Chapter 5: Object-Oriented Design

5.1 Introduction

During the design phase, the main task is to elevate the models into actual objects that can perform required task. The models referred here are the object-oriented analysis models which were explained in the previous chapter. Besides that, there is a shift in emphasis from the application domain to implementation. Classes identified earlier during the analysis stage provide a framework for the design phase.

Furthermore, the topics which will be covered in this chapter are the processes in O-O design phase, namely designing classes, designing the access layer, and designing the user interface.

5.2 Design Axioms and Corollaries

The Unified Approach of developing an Object-Oriented system has proposed two types of design axioms and six corollaries in designing classes of object (Texel and Williams, 1997). In order to develop a well-designed Object-Oriented Data Automation System for Microsoft Excel Files, it is important to take note and follow these axioms and corollaries.

The axioms and corollaries should be clearly defined before proceeding to the design of classes. By definition, an axiom is a fundamental truth that always observed to be valid and for which there is no counter example or exception. The design axioms suggested in Unified Approach are defined by Suh (Suh, 1990). Table 5.1 below explains two types of axioms that should be applied during O-O design phase.

Table 5.1 O-O design axioms

Axiom	Description	Design Rules
Axiom 1	Deals with relationships between system components (such as classes, requirements, software components)	The independence axiom. Maintain the independence of components
Axiom 2	Deals with the complexity of design	The information axiom. Minimize the information content of the design

From the two design axioms, many corollaries can be derived as a direct consequence of the axioms. These corollaries may be more useful in deciding a more specific design of classes. Corollaries can be applied to actual situations more easily than the original axioms. Corollaries are known as the design rules as well. Table 5.2 below explains six corollaries, which are derived from the two axioms presented in Table 5.1.

Table 5.2 O-O design corollaries

Corollary	**Name**	**Description**
Corollary 1	Uncoupled Design with Less Information Content	All classes have to be designed with strong objects (or software components) cohesiveness.
Corollary 2	Single Purpose	Each class must have a specific purpose. Do not combine different methods/behaviours of different objects into one class.
Corollary 3	Large Number of Simple Classes	It is good to have a large numbers of simpler classes.
Corollary 4	Strong Mapping	There must be a strong mapping links classes identified during O-O analysis phase and classes designed during design phase.
Corollary 5	Standardization	All classes designed in this phase have to be documented properly so that it can be reused in future enhancements.
Corollary 6	Design with inheritance	Superclass is built to store common behaviour (methods). There must be a logical sense between the superclass-subclass structures.

Basically, the six corollaries presented in Table 5.2 above are originated from the two design axioms that were presented in Table 5.1. Figure 5.1 shows the origins of the corollaries.

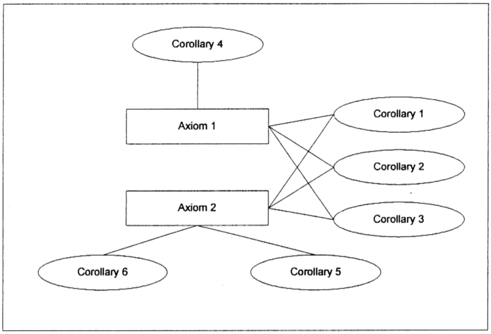


Figure 5.1 The origins of the corollaries

As a conclusion, the axioms and corollaries suggested by Suh have been used and practiced in order to produce all the object classes which will be explained in the following sections.

5.3 Designing Classes (attributes, methods, associations, structures, protocols)

At this stage, the process of designing classes is carried out by adding more new classes, attributes or methods. The main contribution of adding new classes, attributes or methods is to enhance the quality of the system implementation as well as the user interfaces.

In order to achieve the objective, it is important to apply the design axioms and corollaries in designing classes, their attributes, methods, associations, structure and protocols. Static UML class diagrams were developed and presented in Chapter 4 while executing the Object-Oriented analysis activities. Hence, the activities involved in this stage would be to refine and complete the static UML class diagrams.

There are three steps in this stage of refining the design of classes:

5.3.1 Refining Attributes

During the analysis stage, 13 objects have been identified. These objects have to be refined and some of them belong to the access and view layer classes. The static UML class diagrams presented in section 4.7 have included the information regarding attributes, which describes the behaviour of the classes.

Here, four classes will be redefined as the data type for each attribute in them are not clearly defined yet. The four classes are the ExcelSeparator class, the ExcelCombiner class, the ExcelSeparatorService class and the ExcelCombinerService class. Eventually, the rest of the classes which have been defined during the O-O analysis stage will be re-evaluated in the remaining sections of this chapter.

The representation of the attributes for each class will be composed by three types of symbols. A "+" sign represents the public visibility of the attributes, where they are accessible to all classes. A "#" sign represents the protected visibility of the attributes, *where they are accessible to subclasses and the operations of the class. The third type of* visibility is the private visibility, which is represented by a minus (-) sign. The private visibility allows the accessibility only to the operations of the class.

5.3.1.1 Refining Attributes for the ExcelSeparator Class

During Object-Oriented analysis, the following attributes have been identified for this class:

Table 5.3 Attributes for the ExcelSeparator Class Identified during O-O Analysis Stage

Attributes
Lcid
App
Sheet
Book
Range
Maxcolumn
noParam
Alpha
execelSource
startPoint
Endpoint
startCell1
endCell1

At this stage, more information has to be added and the attributes have to be redefined. The data types of the attributes have to be identified in order to enable implementation of this class.

Table 5.4 Redefined Attributes for the ExcelSeparator Class

Attribute	Data Type	Default Value
-lcid	Integer	0
-app	Application	Null
-sheet	_Worksheet	Null
-book	_Workbook	Null
-range	Range	Null
-maxcolumn	Integer	10
-noParam	Variant	Null
-alpha	String[ ]	An array of string with 26 members {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"}
-execelSource	String	Null
-startPoint	String	Null
-endPoint	String	Null
-startCell1	String	Null
-endCell1	String	Null

5.3.1.2 Refining Attributes for the ExcelCombiner Class

The attributes that have been identified earlier in the O-O analysis stage are presented in Table 5.5. These attributes were not defined clearly as the data types were not explained in details.

Table 5.5 Attributes for the ExcelCombiner Class Identified during O-O Analysis Stage

Attributes
Lcid
Spp
myApp
Sheet
mySheet
Book
myBook
Range
Type
Error
noParam
Alpha
Filel
file2
startCell1
endCell1
startCell2
endCell2

More detailed descriptions of the attributes that belong to the ExcelSeparator class are presented in Table 5.6 showed below. Some of the data types of the attributes are the classes defined in the package it.bigatti.execl8 and the package com.jacob.com.

Table 5.6 Refined Attributes for the ExcelCombiner Class

Attribute	Data Type	Default Value
-lcid	Integer	0
-app	Application	Null
-myApp	Application	Null
-sheet	_Worksheet	Null
-mySheet	_Worksheet	Null
-book	_Workbook	Null
-myBook	_Workbook	Null
-range	Range	Null
-type	String	Null
-error	String	Null
-noParam	Variant	Null
-alpha	String[ ]	An array of string with 26 members {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"}
-filel	String	Null
-file2	String	Null
-startCell1	String	Null
-endCell1	String	Null
-startCell2	String	Null
-endCell2	String	Null

5.3.1.3 Refining Attributes for the ExcelSeparatorService Class

Basically, the attributes that belong to this class have been defined clearly during the O-O analysis stage. However, in order to enable the association "invokes" between this class and the ExcelSeparator class, there must be an instance of the ExcelSeparator class declared in the ExcelSeparatorService class. Table 5.7 shows the refined attributes for this class.

Table 5.7 Refined Attributes for the ExcelSeparatorService Class

Attribute	Data Type	Default Value
-sourceFile	String	Null
-headers	String[ ]	Null
-tempFileName	String[ ]	Null
-count	Integer	0
-destinationPath	String	Null
-error	String	Null
-objSeparator	ExcelSepartor	Null
-startCells	String[ ]	Null
-endCells	String[ ]	Null

5.3.1.4 Refining Attributes for the ExcelCombinerService Class

At this stage, the ExcelCombinerService class is defined in order to enable the association "invokes" with the ExcelCombiner class. Hence, an instance of the ExcelCombiner class is added as one attribute to this class. A list of attributes for this class is presented in Table 5.8.

Table 5.8 Refined Attributes for the ExcelCombinerService Class

Attribute	Data Type	Default Value
-listOfFiles	String[ ]	Null
-errorMsg	String	Null
-combineType	String	Null
-objCombiner	ExcelCombiner	Null
-startCells	String[ ]	Null
-endCells	String[ ]	Null

5.3.2 Designing methods and protocols

The UML activity diagrams designed in Chapter 4 have reflected the methods that should be included in all the classes for this project. All the activities have to be converted to a programming language (in this context, Java) for implementing the OODA system for Microsoft Excel Files. A class can provide several types of methods (Texel and Williams, 1997). The methods are shown in Table 5.9.

Table 5.9 Types of methods for a class

No	Type of method	Description
1	Constructor	Method that creates instances (object) of the class.
2	Destructor	Method that destroys instances.
3	Conversion	Method that converts a value from one unit of measure to another.
4	Copy	Method that copies the contents of one instance to another instance.
5	Attribute set	Method that sets the values of one or more attributes.
6	Attribute get	Method that returns the values of one or more attributes.
7	I/O	Method that provides or receives data to or from a device.
8	Domain specific	Method that is specific to the application.

The types of methods shown in the above table provide a clear guideline for designing methods contained in a single class. The following operation representation of the method types has been suggested by the UML. The syntax is as below:

Visibility name: (parameter-list):return-type-expression

5.3.2.1 Designing methods for the ExcelSeparator Class

Basically, there are seven methods that have been identified to represent the roles played by the ExcelSeparator class in Chapter 4. At this point, the design of the methods in this class is conceptually complete. The following paragraphs describe the methods in details.

The constructor ExcelSeparator which has the input parameters consist of an Excel File name (String), a starting point for separation (String) and an ending point for separation (String), is responsible for setting the private attributes (attribute excelSource, startPoint and endPoint) of this class. This method will be invoked when there is a need to create an instance of this class.

Next, a private method called openExcel performs an invocation to the Application objects (provided by the package it.bigatti.excel8) for initialising an Excel application object which will then open the Excel source file. This method is designed to be accessible within this class only.

A private method called modifyFile is designed to cater for the main separation step. In this class, a search operation is performed to identify the range of data that need to be segregated and pasted in a new Excel file. There are two types of search operations that are implemented in this system. The first search operation is based on searching the exact string assigned to the startPoint and endPoint variables. If either of the strings is not found in the Excel document, a false value will be returned to the function called.

However, if the search operation returns a true value, that means the range of data has to be copied and duplicated in a new Excel file.

The second search operation is based on searching by the name of the cells. The system user will need to input the name of an Excel cell which is in the form of "A1", where the alphabet "A" indicates the column name and the number 1 indicates the row number. Two variables (startCell1 and endCell1) are allocated to store the starting cell name and the ending cell name. Figure 5.2 illustrates an activity diagram for this method.

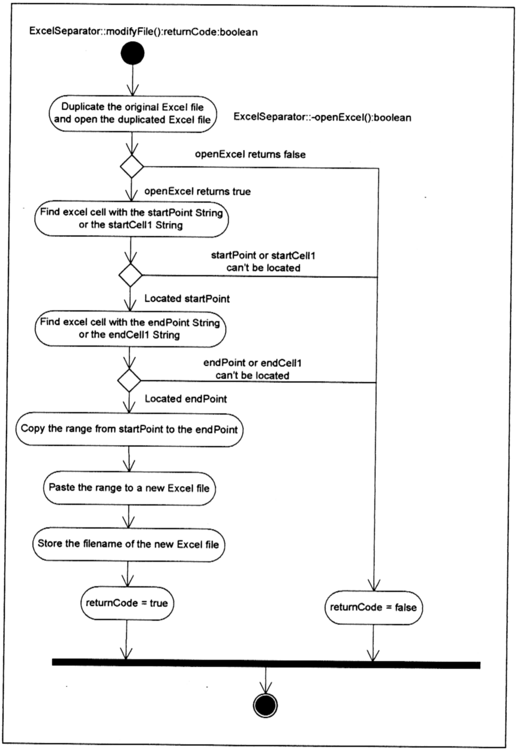

ExcelSeparator::modifyFile():returnCode:boolean
Duplicate the original Excel file and open the duplicated Excel file
ExcelSeparator::-openExcel():boolean
openExcel returns false
openExcel returns true
Find excel cell with the startPoint String or the startCell1 String
startPoint or startCell1 can't be located
Located startPoint
Find excel cell with the endPoint String or the endCell1 String
endPoint or endCell1 can't be located
Located endPoint
Copy the range from startPoint to the endPoint
Paste the range to a new Excel file
Store the filename of the new Excel file
returnCode = true
returnCode = false

Figure 5.2 Activity Diagram for modifyFile method

A close method is needed in order to terminate the Excel application object by first calling the save file method. This method has to be called for freeing up the memory taken by this class while executing an Excel file separation procedure.

A method called run is prepared for invoking the private method openExcel and if the method returns a true value, proceed to the method modifyFile. Lastly, a public method doSeparate (see Figure 5.3) is designed for initialising a ComThread instance (provided in the package com.jacob.com) and followed by creating an instance of this class. The instance will invoke run method. If the run method returns false, the close method is invoked for terminating the Excel Application object. Otherwise, the separated range of Excel data is saved with a randomly generated file name produced by the TempFileCreator object.

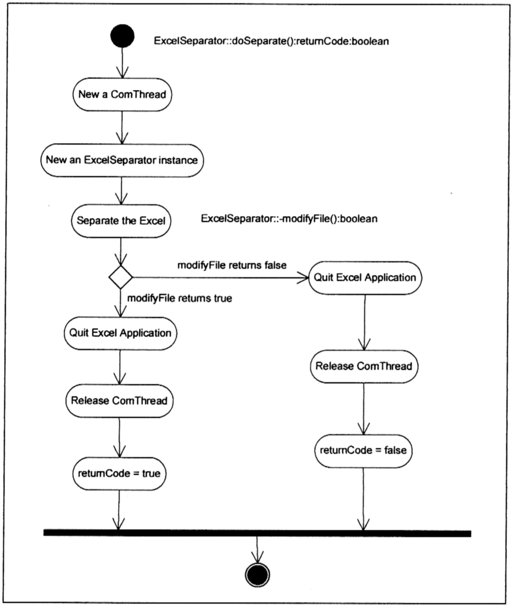


Figure 5.3 Activity Diagram for doSeparate method in ExcelSeparator class

5.3.2.2 Designing methods for the ExcelCombiner Class

The main role played by the ExcelCombiner class is to execute the combining process on several Excel files. Here, the combined Excel file will be saved with a new filename generated from a temporary file creator object. A constructor method with two input parameters, a string called file1 and a string called file2, is designed for setting the private attributes file1 and file2.

This class also has a method that sets combination type, called setCombineType. This method will determine whether both file1 and file2 will be combined horizontally or vertically. A private method openExcel which returns a boolean value is designed to create an instance of an Excel application object for opening an Excel document (provided by the file1 attribute).

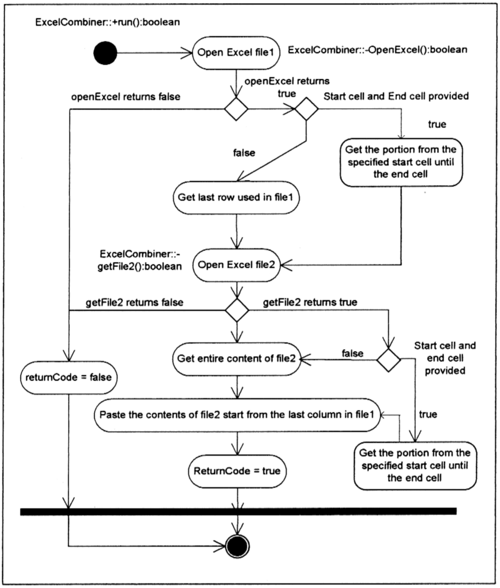
Besides that, system users can also specify the start cell and the end cell in either file1 or file2. If a specific pair of start cell and end cell is provided, this class will handle the Excel file combination by duplicating the portion (capture all the contents from the start cell until the end cell) and combine it with the other part of an Excel file.

Next, a private method called getFile2 with a string value as the input parameter is created. The string value represents the file name for an Excel file. This private method returns a _Worksheet object by using the Excel application object provided by the package it.bigatti.excel8.

A function called run (see Figure 5.4) is designed for combining two Excel files vertically. This can be done by first identifying the last row used in file1. Add a row below and follow by pasting the content in file2 onto the cells in file1. Eventually, there is another public method called run2 (see Figure 5.5) designed for combining two Excel files horizontally. The last column used in file1 must be first identified. Then, add one more column to the right. Finally, paste the content of file2 onto the cells (blank area) in file1.

It is important to have a method that is responsible for terminating the Excel application object. Therefore, a public method called close is designed. Meanwhile, since there is a private attribute called error contained in this class, a setErrorMsg method and a getErrorMsg method were designed to set and get the value of this attribute.

Finally, a public method doCombine (see Figure 5.6) that returns a boolean value is designed for invoking the combining methods (run and run2). The doCombine method will call the appropriate methods by referring to the value contained in the type attribute (either horizontal value or vertical value).



ExcelCombiner::+run():boolean
Open Excel file1
ExcelCombiner::-OpenExcel():boolean
openExcel returns true
openExcel returns false
Start cell and End cell provided
true
false
Get the portion from the specified start cell until the end cell
Get last row used in file1
ExcelCombiner::-getFile2():boolean
Open Excel file2
getFile2 returns false
getFile2 returns true
Get entire content of file2
false
Start cell and end cell provided
returnCode = false
Paste the contents of file2 start from the last column in file1
true
Get the portion from the specified start cell until the end cell
ReturnCode = true

Figure 5.4 Activity Diagram for run method in ExcelCombiner class

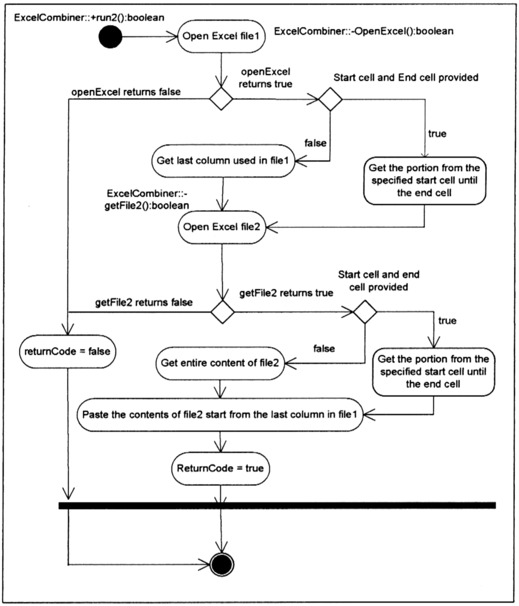
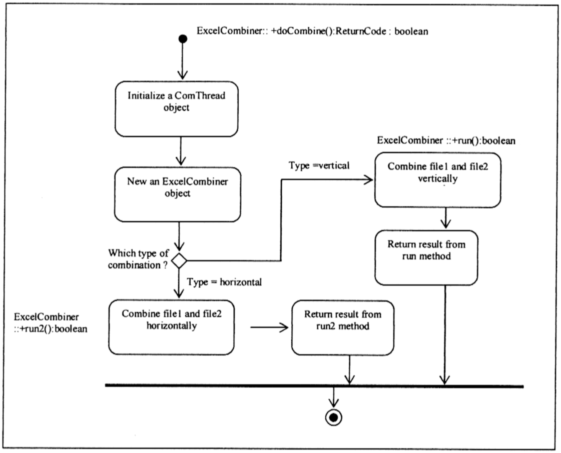

ExcelCombiner::+run2():boolean
Open Excel file1
ExcelCombiner::-OpenExcel():boolean
openExcel returns true
Start cell and End cell provided
openExcel returns false
true
false
Get last column used in file1
Get the portion from the specified start cell until the end cell
ExcelCombiner::-getFile2():boolean
Open Excel file2
Start cell and end cell provided
getFile2 returns true
getFile2 returns false
true
returnCode = false
false
Get entire content of file2
Get the portion from the specified start cell until the end cell
Paste the contents of file2 start from the last column in file1
ReturnCode = true

Figure 5.5 Activity Diagram for run2 method in ExcelCombiner class



ExcelCombiner:: +doCombine():ReturnCode : boolean
Initialize a ComThread object
New an ExcelCombiner object
Which type of combination ?
Type =vertical
Type = horizontal
ExcelCombiner ::+run():boolean
Combine file1 and file2 vertically
Return result from run method
ExcelCombiner ::+run2():boolean
Combine file1 and file2 horizontally
Return result from run2 method

Figure 5.6 Activity Diagram for doCombine method in ExcelCombiner class

5.3.2.3 Designing methods for the ExcelSeparatorService Class

This class is responsible for accepting the invocation from the interface layer and calling the execution of the ExcelSeparator class. In this class, a constructor is prepared for setting the values for the attributes. These attributes are the sourceFile, headers, count and destinationPath. A private method called setTempFileName is designed for duplicating several Excel files and save them with a different filename. The getTempFileName is a public method that returns an array of the randomly generated filenames.

A method called separate is designed for invoking the public methods provided by the ExcelSeparator class. Firstly, this method will call the private method setTempFileName. If the setTempFileName returns an array of filenames, then an instance of the ExcelSeparator class will be created and invokes its doSeparate method. The outcome of the doSeparate method will be assigned to a boolean variable. This boolean variable will be returned as the output of this method. If the output is true, then it indicates a success status of separating an Excel file. Otherwise, an error must have occurred and a public method getError is invoked for getting the error message.

5.3.2.4 Designing methods for the ExcelCombinerService class

This class is designed for accepting the request for combining several Excel files and invoking public methods provided by the ExcelCombiner class. A constructor that has an array of strings and a string as the input parameters is designed. The constructor serves the purpose of setting an array of Excel filenames that need to be combined and setting the type of combination (either horizontally or vertically).

Besides that, a private method called verifyFileFormat is created for verifying the file extensions provided by the system user that have to be combined together. If the file extension is not “XLS”, then the process of combining Excel files will be terminated and returns a false value.

A method called getErrorMsg is designed to return a string value. The string value would be the error message that occurred if there is an exception thrown by any of the methods.

Finally, a public method called combine is designed for creating an instance of the ExcelCombiner class and execute the public methods provided. This method will return a boolean value. A true value indicates a success status in combining several Excel files while a false value means there is an exception occurred and the combining process has failed.

5.3.3 Packages and managing classes

All the elements including the classes can be grouped into UML packages. The packages themselves may be nested within other packages. In this project, the main package would be named as the OODA System for Microsoft Excel Files. Inside this package, two packages are used significantly in order to complete the objectives of this project. These two packages are the com.jacob.com package and the it.bigatti.excel8 package. Figure 5.7 illustrates a more complete class diagram for the OODA system for Microsoft Excel Files.

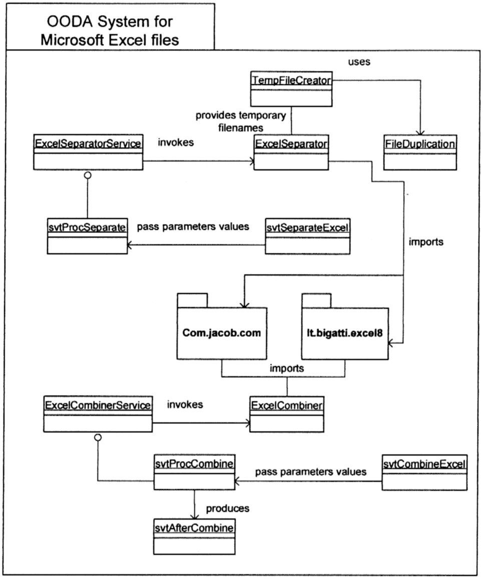


Figure 5.7 The OODA System for Microsoft Excel files package

5.4 Designing Access Layer

The Unified Approach (UA) has proposed the Layered Approach in designing classes. This Layered Approach defines a type of classes that is responsible for accessing data storage and these classes will not be accessible directly from the user interface layer objects. This type of classes is defined as access layer classes.

Designing access layer is about communicating with a Database Management System (DBMS). A DBMS is a set of programs that are responsible for the creation and maintenance of a collection of interrelated data. In this project, the DBMS used is the Microsoft Excel program. Microsoft Excel has been used for accessing, manipulating, protecting and managing data.

For this project, Microsoft Excel provides persistent data storage facility. Hence, it is necessary to create classes that can access and manipulate the data. Since Microsoft Excel is not a Relational DBMS (RDBMS), the classes designed will not be handling Structured Query Language (SQL) statements.

Two packages of classes are used as the access layer for this project. These two packages are the com.jacob.com package and the it.bigatti.excel8 package. Figure 5.8 illustrates the relationship and main objects used for manipulating data stored in a Microsoft Excel file.

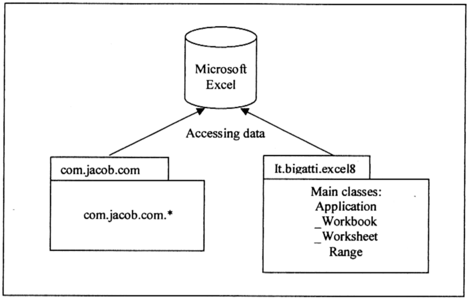


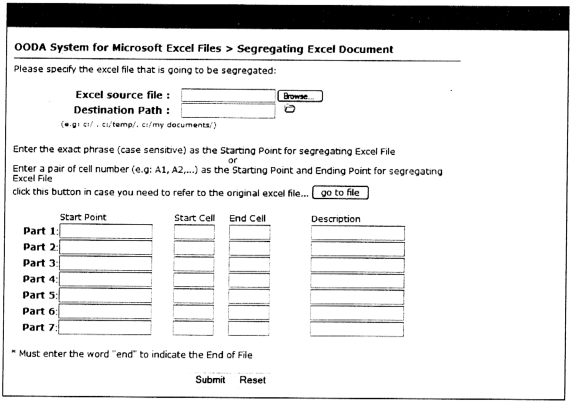
Figure 5.8 The Access Layer's Packages

5.5 Designing User Interface

After designing the access layer for accessing Microsoft Excel files, it is also important to look into the view layer (presentation layer) of this system. Interface objects construct the view layer classes. This can be done by referring to the activity diagrams developed during the O-O analysis stage.

Eventually, the design axioms and corollaries defined earlier must be followed in order to produce a good presentation prototype. It is good to have a user interface (UI) that is simple and transparent controlled by the system user. The main goal of designing the UI is to display and obtain information needed in an accessible and efficient manner.

Following, there are four diagrams that show the user interface design of the OODA System for Microsoft Excel Files in Figures 5.9 to 5.12.



OODA System for Microsoft Excel Files > Segregating Excel Document

Please specify the excel file that is going to be segregated:

Excel source file : [ ] Browse...
Destination Path : [ ]
(e.g: c:/ , c:/temp/, c:/my documents/)

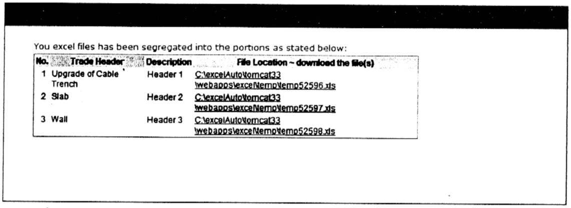
Enter the exact phrase (case sensitive) as the Starting Point for segregating Excel File
or
Enter a pair of cell number (e.g: A1, A2,...) as the Starting Point and Ending Point for segregating Excel File

click this button in case you need to refer to the original excel file... go to file

	Start Point	Start Cell	End Cell	Description
Part 1:				
Part 2:				
Part 3:				
Part 4:				
Part 5:				
Part 6:				
Part 7:				

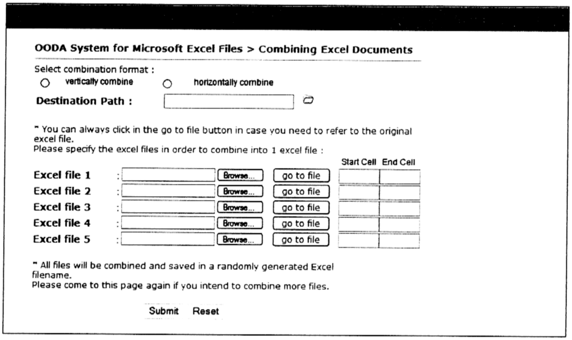
* Must enter the word "end" to indicate the End of File

Figure 5.9 User Interface for Calling ExcelSeparatorService Class


You excel files has been segregated into the portions as stated below:

No.	Trade Header	Description	File Location ~ download the file(s)
1	Upgrade of Cable Trench	Header 1	C:\excelAuto\tomcat33 \webapps\excelTemp\temp52596.xls
2	Slab	Header 2	C:\excelAuto\tomcat33 \webapps\excelTemp\temp52597.xls
3	Wall	Header 3	C:\excelAuto\tomcat33 \webapps\excelTemp\temp52598.xls

Figure 5.10 User Interface After Executing ExcelSeparatorService Class


OODA System for Microsoft Excel Files > Combining Excel Documents

Select combination format :
○ vertically combine ○ horizontally combine

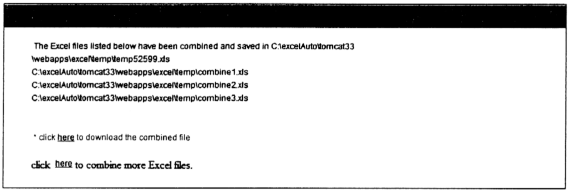
Destination Path :

* You can always click in the go to file button in case you need to refer to the original excel file.
Please specify the excel files in order to combine into 1 excel file :

				Start Cell	End Cell
Excel file 1	:		Browse...	go to file		
Excel file 2	:		Browse...	go to file		
Excel file 3	:		Browse...	go to file		
Excel file 4	:		Browse...	go to file		
Excel file 5	:		Browse...	go to file		

* All files will be combined and saved in a randomly generated Excel filename.
Please come to this page again if you intend to combine more files.

Figure 5.11 User Interface for Calling ExcelCombinerService Class



The Excel files listed below have been combined and saved in C:\excelAuto\tomcat33 \webapps\excel\temp\temp52599.xls
C:\excelAuto\tomcat33\webapps\excel\temp\combine1.xls
C:\excelAuto\tomcat33\webapps\excel\temp\combine2.xls
C:\excelAuto\tomcat33\webapps\excel\temp\combine3.xls

* click here to download the combined file

click here to combine more Excel files.

Figure 5.12 The User Interface after Executing ExcelCombinerService Class

5.6 Summary

Object-Oriented design is an iterative process. A good design will lead to a more efficient and effective system construction. Finally, the objectives of the system can be achieved and the system meets most of the user requirements. Therefore, it is a crucial task to come out with a fine design before proceed to the real system development phase.

In this chapter, all the classes (from the access layer, to the business layer and the view layer) have been designed properly. The relationships among all the classes have been defined as well. Diagrams showing the views of the system have also been presented. With all these elements, it will help the system developer to construct and build the real java code in order to implement the OODA System for the Microsoft Excel files. The following chapter will describe the processes involved in developing this project.