses a `Salesman` object to get the sales data. It calls methods on the `Salesman` to get the name, AFM, total sales, kind sales, and commission.

Class Name: XMLReportGenerator from XMLReportGenerator.java	
Responsibilities	**Collaborations**
- XML Document Creation: It creates a new XML document using a `DocumentBuilder`. This involves creating a `DocumentBuilderFactory`, creating a `DocumentBuilder` from the factory, and then creating a `Document` from the builder. **- Data Processing**: It processes the sales data and adds it to the XML document. This involves creating XML elements, setting their text content, and appending them to the document. **- XML Document Writing**: It writes the XML document to a file. This involves creating a `Transformer` from a `TransformerFactory`, setting the output properties of the `Transformer`, creating a `DOMSource` from the document, creating a	**- DocumentBuilderFactory, DocumentBuilder, and Document Classes:** The `XMLReportGenerator` class uses these classes to create a new XML document. It creates a `DocumentBuilderFactory`, creates a `DocumentBuilder` from the factory, and then creates a `Document` from the builder. **- Element Class:** The `XMLReportGenerator` class uses the `Element` class to create XML elements and append them to the document.

`StreamResult` from the file, and then transforming the `DOMSource` to the `StreamResult`.	**- TransformerFactory, Transformer, DOMSource, and StreamResult Classes**: The `XMLReportGenerator` class uses these classes to write the XML document to a file. It creates a `Transformer` from a `TransformerFactory`, sets the output properties of the `Transformer`, creates a `DOMSource` from the document, creates a `StreamResult` from the file, and then transforms the `DOMSource` to the `StreamResult`. **- File Class:** The `XMLReportGenerator` class uses a `File` object to represent the file where the XML document will be written. **- Salesman Class:** The `XMLReportGenerator` class uses a `Salesman` object to get the sales data. It calls methods on the `Salesman` to get the name, AFM, total sales, kind sales, and commission.

Class Name: HTMLReportGenerator from HTMLReportGenerator.java	
Responsibilities	**Collaborations**
- File Selection: It provides a method for the user to select a file where the report will be saved. This method uses a `JFileChooser` to display a file selection dialog. **- File Writing**: It provides a method for writing the sales data to the selected file. This method uses a `BufferedWriter` to write the data to the file. **- Data Formatting:** It formats the sales data into an HTML report. This involves calling methods on a `Salesman` object to get the data, and writing it to the file in a user-friendly format.	**- JFileChooser Class:** The `HTMLReportGenerator` class uses a `JFileChooser` object to display a file selection dialog. It gets the selected file from the `JFileChooser`. **- File Class:** The `HTMLReportGenerator` class uses a `File` object to represent the selected file. It gets the absolute path of the `File`, and checks and modifies the file extension. **- FileWriter and BufferedWriter Classes**: The `HTMLReportGenerator class uses a `FileWriter` and `BufferedWriter` to write the sales data to the selected file.

	- JOptionPane Class: The `HTMLReportGenerator` class uses a `JOptionPane` to display a message dialog when an exception is caught. **- Salesman Class**: The `HTMLReportGenerator` class uses a `Salesman` object to get the sales data. It calls methods on the `Salesman` to get the name, AFM, total sales, kind sales, and commission.