GRSManager: Order Processing Automation

Business Problem

Summarize the problem this program is attempting to resolve by describing the specific challenges that exist and how the software solution addresses them.

* Why do research customers not email their order forms to a general email inbox like clinical customers?
  + The pricing for research customers is different from the standard commercial pricing offered to clinical customers and is different between research customers.
  + ERT is over 50% of the business and chooses to upload spreadsheets to a portal that Circassia employees can access to get new orders.
* What are some unique requirements of the GRS order process?
  + Only domestic shipments are entered into NAV. If an order is being shipped OUS, the information is copied over to a word document and emailed to Circassia Sweden order processing. Circassia Sweden enters the order in NAV.
  + In the case of domestic shipments, the order is entered into NAV and then an email with order and customer number is sent to Morrisville logistics informing them of a new order that needs packaging and scheduling.
  + There are 3-5 pre-entry (in NAV) steps that are required for GRS orders which require manually copy/pasting, re-entering, and reformatting order information from one program to another before being entered in NAV.

Analysis and Requirements

**Model Details**

The model organizes different elements of the order process into classes which are used to manage the creation, tracking, and storage of order data. It provides the customer and the vendor simple interfaces for entering and processing orders, respectively. Table 1 lists the non-GUI, non-utility classes used to represent different processes within order management. The default values for all the factors are listed in the Default Value column.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbol** | **Factor** | **Default Value** | **Range** | **Description** |
| x | Prey count | 10 (start) | 0 to 300 | Number of prey |
| a | Prey birth rate | 0.165 | 0.0 to 1.0 | Birth rate of prey |
| b | Prey death rate | 0.0006 | 0.0 to 1.0 | Death rate of prey caused by predation |
| y | Predator count | 250 (start) | 0 to 500 | Number of predators |
| c | Predator death rate | 0.23 | 0.0 to 1.0 | Death rate of predators |
| p | Predator birth rate | 0.00068 | 0.0 to 1.0 | Population growth rate of predator due to availability of prey |
| z | Scavenger count | 400 (start) | 4 to 500 | Number of scavengers |
| e | Scavenger death rate | 0.1 | 0.0 to 1.0 | Death rate of scavengers |
| f | Scavenger birth rate from kills | 0.000002 | 0.0 to 1.0 | Population growth rate from scavenging corpses of prey that die from predation |
| g | Scavenger birth rate from prey | 0.0003 | 0.0 to 1.0 | Population growth rate from scavenging corpses of prey that die naturally |
| h | Scavenger birth rate from predators | 0.0006 | 0.0 to 1.0 | Population growth rate from scavenging corpses of predators |
| i | Scavenger population limiting factor | 0.001 | 0.0 to 1.0 | Limits population growth of scavengers. *This value is fixed and cannot be changed.* |
| **Table 1. Default Simulation Factors/Parameters** | | | | |

**User Interface**

The user interface has text fields for input of all of values listed in Table 1 except for the last one (Scavenger population limiting factor). The user is expected to enter integer values for the counts and double values for the remaining rate factors. Actions are initiated via three buttons at the top of the application window.

When the user starts the simulation, the graph shows the progress of the simulation step-by-step, with three points plotted for each step: one for the predator count, one for the prey count, and one for the scavenger count. There are 600 steps. The points are color coded as illustrated by Figure 1, which uses red for predator counts, black for prey counts, and blue for scavenger counts.

When a simulation run is complete, the text fields for the counts show the final population count values. For example, in the snapshot above, the population counts began with their default values: 10, 250, and 400. Figure 1 shows the final counts: 244 predators, 343 prey, and 311 scavengers.

Requirements Use Cases

**Use Case 1: Startup**

***Preconditions:***The user has just started the program.

***Main Flow:***The login panel opens with the two blank text fields for user id and password.

The user enters their login credentials [S1]. The user clicks Submit [S2] to close the window. Figure 2 shows the the initial window with default values. Figure 3 shows the initial window after the user changes all species' names from their default values and the scavenger's color to green.

***Sub flows:***

* [S1] The user enters new species names (as desired) [S2].
* [S2] The user picks different colors (as desired) [S3].
* [S3] The user clicks Close. The main simulation window opens with an empty graph area and default simulation parameters in the text fields [UC2]. See Table 1 and Figure 4.

***Alternative Flows:***

* [E1] There is a non-number in a Count field when the user clicks Start (Figure 5). The program displays a dialog box warning of the error and the simulation does not run. [E5]
* [E2] There is a negative number in a Count field when the user clicks Start (Figure 6). The program displays a dialog box warning of the error and the simulation does not run. [E5]

**Use Case 18: Vendor Interacts with an Open Order Schedule**

***Preconditions:*** An administrator has previously setup a vendor employee and a research company employee as users of the system. The administrator has loaded product information and order records. The research company employee has logged in and entered at least one order. A vendor employee is currently logged in.

***Main Flow:*** The user selects the open order [S1] and clicks the “Processed” button[S2].

***Sub Flows:***

* [S1] The user selects the open order displayed in the order schedule
* [S2] With the order selected, the user clicks the “Processed” button

***Alternative Flows:***

* [E1] There is no order selected in the order schedule and the user clicks the “Processed” button. The program displays a dialog box warning with the text “No order has selected.”

**Program Execution**

Step 1: Start-Up

When GRSManagerGUI is first constructed it initializes the following in this order:

1. LoginPanel is constructed with GridBagLayout.
2. AdministratorPanel is constructed and initializes the following panels in this order:
   1. VendorEmployeeDirectoryPanel is constructed and initializes first instance of GRSManager by calling the static GRSManager method getInstance(). GRSManager contains a static instance variable of itself so the getInstance() method checks this instance variable for the current GRSManager and returns it if it exists. If it is null, no GRSManager has been constructed, it calls GRSManager’s private constructor via the *new* keyword.
      1. GRSManager constructs the following data structure classes in this order:
         1. NioxCatalog
            1. LinkedListRecursive<Product> - an empty product list
         2. CompanyDirectory
            1. LinkedListRecursive<Company> - an empty Company list
         3. OrderRecord
            1. String [][] – an empty 2D string array
         4. Administrator
         5. UserDirectory
            1. LinkedAbstractList<User> - an empty list of User
      2. After GRSManager’s construction and subsequent creation of the above objects, **they are now available for access by all other GUI components via the static getInstance() method. User actions call GRSManager methods, which may call data structure class methods.** VendorEmployeeDirectoryPanel assigns its own instances of UserDirectory and CompanyDirectory to GRSManager’s (which are empty lists of Users and Companies, respectively). VendorEmployeeDirectoryPanel sets up the rest of the GUI components for its panel and returns control to AdministratorPanel’s constructor.
   2. ResearchEmployeeDirectoryPanel
      1. Requests instance of GRSManager (previously constructed during VendorEmployeeDirectoryPanel’s request for UserDirectory and CompanyDirectory), and receives empty LinkedLists of users and companies.
      2. Sets up the rest of its GUI components and returns control to AdministratorPanel’s constructor.
   3. ProductCatalogPanel
      1. Requests same instance of GRSManager and gets the product catalog class (NioxCatalog) and sets its one instance of NioxCatalog to the empty product list that is returned.
      2. Constructs remaining GUI components and returns control to AdministratorPanel’s constructor.
   4. OrderRecordPanel
      1. Requests same instance of GRSManager and gets the empty list of users and companies by setting its own instance variables for UserDirectory and CompanyDirectory to those returned by GRSManager.
      2. Sets up GUI components and calls the updateTables() method. updateTables() calls the updateData() method in the private class OrderDirectoryTableModel.
      3. updateData() in OrderDirectoryTableModel requests GRSManager’s OrderRecord first and then a 2D array of data by calling getShortOrderInfo. At this point, GRSManager’s instance of OrderRecord has an empty 2D array of Strings so null is copied over into the 2D array returned by getShortOrderInfo.
      4. Control returns to AdministratorPanel which finishes setting up and adding GUI components. Control returns to GRSManagerGUI.
3. VendorPanel. The Login and Administrator Panels have been constructed. Along with them the panels VendorEmployeeDirectoryPanel, ResearchEmployeeDirectoryPanel, ProductCatalogPanel, and OrderRecordPanel are all added to a card layout in the AdministratorPanel. The GRSManager has been initialized with references to data structure classes.
   1. VendorPanel’s constructor initializes
      1. VendorCompanyOrderSchedulePanel.
         1. When constructed, VCOSP requests GRSManager’s currentUser instance variable. At this point it is null because no one has logged into the system. It also requests the NioxCatalog from GRSManager which has an empty LinkedList of products.
         2. Sets up GUI components and calls the updateTables() method which calls the udpateData() methods via OrderTableModel instance variable named orderTableModel(). It requests the open order array from the OrderRecord instance variable named orderRecord which returns an empty 2D array of Strings. It is empty because, at this point, no order records have been uploaded into the program. This is done through the OrderRecordPanel contained in the Administrator class.
         3. VendorCompanyOrderSchedulePanel’s constructor finishes setting up and adding GUI components and returns control to VendorPanel()’s constructor
      2. VendorPanel constructor finishes setting up GUI components and returns control to GRSManagerGUI’s constructor
4. ResearchPanel’s constructor is called
   1. ResearchCompanyOrderEntryPanel’s constructor is called
      1. RCOEP’s currentUser instance variable is assigned the value from GRSManager (cast as employee). At this point, no one has logged in, so the value is null.
      2. RCOEP’s catalog instance variable is assigned the value from the GRSManager, which gives it access to an empty LinkedList of Product’s. Nothing has been loaded into the NioxCatalog (this is done in the ProductCatalogPanel).
      3. RCOEP’s orderRecord instance variable is assigned the value from the GRSManager, which gives it access to an empty LinkedList of Orders
      4. GroupLayout is set and GUI components are added to the layout
      5. updateTables() method is called which calls the updateData() method in ProductRollTableModel instance variable named productRollTableModel. The code is commented out, so control returns to RCOEP’s constructor which adds a few more GUI components and then returns control to ResearchPanel
   2. ResearchCompanyOpenOrderPanel’s constructor is called
      1. RCOOP’s currentUser instance variable is assigned the value from GRSManager (cast as employee). At this point, no one has logged in, so the value is null.
      2. OrderRecord instance variable is assigned the value from GRSManager’s OrderRecord instance variable
      3. updateTable() method in RCOOP called in the constructor, which calls OpenOrderTableModel’s updateData(), which requests 2D array of strings for open orders from the OrderRecord instance variable. Control returns to RCOOP’s constructor. Control returns to ResearchPanel
   3. Adds the GUI components and returns control to GRSManagerGUI

The GRSManagerGUI sets up GUI components, shows the LoginPanel on the card layout and sets the JFrame to visible.

Step 2: Administrator Login

1. User enters user name: Administrator, password: admin and clicks enter
2. LoginPanel.actionPerformed() is called, source of the action is determined, and an instance of GRSManager is requested.
3. GRSManager.login() is called and passed the string username and password. The user is not an instance of Employee so the first two conditions are skipped and cardLayout shows the AdministratorPanel
4. Controls returns to the GUI which is “listening” for user action

Step 3: Add Research Employees

1. The Administrator panel opens and displays the ResearchEmployeeDirectoryPanel
2. User enters research employee information into the text fields and clicks, add employee button
3. ResearchEmployeeDirectoryPanel.actionPerformed() is called -> it sees that the source is the addResearchEmployee button and gets the text from the appropriate text fields
   1. Gets the ship-to or bill-to from the combo box, catches npe if nothing has been selected
   2. Hashes the password
   3. If BillTo was selected, ResearchCompany is constructed with address information. ResearchCompany calls parent class, Company’s, constructor. Company’s constructor creates a new list of Locations, constructs a new BillTo (child class of Location) and adds it to the list
   4. Request the BillTo location for this Company and call the getBillTo() to get the BillTo location and call the addEmploye() method to add the Employee to the list of employees in the BillTo location
   5. Add ResearchCompany to company directory by calling CompanyDirectory.addResearchcompany().
   6. If ShipTo was selected, JOptionPane pops up and asks for the study number, the study number is parsed into local long variable named study and a ResearchSite (child class of Ship-To) is created.
   7. ResearchSite.addEmployee() is called and first,last,id,email,hashPW are passed to the addEmployee() method.
   8. The GUI textfields are all reset and the updateData() method in ResearchEmployeeDirectoryTableModel is called
   9. Employee is added to userDirectory

**OrderRecord.updateOrderList()**

Beginning with the second row, this method iterates over the order record array and constructs an Order from the data in each column of the row. The column titles are listed in the first row and within this list exist product titles, which are a consecutive set of titles that correspond to a product that the customer can order. In each of the following rows, integers in the product title columns indicate which and how many products were selected. Using OrderRecord’s “first” and “last” instance variables, which were set when the order record titles were read into the program, updateOrderList() can determine the beginning and ending titles on the upper and lower bounds of the product title range.

**OrderRecordIO.readOrderRecord()**

Beginning with the second row in the 2-D array parameter, each character of a comma-delimited text file containing all order records, is read into the corresponding row/column of the 2-D array. There are 52 comma-delimited data fields so the 2D array will need 52 columns. Some data values are contained within quotation marks and when this occurs, all characters (including commas) are part of the entry. Client code must pass readOrderRecord() a filename, a 2-D array of sufficient size, a list of ProductTitles, and an integer for the number of columns. readOrderRecord() initializes a FileReader and BufferedReader for reading the text file and a local char variable which will store the delimiter. As previously described, if reader encounters a quotation mark, it will be assigned to the local delimiter to indicate that all text read from this point in the file until the ending quotation mark, including commas, should be included in the value. After the ending quotation mark is read the local delimiter is reassigned a comma, as this is the original delimiter in the file. Local variables for column, row, and a char for the most recently read character are initialized before the readOrderRecord() enters a while loop whose condition fails when reader’s read method returns -1, indicating no more characters remain. In order, the following are the conditions each character passes through in the loop:

1. **Column Count:** When reader has read the last character of the last data field in a given row, the column count should equal the value (52) passed into the last column parameter of readOrderRecord(). StringBuilder’s toString() method is called to assign the characters in StringBuilder to the location at [row][column] the 2D array. The column count is then incremented past the last data field in the text and last column in the array. Attempting to assign data to a column greater than 52 will result in unchecked exception IndexOutOfBoundsException being thrown, so this is the first check done at the top of the loop. If the current column count is greater than the last column, we move to the next row, reset the column count to the first column, and use the *continue* keyword to bypass the rest of the loop body. If column count is <= last column, the condition fails and control moves to the next condition in the loop.
2. **Whitespace:** If the character is a whitespace ( ‘\n’, ‘\t’, ‘\_’ ) a single space is added to StringBuilder and the continue keyword is used to jump to the bottom of the loop. If the character is anything other than whitespace, the condition fails and control moves to the
3. **Ending Comma:** When the first comma is read in the first row of the text file, it marks the end of the first data field. In this initial case, delimiter equals 0, StringBuilder has characters so its length will be greater than 0, and the current character is a comma marking the end of the data field. These three checks must each be true for the second condition of the logical OR in the if statement to evaluate true.
   1. In subsequent passes through the loop, it is the first condition of the logical OR which will evaluate to true since current character will be a comma, marking the end of a data field, the local delimiter variable will have been set to a comma if this was the last occurrence of delimiter (if not, it will be a quotation), and the StringBuilder will not be empty. The characters in that data field will be assigned to the location at [row][column] in the 2D array.
4. **Beginning Quotation:** If the above condition fails control passes to the next if statement. Since no row begins with a quotation mark, the local delimiter variable will not be 0 when the current character is also a quotation mark.
   1. The local delimiter being a comma means this was the last delimiter encountered, if the current character is now a quotation mark then all characters read since the last comma and all characters read from this beginning quotation to the ending quotation are part of the data value. The local delimiter is assigned the quotation mark, control exists the if statement and returns to the top of the loop to read the next character.
5. **Between Quotation:** If the previous condition fails control passes to the next if statement. In this case, reader has encountered a beginning quotation mark. Local delimiter has been set to the quotation mark and every character since has been added to StringBuilder. In the first condition of the logical OR, the current character is not the ending quotation and is, therefore, added to StringBuilder. In the second condition of the logical OR, local delimiter is 0 and any character other than a quotation is…
6. **Ending Quotation:** Local delimiter has been set to a beginning quotation mark and every character since this assignment has been added to StringBuilder. The current character is the ending quotation mark. StringBuilder contents are assigned to the [row][column] location in the 2D array. The next character following a quotation mark is always a comma. Reader’s read() method is called to get this comma. The column count is incremented and control returns to the top of the loop where the while condition is checked again. Reader reads and assigns the character following the comma to current character. If it is not -1, control enters the while loop.
7. **Empty Column:** If the previous if statement describing the ending quotation condition fails, control passes to the final if statement where the current character is a comma and the last delimiter encountered (and stored in local delimiter variable) was a comma. Both expressions are included in the “Ending Comma” expression described an earlier if statement. In the “Ending Comma” expression we also check to see that StringBuilder has characters. Here we are describing an empty column, where one comma comes right after another.

**OrderRecordIO.readOrderTitles()**

Beginning with the first row in the 2-D array parameter, each character of a comma-delimited text file containing the titles of all order records, is read into the first row of the 2D array. There are 52 comma-delimited titles o the 2D array will need 52 columns. Some data values are contained within quotation marks and when this occurs, all characters (including commas) are part of the entry. Client code must pass readOrderRecord() a filename, a 2-D array of sufficient size, a list of ProductTitles, and an integer for the number of columns. readOrderRecord() initializes a FileReader and BufferedReader for reading the text file and a local char variable which will store the delimiter. As previously described, if reader encounters a quotation mark, it will be assigned to the local delimiter to indicate that all text read from this point in the file until the ending quotation mark, including commas, should be included in the value. After the ending quotation mark is read the local delimiter is reassigned a comma, as this is the original delimiter in the file. Local variables for column, row, and a char for the most recently read character are initialized before the readOrderRecord() enters a while loop whose condition fails when reader’s read method returns -1, indicating no more characters remain. In order, the following are the conditions each character passes through in the loop:

1. **Column Count:** When reader has read the last character of the last data field in a given row, the column count should equal the value (52) passed into the last column parameter of readOrderRecord(). StringBuilder’s toString() method is called to assign the characters in StringBuilder to the location at [row][column] the 2D array. The column count is then incremented past the last data field in the text and last column in the array. Attempting to assign data to a column greater than 52 will result in unchecked exception IndexOutOfBoundsException being thrown, so this is the first check done at the top of the loop. If the current column count is greater than the last column, we move to the next row, reset the column count to the first column, and use the *continue* keyword to bypass the rest of the loop body. If column count is <= last column, the condition fails and control moves to the next condition in the loop.
2. **Whitespace:** If the character is a whitespace ( ‘\n’, ‘\t’, ‘\_’ ) a single space is added to StringBuilder and the continue keyword is used to jump to the bottom of the loop. If the character is anything other than whitespace, the condition fails and control moves to the
3. **Ending Comma:** When the first comma is read in the first row of the text file, it marks the end of the first data field. In this initial case, delimiter equals 0, StringBuilder has characters so its length will be greater than 0, and the current character is a comma marking the end of the data field. These three checks must each be true for the second condition of the logical OR in the if statement to evaluate true.
   1. In subsequent passes through the loop, it is the first condition of the logical OR which will evaluate to true since current character will be a comma, marking the end of a data field, the local delimiter variable will have been set to a comma if this was the last occurrence of delimiter (if not, it will be a quotation), and the StringBuilder will not be empty. The characters in that data field will be assigned to the location at [row][column] in the 2D array.
4. **Beginning Quotation:** If the above condition fails control passes to the next if statement. Since no row begins with a quotation mark, the local delimiter variable will not be 0 when the current character is also a quotation mark.
   1. The local delimiter being a comma means this was the last delimiter encountered, if the current character is now a quotation mark then all characters read since the last comma and all characters read from this beginning quotation to the ending quotation are part of the data value. The local delimiter is assigned the quotation mark, control exists the if statement and returns to the top of the loop to read the next character.
5. **Between Quotation:** If the previous condition fails control passes to the next if statement. In this case, reader has encountered a beginning quotation mark. Local delimiter has been set to the quotation mark and every character since has been added to StringBuilder. In the first condition of the logical OR, the current character is not the ending quotation and is, therefore, added to StringBuilder. In the second condition of the logical OR, local delimiter is 0 and any character other than a quotation is…
6. **Ending Quotation:** Local delimiter has been set to a beginning quotation mark and every character since this assignment has been added to StringBuilder. The current character is the ending quotation mark. StringBuilder contents are assigned to the [row][column] location in the 2D array. The next character following a quotation mark is always a comma. Reader’s read() method is called to get this comma. The column count is incremented and control returns to the top of the loop where the while condition is checked again. Reader reads and assigns the character following the comma to current character. If it is not -1, control enters the while loop.
7. **Empty Column:** If the previous if statement describing the ending quotation condition fails, control passes to the final if statement where the current character is a comma and the last delimiter encountered (and stored in local delimiter variable) was a comma. Both expressions are included in the “Ending Comma” expression described an earlier if statement. In the “Ending Comma” expression we also check to see that StringBuilder has characters. Here we are describing an empty column, where one comma comes right after another.

The VendorCompanyOrderSchedulePanel is the interface where a vendor can view, select, and mark complete a list of open orders. It consists of a JPanel with GridBagLayout and one button in the far left corner labeled “Processed.” Below this JPanel is a JTable with order information for orders that are open. An order is open if it has not been processed by the vendor. This list of open orders is the vendor’s order schedule. When an order is processed, the vendor selects that order in the table and clicks the “Processed” button. Internally, the Order’s status is changed to “Processed,” and the Order is removed from the table.