< SALES MANAGEMENT SYSTEM>

OVERALL REPORT

VERSION <2.0>

VASILEIOS IOANNIS BOUZAMPALIDIS AM:4744

IOANNIS FOURNARIS AM:4828

TABLE OF CONTENTS

INTRODUCTION

Project Objective:

The primary objective of this project is to refactor
an existing code, by employing established refactoring methods and algorithms.
The aim is to enhance the functionality and efficiency of the existing codebase,
thereby ensuring improved performance and operational effectiveness.

REFACTORED DESIGN

USE CASES

Use case ID	insertFromTXT
Actors	User
Pre conditions	• The user initiates an action to insert data from a TXT file.
Main flow of events	1. The use case starts when the user triggers the action to insert data from a TXT file. 2. The system opens a file chooser dialog (JFileChooser) to allow the user to select a TXT file. 3. The user selects a TXT file and confirms the selection. 4. The system retrieves the selected TXT file and creates a TXTInput object (inputFileTXT) to read the file. 5. The system reads the contents of the TXT file using the readFile() method of the TXTInput object. 6. The system retrieves the agent information from the TXT file using the getAgent() method of the TXTInput object. 7. The system sets the file type of the agent to "TXT" using the setFileType("TXT") method.

	8. The system sets the file to append for the agent's file appender using the setFileToAppend() method. 9. The system checks if the agent already exists in the allAgents list by comparing names. 9.1. If the agent already exists, the system sets agentDuplicate to true. 9.2. If the agent does not exist, the system proceeds to the next step. 10. If agentDuplicate is true: 10.1. The system displays a message dialog informing the user that the profile is already loaded. 11. If agentDuplicate is false: 11.1. The system adds the agent to the allAgents list. 11.2. The system adds the agent's name to the list model (listModel). 11.3. The system updates the displayed agents list (agentsList) with the updated list model. 11.4. The system sets fileTypeFlag to "TXT."
Alternative flow 1	If the user cancels the file selection or no file is selected: 1. The system displays a message dialog informing the user that a TXT file was not selected. 2. The use case ends.
Alternative flow 2	If a NullPointerException is caught (e.g., the user closes the file chooser without making a selection): 1. The system displays a message dialog informing the user that a TXT file was not selected. 2. The use case ends.
Post conditions	1. The system has successfully inserted data from the selected TXT file into the

	application. 2.The agent information is added to the allAgents list. 3.The agent's name is added to the displayed list of agents (agentsList). 4.If the agent already exists, the system informs the user, and no duplicate entries are created. 5.If the user cancels the file selection, the system provides appropriate feedback. 6.If there is an issue with the TXT file format, the system notifies the user.

Use case ID	InsertFromXML
Actors	User
Pre conditions	• The user initiates an action to insert data from a XML file.
Main flow of events	1. The use case starts when the user triggers the action to insert data from an XML file. 2. The system opens a file chooser dialog (JFileChooser) to allow the user to select an XML file. 3. The user selects an XML file and confirms the selection. 4. The system retrieves the selected XML file and creates an XMLInput object (inputFileXML) to read the file. 5. The system reads the contents of the XML file using the readFile() method of the XMLInput object. 6. The system retrieves the agent information from the XML file using

	the getAgent() method of the XMLInput object. 7. The system sets the file type of the agent to "XML" using the setFileType("XML") method. 8. The system sets the file to append for the agent's file appender using the setFileToAppend() method. 9. The system checks if the agent already exists in the allAgents list by comparing names. 9.1. If the agent already exists, the system sets agentDuplicate to true. 9.2. If the agent does not exist, the system proceeds to the next step. 10. If agentDuplicate is true: 10.1. The system displays a message dialog informing the user that the profile Is already loaded. 11. If agentDuplicate is false: 11.1. The system adds the agent to the allAgents list. 11.2. The system adds the agent's name to the list model (listModel). 11.3. The system updates the displayed agents list (agentsList) with the updated list model. 11.4. The system sets fileTypeFlag to "XML."
Alternative flow 1	If the user cancels the file selection or no file is selected: 1. The system displays a message dialog informing the user that an XML file was not selected. 2. The use case ends.
Alternative flow 2	If an IllegalArgumentException is caught (e.g., illegal format in the XML file): 1. The system displays a message dialog informing the user that the XML file cannot be loaded due to an illegal format. 2. The use case ends.

Post conditions	1. The system has successfully inserted data from the selected XML file into the application. 2. The agent information is added to the allAgents list. 3. The agent's name is added to the displayed list of agents (agentsList). 4. If the agent already exists, the system informs the user, and no duplicate entries are created. 5. If the user cancels the file selection, the system provides appropriate feedback. 6. If there is an issue with the XML file format, the system notifies the user.

Use case ID	InsertFromHTML
Actors	User
Pre conditions	• The user initiates an action to insert data from an HTML file.
Main flow of events	1. The use case starts when the user triggers the action to insert data from an HTML file. 2. The system opens a file chooser dialog (JFileChooser) to allow the user to select an HTML file. 3. The user selects an HTML file and confirms the selection. 4. The system retrieves the selected HTML file and creates an

	HTMLInputReader object (inputFileHTML) to read the file. 5. The system reads the contents of the HTML file using the readFile() method of the HTMLInputReader object. 6. The system retrieves the agent information from the HTML file using the getAgent() method of the HTMLInputReader object. 7. The system sets the file type of the agent to "HTML" using the setFileType("HTML") method. 8. The system sets the file to append for the agent's file appender using the setFileToAppend() method. 9. The system checks if the agent already exists in the allAgents list by comparing names. 9.1. If the agent already exists, the system sets agentDuplicate to true. 9.2. If the agent does not exist, the system proceeds to the next step. 10. If agentDuplicate is true: 10.1. The system displays a message dialog informing the user that the profile is already loaded. 11. If agentDuplicate is false: 11.1. The system adds the agent to the allAgents list. 11.2. The system adds the agent's name to the list model (listModel). 11.3. The system updates the displayed agents list (agentsList) with the updated list model. 11.4. The system sets fileTypeFlag to "HTML."
Alternative flow 1	If the user cancels the file selection or no file is selected: 1. The system displays a message dialog informing the user that an HTML file was not selected. 2. The use case ends.
Alternative flow 2	If an IllegalArgumentException is caught (e.g., illegal format in the HTML file): 1. The system displays a message dialog informing the user that the HTML

	file cannot be loaded due to an illegal format. 2. The use case ends.
Post conditions	1. The system has successfully inserted data from the selected HTML file into the application. 2. The agent information is added to the allAgents list. 3. The agent's name is added to the displayed list of agents (agentsList). 4. If the agent already exists, the system informs the user, and no duplicate entries are created. 5. If the user cancels the file selection, the system provides appropriate feedback. 6. If there is an issue with the HTML file format, the system notifies the user.

Use case ID	selectSalesManager
Actors	User
Pre conditions	• The UI includes a list of agents (agentsList) and responds to mouse events.
Main flow of events	1. The use case starts when the user clicks on an sales manager in the list (agentsList). 2. The system triggers the selectAgent method in response to the mouse event. 3. The system checks if an agent is selected by verifying that the selected index in the list is greater than or equal to 0.

	3.1. If a sales manager is selected: 3.1.1 The system retrieves the name of the selected agent from the agentsList. 3.1.2 The system iterates through the allAgents list to find the corresponding agent object based on the selected name. 3.1.3 The system sets the selectedAgent variable to the found agent. 3.1.4 The loop breaks once the sales manager is found. 3.2. If no sales manager is selected, the method ends without any further actions.
Alternative flow 1	If the agentsList is empty or the user clicks on an area outside the list: 1. The system does not find a selected index (index < 0). 2. The method ends without further actions.
Alternative flow 2	-
Post conditions	1. If a sales manager is selected: 1.1 The selectedAgent variable is set to the agent object corresponding to the selected name. 2. The application is now aware of the selected agent for further processing or display. 3. If no sales manager is selected or the user clicks outside the list: 3.1 The method ends without any further actions.

Use case ID	okButtonPressed (receipt selection dialog)
Actors	User
Pre conditions	• The UI includes checkboxes and radio buttons for user input. • The selectedAgent variable is set to a valid agent object.
Main flow of events	1. The use case starts when the user clicks the "OK" button. 2. The system triggers the okButtonPressed method in response to the button click. 3. The system checks the state of various checkboxes and radio buttons to determine which calculations to perform. 3.1. The system checks the state of totalSalesCheckBox. 3.1.1 If selected, the system calculates the total sales value using the calculateSalesValue method and updates the totalSales variable. 3.2. The system checks the state of totalItemsCheckBox. 3.2.1 If selected, the system calculates the total items value using the calculateItemsValue method and updates the totalItems variable. 3.3. The system checks the state of radio buttons (shirtRadio, skirtRadio, coatRadio, trousersRadio) for specific clothing kinds. 3.3.1 If selected, the system calculates the sales value for the corresponding clothing kind using the calculateSalesValue method and updates the respective variables (shirtSales, skirtSales, coatsSales, trousersSales). 3.4. The system checks the state of commissionCheckBox. 3.4.1 If selected, the system calculates the commission value using the

	calculateSalesValue method and updates the commission variable. 4. The system creates a ResultWindow object (rs) with the calculated values, the selectedAgent, and other necessary parameters. 5. The system sets the visibility of the ResultWindow to true and hides the current window. 6. The use case ends.
Alternative flow 1	If the selectedAgent variable is not set to a valid agent object: The system does not have valid data for calculations. The method ends without further actions.
Alternative flow 2	-
Post conditions	1. The system has calculated the requested values based on user input. 2. A ResultWindow is displayed with the calculated values and information about the selected agent. 3. The current window is set to not visible.

Use case ID	outputTXTButtonPressed (report result window)
Actors	User
Pre conditions	• The UI includes a button that triggers the action to save a TXT file. • The selectedAgent variable is set to a valid agent object.
Main flow of events	1. The use case starts when the user clicks the button for saving a TXT file. 2. The system triggers the outputTXTButtonPressed method in response to the button click. 3. The system opens a file chooser dialog (JFileChooser) for the user to select the location and name of the TXT file to be saved. 4. The user selects a location and provides a name for the TXT file. 5. The system checks if the user has approved the file selection (userSelection == JFileChooser.APPROVE_OPTION). 5.1. If approved: 5.1.1 The system retrieves the selected file using fileChooser.getSelectedFile(). 5.1.2 The system creates an instance of the TXTReport class (makeTXTFile) with the selected agent and file. 5.1.3 The system invokes the saveFile method of makeTXTFile to save the TXT report to the selected file. 5.1.4 The system displays a message dialog informing the user that the TXT file was saved successfully. 5.2. If not approved: 5.2.1 The method ends without further actions.
Alternative flow 1	If the selectedAgent variable is not set to a valid agent object: 1. The system does not have valid data to create a TXT report. 2. The method ends without further actions.

Alternative flow 2	-
Post conditions	1. If the user approves the file selection, a TXT report is created and saved to the specified location. 2. The system displays a message dialog indicating whether the TXT file was saved successfully. 3. If the user does not approve the file selection, the method ends without further actions.

Use case ID	outputXMLButtonPressed (report result window)
Actors	User
Pre conditions	• The UI includes a button that triggers the action to save an XML file. • The selectedAgent variable is set to a valid agent object.
Main flow of events	1. The use case starts when the user clicks the button for saving an XML file. 2. The system triggers the outputXMLButtonPressed method in response to the button click. 3. The system opens a file chooser dialog (JFileChooser) for the user to select the location and name of the XML file to be saved. 4. The user selects a location and provides a name for the XML file. 5. The system checks if the user has approved the file selection

	(userSelection == JFileChooser.APPROVE_OPTION). 5.1. If approved: 5.1.1 The system retrieves the selected file using fileChooser.getSelectedFile(). 5.1.2 The system creates an instance of the XMLReport class (makeXMLFile) with the selected agent and file. 5.1.3 The system invokes the saveFile method of makeXMLFile to save the XML report to the selected file. 5.1.4 The system displays a message dialog informing the user that the XML file was saved successfully. 5.2. If not approved: 5.2.1 The method ends without further actions.
Alternative flow 1	If the selectedAgent variable is not set to a valid agent object: 1. The system does not have valid data to create an XML report. 2. The method ends without further actions.
Alternative flow 2	-
Post conditions	1. If the user approves the file selection, an XML report is created and saved to the specified location. 2. The system displays a message dialog indicating whether the XML file was saved successfully. 3. If the user does not approve the file selection, the method ends without further actions.

Use case ID	outputHTMLButtonPressed (report result window)
Actors	User
Pre conditions	• The UI includes a button that triggers the action to save an HTML file. • The selectedAgent variable is set to a valid agent object.
Main flow of events	1. The use case starts when the user clicks the button for saving an HTML file. 2. The system triggers the outputHTMLButtonPressed method in response to the button click. 3. The system opens a file chooser dialog (JFileChooser) for the user to select the location and name of the HTML file to be saved. 4. The user selects a location and provides a name for the HTML file. 5. The system checks if the user has approved the file selection (userSelection == JFileChooser.APPROVE_OPTION). 5.1. If approved: 5.1.1 The system retrieves the selected file using fileChooser.getSelectedFile(). 5.1.2 The system creates an instance of the HTMLReport class (makeHTMLFile) with the selected agent and file. 5.1.3 The system invokes the saveFile method of makeHTMLFile to save the HTML report to the selected file. 5.1.4 The system displays a message dialog informing the user that the HTML file was saved successfully. 5.2. If not approved:

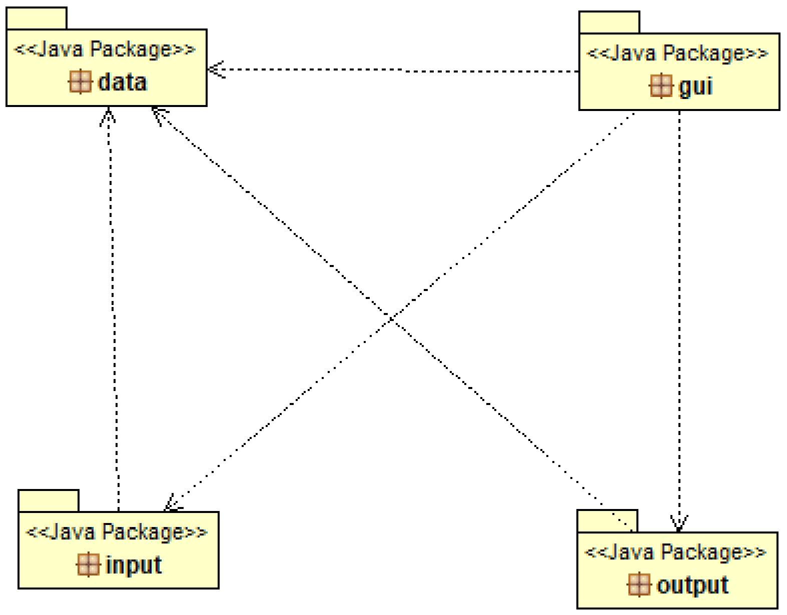
	5.2.1 The method ends without further actions.
Alternative flow 1	If the selectedAgent variable is not set to a valid agent object: 1. The system does not have valid data to create an HTML report. 2. The method ends without further actions.
Alternative flow 2	-
Post conditions	1. If the user approves the file selection, an HTML report is created and saved to the specified location. 2. The system displays a message dialog indicating whether the HTML file was saved successfully. 3. If the user does not approve the file selection, the method ends without further actions.

Use case ID	addReceipt (receipt selection dialog)
Actors	User
Pre conditions	• The UI includes input fields for receipt information. • The selectedAgent variable is set to a valid agent object.
Main flow of events	1. The use case starts when the user triggers the action to add a receipt. 2. The system triggers the addReceipt method in response to the user action. 3. The system creates a new Receipt object (receipt). 4. The system checks the value of the kindTextField to determine the type of clothing for the receipt. 4.1. If the text in kindTextField is "Shirts," set the kind of the receipt to "Shirts." 4.2. If the text in kindTextField is "Skirts," set the kind of the receipt to "Skirts." 4.3. If the text in kindTextField is "Trousers," set the kind of the receipt to "Trousers." 4.4. If the text in kindTextField is "Coats," set the kind of the receipt to "Coats." 5. The system tries to set the following attributes of the receipt object: 5.1. Receipt ID (receiptIDTextField) 5.2. Date (dateTextField) 5.3. Sales amount (salesTextField) 5.4. Number of items (itemsTextField) 5.5. Company name (companyTextField) 5.6. Company country (countryTextField) 5.7. Company city (cityTextField) 5.8. Company street (streetTextField) 5.9. Street number (numberTextField)

	6. The system adds the receipt to the list of receipts of the selectedAgent. 7. The system increments the count of receipts (numOfReceipts) and updates the corresponding UI element (numOfReceiptsTextField). 8. The system displays a message dialog indicating that the receipt was added successfully.
Alternative flow 1	If any of the numeric input fields (e.g., receiptIDTextField, salesTextField, itemsTextField, numberTextField) does not contain a valid number: 1. The system catches a NumberFormatException. 2. The system displays a message dialog indicating that the receipt could not be added.
Alternative flow 2	-
Post conditions	1. A new receipt is added to the list of receipts of the selectedAgent. 2. The count of receipts (numOfReceipts) is incremented. 3. The UI is updated to reflect the new count of receipts. 4. A message dialog is displayed indicating whether the receipt was added successfully.

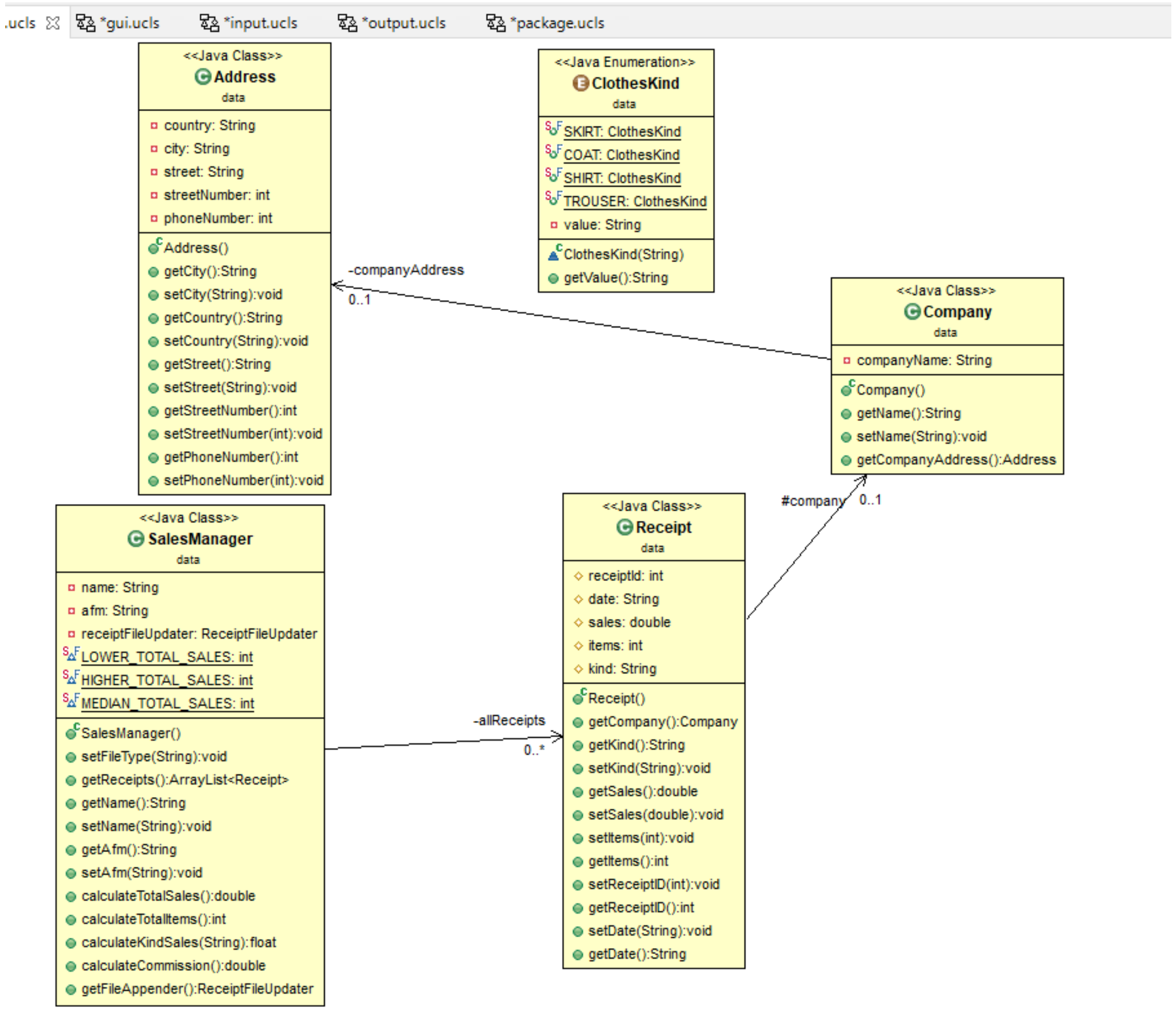
ARCHITECTURE

PACKAGE UML DIAGRAM:

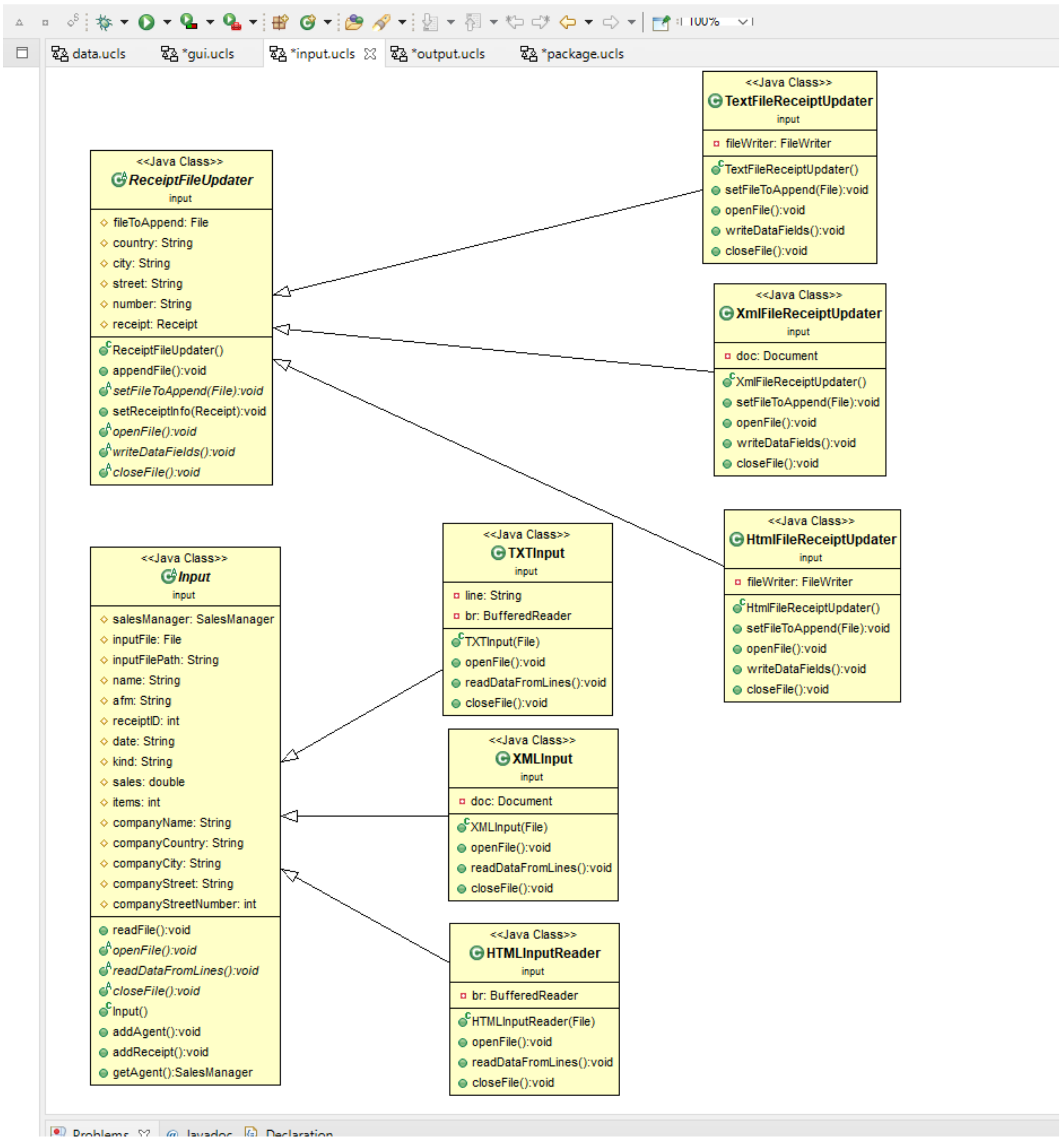


DETAILED DESIGN

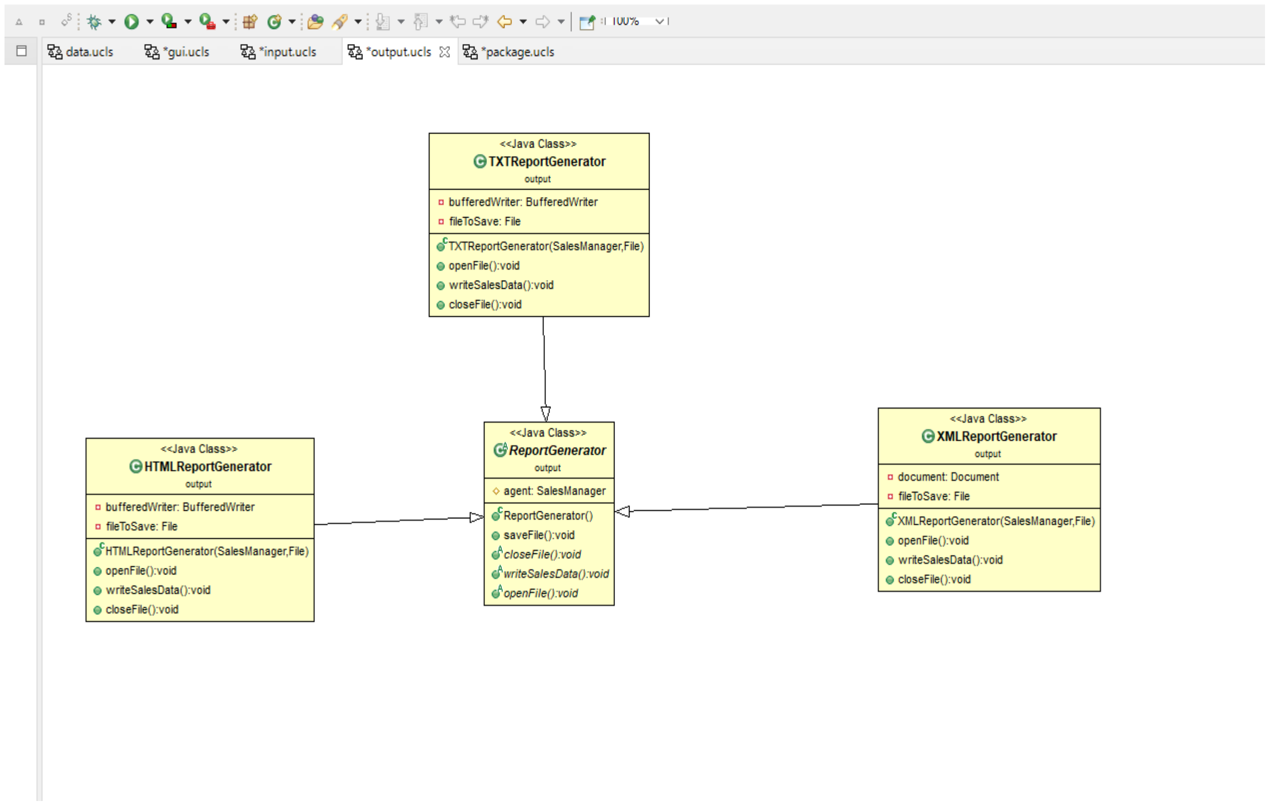
DATA PACKAGE UML CLASS DIAGRAM:



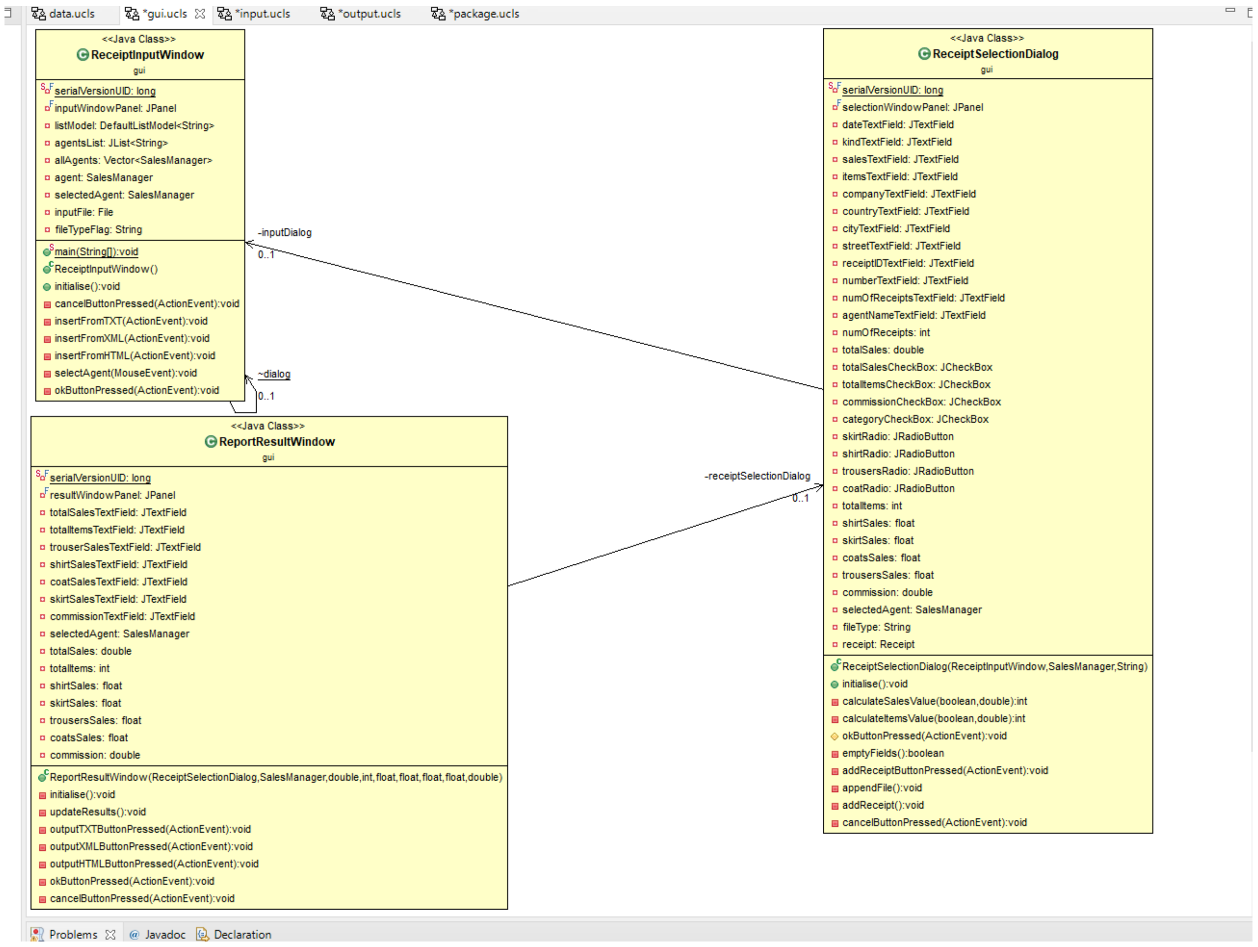
INPUT PACKAGE UML CLASS DIAGRAM:



OUTPUT PACKAGE UML CLASS DIAGRAM:



GUI PACKAGE UML CLASS DIAGRAM:


data.ucls
*gui.ucls
*input.ucls
*output.ucls
*package.ucls
<<Java Class>>
ReceiptInputWindow
gui
serialVersionUID: long
inputWindowPanel: JPanel
listModel: DefaultListModel<String>
agentsList: JList<String>
allAgents: Vector<SalesManager>
agent: SalesManager
selectedAgent: SalesManager
inputFile: File
fileTypeFlag: String
main(String[]):void
ReceiptInputWindow()
initialise():void
cancelButtonPressed(ActionEvent):void
insertFromTXT(ActionEvent):void
insertFromXML(ActionEvent):void
insertFromHTML(ActionEvent):void
selectAgent(MouseEvent):void
okButtonPressed(ActionEvent):void
-inputDialog
0..1
~dialog
0..1
<<Java Class>>
ReportResultWindow
gui
serialVersionUID: long
resultWindowPanel: JPanel
totalSalesTextField: JTextField
totalItemsTextField: JTextField
trouserSalesTextField: JTextField
shirtSalesTextField: JTextField
coatSalesTextField: JTextField
skirtSalesTextField: JTextField
commissionTextField: JTextField
selectedAgent: SalesManager
totalSales: double
totalItems: int
shirtSales: float
skirtSales: float
trousersSales: float
coatsSales: float
commission: double
ReportResultWindow(ReceiptSelectionDialog,SalesManager,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
-receiptSelectionDialog
0..1
<<Java Class>>
ReceiptSelectionDialog
gui
serialVersionUID: long
selectionWindowPanel: 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
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
selectedAgent: SalesManager
fileType: String
receipt: Receipt
ReceiptSelectionDialog(ReceiptInputWindow,SalesManager,String)
initialise():void
calculateSalesValue(boolean,double):int
calculateItemsValue(boolean,double):int
okButtonPressed(ActionEvent):void
emptyFields():boolean
addReceiptButtonPressed(ActionEvent):void
appendFile():void
addReceipt():void
cancelButtonPressed(ActionEvent):void
Problems
Javadoc
Declaration

REFACTORING THE OLD CODE:

data package:

1. Regarding the issue of **Lazy Classes** (Coat, Shirt, Skirt, Trouser), a resolution has been implemented by eliminating these individual classes. Instead, an ENUM-type class has been created, including the distinct categories of clothing, namely Coat, Shirt, Skirt, and Trouser.

2. Regarding the issue in **Agent class**: Initially the name of the class does not reflect the role of the class, resulting in the renaming of the class to **SalesManager**. Subsequently, for resolving code Duplication within the method responsible for calculating the kind sales, **Parameterized Method** is used, retaining the core functionality and eliminating redundant methods. In addressing the issue of code Duplication within the commission calculation, **Symbolic Constants** have been introduced. This approach involves assigning appropriate values to these constants, thereby effectively fixing the Duplication. Furthermore, **Java ArrayList** replaced the deprecated Java Vector.

input package:

3. Regarding the issue in **FileAppender, FileAppenderTXT, FileAppenderXML classes**: Initially the name of the classes does not reflect the role of the classes, resulting in the renaming of the classes to **ReceiptFileUpdater, TextFileReceiptUpdater, XmlFileReceiptUpdater**. Since the classes perform IO operations, they are relocated in the **Input package.** Furthermore, the ReceiptFileUpdater class has been addressed for the Primitive Obsession problem. This issue has been rectified by introducing a **Receipt object** to encapsulate primitive data. The unnecessary fields are removed and the corresponding setter methods have been replaced with a **single setter method** for the Receipt object. Additionally, the TextFileReceiptUpdater, XmlFileReceiptUpdater classes had a shared algorithmic structure within their appendFile() method. To fix this redundancy a **Template Method** has been implemented in the ReceiptFileUpdater class. This involves creating a method that calls three abstract methods. These abstract methods are then implemented accordingly in TextFileReceiptUpdater, XmlFileReceiptUpdater classes, thereby eliminating duplication in the algorithmic steps.

4. Regarding the issue in **Input, TXTInput, XMLInput classes:** These classes previously shared a common algorithmic structure for the parsing algorithms, after the refactoring of the TXTInput class to a simpler algorithm. To resolve this issue, a **Template Method** has been introduced in the Input class, calling three abstract methods. These abstract methods are then implemented accordingly in the TXTInput and XMLInput classes, effectively eliminating duplication in the algorithmic steps. Moreover, the addReceipt() method in the Input has been rectified subsequent to the removal of Lazy Classes and the usage of the setKind method of the Receipt class.

output package:

5. Regarding the issue in **Report, TXTReport, XMLReport classes**: Initially the name of the classes does not reflect the role of the classes, resulting in the renaming of the classes to **ReportGenerator, TXTReportGenerator, XMLReportGenerator**. These classes previously shared a common algorithmic structure for the saving of a report file of receipts. To resolve this issue, a **Template Method** has been introduced in the ReportGenerator class, calling three abstract methods. These abstract methods are then implemented accordingly in the TXTReportGenerator and XMLReportGenerator classes, effectively eliminating duplication in the algorithmic steps.

gui package:

6. Regarding the issue of saving a user selected file, in the **ReportResultWindow class**(observe the renaming of the class in 7), modifications have been made to the methods **outputTXTButtonPressed** and **outputXMLButtonPressed**. These adjustments now enable the user to specify both the path and the filename under which they intend to save a report.

7. Regarding the issue in **InputWindow, SelectionWindow, ResultWindow classes**: Initially the name of the classes does not reflect the role of the classes, resulting in the renaming of the classes to **ReceiptInputWindow, ReceiptSelectionDialog, ReportResultWindow**.

8. Regarding the issue in **ReceiptSelectionDialog class:** The **addReceiptButtonPressed** method previously had a very complex conditional statement. To address this complexity, a resolution has been implemented by introducing the **emptyFields** method. This newly introduced method assesses whether the fields are empty or not and is utilized within the addReceiptButtonPressed method. This modification serves to simplify the conditional statement and enhance the overall clarity and maintainability of the code. Furthermore, this modification **removed the chain calls of the appendFile** method and the refactoring of the **ReceiptFileUpdater** in the input package significantly reduced the size of the method in the **ReceiptSelectionDialog class** . Moreover, in addressing the complexity in the **okButtonPressed** method, the multiple if-else statements are have been streamlined by incorporating the **calculateSalesValue** and the **calculateItemsValue** methods. The first returns the selected sales value, or -1 if none is chosen and the second, returns the selected items value or -1 if none are chosen.. Both methods are utilized within the okButtonPressed method, effectively eliminating the need for multiple if-else statements.

EXTENSION TASK:

9. For the facilitation of application **input** via an **HTML file**, several enhancements have been introduced. An **HTML file** has been generated, containing a sales manager and associated receipts. Within the input package, a **HtmlFileReceiptUpdater** class has been established, mirroring the functionalities of the **TextFileReceiptUpdater** and **XmlFileReceiptUpdater** classes but for the HTML format. Additionally, a **HTMLInputReader** class has been implemented within the same package, performing analogous tasks as the **TXTInput** and **XMLInput** classes but for HTML files. Furthermore, in the gui package, relevant methods have been implemented. An **HTML input button** has been incorporated, alongside the creation of the **insertFromHTML** method. This method mirrors the operations of **insertFromXML** and **insertFromTXT** but utilizes the **HTMLInputReader**.

10. To facilitate the **storage** of a sales representative **report** in an **HTML file**, a series of modifications have been introduced. Within the output package, an **HTMLReportGenerator** class has been created, emulating the functionalities of the **TXTReportGenerator** and **XMLReportGenerator** classes but specifically designed for **HTML files**. Additionally, in the gui package, pertinent methods have been implemented. An **HTML report button** has been incorporated, and an **outputHTMLButtonPressed** method has been introduced. This method mirrors the functionalities of the **outputXMLButtonPressed** and **outputTXTButtonPressed**, but for **HTML files**.

CLASSES RESPONSIBILITIES AND COLLABORATIONS (CRC CARDS)

Class Name: Address	
Responsibilities	**Collaborations**
1. Manage location information, including country, city, street, and street number. 2. Handle and store phone number information. 3. Provide methods to retrieve and update city details.	

4. Offer functionality for retrieving and updating country details. 5. Facilitate operations related to street information retrieval and modification. 6. Handle street number data, providing accessors and mutators. 7. Manage phone number data, offering methods for retrieval and modification.	

Class Name: ClothesKind	
Responsibilities	**Collaborations**
1. Classify different types of clothing materials, including Skirt, Coat, Shirt, and Trouser. 2. Assign a human-readable value to each clothing material for better representation. 3. Provide accessors to retrieve the value associated with each clothing material.	

Class Name: Company	
Responsibilities	**Collaborations**
1. Manage and store the name of the company. 2. Instantiate and manage the company's address using the Address class. 3. Provide accessors to retrieve the name of the company. 4. Offer a mutator to set and update the name of the company. 5. Provide accessors to retrieve the company's address.	• Depends on the Address class: Utilizes the Address class to represent and manage the company's address information.

Class Name: Receipt	
Responsibilities	**Collaborations**
1. Manage and store the unique identifier (ID) of the receipt. 2. Store and manage the date of the receipt. 3. Store and manage the total sales amount associated with the receipt. 4. Store and manage the number of items on the receipt. 5. Instantiate and manage the company associated with the receipt using the Company class. 6. Store and manage the kind of receipt (e.g., "No specific kind"). 7. Provide accessors to retrieve the associated company.	• Depends on the Company class: Utilizes the Company class to represent and manage the company associated with the receipt.

8. Provide accessors to retrieve the kind of receipt. 9. Offer a mutator to set and update the kind of receipt. 10. Provide accessors to retrieve the total sales amount. 11. Offer a mutator to set and update the total sales amount. 12. Provide accessors to retrieve the number of items on the receipt. 13. Offer a mutator to set and update the number of items on the receipt. 14. Provide accessors to retrieve the receipt ID. 15. Offer a mutator to set and update the receipt ID. 16. Provide accessors to retrieve the date of the receipt. 17. Offer a mutator to set and update the date of the receipt.	

Class Name: SalesManager	
Responsibilities	**Collaborations**
1. Manage and store the name of the sales manager. 2. Manage and store the AFM (Tax Identification Number) of the sales manager. 3. Manage a collection of receipts (allReceipts) using an ArrayList. 4. Instantiate the FileAppender based on the specified file type (TXT, XML, or HTML). 5. Provide accessors to retrieve the collection of receipts. 6. Provide accessors to retrieve the name of	• Depends on the Receipt class: Utilizes the Receipt class to represent and manage individual receipts within the allReceipts collection. • Depends on the FileAppender, FileAppenderTXT, FileAppenderXML, and FileAppenderHTML classes: Utilizes these classes to handle file appending based on the specified file type.

the sales manager. 7. Offer a mutator to set and update the name of the sales manager. 8. Provide accessors to retrieve the AFM of the sales manager. 9. Offer a mutator to set and update the AFM of the sales manager. 10. Calculate the total sales amount from all receipts. 11. Calculate the total number of items from all receipts. 12. Calculate the total sales of a specific kind of clothing. 13. Calculate the commission based on total sales, considering different commission rates for specified	

ranges. 14. Provide accessors to retrieve the file appender.	

Class Name: ReceiptInputWindow	
Responsibilities	**Collaborations**
1. Manage the graphical user interface for input operations. 2. Provide buttons for loading receipts from different file types (TXT, XML, HTML). 3. Handle the selection of a sales manager profile from the displayed list. 4. Instantiate and manage the SalesManager instances representing	• Depends on the SalesManager class: Utilizes instances of the SalesManager class to represent different sales managers. • Depends on the ReceiptFileUpdater, TextFileReceiptUpdater, XmlFileReceiptUpdater, and HtmlFileReceiptUpdater classes: Utilizes these classes for file appending based on the specified file type. • Depends on the ReceiptSelectionDialog class:

different agents. 5. Manage the file type flag (e.g., "TXT", "XML") based on the selected file type. 6. Display a list of loaded sales manager profiles. 7. Display a window allowing the user to select a loaded sales manager profile for further operations.	Displays the ReceiptSelectionDialog for further operations once a sales manager profile is selected. • Depends on HTMLInputReader, TXTInput, and XMLInput classes: Utilizes these classes to read data from HTML, TXT, and XML files, respectively.

Class Name: ReportResultWindow	
Responsibilities	**Collaborations**
1. Manage the graphical user interface for displaying results and saving reports. 2. Provide buttons for saving reports in different formats (TXT, XML, HTML). 3. Display the total sales, total items, and	• Depends on the SalesManager class: Utilizes the SalesManager instance for which the results are displayed. • Depends on the ReceiptSelectionDialog class:

sales for each clothing type for the selected sales manager. 4. Display the commission earned by the sales manager. 5. Allow the user to save reports in TXT, XML, and HTML formats. 6. Interact with instances of the TXTReportGenerator, XMLReportGenerator, and HTMLReportGenerator classes to create and save reports. 7. Handle button press events for saving reports and closing the application.	Allows the user to return to the ReceiptSelectionDialog for further operations. • Depends on the TXTReportGenerator, XMLReportGenerator, and HTMLReportGenerator classes: Utilizes these classes to generate and save reports in different formats.

Class Name: ReceiptSelectionDialog	
Responsibilities	**Collaborations**
1. Display a GUI window for selecting various options related to sales data. 2. Initialize and manage the components within the GUI,	• Collaborates with the ReceiptInputWindow class to switch between different GUI windows. • Collaborates with the

	including checkboxes, radio buttons, text fields, and buttons. 3. Handle user interactions, such as button clicks and checkbox selections. 4. Calculate and store total sales, total items, and sales for specific clothing categories based on user selections. 5. Provide a mechanism to add new receipts, entering details like receipt ID, date, kind, sales, items, company information, and address. 6. Update the displayed number of receipts. 7. Handle events such as OK button press, cancel button press, and add receipt button press. 8. Collaborate with the ReceiptInputWindow, SalesManager, and Receipt classes to manage and display	SalesManager class to access and manipulate sales-related data. • Collaborates with the Receipt class to create and store new receipts. • Collaborates with the ReportResultWindow class to display the results of selected options.

relevant information. 9. Use the ReportResultWindow class to display the results of selected options.	
Class Name: ReceiptFileUpdater	
Responsibilities	**Collaborations**
1. Serve as an abstract class for file appending operations related to receipt data. 2. Declare and define a template method appendFile() that orchestrates the file appending process by calling abstract methods. 3. Declare and define an abstract method setFileToAppend(File fileToAppend) for setting the file to which data will be appended. 4. Provide a method setReceiptInfo(Receipt receipt) to set the Receipt object to be appended, eliminating primitive obsession. 5. Declare abstract methods (openFile(), writeDataFields(), closeFile()) that must be implemented by concrete	• Collaborates with concrete subclasses that extend ReceiptFileUpdater: HtmlFileReceiptUpdater, XmlFileReceiptUpdater, TextFileReceiptUpdater to implement file-specific operations. • Collaborates with the Receipt class by using an instance of it to gather and append receipt data.

subclasses for specific file append operations.	

Class Name: HtmlFileReceiptUpdater	
Responsibilities	**Collaborations**
1. Implement the file appending operations specific to HTML format for receipt data. 2. Override the abstract method setFileToAppend(File fileToAppend) to set the HTML file to which data will be appended. 3. Override the abstract method openFile() to open the HTML file for writing and print debugging information. 4. Override the abstract method writeDataFields() to write receipt data in HTML format to the file. 5. Override the abstract method closeFile() to close the HTML file after appending data.	• Collaborates with the ReceiptFileUpdater abstract class to inherit the template method and abstract method definitions. • Collaborates with the Receipt class to gather data for appending in HTML format. • Collaborates with the Java I/O classes (File, FileWriter, IOException) to perform file operations.

Class Name: TextFileReceiptUpdater	
Responsibilities	**Collaborations**
1. Implement the file appending operations specific to plain text (TXT) format for receipt data. 2. Override the abstract method setFileToAppend(File fileToAppend) to set the TXT file to which data will be appended. 3. Override the abstract method openFile() to open the TXT file for writing and print debugging information. 4. Override the abstract method writeDataFields() to write receipt data in plain text format to the file. 5. Override the abstract method closeFile() to close the TXT file	• Collaborates with the ReceiptFileUpdater abstract class to inherit the template method and abstract method definitions. • Collaborates with the Receipt class to gather data for appending in plain text format. • Collaborates with the Java I/O classes (File, FileWriter, IOException) to perform file operations.

after appending data.	

Class Name: XmlFileReceiptUpdater	
Responsibilities	**Collaborations**
1. Implement the file appending operations specific to XML format for receipt data. 2. Override the abstract method setFileToAppend(File fileToAppend) to set the XML file to which data will be appended. 3. Override the abstract method openFile() to initialize the XML document object for appending data and handle exceptions appropriately. 4. Override the abstract method writeDataFields() to	• Collaborates with the ReceiptFileUpdater abstract class to inherit the template method and abstract method definitions. • Collaborates with the Receipt class to gather data for appending in XML format. • Collaborates with the Java XML-related classes (DocumentBuilder, DocumentBuilderFactory, Document, Element, Node, Transformer, TransformerFactory, DOMSource, StreamResult) to perform XML file operations.

create XML elements and append receipt data to the document. 5. Override the abstract method closeFile() to transform and write the XML document to the specified file.	

Class Name: HTMLInputReader	
Responsibilities	**Collaborations**
1. Implement the file reading operations specific to HTML format for receipt data. 2. Override the abstract method openFile() to initialize the BufferedReader for reading from the HTML file and handle exceptions appropriately. 3. Override the abstract method readDataFromLines() to read data from HTML lines and populate the corresponding fields of the Input class	• Collaborates with the Input class (or a related class) to populate fields with the data read from the HTML file. • Collaborates with the Java I/O classes (BufferedReader, File, FileReader, IOException) to perform HTML file reading operations.

(or a related class). 4. Override the abstract method closeFile() to close the BufferedReader after reading from the HTML file.	

Class Name: Input	
Responsibilities	**Collaborations**
1. Serve as a template class for different input file formats (e.g., HTML, XML, TXT). 2. Implement a template method readFile() that orchestrates the file reading process. 3. Declare abstract methods openFile(), readDataFromLines(), and closeFile() to be implemented by concrete subclasses. 4. Manage a SalesManager instance to store data read from input files. 5. Declare and manage fields for various receipt-related data,	• Collaborates with concrete subclasses (e.g., HTMLInputReader, XmlFileReceiptUpdater, etc.) to provide specific implementations for file reading and writing operations. • Interacts with the SalesManager class to store information about agents and receipts.

such as name, AFM, receipt ID, date, kind, sales, items, company name, company country, city, street, and street number. 6. Provide methods like addAgent() and addReceipt() to populate the SalesManager with agent and receipt data, respectively.	

Class Name: TXTInput	
Responsibilities	**Collaborations**
1. Extends the Input abstract class to provide a concrete implementation for TXT file input. 2. Implements the abstract methods openFile(), readDataFromLines(), and closeFile() to read data from TXT files and populate the SalesManager. 3. Uses a BufferedReader to efficiently read lines from the TXT file.	• Collaborates with the parent Input class, inheriting its structure and utilizing its methods for handling agent and receipt data.

4. Parses the lines to extract information about agents and receipts.	
Class Name: XMLInput	
Responsibilities	**Collaborations**
1. Extends the Input abstract class to provide a concrete implementation for XML file input. 2. Implements the abstract methods openFile(), readDataFromLines(), and closeFile() to read data from XML files and populate the SalesManager. 3. Uses the Java DOM (Document Object Model) API to parse and manipulate XML documents. 4. Handles exceptions and displays an error message using JOptionPane in case of issues during XML parsing.	• Collaborates with the parent Input class, inheriting its structure and utilizing its methods for handling agent and receipt data. • Collaborates with the Document, Element, and NodeList classes from the DOM API for XML parsing.

Class Name: HTMLReportGenerator	
Responsibilities	**Collaborations**
1. Extends the Report abstract class to provide a concrete implementation for HTML report generation. 2. Implements the abstract methods openFile(), writeSalesData(), and closeFile() to generate an HTML report and write it to a file. 3. Uses a BufferedWriter to efficiently write data to the HTML file. 4. Handles exceptions and displays an error message using JOptionPane in case of issues during file creation or writing.	• Collaborates with the parent ReportGenerator class, inheriting its structure and utilizing its methods for handling report generation. • Collaborates with the SalesManager class to obtain sales and commission data.

Class Name: ReportGenerator	
Responsibilities	**Collaborations**
1. Acts as a template for generating reports, providing a standard process through the saveFile() template method. 2. Declares abstract methods that represent steps in the report generation process but vary based on the type of report (HTML, XML, etc.). 3. Holds a reference to a SalesManager object, indicating that reports are generated based on the data from a sales manager.	• Collaborates with concrete subclasses that extend Report to provide specific implementations for different types of reports (HTMLReportGenerator, XMLReportGenerator, TXTReportGenerator). • Collaborates with the SalesManager class to obtain sales-related data for generating the report.

Class Name: TXTReportGenerator	
Responsibilities	**Collaborations**
1. Generates a sales report in plain text format (TXT) based on the provided SalesManager data. 2. Writes relevant sales information, including total sales, sales by clothing type, and commission, to a specified TXT file.	• Collaborates with the SalesManager class to obtain sales-related data. • Extends the abstract ReportGenerator class, providing specific implementations for generating TXT reports.

Class Name: XMLReportGenerator	
Responsibilities	**Collaborations**
1. Generates a sales report in XML format based on the provided SalesManager data. 2. Writes relevant sales information, including total sales, sales by clothing type, and commission, to a specified XML file.	• Collaborates with the SalesManager class to access sales-related data required for generating the report. • Extends the abstract ReportGenerator class, providing specific implementations for generating XML reports.

WARNING!!: FOR THE TESING ONLY:

For the input-output package tests change the test file path and name according to the place you saved them (or you want the reports to be saved).