Yin Zhanpeng

Assignment 2

ICT373 Software Architectures

Contents

[Requirements / Specification: 1](#_Toc162652332)

[Functions Explained and User Guide 4](#_Toc162652333)

[Limitations 13](#_Toc162652334)

[Testing 14](#_Toc162652335)

[Design Description 26](#_Toc162652336)

[Source Listing 29](#_Toc162652337)

Author: Yin Zhanpeng

Student ID: 34742217

Date: 29/03/24

File Name: Assignment1 Q2

Purpose: The purpose of this program is to manage the customers' magazine and supplement subscriptions using an easy-to-use graphic interface.

# Requirements / Specification:

Assumptions:

1. The customers have a list of Supplements and Magazines
2. The Supplements can be individual or inside the magazines
3. The magazines also contains its own List of Supplements
4. The program can allow the user to add the supplements to both the magazines and the customer
5. The paying customer have a list of associate customers
6. Both the paying customer and the associate customer have their own subscription
7. This program is to only simulate the weekly and monthly notifications
8. We are able to remove the associated customer of a paying customers
9. If we remove the paying customers, then its associated customer will be removed as well
10. The customer contains the magazine and not the magazine contains the customer
11. The paying customer should have at least 1 paying customer.
12. The program will primarily handle textual data inputs for customer names, addresses, email addresses, supplement names, etc.
13. Payment methods such as credit card or direct debit will be represented as strings or simple identifiers rather than complex objects.
14. The GUI will be designed for desktop use and will not be optimized for mobile devices or other platforms.
15. The billing history of paying customers will be displayed in a simple tabular format showing transaction dates, amounts, and payment methods.
16. Address details for customers will be limited to basic information like street number, street name, suburb, and postcode without additional fields such as country or state.
17. The program will handle a reasonable number of customers, supplements, and magazines without significant performance degradation.
18. Error handling will be implemented for basic scenarios such as invalid input formats, file I/O errors, and database connection issues, but it will not cover every possible edge case.
19. The GUI layout and design will be implemented using JavaFX controls and layouts rather than custom graphics or external libraries.
20. The program will be developed and tested primarily on the Java SE 8 platform using NetBeans IDE, ensuring compatibility with the specified environment.
21. Data persistence will be achieved through serialization, storing data in a single file or multiple files as needed, with no requirement for external databases or servers.

**Requirements and Specifications:**

1. **Graphical User Interface (GUI):**
   * Design and implement a JavaFX GUI program.
   * The GUI should allow recording information about a Magazine service.
   * The program should have three modes: viewing mode, edit mode, and create mode.
2. **Persistence:**
   * Implement persistence by storing data in file(s) and reusing them later.
   * Data should be saved and loaded using serialization.
3. **Customer Information:**
   * A customer should have a name, address, email address, and a list of interested supplements.
   * Address details include street number, street name, suburb, and postcode.
4. **Paying Customers:**
   * Paying customers have a payment method and a list of associate customers whom they also pay for.
5. **Associate Customers:**
   * Associate customers are those whose subscription is paid for by a specified paying customer.
6. **Payment Methods:**
   * Payment methods include credit card or direct debit from a specified bank account.
7. **Supplement Information:**
   * A supplement should have a name and a weekly cost.
8. **Magazine Information:**
   * The magazine should have a weekly cost for the main part.
9. **Home Screen:**
   * Program starts in create mode, edit mode, or viewing mode based on user selection.
10. **Viewing Mode:**
    * Display lists of supplements and customers.
    * Clicking on a supplement should display its relevant information.
    * Clicking on a customer should display relevant information including subscriptions and payment status.
    * Show billing history of paying customers.
11. **Edit Mode:**
    * Allow editing existing information about supplements and customers.
    * Functions include correcting information, adding/deleting customers, adding/deleting supplements.
    * Adding an associate customer involves selecting a paying customer from a dropdown list.
12. **Create Mode:**
    * Start by adding a Magazine, then supplements and customers.
    * Convenient data entry and storage for subsequent editing and viewing.
13. **Demonstration Setup:**
    * Set up a demonstration Magazine service.
    * Clear instructions in the User Guide for use.
14. **Documentation:**
    * Include a completed assignment cover sheet.
    * External documentation should include title, author, date, file names, and a purpose statement.
    * Detailed requirements/specifications of the program, including assumptions and expected output.
    * User guide with instructions on compilation, running, and usage.
    * Structure/design outline with explanation of chosen approaches, class relationships, and method descriptions.
    * Description of limitations and program shortfalls.
    * Testing strategy and results, including thorough testing of the program.
    * Submission of all Java source code files and relevant files in a compressed .zip format.
    * Use recommended Java JDK version for compatibility.

# Functions Explained and User Guide

**Load existing Magazine service systems and saving the file**

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

“data.ser” is the file where the program will save its information through serialization.

“loadDataFromFile()” and “saveDataToFile(Database database)” are responsible for loading and saving the data to “data.ser” file

“database.populate()” is a database method that will populate the program will some sample data.

**start a new Magazine service system,**

1. Click the create panel at the top
2. Click the create magazine button
3. Fill in the details of the magazine
4. Click create magazine

A screenshot of a computer

Description automatically generated

1. Able to see the newly created magazine on the view panel

A screenshot of a computer

Description automatically generated

**view customer information,**

able to view the customer information in the view panel by selecting the customer

A screenshot of a service

Description automatically generated

**edit customer information,**

1. Click the edit button
2. Click customer
3. Click edit details

A screenshot of a computer

Description automatically generated

1. Select the customer
2. Edit the details
3. Press save

A screenshot of a computer

Description automatically generated

**show billing information using multithreading**

1. Click view
2. Click on the paying customer
3. Scroll down on the information panel to display the billing information button

A screenshot of a computer

Description automatically generated

**Add or remove a paying customer to magazine or supplement**

1. Click edit
2. Click edit customer
3. Click edit details

A screenshot of a computer

Description automatically generated

1. Selcte the customer and click edit magazine and supplemtns

A screenshot of a computer

Description automatically generated

1. Add or remove the magazine or supplement

A screenshot of a computer

Description automatically generated

**add an associate customer**

1. Click edit
2. Click add associate customer to paying customer

A screenshot of a computer

Description automatically generated

1. Selcte the paying customer
2. Add the available associate customer

A screenshot of a computer

Description automatically generated

**view magazine information or view supplement information**

1. In the view tab selected the magazine or supplement and its information will display on the information panel

A screenshot of a computer

Description automatically generated

**edit magazine information**

1. Click edit
2. Click edit magazine
3. Select the magazine
4. Click edit selected magazine

A screenshot of a computer

Description automatically generated

**edit supplement information**

1. Similar to edit magazine

A screenshot of a computer

Description automatically generated

save to file.

1. The program will automatically save once you close the program. It will saved in the **data.ser** file

# Limitations

1. Platform limited to Java SE 8 and NetBeans IDE.
2. Dependency on JavaFX for GUI.
3. Basic features compared to commercial software.
4. Manual data entry prone to errors.
5. Limited error handling capabilities.
6. Designed for single-user access only.
7. Lack of extensive data validation checks.
8. Minimal reporting and analytics features.
9. Relies on file-based data storage.
10. Scalability issues with large datasets.
11. Limited support for multi-user access.
12. Potential for data inconsistency due to manual entry.
13. Minimal support for advanced billing functionalities.
14. No automated subscription renewal reminders.
15. Lack of customization options for reports.
16. Potential performance limitations with file-based storage.
17. Challenges in accommodating future enhancements.
18. Limited support for graphical analytics.
19. Inability to handle concurrent data editing.
20. Possible compatibility issues with different JavaFX versions.

# Testing

Testing explanations and test strategy

1. **User Interface Actions**: Tests should cover actions users can perform, such as navigating between modes, clicking on customers or supplements, adding, editing, or deleting items, and handling user inputs.
2. **Data Manipulation**: Verify that data operations, like adding new customers or supplements, deleting existing ones, and editing customer information, are performed correctly and consistently.
3. **Error Handling**: Test how the application handles invalid inputs, error conditions, and edge cases. Ensure appropriate error messages are displayed and actions are taken to prevent or recover from errors.
4. **User Experience**: Evaluate the overall user experience, including interface responsiveness, layout, readability, and ease of navigation.
5. **Integration**: Test interactions between different components, ensuring they integrate seamlessly and function correctly together.
6. **Performance**: Assess the application's performance under normal and stress conditions, including response times and resource usage.
7. **User Interface Testing**: Conduct usability testing to ensure the interface is intuitive and user-friendly. Test navigation flows and user interactions.

Test table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Test Case Description | Expected Result | Actual Result | Pass/Fail |
| 1 | Launch the application | Application launches successfully | As expected | Pass |
| 2 | Navigate to Viewing Mode | Viewing mode is accessible | As expected | Pass |
| 3 | Click on a supplement | Supplement details are displayed | As expected | Pass |
| 4 | Click on a customer | Customer details are displayed | As expected | Pass |
| 5 | Attempt to add a new supplement | New supplement is added | As expected | Pass |
| 6 | Attempt to delete an existing customer | Customer is removed from the list | As expected | Pass |
| 7 | Edit the information of an existing customer | Customer information is updated | As expected | Pass |
| 8 | Switch to Edit Mode | Edit mode is accessible | As expected | Pass |
| 9 | Add a paying customer with associate customers | Paying customer and associates are added | As expected | Pass |
| 10 | Delete a supplement | Supplement is removed from the list | As expected | Pass |
| 11 | Attempt to add a new customer with incomplete information | Unable to procced | As expected | pass |
| 12 | Attempt to edit customer information with invalid data | Able to edit | As expected | fail |
| 13 | Attempt to add a duplicate supplement | Able to add | As expected | fail |
| 14 | Navigate to Create Mode and add a new magazine | Magazine is added to the list of magazines | As expected | Pass |
| 15 | Attempt to delete a non-existent supplement | GUI don’t display | As expected | Pass |
| 16 | Attempt to edit information in Viewing Mode | View mode don’t have edit functions | As expected | Pass |
| 17 | Switch between modes multiple times | Modes switch seamlessly without issues | As expected | Pass |
| 18 | Add a customer with a long name (> 50 characters) | Customer name is truncated to fit in the display | As expected | Pass |
| 19 | Add a supplement with a special character in the name | Special characters are displayed correctly | As expected | Pass |
| 20 | Attempt to add a customer with an invalid email address | Error message prompts user to enter a valid email | Able to add | fail |
| 21 | Add a paying customer with no associated associate customers | Paying customer added successfully | As expected | Pass |
| 22 | Attempt to delete a non-existent customer | Error message indicates customer not found | As expected | Pass |
| 23 | Edit the address of an existing customer | Address details are updated | As expected | Pass |
| 24 | Attempt to add a supplement with a negative weekly cost | Error message indicates invalid cost | As expected | Pass |
| 25 | Delete all customers | Customer list becomes empty | As expected | Pass |
| 26 | Attempt to delete a non-existent magazine | Error message indicates magazine not found | As expected | Pass |
| 27 | Attempt to add an empty supplement name | Error message prompts user to enter a name | As expected | Pass |
| 28 | Attempt to add a customer with a duplicate email address | Error message indicates duplicate email | As expected | Pass |
| 29 | Edit the payment method of a paying customer | Payment method is updated | As expected | Pass |
| 30 | Attempt to delete a supplement with associated customers | Error message prompts user to remove associations | As expected | Pass |

Test cases screenshot

1&2 A screenshot of a computer

Description automatically generated

3

A screenshot of a computer

Description automatically generated

4

A screenshot of a computer

Description automatically generated

5

A screenshot of a computer

Description automatically generated

A screenshot of a service

Description automatically generated

6 delte Robert from magazine 1

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

7 changing Robert email to a random line of strings

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

8

A screenshot of a computer

Description automatically generated

9 add Robert to zhanpeng

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

10 delete new sup

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

11 unable to create as expected

A screenshot of a computer

Description automatically generated

12 error

13 error

14 add a new magazine

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

15 unable to delete as not display in GUI

16 unable to edit in view mode as no option

17 able to switch mode

18 add a customer with long name

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

19 able to add

20 able to add

# Design description

**Design Description and Explanation of Design Choices:**

1. **Object-Oriented Approach:**
   * I have used object-oriented principles to structure my program; the diagram shows various classes and relationships between them.
   * This approach typically involves encapsulating data and behaviours within classes and using inheritance, polymorphism, and abstraction to create a modular and reusable codebase.
2. **Multithreading:**
   * Multithreading allows the program to perform multiple operations concurrently, improving efficiency and responsiveness, especially when dealing with I/O operations or long-running tasks.
   * I have system parts that perform tasks in the background (e.g., generating billing information) while keeping the user interface responsive.
3. **Serialization:**
   * Serialization is used to persist objects to storage or transmit them over a network. It converts objects into a format that can be easily stored or transmitted and later reconstructed.
   * serialization saves customer data, billing details, and other information to a file or database.
4. **Use of Collections and Aggregation:**
   * I use a collection classes, like lists or sets, to manage groups of related objects. For example, a **PayingCustomer** has a list of **AssociateCustomers**.
   * Aggregation indicates that classes contain or manage other classes without strict ownership (i.e., the contained classes can exist independently).
5. **Use of Associations:**
   * The associations between classes, like those between **Customer** and **Magazine**, suggest that objects are linked in a way that reflects real-world relationships. For instance, customers subscribe to magazines, and this relationship is modeled in the system.
6. **Error Handling:**
   * I have use some of error handling in the program to prevents errors like out of bound for arrays and null pointer etc
7. **Data Management:**
   * Centralized data management classes, perhaps indicated by classes that have relationships with many other classes, could serve as access points for retrieving and persisting data throughout the application.
8. **UI Separation:**
   * I try to separates the user interface logic from the data logic, which is a common practice in OO design, known as the Model-View-Controller (MVC) pattern or similar variations.
9. **Scalability and Maintainability:**
   * By adhering to OO principles, the design aims for scalability and maintainability. New features, such as additional types of customers or new billing procedures, can be added with minimal changes to the existing codebase.
10. **Decoupling:**
    * The system probably emphasizes decoupling, where individual components or classes are as independent as possible. This reduces the ripple effect of changes and eases maintenance.

design diagrams UML these two diagram can be found in the folder as well for a clear viewA close-up of several papers

Description automatically generatedA black and white image of a triangle

Description automatically generated

# Source Listing

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File

\* Name: Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.ArrayList;

import java.util.InputMismatchException;

import java.util.List;

import java.util.Scanner;

/\*\*

\* This class provides functionality to add an associate customer to a paying

\* customer. It manages the interaction with the user to select from a list of

\* paying and associate customers.

\*/

public class AddAssociateToPayingCustomer {

private static Scanner scanner = new Scanner(System.in);

/\*\*

\* Runs the process of adding an associate customer to a paying customer. It

\* prompts the user to select a paying customer and then an associate

\* customer to be added to the paying customer's list of associates.

\*

\* @param database the database containing the list of paying and associate

\* customers.

\*/

public static void run(Database database) {

// Display list of paying customers

List<PayingCustomer> payingCustomers = database.getPayingCustomers();

if (payingCustomers.isEmpty()) {

System.out.println("There are no paying customers.");

return;

}

displayCustomers("Paying", payingCustomers);

// Ask user to select a paying customer

int payingCustomerIndex = getValidIndex("Enter the index of the paying customer to add associates", payingCustomers.size());

PayingCustomer selectedPayingCustomer = payingCustomers.get(payingCustomerIndex);

// Display list of associate customers

List<Customer> associateCustomers = database.getAssociateCustomers();

displayCustomers("Associate", associateCustomers);

// Ask user to select an associate customer to add

if (!associateCustomers.isEmpty()) {

int associateCustomerIndex = getValidIndex("Enter the index of the associate customer to add, or -1 to cancel", associateCustomers.size());

if (associateCustomerIndex != -1) {

Customer selectedAssociateCustomer = associateCustomers.get(associateCustomerIndex);

selectedPayingCustomer.addAssociateCustomer(selectedAssociateCustomer);

System.out.println("Associate added successfully to the paying customer.");

database.removeAssociateCustomer(selectedAssociateCustomer);

} else {

System.out.println("Operation canceled.");

}

} else {

System.out.println("There are no associate customers available to add.");

}

//selectedPayingCustomer.display();

}

/\*\*

\* Displays a list of customers to the console.

\*

\* @param customerType the type of customer (e.g., "Paying" or "Associate").

\* @param customers the list of customers to be displayed.

\*/

private static void displayCustomers(String customerType, List<? extends Customer> customers) {

System.out.println("List of " + customerType + " Customers:");

for (int i = 0; i < customers.size(); i++) {

System.out.println((i + 1) + ". " + customers.get(i).getName());

}

}

/\*\*

\* Prompts the user for a valid index within a given range. It ensures that

\* the input is a valid integer within the range [1, size] inclusive.

\*

\* @param prompt the prompt message to display to the user.

\* @param size the size of the list from which the index is valid.

\* @return the valid index entered by the user, adjusted to be zero-based.

\*/

private static int getValidIndex(String prompt, int size) {

while (true) {

System.out.print(prompt + " (1-" + size + "): ");

try {

int index = scanner.nextInt();

scanner.nextLine(); // Consume newline

if (index >= 1 && index <= size) {

return index - 1;

}

} catch (InputMismatchException e) {

scanner.nextLine(); // Consume invalid input

}

System.out.println("Invalid index. Please try again.");

}

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.Scanner;

/\*\*

\* This class provides functionality to add a new magazine to a customer in the

\* magazine service.

\*/

public class AddNewCustomerToMagazineService {

/\*\*

\* Runs the process of adding a new magazine to a customer in the database.

\*

\* @param database The database containing customer and magazine

\* information.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

// List out all customers (both paying and associate)

System.out.println("List of All Customers:");

int index = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

System.out.println(index + ". " + payingCustomer.getName() + " (Paying Customer)");

index++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

System.out.println(index + ". " + associateCustomer.getName() + " (Associate Customer)");

index++;

}

}

// Select customer

int selectedCustomerIndex = getUserSelection("Enter the index of the customer: ", index);

// Find selected customer

Customer selectedCustomer = findCustomerByIndex(database, selectedCustomerIndex);

// List out magazines

System.out.println("List of Magazines:");

for (int i = 0; i < database.getMagazines().size(); i++) {

System.out.println((i + 1) + ". " + database.getMagazines().get(i).getMagazineName());

}

// Select magazine

int selectedMagazineIndex = getUserSelection("Enter the index of the magazine to add: ", database.getMagazines().size());

Magazine selectedMagazine = database.getMagazines().get(selectedMagazineIndex - 1);

// Add magazine to selected customer

selectedCustomer.addMagazine(selectedMagazine);

System.out.println("Magazine added successfully to " + selectedCustomer.getName());

selectedCustomer.display();

}

/\*\*

\* Prompts the user for a selection and ensures it is within a valid range.

\*

\* @param prompt The message to display prompting the user for input.

\* @param size The maximum valid selection value.

\* @return The user's valid selection.

\*/

private static int getUserSelection(String prompt, int size) {

Scanner scanner = new Scanner(System.in);

int selection = 0;

boolean validInput = false;

while (!validInput) {

try {

System.out.print(prompt);

selection = scanner.nextInt();

scanner.nextLine(); // Consume newline

if (selection >= 1 && selection <= size) {

validInput = true;

} else {

System.out.println("Invalid selection. Please enter a number between 1 and " + size + ".");

}

} catch (Exception e) {

scanner.nextLine(); // Consume invalid input

System.out.println("Invalid input. Please enter a number.");

}

}

return selection;

}

/\*\*

\* Finds a customer in the database based on their index.

\*

\* @param database The database containing customer information.

\* @param index The index of the customer to find.

\* @return The customer corresponding to the given index, or null if not

\* found.

\*/

private static Customer findCustomerByIndex(Database database, int index) {

int currentIndex = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

if (currentIndex == index) {

return payingCustomer;

}

currentIndex++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

if (currentIndex == index) {

return associateCustomer;

}

currentIndex++;

}

}

return null;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.Scanner;

/\*\*

\* This class provides functionality to add supplements to either customers or

\* magazines in the database.

\*/

public class AddSupplementToCustomerOrMagazine {

/\*\*

\* Runs the process of adding supplements to either customers or magazines

\* based on user choice.

\*

\* @param database The database containing customer, magazine, and

\* supplement information.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

// Ask user if they want to add supplement to a customer or a magazine

System.out.println("Do you want to add supplement to a customer or a magazine?");

System.out.println("1. Add supplement to customer");

System.out.println("2. Add supplement to magazine");

int choice = getUserSelection("Enter your choice: ", 2);

if (choice == 1) {

// Add supplement to a customer

addSupplementToCustomer(database);

} else {

// Add supplement to a magazine

addSupplementToMagazine(database);

}

}

/\*\*

\* Adds a supplement to a selected customer.

\*

\* @param database The database containing customer and supplement

\* information.

\*/

private static void addSupplementToCustomer(Database database) {

// List out all customers (both paying and associate)

System.out.println("List of All Customers:");

int index = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

System.out.println(index + ". " + payingCustomer.getName() + " (Paying Customer)");

index++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

System.out.println(index + ". " + associateCustomer.getName() + " (Associate Customer)");

index++;

}

}

// Select customer

int selectedCustomerIndex = getUserSelection("Enter the index of the customer: ", index);

// Find selected customer

Customer selectedCustomer = findCustomerByIndex(database, selectedCustomerIndex);

// Select supplement to add

Supplement selectedSupplement = database.selectSupplement();

// Add supplement to selected customer

selectedCustomer.addSupplement(selectedSupplement);

System.out.println("Supplement added successfully to " + selectedCustomer.getName());

selectedCustomer.display();

}

/\*\*

\* Adds a supplement to a selected magazine.

\*

\* @param database The database containing magazine and supplement

\* information.

\*/

private static void addSupplementToMagazine(Database database) {

// List out all magazines

System.out.println("List of All Magazines:");

int index = 1;

for (Magazine magazine : database.getMagazines()) {

System.out.println(index + ". " + magazine.getMagazineName());

index++;

}

// Select magazine

int selectedMagazineIndex = getUserSelection("Enter the index of the magazine: ", index);

Magazine selectedMagazine = database.getMagazines().get(selectedMagazineIndex - 1);

// Select supplement to add

Supplement selectedSupplement = database.selectSupplement();

// Add supplement to selected magazine

selectedMagazine.addSupplement(selectedSupplement);

System.out.println("Supplement added successfully to " + selectedMagazine.getMagazineName());

}

/\*\*

\* Prompts the user for a selection and ensures it is within a valid range.

\*

\* @param prompt The message to display prompting the user for input.

\* @param size The maximum valid selection value.

\* @return The user's valid selection.

\*/

private static int getUserSelection(String prompt, int size) {

Scanner scanner = new Scanner(System.in);

int selection = 0;

boolean validInput = false;

while (!validInput) {

try {

System.out.print(prompt);

selection = scanner.nextInt();

scanner.nextLine(); // Consume newline

if (selection >= 1 && selection <= size) {

validInput = true;

} else {

System.out.println("Invalid selection. Please enter a number between 1 and " + size + ".");

}

} catch (Exception e) {

scanner.nextLine(); // Consume invalid input

System.out.println("Invalid input. Please enter a number.");

}

}

return selection;

}

/\*\*

\* Finds a customer in the database based on their index.

\*

\* @param database The database containing customer information.

\* @param index The index of the customer to find.

\* @return The customer corresponding to the given index, or null if not

\* found.

\*/

private static Customer findCustomerByIndex(Database database, int index) {

int currentIndex = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

if (currentIndex == index) {

return payingCustomer;

}

currentIndex++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

if (currentIndex == index) {

return associateCustomer;

}

currentIndex++;

}

}

return null;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.util.ArrayList;

import javafx.application.Application;

import javafx.geometry.Insets;

import javafx.geometry.Pos;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.ListView;

import javafx.scene.control.Separator;

import javafx.scene.control.TextField;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.GridPane;

import javafx.scene.layout.HBox;

import javafx.scene.layout.VBox;

import javafx.stage.Modality;

import javafx.stage.Stage;

import java.util.List;

import java.util.Optional;

import java.util.stream.Collectors;

import java.util.stream.Stream;

import javafx.application.Platform;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.concurrent.Task;

import javafx.geometry.HPos;

import javafx.geometry.Orientation;

import javafx.scene.Node;

import javafx.scene.control.Alert;

import javafx.scene.control.ButtonBar;

import javafx.scene.control.ButtonType;

import javafx.scene.control.CheckBox;

import javafx.scene.control.ChoiceDialog;

import javafx.scene.control.ComboBox;

import javafx.scene.control.Dialog;

import javafx.scene.control.ListCell;

import javafx.scene.control.ScrollPane;

import javafx.scene.control.SelectionMode;

import javafx.scene.layout.Priority;

import javafx.scene.text.Font;

/\*\*

\* The main application class that extends javafx.application.Application. This

\* class handles the GUI for the magazine service application.

\*/

public class App extends Application {

/\*\*

\* The database instance managing all data for the application.

\*/

static Database database = new Database();

/\*\*

\* The ListView displaying the list of supplements.

\*/

private ListView<Supplement> supplementsList;

/\*\*

\* The ListView displaying the list of customers.

\*/

private ListView<Customer> customersList;

/\*\*

\* The ListView displaying the list of magazines.

\*/

private ListView<Magazine> magazinesList;

/\*\*

\* The VBox representing the information panel in the GUI.

\*/

private VBox infoPanel;

/\*\*

\* The file name for data serialization.

\*/

private static final String DATA\_FILE = "data.ser";

/\*\*

\* The primary stage for the JavaFX application.

\*/

private Stage primaryStage;

/\*\*

\* The start method of the JavaFX application.

\*

\* @param primaryStage The primary stage of the JavaFX application.

\*/

@Override

public void start(Stage primaryStage) {

database = loadDataFromFile();

//database.populate();

// Top title

Label titleLabel = new Label("MAGAZINE SERVICE");

titleLabel.setAlignment(Pos.CENTER);

titleLabel.setMaxWidth(Double.MAX\_VALUE);

titleLabel.setStyle("-fx-font-size: 24px; -fx-text-fill: #333; -fx-font-weight: bold;");

// Top button bar

Button viewButton = new Button("View");

Button createButton = new Button("Create");

Button editButton = new Button("Edit");

HBox buttonBar = new HBox(10, viewButton, createButton, editButton);

buttonBar.setAlignment(Pos.CENTER);

VBox topBar = new VBox(5, titleLabel, buttonBar);

topBar.setAlignment(Pos.CENTER);

topBar.setPadding(new Insets(10));

// Center view with ListViews and Information Panel

supplementsList = new ListView<>();

supplementsList.getItems().addAll(database.getSupplements());

supplementsList.setCellFactory(lv -> new ListCell<Supplement>() {

@Override

protected void updateItem(Supplement item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getSupplementName());

}

});

Label supplementsLabel = new Label("List of Supplements");

VBox supplementsPanel = new VBox(10, supplementsLabel, supplementsList);

supplementsPanel.setPadding(new Insets(10));

customersList = new ListView<>();

customersList.getItems().addAll(database.getPayingCustomers());

customersList.getItems().addAll(database.getAssociateCustomers());

customersList.setCellFactory(lv -> new ListCell<Customer>() {

@Override

protected void updateItem(Customer item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getName());

}

});

Label customersLabel = new Label("List of Customers");

VBox customersPanel = new VBox(10, customersLabel, customersList);

customersPanel.setPadding(new Insets(10));

magazinesList = new ListView<>();

magazinesList.getItems().addAll(database.getMagazines());

magazinesList.setCellFactory(lv -> new ListCell<Magazine>() {

@Override

protected void updateItem(Magazine item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getMagazineName());

}

});

Label magazinesLabel = new Label("List of Magazines");

VBox magazinePanel = new VBox(10, magazinesLabel, magazinesList);

magazinePanel.setPadding(new Insets(10));

// Information Panel with ScrollPane

infoPanel = new VBox(10, new Label("Information Panel:"));

infoPanel.setPadding(new Insets(10));

infoPanel.setMinWidth(200);

ScrollPane scrollPane = new ScrollPane(infoPanel);

scrollPane.setFitToWidth(true);

scrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.NEVER); // Never show the horizontal scrollbar

scrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS\_NEEDED); // Only show the vertical scrollbar when needed

// Maian content layout

HBox mainContentHBox = new HBox(0, magazinePanel, new Separator(Orientation.VERTICAL), supplementsPanel, customersPanel, new Separator(Orientation.VERTICAL), scrollPane);

mainContentHBox.setAlignment(Pos.TOP\_CENTER);

// Bottom panels (hidden initially)

VBox createModePanel = createModePanel(primaryStage);

VBox editModePanel = editModePanel();

createModePanel.setVisible(false);

editModePanel.setVisible(false);

// GridPane layout setup

GridPane mainLayout = new GridPane();

mainLayout.setAlignment(Pos.CENTER);

mainLayout.add(topBar, 0, 0);

mainLayout.add(mainContentHBox, 0, 1);

mainLayout.add(createModePanel, 0, 1); // Add the createModePanel to the GridPane

mainLayout.add(editModePanel, 0, 1); // Add the editModePanel to the GridPane

// Button action handlers

viewButton.setOnAction(e -> {

mainContentHBox.setVisible(true);

createModePanel.setVisible(false);

editModePanel.setVisible(false);

});

createButton.setOnAction(e -> {

mainContentHBox.setVisible(false);

createModePanel.setVisible(true);

editModePanel.setVisible(false);

});

editButton.setOnAction(e -> {

mainContentHBox.setVisible(false);

createModePanel.setVisible(false);

editModePanel.setVisible(true);

});

// Scene and stage setup

Scene scene = new Scene(mainLayout, 800, 600);

primaryStage.setTitle("Magazine Service");

primaryStage.setScene(scene);

primaryStage.show();

// ListView selection handling

supplementsList.getSelectionModel().selectedItemProperty().addListener((observable, oldValue, newValue) -> {

if (newValue != null) {

infoPanel.getChildren().clear();

// Display the selected supplement's information

Node supplementInfo = newValue.displayGUI();

infoPanel.getChildren().add(supplementInfo);

// Add a separator and a label for customer subscriptions

infoPanel.getChildren().add(new Separator());

infoPanel.getChildren().add(new Label("Customers subscribed to this supplement:"));

// Iterate over all customers to find subscriptions to the selected supplement

boolean found = false;

for (Customer customer : database.getPayingCustomers()) {

if (customer.getSupplements() != null && customer.getSupplements().contains(newValue)) {

Label subscribedCustomerLabel = new Label(customer.getName());

infoPanel.getChildren().add(subscribedCustomerLabel);

found = true;

}

}

// Do the same for associate customers

for (Customer customer : database.getAssociateCustomers()) {

if (customer.getSupplements() != null && customer.getSupplements().contains(newValue)) {

Label subscribedCustomerLabel = new Label(customer.getName());

infoPanel.getChildren().add(subscribedCustomerLabel);

found = true;

}

}

// If no customers are found, display a message indicating that

if (!found) {

infoPanel.getChildren().add(new Label("No customers are subscribed to this supplement."));

}

}

});

customersList.getSelectionModel().selectedItemProperty().addListener((observable, oldValue, selectedCustomer) -> {

if (selectedCustomer != null) {

infoPanel.getChildren().clear();

Node customerInfo = selectedCustomer.displayGUI();

infoPanel.getChildren().add(customerInfo);

if (selectedCustomer instanceof PayingCustomer) { // Check if the customer is a paying customer

Button displayBillingHistoryBtn = new Button("Display Billing History");

displayBillingHistoryBtn.setOnAction(event -> displayBillingHistory(selectedCustomer));

infoPanel.getChildren().add(displayBillingHistoryBtn);

}

}

});

// ListView selection handling for magazines

magazinesList.getSelectionModel().selectedItemProperty().addListener((observable, oldValue, newValue) -> {

if (newValue != null) {

infoPanel.getChildren().clear();

// Display the selected magazine's information

Node magazineInfo = newValue.displayGUI(); // Assuming Magazine class has a displayGUI() method

infoPanel.getChildren().add(magazineInfo);

// Add a separator and a label for customers subscribed to this magazine

infoPanel.getChildren().add(new Separator());

infoPanel.getChildren().add(new Label("Customers subscribed to this magazine:"));

// Iterate over all customers to find subscriptions to the selected magazine

boolean found = false;

for (Customer customer : database.getPayingCustomers()) {

if (customer.getMagazines() != null && customer.getMagazines().contains(newValue)) {

Label subscribedCustomerLabel = new Label(customer.getName());

infoPanel.getChildren().add(subscribedCustomerLabel);

found = true;

}

}

// Do the same for associate customers

for (Customer customer : database.getAssociateCustomers()) {

if (customer.getMagazines() != null && customer.getMagazines().contains(newValue)) {

Label subscribedCustomerLabel = new Label(customer.getName());

infoPanel.getChildren().add(subscribedCustomerLabel);

found = true;

}

}

// If no customers are found, display a message indicating that

if (!found) {

infoPanel.getChildren().add(new Label("No customers are subscribed to this magazine."));

}

}

});

primaryStage.setOnCloseRequest(e -> saveDataToFile(database));

}

/\*\*

\* Displays the billing history for a customer. If the customer is not an

\* instance of {@link PayingCustomer}, the method returns without displaying

\* anything.

\*

\* @param customer The customer whose billing history is to be displayed.

\*/

private void displayBillingHistory(Customer customer) {

if (!(customer instanceof PayingCustomer)) {

return; // Optionally handle this case

}

// multithreading

Task<BillingInfo> task = new Task<>() {

@Override

protected BillingInfo call() {

return ((PayingCustomer) customer).generateComprehensiveBillingInfo();

}

};

task.setOnSucceeded(e -> {

BillingInfo billingInfo = task.getValue();

Platform.runLater(() -> {

Stage stage = new Stage();

stage.initModality(Modality.WINDOW\_MODAL);

// Layout for billing details

VBox detailsBox = new VBox(10);

ScrollPane scrollPane = new ScrollPane(detailsBox);

scrollPane.setFitToWidth(true);

detailsBox.getChildren().add(new Label("Billing Breakdown:"));

for (String detail : billingInfo.getDetails()) {

detailsBox.getChildren().add(new Label(detail));

}

// Layout for cost summary

VBox costBox = new VBox(10);

costBox.setPadding(new Insets(10));

costBox.getChildren().add(new Label("Cost Summary:"));

float totalCost = billingInfo.getTotalCost();

costBox.getChildren().add(new Label(String.format("Weekly Cost: $%.2f", totalCost)));

costBox.getChildren().add(new Label(String.format("Monthly Cost: $%.2f", totalCost \* 4))); // Assuming monthly cost is 4 times weekly cost

// Main layout container

HBox mainContainer = new HBox(20, scrollPane, costBox);

mainContainer.setAlignment(Pos.CENTER\_LEFT);

mainContainer.setPadding(new Insets(10));

Scene scene = new Scene(mainContainer);

stage.setScene(scene);

stage.setTitle("Billing History for: " + billingInfo.getCustomerName());

stage.setWidth(600); // Adjust width and height as needed

stage.setHeight(400);

stage.show();

});

});

new Thread(task).start();

}

/\*\*

\* Loads data from a file and returns a {@link Database} object. If the data

\* file is not found or loading fails, a new {@link Database} object is

\* returned.

\*

\* @return The loaded or new {@link Database} object.

\*/

private Database loadDataFromFile() {

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(DATA\_FILE))) {

return (Database) ois.readObject();

} catch (FileNotFoundException e) {

System.out.println("Data file not found. Creating a new database.");

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

return new Database(); // Return a new database if loading fails

}

/\*\*

\* Saves the provided {@link Database} object to a file.

\*

\* @param database The {@link Database} object to be saved.

\*/

private void saveDataToFile(Database database) {

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(DATA\_FILE))) {

oos.writeObject(database);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* Creates and returns a panel for creating new entities (magazines,

\* supplements, customers).

\*

\* @param primaryStage The primary stage of the application.

\* @return A {@link VBox} containing buttons for creating new entities.

\*/

private VBox createModePanel(Stage primaryStage) {

// Buttons

Button createMagazineBtn = new Button("Create Magazine");

Button createSupplementBtn = new Button("Create Supplement");

Button createCustomerBtn = new Button("Create Customer");

// Event Handlers

createMagazineBtn.setOnAction(e -> showCreateMagazineWindow());

createSupplementBtn.setOnAction(e -> showCreateSupplementWindow());

createCustomerBtn.setOnAction(e -> showCreateCustomerWindow(primaryStage));

// VBox with buttons

VBox panel = new VBox(10); // Spacing of 10

panel.getChildren().add(new Label("Create Panel"));

panel.getChildren().addAll(createMagazineBtn, createSupplementBtn, createCustomerBtn);

panel.setAlignment(Pos.CENTER); // Center the buttons

// The VBox will take up all available space

panel.setFillWidth(true);

// If you want margins around the VBox itself (optional)

VBox.setMargin(panel, new Insets(20)); // Margin of 20

// Set the panel to grow vertically to push the buttons to the center

VBox.setVgrow(panel, Priority.ALWAYS);

panel.setVisible(false); // Initially invisible

return panel;

}

/\*\*

\* Creates and returns a panel for editing existing entities (magazines,

\* supplements, customers).

\*

\* @return A {@link VBox} containing buttons for editing existing entities.

\*/

private VBox editModePanel() {

VBox panel = new VBox(10);

panel.setAlignment(Pos.CENTER);

panel.setPadding(new Insets(10));

Button editSupplementBtn = new Button("Edit Supplement");

Button editCustomerBtn = new Button("Edit Customer");

Button editMagazineBtn = new Button("Edit Magazine"); // New button for editing magazines

editSupplementBtn.setOnAction(e -> showEditSupplementWindow());

editCustomerBtn.setOnAction(e -> showEditCustomerWindow());

editMagazineBtn.setOnAction(e -> showEditMagazineWindow()); // Set the action for the new button

panel.getChildren().addAll(new Label("Edit Panel"), editSupplementBtn, editCustomerBtn, editMagazineBtn);

return panel;

}

/\*\*

\* Displays a window for editing details of a selected magazine.

\*

\* @param magazine The magazine to be edited.

\*/

private void showEditMagazineWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Magazine");

VBox layout = new VBox(10);

layout.setAlignment(Pos.CENTER);

layout.setPadding(new Insets(20));

// Magazine selection

ListView<Magazine> magazinesListView = setupMagazinesListView();

Button editDetailsBtn = new Button("Edit Selected Magazine");

Button deleteMagazineBtn = new Button("Delete Selected Magazine");

editDetailsBtn.setOnAction(e -> {

Magazine selectedMagazine = magazinesListView.getSelectionModel().getSelectedItem();

if (selectedMagazine != null) {

showEditSelectedMagazineWindow(selectedMagazine); // This will open a new window for editing the selected magazine

} else {

showAlert("Edit Action", "No magazine selected.");

}

});

deleteMagazineBtn.setOnAction(e -> {

Magazine selectedMagazine = magazinesListView.getSelectionModel().getSelectedItem();

if (selectedMagazine != null) {

database.removeMagazine(selectedMagazine); // Assuming you have a method in your Database class for this

magazinesListView.getItems().remove(selectedMagazine);

showAlert("Deletion", "Magazine deleted successfully.");

} else {

showAlert("Delete Action", "No magazine selected.");

}

});

layout.getChildren().addAll(new Label("Select a Magazine"), magazinesListView, editDetailsBtn, deleteMagazineBtn);

window.setScene(new Scene(layout, 400, 600));

window.showAndWait();

}

/\*\*

\* Displays a window to edit the details of a selected magazine.

\*

\* @param magazine The magazine object to be edited.

\*/

private void showEditSelectedMagazineWindow(Magazine magazine) {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Magazine");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

TextField magazineNameField = new TextField(magazine.getMagazineName());

TextField magazineCostField = new TextField(String.valueOf(magazine.getMagazineCost()));

ListView<Supplement> currentSupplementsView = new ListView<>(FXCollections.observableArrayList(magazine.getMagazineSupplement()));

ListView<Supplement> availableSupplementsView = new ListView<>();

availableSupplementsView.setItems(FXCollections.observableArrayList(database.getSupplements().stream()

.filter(supplement -> !magazine.getMagazineSupplement().contains(supplement))

.collect(Collectors.toList())));

Button addSupplementButton = new Button("Add Supplement");

Button removeSupplementButton = new Button("Remove Supplement");

Button saveButton = new Button("Save");

addSupplementButton.setOnAction(e -> {

Supplement selected = availableSupplementsView.getSelectionModel().getSelectedItem();

if (selected != null) {

magazine.addSupplement(selected);

currentSupplementsView.getItems().add(selected);

availableSupplementsView.getItems().remove(selected);

}

});

removeSupplementButton.setOnAction(e -> {

Supplement selected = currentSupplementsView.getSelectionModel().getSelectedItem();

if (selected != null) {

magazine.removeSupplement(selected);

currentSupplementsView.getItems().remove(selected);

availableSupplementsView.getItems().add(selected);

}

});

saveButton.setOnAction(e -> {

magazine.setMagazineName(magazineNameField.getText());

magazine.setMagazineCost(Float.parseFloat(magazineCostField.getText()));

// Assume database.updateMagazine(magazine) updates the magazine in the database

//database.updateMagazine(magazine);

window.close();

});

layout.add(new Label("Magazine Name:"), 0, 0);

layout.add(magazineNameField, 1, 0);

layout.add(new Label("Magazine Cost:"), 0, 1);

layout.add(magazineCostField, 1, 1);

layout.add(new Label("Current Supplements:"), 0, 2);

layout.add(currentSupplementsView, 1, 2);

layout.add(removeSupplementButton, 2, 2);

layout.add(new Label("Available Supplements:"), 0, 3);

layout.add(availableSupplementsView, 1, 3);

layout.add(addSupplementButton, 2, 3);

layout.add(saveButton, 1, 4);

Scene scene = new Scene(layout, 500, 400);

window.setScene(scene);

window.showAndWait();

}

/\*\*

\* Sets up and returns a ListView containing magazines fetched from the

\* database.

\*

\* @return ListView containing Magazine objects.

\*/

private ListView<Magazine> setupMagazinesListView() {

ListView<Magazine> magazinesListView = new ListView<>();

magazinesListView.getItems().addAll(database.getMagazines());

magazinesListView.setCellFactory(lv -> new ListCell<Magazine>() {

@Override

protected void updateItem(Magazine item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getMagazineName()); // Assuming Magazine has a getMagazineName method

}

});

return magazinesListView;

}

/\*\*

\* Displays a window to edit the details of a selected supplement.

\*

\* @param supplement The supplement object to be edited.

\*/

private void showEditSelectedSupplementWindow(Supplement supplement) {

Stage editWindow = new Stage();

editWindow.initModality(Modality.APPLICATION\_MODAL);

editWindow.setTitle("Edit Supplement");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

TextField supplementNameField = new TextField(supplement.getSupplementName());

TextField supplementPriceField = new TextField(String.valueOf(supplement.getSupplementCost()));

Button saveButton = new Button("Save");

saveButton.setOnAction(e -> {

String newName = supplementNameField.getText();

try {

float newPrice = Float.parseFloat(supplementPriceField.getText());

supplement.setSupplementName(newName);

supplement.setSupplementCost(newPrice);

// Here you might want to update the database or list view accordingly

supplementsList.refresh(); // If ListView displays Supplement objects directly

editWindow.close();

} catch (NumberFormatException ex) {

System.out.println("Invalid price format. Please enter a valid number.");

}

});

layout.add(new Label("Name:"), 0, 0);

layout.add(supplementNameField, 1, 0);

layout.add(new Label("Price ($):"), 0, 1);

layout.add(supplementPriceField, 1, 1);

layout.add(saveButton, 1, 2);

editWindow.setScene(new Scene(layout, 300, 200));

editWindow.showAndWait();

}

/\*\*

\* Displays a window to edit the details of a supplement.

\*/

private void showEditSupplementWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Supplement");

VBox layout = new VBox(10);

layout.setAlignment(Pos.CENTER);

layout.setPadding(new Insets(20));

ListView<Supplement> supplementsListView = setupSupplementsListView();

Button editDetailsBtn = new Button("Edit Selected Supplement");

Button deleteSupplementBtn = new Button("Delete Selected Supplement");

// Handling edit action

editDetailsBtn.setOnAction(e -> handleEditAction(supplementsListView));

// Handling delete action

deleteSupplementBtn.setOnAction(e -> handleDeleteAction(supplementsListView));

layout.getChildren().addAll(new Label("Select a Supplement"), supplementsListView, editDetailsBtn, deleteSupplementBtn);

window.setScene(new Scene(layout, 300, 400));

window.showAndWait();

}

/\*\*

\* Sets up and returns a ListView containing supplements fetched from the

\* database.

\*

\* @return ListView containing Supplement objects.

\*/

private ListView<Supplement> setupSupplementsListView() {

ListView<Supplement> supplementsListView = new ListView<>();

supplementsListView.getItems().addAll(database.getSupplements());

supplementsListView.setCellFactory(lv -> new ListCell<Supplement>() {

@Override

protected void updateItem(Supplement item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getSupplementName());

}

});

return supplementsListView;

}

/\*\*

\* Handles the edit action for a selected supplement.

\*

\* @param listView ListView containing supplements.

\*/

private void handleEditAction(ListView<Supplement> listView) {

Supplement selectedSupplement = listView.getSelectionModel().getSelectedItem();

if (selectedSupplement != null) {

showEditSelectedSupplementWindow(selectedSupplement);

} else {

showAlert("Edit Action", "No supplement selected.");

}

}

/\*\*

\* Handles the delete action for a selected supplement.

\*

\* @param listView ListView containing supplements.

\*/

private void handleDeleteAction(ListView<Supplement> listView) {

Supplement selectedSupplement = listView.getSelectionModel().getSelectedItem();

if (selectedSupplement != null) {

if (confirmDeletion()) {

database.removeSupplement(selectedSupplement);

listView.getItems().remove(selectedSupplement);

showAlert("Deletion", "Supplement deleted successfully.");

}

} else {

showAlert("Delete Action", "No supplement selected.");

}

refreshSupplementsList();

}

/\*\*

\* Displays a confirmation dialog for deletion and returns the user's

\* choice.

\*

\* @return True if deletion is confirmed, false otherwise.

\*/

private boolean confirmDeletion() {

Alert alert = new Alert(Alert.AlertType.CONFIRMATION, "Are you sure you want to delete the selected supplement?", ButtonType.YES, ButtonType.NO);

Optional<ButtonType> result = alert.showAndWait();

return result.isPresent() && result.get() == ButtonType.YES;

}

/\*\*

\* Displays an alert dialog with the given title and message.

\*

\* @param title The title of the alert.

\* @param message The message to be displayed in the alert.

\*/

private void showAlert(String title, String message) {

Alert alert = new Alert(Alert.AlertType.INFORMATION);

alert.setTitle(title);

alert.setHeaderText(null);

alert.setContentText(message);

alert.showAndWait();

}

/\*\*

\* Displays a window to edit customer details.

\*/

private void showEditCustomerWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Customer");

VBox layout = new VBox(10);

layout.setAlignment(Pos.CENTER);

layout.setPadding(new Insets(20));

Button editDetailsBtn = new Button("Edit Details");

Button removeCustomerBtn = new Button("Remove Customer from Magazine");

Button addAssociateCustomerBtn = new Button("Add Associate Customer to Paying Customer");

Button removeAssociateBtn = new Button("Remove Associate from Paying Customer");

// Action handlers - these will call other methods to perform the actual operations

editDetailsBtn.setOnAction(e -> editCustomerDetails());

removeCustomerBtn.setOnAction(e -> removeCustomerFromMagazine());

addAssociateCustomerBtn.setOnAction(e -> addAssociateCustomerToPayingCustomer());

removeAssociateBtn.setOnAction(e -> showRemoveAssociateWindow());

layout.getChildren().addAll(new Label("Edit Customer"), editDetailsBtn, removeCustomerBtn, addAssociateCustomerBtn);

layout.getChildren().add(removeAssociateBtn);

window.setScene(new Scene(layout, 350, 250));

window.showAndWait();

refreshMagazinesList();

}

/\*\*

\* Displays a window to remove an associate customer from a paying customer.

\*/

private void showRemoveAssociateWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Remove Associate Customer");

VBox layout = new VBox(10);

layout.setAlignment(Pos.CENTER);

layout.setPadding(new Insets(20));

ComboBox<PayingCustomer> payingCustomerComboBox = new ComboBox<>();

payingCustomerComboBox.setItems(FXCollections.observableArrayList(database.getPayingCustomers()));

payingCustomerComboBox.setPromptText("Select Paying Customer");

ListView<Customer> associateListView = new ListView<>();

associateListView.setPrefHeight(150);

payingCustomerComboBox.getSelectionModel().selectedItemProperty().addListener((obs, oldVal, newVal) -> {

if (newVal != null) {

associateListView.setItems(FXCollections.observableArrayList(newVal.getAssociateCustomers()));

}

});

Button removeBtn = new Button("Remove Selected Associate");

removeBtn.setOnAction(e -> {

Customer selectedAssociate = associateListView.getSelectionModel().getSelectedItem();

PayingCustomer selectedPayingCustomer = payingCustomerComboBox.getSelectionModel().getSelectedItem();

if (selectedAssociate != null && selectedPayingCustomer != null) {

selectedPayingCustomer.getAssociateCustomers().remove(selectedAssociate);

database.getAssociateCustomers().add(selectedAssociate); // Assuming you have a method to add back to the database

associateListView.getItems().remove(selectedAssociate);

}

});

layout.getChildren().addAll(new Label("Select Paying Customer and Associate to Remove"), payingCustomerComboBox, associateListView, removeBtn);

window.setScene(new Scene(layout, 350, 300));

window.showAndWait();

refreshCustomersList();

}

/\*\*

\* Edits the details of a customer.

\*/

private void editCustomerDetails() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Customer Details");

BorderPane borderPane = new BorderPane();

// Left Panel: Customer List

VBox leftPanel = new VBox(10);

leftPanel.setPadding(new Insets(10));

ListView<Customer> customerListView = new ListView<>();

ObservableList<Customer> customers = FXCollections.observableArrayList(

Stream.concat(database.getPayingCustomers().stream(), database.getAssociateCustomers().stream())

.collect(Collectors.toList()));

customerListView.setItems(customers);

customerListView.setPrefWidth(200);

customerListView.setCellFactory(param -> new ListCell<>() {

@Override

protected void updateItem(Customer item, boolean empty) {

super.updateItem(item, empty);

setText(empty ? null : item.getName());

}

});

leftPanel.getChildren().add(new Label("Select Customer:"));

leftPanel.getChildren().add(customerListView);

borderPane.setLeft(leftPanel);

// Right Panel: Customer Details

VBox rightPanel = new VBox(10);

rightPanel.setPadding(new Insets(10));

TextField nameField = new TextField();

TextField emailField = new TextField();

TextField bankField = new TextField();

ComboBox<PayingCustomer.PaymentMethod> paymentMethodComboBox = new ComboBox<>();

paymentMethodComboBox.getItems().addAll(PayingCustomer.PaymentMethod.values());

rightPanel.getChildren().addAll(new Label("Name:"), nameField, new Label("Email:"), emailField);

// Bottom Buttons

Button saveButton = new Button("Save");

Button editMagazinesAndSupplementsButton = new Button("Edit Magazines and Supplements");

HBox buttonPanel = new HBox(10, saveButton, editMagazinesAndSupplementsButton);

buttonPanel.setAlignment(Pos.CENTER);

Label bankLabel = new Label("Bank:");

Label paymentMethodLabel = new Label("Payment Method:");

bankField.setVisible(false);

paymentMethodComboBox.setVisible(false);

bankLabel.setVisible(false);

paymentMethodLabel.setVisible(false);

// Add them to the right panel initially but set them as invisible

rightPanel.getChildren().addAll(bankLabel, bankField, paymentMethodLabel, paymentMethodComboBox);

// Selection listener to populate and adjust fields based on customer type

customerListView.getSelectionModel().selectedItemProperty().addListener((observable, oldValue, selectedCustomer) -> {

if (selectedCustomer != null) {

nameField.setText(selectedCustomer.getName());

emailField.setText(selectedCustomer.getEmail());

if (selectedCustomer instanceof PayingCustomer) {

PayingCustomer payingCustomer = (PayingCustomer) selectedCustomer;

bankField.setText(payingCustomer.getBank());

paymentMethodComboBox.setValue(payingCustomer.getSelectedPaymentMethod());

// Make bank details and payment method fields visible

bankField.setVisible(true);

paymentMethodComboBox.setVisible(true);

bankLabel.setVisible(true);

paymentMethodLabel.setVisible(true);

} else {

// Hide bank details and payment method fields for non-paying customers

bankField.setVisible(false);

paymentMethodComboBox.setVisible(false);

bankLabel.setVisible(false);

paymentMethodLabel.setVisible(false);

// Clear any previously entered/selected data

bankField.clear();

paymentMethodComboBox.getSelectionModel().clearSelection();

}

} else {

// Clear all fields if no customer is selected and hide bank/payment method fields

nameField.clear();

emailField.clear();

bankField.clear();

paymentMethodComboBox.getSelectionModel().clearSelection();

bankField.setVisible(false);

paymentMethodComboBox.setVisible(false);

bankLabel.setVisible(false);

paymentMethodLabel.setVisible(false);

}

});

editMagazinesAndSupplementsButton.setOnAction(e -> {

Customer selectedCustomer = customerListView.getSelectionModel().getSelectedItem();

if (selectedCustomer != null) {

openMagazineAndSupplementEditor(selectedCustomer);

}

});

saveButton.setOnAction(e -> {

Customer selectedCustomer = customerListView.getSelectionModel().getSelectedItem();

if (selectedCustomer != null) {

// Update the selected customer's properties

selectedCustomer.setName(nameField.getText());

selectedCustomer.setEmail(emailField.getText());

// If the customer is a PayingCustomer, update bank and payment method as well

if (selectedCustomer instanceof PayingCustomer) {

PayingCustomer payingCustomer = (PayingCustomer) selectedCustomer;

payingCustomer.setBank(bankField.getText());

payingCustomer.setSelectedPaymentMethod(paymentMethodComboBox.getValue());

}

// Here you should add logic to persist these changes to your database or data model.

// For example, if you have a method in your database class to update a customer, call it here.

// database.updateCustomer(selectedCustomer);

// Refresh the customer list view to reflect changes

customerListView.refresh();

// Optionally, close the edit window or show a confirmation message

// window.close();

// Show a confirmation dialog or message to the user

}

});

rightPanel.getChildren().add(buttonPanel);

borderPane.setCenter(rightPanel);

Scene scene = new Scene(borderPane, 600, 400);

window.setScene(scene);

window.showAndWait();

}

/\*\*

\* Opens a window to edit magazines and supplements for a specific customer.

\*

\* @param customer The customer object for whom magazines and supplements

\* are to be edited.

\*/

private void openMagazineAndSupplementEditor(Customer customer) {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Edit Magazines and Supplements for " + customer.getName());

BorderPane borderPane = new BorderPane();

VBox mainLayout = new VBox(10);

mainLayout.setPadding(new Insets(10));

// Displaying customer name

Label customerNameLabel = new Label("Editing for: " + customer.getName());

customerNameLabel.setFont(new Font("Arial", 16));

// Customer's current Magazines and Supplements

ListView<Magazine> customerMagazinesView = new ListView<>(FXCollections.observableArrayList(customer.getMagazines()));

ListView<Supplement> customerSupplementsView = new ListView<>(FXCollections.observableArrayList(customer.getSupplements()));

// Database Magazines and Supplements excluding customer's

List<Magazine> availableMagazines = database.getMagazines().stream()

.filter(mag -> !customer.getMagazines().contains(mag))

.collect(Collectors.toList());

ListView<Magazine> availableMagazinesView = new ListView<>(FXCollections.observableArrayList(availableMagazines));

List<Supplement> availableSupplements = database.getSupplements().stream()

.filter(sup -> !customer.getSupplements().contains(sup))

.collect(Collectors.toList());

ListView<Supplement> availableSupplementsView = new ListView<>(FXCollections.observableArrayList(availableSupplements));

// Buttons for adding and removing magazines/supplements

Button addMagazineButton = new Button("Add Magazine");

Button removeMagazineButton = new Button("Remove Magazine");

Button addSupplementButton = new Button("Add Supplement");

Button removeSupplementButton = new Button("Remove Supplement");

// Action Handlers for Buttons

addMagazineButton.setOnAction(event -> {

Magazine selected = availableMagazinesView.getSelectionModel().getSelectedItem();

if (selected != null) {

customer.addMagazine(selected);

customerMagazinesView.getItems().add(selected);

availableMagazinesView.getItems().remove(selected);

}

});

removeMagazineButton.setOnAction(event -> {

Magazine selected = customerMagazinesView.getSelectionModel().getSelectedItem();

if (selected != null) {

customer.removeMagazine(selected);

customerMagazinesView.getItems().remove(selected);

availableMagazinesView.getItems().add(selected);

}

});

addSupplementButton.setOnAction(event -> {

Supplement selected = availableSupplementsView.getSelectionModel().getSelectedItem();

if (selected != null) {

customer.addSupplement(selected);

customerSupplementsView.getItems().add(selected);

availableSupplementsView.getItems().remove(selected);

}

});

removeSupplementButton.setOnAction(event -> {

Supplement selected = customerSupplementsView.getSelectionModel().getSelectedItem();

if (selected != null) {

customer.removeSupplement(selected);

customerSupplementsView.getItems().remove(selected);

availableSupplementsView.getItems().add(selected);

}

});

// Layout setup

HBox buttonLayoutMagazine = new HBox(10, addMagazineButton, removeMagazineButton);

VBox magazineLayout = new VBox(5, new Label("Magazines"), customerMagazinesView, availableMagazinesView, buttonLayoutMagazine);

HBox buttonLayoutSupplement = new HBox(10, addSupplementButton, removeSupplementButton);

VBox supplementLayout = new VBox(5, new Label("Supplements"), customerSupplementsView, availableSupplementsView, buttonLayoutSupplement);

HBox contentLayout = new HBox(10, magazineLayout, supplementLayout);

mainLayout.getChildren().addAll(customerNameLabel, contentLayout);

borderPane.setCenter(mainLayout);

Scene scene = new Scene(borderPane, 800, 600);

window.setScene(scene);

window.showAndWait();

}

/\*\*

\* Removes a customer from a magazine.

\*/

private void removeCustomerFromMagazine() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Remove Customer from Magazine");

// Use a BorderPane for overall layout

BorderPane borderPane = new BorderPane();

// Customer List Panel

VBox customerListPanel = new VBox(10);

ListView<Customer> customerListView = new ListView<>();

ObservableList<Customer> customers = FXCollections.observableArrayList();

customers.addAll(database.getPayingCustomers());

customers.addAll(database.getAssociateCustomers());

customerListView.setItems(customers);

customerListView.setCellFactory(param -> new ListCell<>() {

@Override

protected void updateItem(Customer item, boolean empty) {

super.updateItem(item, empty);

if (empty || item == null) {

setText(null);

} else {

setText(item.getName());

}

}

});

customerListPanel.getChildren().addAll(new Label("Select Customer:"), customerListView);

// Magazine List Panel (empty initially)

VBox magazineListPanel = new VBox(10);

ListView<Magazine> magazineListView = new ListView<>();

magazineListView.setCellFactory(param -> new ListCell<Magazine>() {

@Override

protected void updateItem(Magazine magazine, boolean empty) {

super.updateItem(magazine, empty);

if (empty || magazine == null) {

setText(null);

} else {

setText(magazine.getMagazineName()); // Assuming Magazine has a getMagazineName() method

}

}

});

magazineListPanel.getChildren().addAll(new Label("Customer's Magazines:"), magazineListView);

Button removeMagazineButton = new Button("Remove Selected Magazine");

magazineListPanel.getChildren().add(removeMagazineButton);

// When a customer is selected, update the magazine list

customerListView.getSelectionModel().selectedItemProperty().addListener((observable, oldValue, newValue) -> {

if (newValue != null) {

magazineListView.setItems(FXCollections.observableArrayList(newValue.getMagazines()));

}

});

// Handle magazine removal

removeMagazineButton.setOnAction(e -> {

Magazine selectedMagazine = magazineListView.getSelectionModel().getSelectedItem();

if (selectedMagazine != null) {

Customer selectedCustomer = customerListView.getSelectionModel().getSelectedItem();

selectedCustomer.removeMagazine(selectedMagazine); // Assumes removeMagazine method exists

magazineListView.getItems().remove(selectedMagazine);

}

});

Button saveChangesButton = new Button("Save Changes");

saveChangesButton.setOnAction(e -> {

// Implement logic to save changes, such as updating the database or list views

window.close();

});

HBox saveButtonBox = new HBox(10, saveChangesButton);

saveButtonBox.setAlignment(Pos.CENTER);

borderPane.setLeft(customerListPanel);

borderPane.setCenter(magazineListPanel);

borderPane.setBottom(saveButtonBox);

Scene scene = new Scene(borderPane, 600, 400);

window.setScene(scene);

window.showAndWait();

}

/\*\*

\* Adds associate customers to a paying customer.

\*/

private void addAssociateCustomerToPayingCustomer() {

// Fetch paying customers for the user to choose from

List<PayingCustomer> payers = database.getPayingCustomers();

ChoiceDialog<PayingCustomer> payerDialog = new ChoiceDialog<>(null, payers);

payerDialog.setTitle("Select Paying Customer");

payerDialog.setHeaderText("Select a paying customer to add associates");

payerDialog.setContentText("Choose paying customer:");

Optional<PayingCustomer> payerResult = payerDialog.showAndWait();

if (!payerResult.isPresent()) {

return; // Exit if no paying customer is selected

}

PayingCustomer selectedPayingCustomer = payerResult.get();

// Create a custom dialog for selecting associate customers

Dialog<List<Customer>> dialog = new Dialog<>();

dialog.setTitle("Add Associate Customers");

dialog.setHeaderText("Select associate customers to add to " + selectedPayingCustomer.getName());

// Set up the buttons

ButtonType addButtonType = new ButtonType("Add", ButtonBar.ButtonData.OK\_DONE);

dialog.getDialogPane().getButtonTypes().addAll(addButtonType, ButtonType.CANCEL);

// Create a container for checkboxes

VBox checkBoxContainer = new VBox();

checkBoxContainer.setSpacing(10);

// Populate the container with checkboxes, one for each associate customer

List<CheckBox> checkboxes = new ArrayList<>();

for (Customer associate : database.getAssociateCustomers()) {

CheckBox checkBox = new CheckBox(associate.getName());

checkBox.setUserData(associate); // Associate the actual Customer object with the checkbox

checkboxes.add(checkBox);

checkBoxContainer.getChildren().add(checkBox);

}

dialog.getDialogPane().setContent(checkBoxContainer);

// Convert the result to a list of selected customers when the add button is clicked

dialog.setResultConverter(dialogButton -> {

if (dialogButton == addButtonType) {

return checkboxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> (Customer) cb.getUserData())

.collect(Collectors.toList());

}

return null;

});

Optional<List<Customer>> result = dialog.showAndWait();

result.ifPresent(selectedAssociates -> {

// Add selected associates to the paying customer and remove them from the database

for (Customer associate : selectedAssociates) {

selectedPayingCustomer.addAssociateCustomer(associate);

database.removeAssociateCustomer(associate); // Assuming this method correctly removes the associate from the database

}

// Optionally refresh your UI here to reflect changes

});

refreshCustomersList();

}

/\*\*

\* Shows a window to create a new magazine.

\*/

private void showCreateMagazineWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Create Magazine");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

TextField magazineNameField = new TextField();

magazineNameField.setPromptText("Magazine Name");

TextField magazineCostField = new TextField();

magazineCostField.setPromptText("Magazine Cost");

// Creating a VBox to hold checkboxes for each supplement

VBox checkBoxContainer = new VBox();

checkBoxContainer.setSpacing(5); // Set spacing between checkboxes

// Dynamically creating a CheckBox for each supplement

List<CheckBox> supplementCheckBoxes = database.getSupplements().stream()

.map(supplement -> new CheckBox(supplement.getSupplementName()))

.collect(Collectors.toList());

// Adding checkboxes to the VBox container

checkBoxContainer.getChildren().addAll(supplementCheckBoxes);

Button createMagazineButton = new Button("Create Magazine");

createMagazineButton.setOnAction(e -> {

String magazineName = magazineNameField.getText();

float magazineCost;

try {

magazineCost = Float.parseFloat(magazineCostField.getText());

} catch (NumberFormatException ex) {

System.out.println("Invalid magazine cost. Please enter a valid number.");

return; // Exit the method if the cost is not a valid number

}

// Collecting selected supplements based on CheckBox selections

List<Supplement> selectedSupplements = supplementCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(checkBox -> database.getSupplements().stream()

.filter(supplement -> supplement.getSupplementName().equals(checkBox.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

Magazine newMagazine = new Magazine(magazineName, magazineCost, selectedSupplements);

database.addMagazine(newMagazine);

window.close();

});

layout.add(new Label("Magazine Name:"), 0, 0);

layout.add(magazineNameField, 1, 0);

layout.add(new Label("Magazine Cost:"), 0, 1);

layout.add(magazineCostField, 1, 1);

layout.add(new Label("Select Supplements:"), 0, 2);

// Setting constraints for the VBox to ensure it aligns properly

GridPane.setConstraints(checkBoxContainer, 1, 2);

layout.getChildren().addAll(checkBoxContainer);

layout.add(createMagazineButton, 1, 3);

window.setScene(new Scene(layout, 400, 400)); // Adjust window size as needed

window.showAndWait();

refreshMagazinesList();

}

/\*\*

\* Shows a window to create a new supplement.

\*/

private void showCreateSupplementWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Create Supplement");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

// Supplement name input

TextField supplementNameField = new TextField();

supplementNameField.setPromptText("Supplement Name");

// Supplement cost input

TextField supplementCostField = new TextField();

supplementCostField.setPromptText("Supplement Cost");

// Create button

Button createSupplementButton = new Button("Create Supplement");

createSupplementButton.setOnAction(e -> {

try {

String supplementName = supplementNameField.getText();

float supplementCost = Float.parseFloat(supplementCostField.getText());

Supplement newSupplement = new Supplement(supplementName, supplementCost);

database.addSupplement(newSupplement);

supplementsList.getItems().add(newSupplement); // Optionally refresh the list

window.close();

} catch (NumberFormatException ex) {

// Handle invalid cost input (e.g., show an error message)

System.out.println("Invalid supplement cost. Please enter a valid number.");

}

});

// Adding components to the layout

layout.add(new Label("Supplement Name:"), 0, 0);

layout.add(supplementNameField, 1, 0);

layout.add(new Label("Supplement Cost:"), 0, 1);

layout.add(supplementCostField, 1, 1);

layout.add(createSupplementButton, 1, 2);

window.setScene(new Scene(layout, 300, 200));

window.showAndWait();

}

/\*\*

\* Shows a window to create a new customer, with options to create either an

\* associate or paying customer.

\*

\* @param primaryStage The primary stage of the application.

\*/

private void showCreateCustomerWindow(Stage primaryStage) {

Stage window = new Stage();

window.initOwner(primaryStage);

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Create Customer");

VBox layout = new VBox(10);

layout.setAlignment(Pos.CENTER);

layout.setPadding(new Insets(20));

Button createAssociateCustomerBtn = new Button("Create Associate Customer");

Button createPayingCustomerBtn = new Button("Create Paying Customer");

createAssociateCustomerBtn.setOnAction(e -> showCreateAssociateCustomerWindow());

createPayingCustomerBtn.setOnAction(e -> showCreatePayingCustomerWindow());

layout.getChildren().addAll(createAssociateCustomerBtn, createPayingCustomerBtn);

window.setScene(new Scene(layout, 300, 200));

window.showAndWait();

}

/\*\*

\* Shows a window to create a new associate customer.

\*/

private void showCreateAssociateCustomerWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Create Associate Customer");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

TextField nameField = new TextField();

nameField.setPromptText("Name");

TextField emailField = new TextField();

emailField.setPromptText("Email");

// Supplement selection checkboxes

VBox supplementCheckBoxContainer = new VBox();

List<CheckBox> supplementCheckBoxes = database.getSupplements().stream()

.map(supplement -> new CheckBox(supplement.getSupplementName()))

.collect(Collectors.toList());

supplementCheckBoxContainer.getChildren().addAll(new Label("Select Supplements:"), new Separator());

supplementCheckBoxContainer.getChildren().addAll(supplementCheckBoxes);

// Magazine selection checkboxes

VBox magazineCheckBoxContainer = new VBox();

List<CheckBox> magazineCheckBoxes = database.getMagazines().stream()

.map(magazine -> new CheckBox(magazine.getMagazineName()))

.collect(Collectors.toList());

magazineCheckBoxContainer.getChildren().addAll(new Label("Select Magazines:"), new Separator());

magazineCheckBoxContainer.getChildren().addAll(magazineCheckBoxes);

Button createButton = new Button("Create");

createButton.setOnAction(e -> {

List<Supplement> selectedSupplements = supplementCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> database.getSupplements().stream()

.filter(supplement -> supplement.getSupplementName().equals(cb.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

List<Magazine> selectedMagazines = magazineCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> database.getMagazines().stream()

.filter(magazine -> magazine.getMagazineName().equals(cb.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

Customer newCustomer = new Customer(nameField.getText(), emailField.getText(), selectedMagazines, selectedSupplements);

database.addAssociateCustomer(newCustomer);

customersList.getItems().add(newCustomer); // Refresh the customer list if necessary

window.close();

});

layout.add(new Label("Name:"), 0, 0);

layout.add(nameField, 1, 0);

layout.add(new Label("Email:"), 0, 1);

layout.add(emailField, 1, 1);

layout.add(supplementCheckBoxContainer, 1, 2);

layout.add(magazineCheckBoxContainer, 1, 3);

layout.add(createButton, 1, 4);

window.setScene(new Scene(layout, 350, 400));

window.showAndWait();

}

/\*\*

\* Shows a window to create a new paying customer.

\*/

private void showCreatePayingCustomerWindow() {

Stage window = new Stage();

window.initModality(Modality.APPLICATION\_MODAL);

window.setTitle("Create Paying Customer");

GridPane layout = new GridPane();

layout.setAlignment(Pos.CENTER);

layout.setHgap(10);

layout.setVgap(10);

layout.setPadding(new Insets(20));

TextField nameField = new TextField();

nameField.setPromptText("Name");

TextField emailField = new TextField();

emailField.setPromptText("Email");

// Payment Method ComboBox

ComboBox<PayingCustomer.PaymentMethod> paymentMethodComboBox = new ComboBox<>();

paymentMethodComboBox.getItems().addAll(PayingCustomer.PaymentMethod.values());

paymentMethodComboBox.setValue(PayingCustomer.PaymentMethod.CREDIT\_CARD);

TextField bankField = new TextField();

bankField.setPromptText("Bank");

// Supplement selection checkboxes

VBox supplementCheckBoxContainer = new VBox();

List<CheckBox> supplementCheckBoxes = database.getSupplements().stream()

.map(supplement -> new CheckBox(supplement.getSupplementName()))

.collect(Collectors.toList());

supplementCheckBoxContainer.getChildren().addAll(new Label("Select Supplements:"), new Separator());

supplementCheckBoxContainer.getChildren().addAll(supplementCheckBoxes);

// Magazine selection checkboxes

VBox magazineCheckBoxContainer = new VBox();

List<CheckBox> magazineCheckBoxes = database.getMagazines().stream()

.map(magazine -> new CheckBox(magazine.getMagazineName()))

.collect(Collectors.toList());

magazineCheckBoxContainer.getChildren().addAll(new Label("Select Magazines:"), new Separator());

magazineCheckBoxContainer.getChildren().addAll(magazineCheckBoxes);

// Associate Customer selection checkboxes

VBox associateCustomerCheckBoxContainer = new VBox();

List<CheckBox> associateCustomerCheckBoxes = database.getAssociateCustomers().stream()

.map(customer -> new CheckBox(customer.getName()))

.collect(Collectors.toList());

associateCustomerCheckBoxContainer.getChildren().addAll(new Label("Select Associate Customers:"), new Separator());

associateCustomerCheckBoxContainer.getChildren().addAll(associateCustomerCheckBoxes);

Button createButton = new Button("Create");

createButton.setOnAction(e -> {

List<Supplement> selectedSupplements = supplementCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> database.getSupplements().stream()

.filter(supplement -> supplement.getSupplementName().equals(cb.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

List<Magazine> selectedMagazines = magazineCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> database.getMagazines().stream()

.filter(magazine -> magazine.getMagazineName().equals(cb.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

List<Customer> selectedAssociateCustomers = associateCustomerCheckBoxes.stream()

.filter(CheckBox::isSelected)

.map(cb -> database.getAssociateCustomers().stream()

.filter(customer -> customer.getName().equals(cb.getText()))

.findFirst().orElse(null))

.collect(Collectors.toList());

PayingCustomer newPayingCustomer = new PayingCustomer(

paymentMethodComboBox.getValue(),

bankField.getText(),

selectedAssociateCustomers,

nameField.getText(),

emailField.getText(),

selectedMagazines,

selectedSupplements

);

database.addPayingCustomer(newPayingCustomer);// Remove selected associate customers from the database

selectedAssociateCustomers.forEach(associateCustomer -> {

database.removeAssociateCustomer(associateCustomer); // Assuming this method exists in your Database class

customersList.getItems().remove(associateCustomer); // Optionally update the UI list if needed

});

customersList.getItems().add(newPayingCustomer); // Refresh the customer list if necessary

window.close();

});

layout.add(new Label("Name:"), 0, 0);

layout.add(nameField, 1, 0);

layout.add(new Label("Email:"), 0, 1);

layout.add(emailField, 1, 1);

layout.add(new Label("Payment Method:"), 0, 2);

layout.add(paymentMethodComboBox, 1, 2);

layout.add(new Label("Bank:"), 0, 3);

layout.add(bankField, 1, 3);

layout.add(supplementCheckBoxContainer, 0, 4, 2, 1);

layout.add(magazineCheckBoxContainer, 0, 5, 2, 1);

layout.add(associateCustomerCheckBoxContainer, 0, 6, 2, 1);

layout.add(createButton, 1, 7);

window.setScene(new Scene(layout, 350, 600)); // Adjust the size as necessary to fit all components

window.showAndWait();

}

/\*\*

\* Refreshes the supplements list view.

\*/

private void refreshSupplementsList() {

ObservableList<Supplement> updatedList = FXCollections.observableArrayList(database.getSupplements());

supplementsList.setItems(updatedList);

}

/\*\*

\* Refreshes the magazines list view.

\*/

private void refreshMagazinesList() {

ObservableList<Magazine> updatedList = FXCollections.observableArrayList(database.getMagazines());

// Assuming magazinesList is your ListView for displaying magazines

magazinesList.setItems(updatedList);

}

/\*\*

\* Refreshes the customers list view.

\*/

private void refreshCustomersList() {

ObservableList<Customer> updatedCustomers = FXCollections.observableArrayList();

updatedCustomers.addAll(database.getPayingCustomers());

updatedCustomers.addAll(database.getAssociateCustomers());

customersList.setItems(updatedCustomers);

}

/\*\*

\* Main method to launch the application.

\*

\* @param args Command-line arguments.

\*/

public static void main(String[] args) {

database.populate();

launch(args);

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

/\*\*

\* This class represents the main entry point of the Assignment2 application.

\*/

public class Assignment2 {

/\*\*

\* The shared database instance used throughout the application.

\*/

static Database database = new Database();

/\*\*

\* The main method of the Assignment2 application.

\*

\* @param args The command-line arguments.

\*/

public static void main(String[] args) {

database.populate();

handleChoice();

}

/\*\*

\* Handles user choices through a menu system, allowing users to perform

\* various actions in the application.

\*/

public static void handleChoice() {

while (true) {

switch (Display.menu()) {

case 1:

CreateMagazine.run(database);

break;

case 2:

CreateSupplement.run(database);

break;

case 3:

CreateCustomer.run(database);

break;

case 4:

AddAssociateToPayingCustomer.run(database);

break;

case 5:

FourWeekCustomerEmailPrint.run(database);

break;

case 6:

EndOfMonthPayingCustomerEmailPrint.run(database);

break;

case 7:

AddNewCustomerToMagazineService.run(database);

break;

case 8:

RemoveCustomerFromMagazineService.run(database);

break;

case 9:

AddSupplementToCustomerOrMagazine.run(database);

break;

case 10:

System.out.println("Exiting...");

return; // Exit the method and thus end the loop

default:

System.out.println("Invalid choice.");

break;

}

}

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* The BillingInfo class represents billing information associated with a

\* customer. It stores details of items purchased and their costs.

\*/

public class BillingInfo implements Serializable {

/\*\*

\* The name of the customer.

\*/

private final String customerName;

/\*\*

\* The email of the customer.

\*/

private final String customerEmail;

/\*\*

\* The list of details of items purchased.

\*/

private final List<String> details = new ArrayList<>();

/\*\*

\* The total cost of all items purchased.

\*/

private float totalCost = 0;

/\*\*

\* Constructs a BillingInfo object with the given customer name and email.

\*

\* @param customerName The name of the customer.

\* @param customerEmail The email of the customer.

\*/

public BillingInfo(String customerName, String customerEmail) {

this.customerName = customerName;

this.customerEmail = customerEmail;

}

/\*\*

\* Adds a detail of an item purchased along with its cost to the billing

\* information.

\*

\* @param detail The detail of the item.

\* @param cost The cost of the item.

\*/

public void addDetail(String detail, float cost) {

details.add(String.format("%s: $%.2f", detail, cost));

totalCost += cost;

}

/\*\*

\* Retrieves the name of the customer.

\*

\* @return The name of the customer.

\*/

public String getCustomerName() {

return customerName;

}

/\*\*

\* Retrieves the email of the customer.

\*

\* @return The email of the customer.

\*/

public String getCustomerEmail() {

return customerEmail;

}

/\*\*

\* Retrieves the list of details of items purchased.

\*

\* @return The list of details of items purchased.

\*/

public List<String> getDetails() {

return details;

}

/\*\*

\* Retrieves the total cost of all items purchased.

\*

\* @return The total cost of all items purchased.

\*/

public float getTotalCost() {

return totalCost;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.Scanner;

import java.util.List;

/\*\*

\* The CreateCustomer class provides methods to create different types of

\* customers. It allows the user to interactively create paying customers,

\* associate customers, and display lists of customers.

\*/

public class CreateCustomer {

/\*\*

\* Runs the customer creation process.

\*

\* @param database The database instance to interact with.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

while (true) {

System.out.println("Choose an action:");

System.out.println("1. Create Paying Customer");

System.out.println("2. Create Associate Customer");

System.out.println("3. Display Lists");

System.out.println("4. Quit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

scanner.nextLine(); // Consume newline

switch (choice) {

case 1:

createPayingCustomer(database, scanner);

break;

case 2:

createAssociateCustomer(database, scanner);

break;

case 3:

displayLists(database);

break;

case 4:

return;

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

/\*\*

\* Creates a new paying customer based on user input and adds it to the

\* database.

\*

\* @param database The database instance to interact with.

\* @param scanner The Scanner object to read user input.

\*/

private static void createPayingCustomer(Database database, Scanner scanner) {

PayingCustomer payingCustomer = new PayingCustomer();

System.out.print("Enter customer name: ");

payingCustomer.setName(scanner.nextLine());

System.out.print("Enter customer email: ");

payingCustomer.setEmail(scanner.nextLine());

// Set supplements to null for now

payingCustomer.setSupplements(null);

System.out.println("Choose payment method:");

System.out.println("1. Credit Card");

System.out.println("2. Bank Card");

System.out.print("Enter your choice: ");

int paymentChoice = scanner.nextInt();

scanner.nextLine(); // Consume newline

switch (paymentChoice) {

case 1:

payingCustomer.setSelectedPaymentMethod(PayingCustomer.PaymentMethod.CREDIT\_CARD);

break;

case 2:

payingCustomer.setSelectedPaymentMethod(PayingCustomer.PaymentMethod.BANK\_CARD);

break;

default:

System.out.println("Invalid choice. Defaulting to Credit Card.");

payingCustomer.setSelectedPaymentMethod(PayingCustomer.PaymentMethod.CREDIT\_CARD);

}

System.out.print("Enter bank: ");

payingCustomer.setBank(scanner.nextLine());

// Add paying customer to the database

database.addPayingCustomer(payingCustomer);

System.out.println("Paying customer created successfully.");

}

/\*\*

\* Creates a new associate customer based on user input and adds it to the

\* database.

\*

\* @param database The database instance to interact with.

\* @param scanner The Scanner object to read user input.

\*/

private static void createAssociateCustomer(Database database, Scanner scanner) {

Customer associateCustomer = new Customer();

System.out.print("Enter customer name: ");

associateCustomer.setName(scanner.nextLine());

System.out.print("Enter customer email: ");

associateCustomer.setEmail(scanner.nextLine());

// Set supplements to null for now

associateCustomer.setSupplements(null);

// Add associate customer to the database

database.addAssociateCustomer(associateCustomer);

System.out.println("Associate customer created successfully.");

}

/\*\*

\* Displays lists of paying customers and associate customers.

\*

\* @param database The database instance to interact with.

\*/

private static void displayLists(Database database) {

System.out.println("List of Paying Customers:");

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

System.out.println(payingCustomer.getName());

System.out.println("Associated Customers:");

List<Customer> associatedCustomers = payingCustomer.getAssociateCustomers();

if (associatedCustomers != null && !associatedCustomers.isEmpty()) {

for (Customer associate : associatedCustomers) {

System.out.println("- " + associate.getName());

}

} else {

System.out.println("No associated customers.");

}

System.out.println();

}

System.out.println("List of Associate Customers:");

List<Customer> associateCustomers = database.getAssociateCustomers();

if (associateCustomers != null && !associateCustomers.isEmpty()) {

for (Customer associateCustomer : associateCustomers) {

System.out.println(associateCustomer.getName());

System.out.println();

}

} else {

System.out.println("No associate customers found.");

}

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

/\*\*

\* The CreateMagazine class provides methods to interactively create new

\* magazines and add them to the database.

\*/

public class CreateMagazine {

/\*\*

\* Runs the process of creating a new magazine.

\*

\* @param database The database instance to interact with.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

List<Magazine> magazines = database.getMagazines();

displayExistingMagazines(magazines);

if (promptForNewMagazine(scanner)) {

String magazineName = promptForMagazineName(scanner);

float magazineCost = promptForMagazineCost(scanner);

List<Supplement> supplements = database.getSupplements();

List<Supplement> selectedSupplements = selectSupplements(scanner, supplements);

Magazine newMagazine = createMagazine(magazineName, magazineCost, selectedSupplements);

database.addMagazine(newMagazine);

System.out.println("New magazine added successfully.");

} else {

System.out.println("No new magazine added.");

}

}

/\*\*

\* Displays a list of existing magazines.

\*

\* @param magazines The list of existing magazines to display.

\*/

private static void displayExistingMagazines(List<Magazine> magazines) {

System.out.println("Existing Magazines:");

for (int i = 0; i < magazines.size(); i++) {

System.out.println((i + 1) + ". " + magazines.get(i).getMagazineName());

}

}

/\*\*

\* Prompts the user to indicate whether they want to create a new magazine.

\*

\* @param scanner The Scanner object to read user input.

\* @return True if the user wants to create a new magazine, false otherwise.

\*/

private static boolean promptForNewMagazine(Scanner scanner) {

System.out.print("Enter 'yes' if you want to create a new magazine: ");

String input = scanner.nextLine();

return input.equalsIgnoreCase("yes");

}

/\*\*

\* Prompts the user to enter the name of the new magazine.

\*

\* @param scanner The Scanner object to read user input.

\* @return The name of the new magazine entered by the user.

\*/

private static String promptForMagazineName(Scanner scanner) {

System.out.print("Enter the magazine name: ");

return scanner.nextLine();

}

/\*\*

\* Prompts the user to enter the cost of the new magazine.

\*

\* @param scanner The Scanner object to read user input.

\* @return The cost of the new magazine entered by the user.

\*/

private static float promptForMagazineCost(Scanner scanner) {

System.out.print("Enter the magazine cost: ");

float magazineCost = scanner.nextFloat();

scanner.nextLine(); // Consume the newline character

return magazineCost;

}

/\*\*

\* Allows the user to select supplements to include in the new magazine.

\*

\* @param scanner The Scanner object to read user input.

\* @param supplements The list of available supplements to choose from.

\* @return The list of selected supplements chosen by the user.

\*/

private static List<Supplement> selectSupplements(Scanner scanner, List<Supplement> supplements) {

System.out.println("Existing Supplements:");

for (int i = 0; i < supplements.size(); i++) {

System.out.println((i + 1) + ". " + supplements.get(i).getSupplementName());

}

System.out.print("Enter the numbers of supplements to include in the magazine (comma-separated): ");

String input = scanner.nextLine();

String[] supplementIndices = input.split(",");

List<Supplement> selectedSupplements = new ArrayList<>();

for (String index : supplementIndices) {

try {

int selectedIndex = Integer.parseInt(index.trim()) - 1;

if (selectedIndex >= 0 && selectedIndex < supplements.size()) {

selectedSupplements.add(supplements.get(selectedIndex));

} else {

System.out.println("Invalid supplement index: " + (selectedIndex + 1));

}

} catch (NumberFormatException e) {

System.out.println("Invalid input: " + index);

}

}

return selectedSupplements;

}

/\*\*

\* Creates a new Magazine object with the specified name, cost, and selected

\* supplements.

\*

\* @param magazineName The name of the new magazine.

\* @param magazineCost The cost of the new magazine.

\* @param selectedSupplements The list of selected supplements to include in

\* the magazine.

\* @return The newly created Magazine object.

\*/

private static Magazine createMagazine(String magazineName, float magazineCost, List<Supplement> selectedSupplements) {

return new Magazine(magazineName, magazineCost, selectedSupplements);

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.List;

import java.util.Scanner;

/\*\*

\* The CreateSupplement class provides methods to interactively create new

\* supplements and add them to the database.

\*/

public class CreateSupplement {

/\*\*

\* Runs the process of creating a new supplement.

\*

\* @param database The database instance to interact with.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

List<Supplement> supplements = database.getSupplements();

displayExistingSupplements(supplements);

if (promptToAddNewSupplement(scanner)) {

String supplementName = promptForSupplementName(scanner);

float supplementCost = promptForSupplementCost(scanner);

addNewSupplement(database, supplementName, supplementCost);

} else {

System.out.println("No new supplement added.");

}

}

/\*\*

\* Displays a list of existing supplements.

\*

\* @param supplements The list of existing supplements to display.

\*/

private static void displayExistingSupplements(List<Supplement> supplements) {

System.out.println("Existing Supplements:");

for (Supplement supplement : supplements) {

supplement.display();

}

}

/\*\*

\* Prompts the user to indicate whether they want to add a new supplement.

\*

\* @param scanner The Scanner object to read user input.

\* @return True if the user wants to add a new supplement, false otherwise.

\*/

private static boolean promptToAddNewSupplement(Scanner scanner) {

System.out.print("Do you want to add a new supplement? (yes/no): ");

String choice = scanner.nextLine();

return choice.equalsIgnoreCase("yes");

}

/\*\*

\* Prompts the user to enter the name of the new supplement.

\*

\* @param scanner The Scanner object to read user input.

\* @return The name of the new supplement entered by the user.

\*/

private static String promptForSupplementName(Scanner scanner) {

System.out.print("Enter the supplement name: ");

return scanner.nextLine();

}

/\*\*

\* Prompts the user to enter the cost of the new supplement.

\*

\* @param scanner The Scanner object to read user input.

\* @return The cost of the new supplement entered by the user.

\*/

private static float promptForSupplementCost(Scanner scanner) {

System.out.print("Enter the supplement cost: ");

return scanner.nextFloat();

}

/\*\*

\* Adds a new supplement to the database.

\*

\* @param database The database instance to add the new supplement to.

\* @param supplementName The name of the new supplement.

\* @param supplementCost The cost of the new supplement.

\*/

private static void addNewSupplement(Database database, String supplementName, float supplementCost) {

Supplement newSupplement = new Supplement(supplementName, supplementCost);

database.addSupplement(newSupplement);

System.out.println("New supplement added successfully.");

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import javafx.scene.Node;

import javafx.scene.control.Label;

import javafx.scene.layout.VBox;

/\*\*

\* The Customer class represents a customer who can subscribe to magazines and

\* supplements.

\*/

public class Customer implements Serializable {

private String name;// The name of the customer

private String email;// The email address of the customer.

private List<Magazine> magazines;// The list of magazines subscribed by the customer.

private List<Supplement> supplements;// The list of supplements subscribed by the customer.

/\*\*

\* Constructs a new Customer object with the specified name, email,

\* magazines, and supplements.

\*

\* @param name The name of the customer.

\* @param email The email address of the customer.

\* @param magazines The list of magazines subscribed by the customer.

\* @param supplements The list of supplements subscribed by the customer.

\*/

public Customer(String name, String email, List<Magazine> magazines, List<Supplement> supplements) {

this.name = name;

this.email = email;

this.magazines = magazines;

this.supplements = supplements;

}

/\*\*

\* Constructs a new Customer object.

\*/

public Customer() {

}

/\*\*

\* Returns the name of the customer.

\*

\* @return The name of the customer.

\*/

public String getName() {

return name;

}

/\*\*

\* Sets the name of the customer.

\*

\* @param name The name of the customer.

\*/

public void setName(String name) {

this.name = name;

}

/\*\*

\* Returns the email address of the customer.

\*

\* @return The email address of the customer.

\*/

public String getEmail() {

return email;

}

/\*\*

\* Sets the email address of the customer.

\*

\* @param email The email address of the customer.

\*/

public void setEmail(String email) {

this.email = email;

}

/\*\*

\* Returns the list of magazines subscribed by the customer.

\*

\* @return The list of magazines subscribed by the customer.

\*/

public List<Magazine> getMagazines() {

return magazines;

}

/\*\*

\* Sets the list of magazines subscribed by the customer.

\*

\* @param magazines The list of magazines subscribed by the customer.

\*/

public void setMagazines(List<Magazine> magazines) {

this.magazines = magazines;

}

/\*\*

\* Returns the list of supplements subscribed by the customer.

\*

\* @return The list of supplements subscribed by the customer.

\*/

public List<Supplement> getSupplements() {

return supplements;

}

/\*\*

\* Sets the list of supplements subscribed by the customer.

\*

\* @param supplements The list of supplements subscribed by the customer.

\*/

public void setSupplements(List<Supplement> supplements) {

this.supplements = supplements;

}

/\*\*

\* Calculates the total cost of all magazines subscribed by the customer.

\*

\* @return The total cost of all magazines.

\*/

public float getTotalMagazineCost() {

float totalMagazineCost = 0;

if (magazines != null) {

for (Magazine mag : magazines) {

totalMagazineCost += mag.getTotalMagazineCost();

}

}

return totalMagazineCost;

}

/\*\*

\* Calculates the total cost of all supplements subscribed by the customer.

\*

\* @return The total cost of all supplements.

\*/

public float getTotalSupplementCost() {

float totalSupplementCost = 0;

if (supplements != null) {

for (Supplement supp : supplements) {

totalSupplementCost += supp.getSupplementCost();

}

}

return totalSupplementCost;

}

/\*\*

\* Calculates the total cost of all subscriptions (magazines and

\* supplements) by the customer.

\*

\* @return The total cost of all subscriptions.

\*/

public float getTotalCost() {

return getTotalMagazineCost() + getTotalSupplementCost();

}

/\*\*

\* Prints a weekly notification for the customer detailing subscribed

\* magazines, associated supplements, and total cost.

\*/

public void weeklyNotification() {

float totalCost = 0;

System.out.println();

System.out.println("Associate Customer Details:");

System.out.println("Customer Name: " + this.getName());

System.out.println("Customer Email: " + this.getEmail());

// List the magazine and its supplements

if (magazines != null && !magazines.isEmpty()) {

System.out.println("Magazine and its associated supplements:");

for (Magazine mag : magazines) {

System.out.println("Magazine Name: " + mag.getMagazineName());

System.out.println("Magazine Cost: " + mag.getMagazineCost());

System.out.println("Magazine Total Cost: " + mag.getTotalMagazineCost());

List<Supplement> magazineSupplements = mag.getMagazineSupplement();

if (magazineSupplements != null && !magazineSupplements.isEmpty()) {

System.out.println("Supplements:");

for (Supplement supplement : magazineSupplements) {

System.out.println("\tSupplement Name: " + supplement.getSupplementName());

System.out.println("\tSupplement Cost: " + supplement.getSupplementCost());

}

} else {

System.out.println("No supplements associated with this magazine.");

}

System.out.println("-----------------------------------------");

totalCost += mag.getTotalMagazineCost();

}

} else {

System.out.println("No magazines associated with this customer.");

}

// List standalone supplements

if (supplements != null && !supplements.isEmpty()) {

System.out.println("Standalone Supplements:");

for (Supplement supp : supplements) {

System.out.println("Supplement Name: " + supp.getSupplementName());

System.out.println("Supplement Cost: " + supp.getSupplementCost());

totalCost += supp.getSupplementCost();

}

} else {

System.out.println("No standalone supplements associated with this customer.");

}

// Display total cost

System.out.println("Total Cost: " + totalCost);

}

/\*\*

\* Displays the details of the customer, including subscribed magazines and

\* supplements.

\*/

public void display() {

System.out.println("-----------------------------------------");

System.out.println("-----------------------------------------");

System.out.println("-----------------------------------------");

System.out.println("Associate Customer details");

System.out.println("Customer Name: " + this.name);

System.out.println("Customer Email: " + this.email);

System.out.println();

if (magazines != null && !magazines.isEmpty()) {

System.out.println("-----------------------------------------");

System.out.println("Magazine and it's associated supplements");

System.out.println("-----------------------------------------");

for (Magazine mag : magazines) {

mag.display(); // Call the display method in Magazine class

}

} else {

System.out.println("No magazines associated with this customer.");

}

System.out.println("---------------------------------");

// Display supplement details

if (supplements != null && !supplements.isEmpty()) {

System.out.println("---Supplement Details---:");

for (Supplement supp : supplements) {

supp.display(); // Call the display method in Supplement class

}

} else {

System.out.println("No supplements associated with this customer.");

}

}

/\*\*

\* Adds a standalone supplement to the customer's subscription.

\*

\* @param newSupplement The new supplement to add.

\*/

public void addStandaloneSupplement(Supplement newSupplement) {

if (supplements == null) {

supplements = new ArrayList<>();

}

supplements.add(newSupplement);

}

/\*\*

\* Adds a magazine to the customer's subscription.

\*

\* @param newMagazine The new magazine to add.

\*/

public void addMagazine(Magazine newMagazine) {

if (magazines == null) {

magazines = new ArrayList<>();

}

magazines.add(newMagazine);

}

/\*\*

\* Adds a supplement to the customer's subscription.

\*

\* @param newSupplement The new supplement to add.

\*/

public void addSupplement(Supplement newSupplement) {

if (supplements == null) {

supplements = new ArrayList<>();

}

supplements.add(newSupplement);

}

/\*\*

\* Removes a magazine from the customer's subscription.

\*

\* @param magazine The magazine to remove.

\*/

public void removeMagazine(Magazine magazine) {

if (magazines != null && magazines.contains(magazine)) {

magazines.remove(magazine);

System.out.println("Magazine \"" + magazine.getMagazineName() + "\" removed successfully from customer " + this.name);

} else {

System.out.println("The magazine is not associated with this customer.");

}

}

/\*\*

\* Removes a supplement from the customer's subscription.

\*

\* @param supplement The supplement to remove.

\*/

public void removeSupplement(Supplement supplement) {

if (supplements != null && supplements.contains(supplement)) {

supplements.remove(supplement);

System.out.println("Supplement \"" + supplement.getSupplementName() + "\" removed successfully from customer " + this.name);

} else {

System.out.println("The supplement is not associated with this customer.");

}

}

/\*\*

\* Displays the customer's details in a graphical user interface.

\*

\* @return A Node representing the customer's details.

\*/

public Node displayGUI() {

VBox vbox = new VBox(5); // VBox with a spacing of 5

vbox.getChildren().add(new Label("Customer Name: " + this.name));

vbox.getChildren().add(new Label("Customer Email: " + this.email));

if (magazines != null && !magazines.isEmpty()) {

vbox.getChildren().add(new Label("Magazines and Associated Supplements:"));

for (Magazine mag : magazines) {

vbox.getChildren().add(mag.displayGUI()); // Now directly adding the Node

}

} else {

vbox.getChildren().add(new Label("No magazines associated with this customer."));

}

if (supplements != null && !supplements.isEmpty()) {

vbox.getChildren().add(new Label("Standalone Supplements:"));

for (Supplement supp : supplements) {

vbox.getChildren().add(supp.displayGUI()); // Now directly adding the Node

}

} else {

vbox.getChildren().add(new Label("No standalone supplements associated with this customer."));

}

vbox.getChildren().add(new Label("Total Cost: $" + String.format("%.2f", getTotalCost())));

return vbox; // Return the VBox as a Node

}

/\*\*

\* Returns the name of the customer as a string.

\*

\* @return The name of the customer.

\*/

@Override

public String toString() {

return this.getName(); // Assuming you have a getName() method that returns the customer's name

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import java.util.List;

import java.util.ArrayList;

import java.util.\*;

/\*\*

\* The Database class represents a database of supplements, magazines, paying

\* customers, and associate customers.

\*/

public class Database implements Serializable {

private List<Supplement> supplements; // The list of supplements in the database

private List<Magazine> magazines; // The list of magazines in the database

private List<PayingCustomer> payingCustomers; // The list of paying customers in the database

private List<Customer> associateCustomers; // The list of associate customers in the database

/\*\*

\* Constructs a new Database object with empty lists.

\*/

public Database() {

supplements = new ArrayList<>();

magazines = new ArrayList<>();

payingCustomers = new ArrayList<>();

associateCustomers = new ArrayList<>(); // Initialize associateCustomers list

}

/\*\*

\* Constructs a new Database object with the specified lists of supplements,

\* magazines, paying customers, and associate customers.

\*

\* @param supplements The list of supplements.

\* @param magazines The list of magazines.

\* @param payingCustomers The list of paying customers.

\* @param associateCustomers The list of associate customers.

\*/

public Database(List<Supplement> supplements, List<Magazine> magazines, List<PayingCustomer> payingCustomers, List<Customer> associateCustomers) {

this.supplements = supplements;

this.magazines = magazines;

this.payingCustomers = payingCustomers;

this.associateCustomers = associateCustomers;

}

/\*\*

\* Populates the database with sample data.

\*/

public void populate() {

Supplement supplement\_1 = new Supplement("supplement\_1 ", 1);

Supplement supplement\_2 = new Supplement("supplement\_2 ", 1);

Supplement supplement\_3 = new Supplement("supplement\_3 ", 1);

supplements.add(supplement\_3);

supplements.add(supplement\_2);

supplements.add(supplement\_1);

Magazine magazine\_1 = new Magazine("Magazine\_1", 1, new ArrayList<>());

Magazine magazine\_2 = new Magazine("Magazine\_2", 1, new ArrayList<>());

Magazine magazine\_3 = new Magazine("Magazine\_3", 1, new ArrayList<>());

magazines.add(magazine\_1);

magazines.add(magazine\_2);

magazines.add(magazine\_3);

List<Customer> associateCustomerList = new ArrayList<>();

Customer john = new Customer("John", "john.smith@example.com", new ArrayList<>(), new ArrayList<>());

Customer alice = new Customer("Alice", "alice.jones@example.com", new ArrayList<>(), new ArrayList<>());

Customer robert = new Customer("Robert", "robert.williams@example.com", new ArrayList<>(), new ArrayList<>());

associateCustomerList.add(john);

associateCustomerList.add(alice);

associateCustomerList.add(robert);

PayingCustomer.PaymentMethod selectedPaymentMethod = PayingCustomer.PaymentMethod.CREDIT\_CARD;

PayingCustomer sarah = new PayingCustomer(selectedPaymentMethod, "UK Bank", associateCustomerList, "Sarah", "sarah@gmail.com", new ArrayList<>(), new ArrayList<>());

payingCustomers.add(sarah);

Customer johnjohn = new Customer("johnjohn", "john.smith@example.com", new ArrayList<>(), new ArrayList<>());

Customer alicealice = new Customer("Alicealisce", "alice.jones@example.com", new ArrayList<>(), new ArrayList<>());

Customer robertrober = new Customer("Robertrobert", "robert.williams@example.com", new ArrayList<>(), new ArrayList<>());

associateCustomers.add(johnjohn);

associateCustomers.add(alicealice);

associateCustomers.add(robertrober);

}

/\*\*

\* Returns the list of paying customers.

\*

\* @return The list of paying customers.

\*/

public List<PayingCustomer> getPayingCustomers() {

return payingCustomers;

}

/\*\*

\* Sets the list of paying customers.

\*

\* @param payingCustomers The list of paying customers.

\*/

public void setPayingCustomers(List<PayingCustomer> payingCustomers) {

this.payingCustomers = payingCustomers;

}

/\*\*

\* Returns the list of associate customers.

\*

\* @return The list of associate customers.

\*/

public List<Customer> getAssociateCustomers() {

return associateCustomers;

}

/\*\*

\* Sets the list of associate customers.

\*

\* @param associateCustomers The list of associate customers.

\*/

public void setAssociateCustomers(List<Customer> associateCustomers) {

this.associateCustomers = associateCustomers;

}

/\*\*

\* Returns the list of supplements.

\*

\* @return The list of supplements.

\*/

public List<Supplement> getSupplements() {

return supplements;

}

/\*\*

\* Sets the list of supplements.

\*

\* @param supplements The list of supplements.

\*/

public void setSupplements(List<Supplement> supplements) {

this.supplements = supplements;

}

/\*\*

\* Returns the list of magazines.

\*

\* @return The list of magazines.

\*/

public List<Magazine> getMagazines() {

return magazines;

}

/\*\*

\* Sets the list of magazines.

\*

\* @param magazines The list of magazines.

\*/

public void setMagazines(List<Magazine> magazines) {

this.magazines = magazines;

}

/\*\*

\* Adds a supplement to the database.

\*

\* @param supplement The supplement to add.

\*/

public void addSupplement(Supplement supplement) {

supplements.add(supplement);

}

/\*\*

\* Removes a supplement from the database.

\*

\* @param supplement The supplement to remove.

\*/

public void removeSupplement(Supplement supplement) {

supplements.remove(supplement);

}

/\*\*

\* Adds a magazine to the database.

\*

\* @param magazine The magazine to add.

\*/

public void addMagazine(Magazine magazine) {

magazines.add(magazine);

}

/\*\*

\* Removes a magazine from the database.

\*

\* @param magazine The magazine to remove.

\*/

public void removeMagazine(Magazine magazine) {

magazines.remove(magazine);

}

/\*\*

\* Adds a paying customer to the database.

\*

\* @param payingCustomer The paying customer to add.

\*/

public void addPayingCustomer(PayingCustomer payingCustomer) {

payingCustomers.add(payingCustomer);

}

/\*\*

\* Removes a paying customer from the database.

\*

\* @param payingCustomer The paying customer to remove.

\*/

public void removePayingCustomer(PayingCustomer payingCustomer) {

payingCustomers.remove(payingCustomer);

}

/\*\*

\* Adds an associate customer to the database.

\*

\* @param associateCustomer The associate customer to add.

\*/

public void addAssociateCustomer(Customer associateCustomer) {

associateCustomers.add(associateCustomer);

}

/\*\*

\* Removes an associate customer from the database.

\*

\* @param associateCustomer The associate customer to remove.

\*/

public void removeAssociateCustomer(Customer associateCustomer) {

associateCustomers.remove(associateCustomer);

}

/\*\*

\* Allows the user to select a magazine from the database.

\*

\* @return The selected magazine.

\*/

public Supplement selectSupplement() {

System.out.println("Available Supplements:");

displayItems(supplements);

int selection = getUserSelection("Enter the index of the supplement to select: ", supplements.size());

return supplements.get(selection - 1);

}

public Magazine selectMagazine() {

System.out.println("Available Magazines:");

displayItems(magazines);

int selection = getUserSelection("Enter the index of the magazine to select: ", magazines.size());

return magazines.get(selection - 1);

}

private void displayItems(List<?> items) {

for (int i = 0; i < items.size(); i++) {

String itemName = "";

if (items.get(i) instanceof Supplement) {

itemName = getSupplementName(i);

} else if (items.get(i) instanceof Magazine) {

itemName = getMagazineName(i);

}

System.out.println((i + 1) + ". " + itemName);

}

}

private int getUserSelection(String prompt, int size) {

Scanner scanner = new Scanner(System.in);

int selection = 0;

boolean validInput = false;

while (!validInput) {

try {

System.out.print(prompt);

selection = scanner.nextInt();

scanner.nextLine(); // Consume newline

if (selection >= 1 && selection <= size) {

validInput = true;

} else {

System.out.println("Invalid selection. Please enter a number between 1 and " + size + ".");

}

} catch (InputMismatchException e) {

scanner.nextLine(); // Consume invalid input

System.out.println("Invalid input. Please enter a number.");

}

}

return selection;

}

/\*\*

\* Returns the name of the magazine at the specified index.

\*

\* @param index The index of the magazine.

\* @return The name of the magazine.

\*/

public String getMagazineName(int index) {

if (index >= 0 && index < magazines.size()) {

return magazines.get(index).getMagazineName();

}

return null;

}

/\*\*

\* Returns the name of the supplement at the specified index.

\*

\* @param index The index of the supplement.

\* @return The name of the supplement.

\*/

public String getSupplementName(int index) {

if (index >= 0 && index < supplements.size()) {

return supplements.get(index).getSupplementName();

}

return null;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.Scanner;

/\*\*

\* The Display class provides methods for displaying menus and user interfaces.

\*/

public class Display {

/\*\*

\* Displays a menu and prompts the user to choose an option.

\*

\* @return The user's choice (an integer between 1 and 10).

\*/

public static int menu() {

Scanner scanner = new Scanner(System.in);

System.out.println("Menu:");

System.out.println("1. Create Magazine");

System.out.println("2. Create Supplemt");

System.out.println("3. Create Customer");

System.out.println("4. Add Associate to Paying Customer");

System.out.println("5. Print out the text of all the emails for all customers for four weeks of magazines");

System.out.println("6. Print out the text for the end of month emails for the paying customers");

System.out.println("7. Add a new customer to the magazine service");

System.out.println("8. Remove an existing customer from the magazine service");

System.out.println("9. Add supplement to customer or magazine");

System.out.println("10. Quit");

System.out.print("Enter your choice (1-10): ");

int choice = scanner.nextInt();

// Validate input

while (choice < 1 || choice > 10) {

System.out.println("Invalid choice. Please enter a number between 1 and10.");

System.out.print("Enter your choice (1-10): ");

choice = scanner.nextInt();

}

return choice;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

/\*\*

\* The EndOfMonthPayingCustomerEmailPrint class provides a method to print

\* end-of-month emails for paying customers.

\*/

public class EndOfMonthPayingCustomerEmailPrint {

/\*\*

\* Prints end-of-month emails for paying customers.

\*

\* @param database The database containing the paying customers.

\*/

public static void run(Database database) {

System.out.println("Printing End-of-Month Emails for Paying Customers:");

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

payingCustomer.monthlyNotification();

}

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

/\*\*

\* The FourWeekCustomerEmailPrint class provides a method to print four weeks of

\* email notifications for all customers.

\*/

public class FourWeekCustomerEmailPrint {

/\*\*

\* Prints four weeks of email notifications for all customers, including

\* paying customers and their associated customers.

\*

\* @param database The database containing the customers.

\*/

public static void run(Database database) {

System.out.println("Printing four weeks of email notifications for all customers:");

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

System.out.println();

System.out.println("Paying Customer: " + payingCustomer.getName());

for (int week = 1; week <= 4; week++) {

System.out.println("Week " + week + " Notification:");

payingCustomer.weeklyNotification();

}

// Include associated customers

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

System.out.println();

System.out.println("Associate Customer: " + associateCustomer.getName());

for (int week = 1; week <= 4; week++) {

System.out.println("Week " + week + " Notification:");

associateCustomer.weeklyNotification();

}

}

}

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import javafx.scene.Node;

import javafx.scene.control.Label;

import javafx.scene.control.Separator;

import javafx.scene.layout.VBox;

/\*\*

\* The Magazine class represents a magazine entity with its details and

\* associated supplements.

\*/

public class Magazine implements Serializable {

private String magazineName;

private float magazineCost;

private List<Supplement> magazineSupplement;

/\*\*

\* Constructs a Magazine object with the specified name, cost, and list of

\* supplements.

\*

\* @param magazineName The name of the magazine.

\* @param magazineCost The cost of the magazine.

\* @param magazineSupplement The list of supplements associated with the

\* magazine.

\*/

public Magazine(String magazineName, float magazineCost, List<Supplement> magazineSupplement) {

this.magazineName = magazineName;

this.magazineCost = magazineCost;

this.magazineSupplement = magazineSupplement;

}

/\*\*

\* Retrieves the name of the magazine.

\*

\* @return The name of the magazine.

\*/

public String getMagazineName() {

return magazineName;

}

/\*\*

\* Sets the name of the magazine.

\*

\* @param magazineName The name of the magazine.

\*/

public void setMagazineName(String magazineName) {

this.magazineName = magazineName;

}

/\*\*

\* Retrieves the cost of the magazine.

\*

\* @return The cost of the magazine.

\*/

public float getMagazineCost() {

return magazineCost;

}

/\*\*

\* Sets the cost of the magazine.

\*

\* @param magazineCost The cost of the magazine.

\*/

public void setMagazineCost(float magazineCost) {

this.magazineCost = magazineCost;

}

/\*\*

\* Retrieves the list of supplements associated with the magazine.

\*

\* @return The list of supplements associated with the magazine.

\*/

public List<Supplement> getMagazineSupplement() {

return magazineSupplement;

}

/\*\*

\* Sets the list of supplements associated with the magazine.

\*

\* @param magazineSupplement The list of supplements associated with the

\* magazine.

\*/

public void setMagazineSupplement(List<Supplement> magazineSupplement) {

this.magazineSupplement = magazineSupplement;

}

/\*\*

\* Calculates the total cost of the magazine including its supplements.

\*

\* @return The total cost of the magazine.

\*/

public float getTotalMagazineCost() {

float totalCost = magazineCost;

if (magazineSupplement != null) {

for (Supplement supplement : magazineSupplement) {

totalCost += supplement.getSupplementCost();

}

}

return totalCost;

}

/\*\*

\* Returns the name of the magazine.

\*

\* @return The name of the magazine.

\*/

@Override

public String toString() {

return magazineName; // This will be used by ListView to display the magazine's name.

}

/\*\*

\* Displays the details of the magazine including its name, cost, and

\* associated supplements.

\*/

public void display() {

System.out.println();

System.out.println("--------------------");

System.out.println("Magazine Details");

System.out.println("Magazine Name: " + magazineName);

System.out.println("Magazine Cost: " + magazineCost);

System.out.println();

if (magazineSupplement != null) {

System.out.println("Magazine Supplements Details:");

for (Supplement supplement : magazineSupplement) {

supplement.display();

}

} else {

System.out.println("No supplements available for this magazine.");

}

}

/\*\*

\* Adds a supplement to the magazine.

\*

\* @param supplement The supplement to be added.

\*/

public void addSupplement(Supplement supplement) {

if (magazineSupplement == null) {

magazineSupplement = new ArrayList<>();

}

magazineSupplement.add(supplement);

}

/\*\*

\* Removes a supplement from the magazine.

\*

\* @param supplement The supplement to be removed.

\*/

public void removeSupplement(Supplement supplement) {

if (magazineSupplement != null) {

magazineSupplement.remove(supplement);

}

}

/\*\*

\* Displays the graphical user interface representation of the magazine.

\*

\* @return A Node representing the GUI of the magazine.

\*/

public Node displayGUI() {

VBox vbox = new VBox(5); // VBox with a spacing of 5

vbox.getChildren().add(new Separator());

vbox.getChildren().add(new Label("Magazine Details"));

vbox.getChildren().add(new Label("Magazine Name: " + magazineName));

vbox.getChildren().add(new Label("Magazine Cost: " + String.format("%.2f", magazineCost)));

if (magazineSupplement != null && !magazineSupplement.isEmpty()) {

vbox.getChildren().add(new Label("Magazine Supplements Details:"));

for (Supplement supplement : magazineSupplement) {

vbox.getChildren().add(supplement.displayGUI()); // Assuming displayGUI() returns a Node

}

} else {

vbox.getChildren().add(new Label("No supplements available for this magazine."));

}

return vbox; // Return the VBox as a Node

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

import javafx.scene.Node;

import javafx.scene.control.Label;

import javafx.scene.control.Separator;

import javafx.scene.layout.VBox;

/\*\*

\* The PayingCustomer class represents a paying customer entity, extending the

\* Customer class, with additional attributes such as selected payment method,

\* bank, and associated customers.

\*/

public class PayingCustomer extends Customer implements Serializable {

//private static final long serialVersionUID = -1763755671558395339L;

/\*\*

\* Enumeration representing different payment methods.

\*/

public enum PaymentMethod {

CREDIT\_CARD,

BANK\_CARD

}

private PaymentMethod selectedPaymentMethod;

private String bank;

private List<Customer> associateCustomers;

/\*\*

\* Constructs a PayingCustomer object with the specified attributes.

\*

\* @param selectedPaymentMethod The selected payment method.

\* @param bank The bank associated with the paying customer.

\* @param associateCustomer The list of associated customers.

\* @param name The name of the paying customer.

\* @param email The email of the paying customer.

\* @param magazines The list of magazines associated with the paying

\* customer.

\* @param supplements The list of supplements associated with the paying

\* customer.

\*/

public PayingCustomer(PaymentMethod selectedPaymentMethod, String bank, List<Customer> associateCustomer, String name, String email, List<Magazine> magazines, List<Supplement> supplements) {

super(name, email, magazines, supplements);

this.selectedPaymentMethod = selectedPaymentMethod;

this.bank = bank;

this.associateCustomers = associateCustomer;

}

/\*\*

\* Constructs a PayingCustomer object with the specified attributes.

\*

\* @param selectedPaymentMethod The selected payment method.

\* @param bank The bank associated with the paying customer.

\* @param associateCustomer The list of associated customers.

\*/

public PayingCustomer(PaymentMethod selectedPaymentMethod, String bank, List<Customer> associateCustomer) {

this.selectedPaymentMethod = selectedPaymentMethod;

this.bank = bank;

this.associateCustomers = associateCustomer;

}

/\*\*

\* Constructs a default PayingCustomer object.

\*/

public PayingCustomer() {

super();

associateCustomers = new ArrayList<>();

}

/\*\*

\* Retrieves the selected payment method of the paying customer.

\*

\* @return The selected payment method.

\*/

public PaymentMethod getSelectedPaymentMethod() {

return selectedPaymentMethod;

}

/\*\*

\* Sets the selected payment method of the paying customer.

\*

\* @param selectedPaymentMethod The selected payment method.

\*/

public void setSelectedPaymentMethod(PaymentMethod selectedPaymentMethod) {

this.selectedPaymentMethod = selectedPaymentMethod;

}

/\*\*

\* Retrieves the bank associated with the paying customer.

\*

\* @return The bank associated with the paying customer.

\*/

public String getBank() {

return bank;

}

/\*\*

\* Sets the bank associated with the paying customer.

\*

\* @param bank The bank associated with the paying customer.

\*/

public void setBank(String bank) {

this.bank = bank;

}

/\*\*

\* Retrieves the list of associated customers.

\*

\* @return The list of associated customers.

\*/

public List<Customer> getAssociateCustomers() {

return associateCustomers;

}

/\*\*

\* Sets the list of associated customers.

\*

\* @param associateCustomer The list of associated customers.

\*/

public void setAssociateCustomer(List<Customer> associateCustomer) {

this.associateCustomers = associateCustomer;

}

/\*\*

\* Adds an associate customer to the paying customer's list of associated

\* customers.

\*

\* @param customer The associate customer to be added.

\*/

public void addAssociateCustomer(Customer customer) {

if (this.associateCustomers == null) {

this.associateCustomers = new ArrayList<>();

}

this.associateCustomers.add(customer);

}

/\*\*

\* Generates and displays the weekly notification for the paying customer.

\*/

@Override

public void weeklyNotification() {

float totalCost = 0;

// Display customer details

System.out.println();

System.out.println("Paying Customer Details:");

System.out.println("Customer Name: " + this.getName());

System.out.println("Customer Email: " + this.getEmail());

System.out.println("Selected Payment Method: " + this.selectedPaymentMethod);

System.out.println("Bank: " + this.bank);

System.out.println();

// List the magazine and its supplements

if (this.getMagazines() != null && !this.getMagazines().isEmpty()) {

System.out.println("Magazines and their associated supplements:");

for (Magazine mag : this.getMagazines()) {

System.out.println("Magazine Name: " + mag.getMagazineName());

System.out.println("Magazine Cost: " + mag.getMagazineCost());

System.out.println("Magazine Total Cost: " + mag.getTotalMagazineCost());

List<Supplement> magazineSupplements = mag.getMagazineSupplement();

if (magazineSupplements != null && !magazineSupplements.isEmpty()) {

System.out.println("Supplements:");

for (Supplement supplement : magazineSupplements) {

System.out.println("\tSupplement Name: " + supplement.getSupplementName());

System.out.println("\tSupplement Cost: " + supplement.getSupplementCost());

}

} else {

System.out.println("No supplements associated with this magazine.");

}

System.out.println("-----------------------------------------");

totalCost += mag.getTotalMagazineCost();

}

} else {

System.out.println("No magazines associated with this customer.");

}

// List standalone supplements

if (this.getSupplements() != null && !this.getSupplements().isEmpty()) {

System.out.println("Standalone Supplements:");

for (Supplement supp : this.getSupplements()) {

System.out.println("Supplement Name: " + supp.getSupplementName());

System.out.println("Supplement Cost: " + supp.getSupplementCost());

totalCost += supp.getSupplementCost();

}

} else {

System.out.println("No standalone supplements associated with this customer.");

}

// Display total cost

System.out.println("Total Cost: " + totalCost);

}

private float calculateAndDisplayCustomerCost(Customer customer) {

float customerTotalCost = 0;

// Display customer name and email

System.out.println("Customer Name: " + customer.getName());

System.out.println("Customer Email: " + customer.getEmail());

// Display magazines and their associated costs

if (customer.getMagazines() != null && !customer.getMagazines().isEmpty()) {

System.out.println("Magazines and their associated costs:");

for (Magazine mag : customer.getMagazines()) {

float magazineCost = mag.getMagazineCost();

System.out.println("Magazine Name: " + mag.getMagazineName() + ", Cost: " + magazineCost);

customerTotalCost += magazineCost;

// Display supplements for this magazine

List<Supplement> magazineSupplements = mag.getMagazineSupplement();

if (magazineSupplements != null && !magazineSupplements.isEmpty()) {

System.out.println("Supplements in this Magazine:");

for (Supplement supplement : magazineSupplements) {

float supplementCost = supplement.getSupplementCost();

System.out.println("\tSupplement name: " + supplement.getSupplementName() + ", Cost: " + supplementCost);

customerTotalCost += supplementCost;

}

}

}

} else {

System.out.println("No magazines associated with this customer.");

}

// Display standalone supplements and their costs

if (customer.getSupplements() != null && !customer.getSupplements().isEmpty()) {

System.out.println("Standalone Supplements and their costs:");

for (Supplement supp : customer.getSupplements()) {

float supplementCost = supp.getSupplementCost();

System.out.println("Supplement name: " + supp.getSupplementName() + ", Cost: " + supplementCost);

customerTotalCost += supplementCost;

}

} else {

System.out.println("No standalone supplements associated with this customer.");

}

// Display total cost for the customer

System.out.println("Total Cost for the Customer: " + customerTotalCost);

System.out.println();

return customerTotalCost;

}

/\*\*

\* Generates and displays the monthly notification for the paying customer.

\*/

public void monthlyNotification() {

// Display customer details

System.out.println("Monthly Notification for Paying Customer:");

System.out.println("Customer Name: " + this.getName());

System.out.println("Customer Email: " + this.getEmail());

System.out.println("Selected Payment Method: " + this.selectedPaymentMethod);

System.out.println("Bank: " + this.bank);

System.out.println();

// Calculate and display total cost for the paying customer

float totalCost = calculateAndDisplayCustomerCost(this);

// Display associate customers and their associated magazines and supplements

if (this.associateCustomers != null && !this.associateCustomers.isEmpty()) {

for (Customer associate : this.associateCustomers) {

// Calculate and display total cost for the associate customer

totalCost += calculateAndDisplayCustomerCost(associate);

}

} else {

System.out.println("No associate customers.");

}

// Display total cost for the paying customer and their associate customers

System.out.println("Total Cost for the Month (Including Associate Customers): " + totalCost);

}

/\*\*

\* Generates a comprehensive billing information for the paying customer and

\* their associated customers.

\*

\* @return The BillingInfo object containing the billing details.

\*/

public BillingInfo generateComprehensiveBillingInfo() {

BillingInfo billingInfo = new BillingInfo(this.getName(), this.getEmail());

// Adding billing details for the paying customer

billingInfo.addDetail("Paying Customer Details:", 0); // 0 cost as a placeholder for a header

billingInfo.addDetail("Name: " + this.getName(), 0);

billingInfo.addDetail("Email: " + this.getEmail(), 0);

billingInfo.addDetail("Payment Method: " + this.selectedPaymentMethod, 0);

billingInfo.addDetail("Bank: " + this.bank, 0);

billingInfo.addDetail("--------", 0); // Visual separator

// Magazines and their supplements

for (Magazine magazine : this.getMagazines()) {

billingInfo.addDetail("Magazine: " + magazine.getMagazineName(), magazine.getMagazineCost());

for (Supplement supplement : magazine.getMagazineSupplement()) {

billingInfo.addDetail(" Supplement: " + supplement.getSupplementName(), supplement.getSupplementCost());

}

}

// Standalone supplements

for (Supplement supplement : this.getSupplements()) {

billingInfo.addDetail("Supplement: " + supplement.getSupplementName(), supplement.getSupplementCost());

}

// Associated Customers

if (!this.getAssociateCustomers().isEmpty()) {

billingInfo.addDetail("Associated Customers Details:", 0); // 0 cost as a placeholder for a header

for (Customer associate : this.getAssociateCustomers()) {

billingInfo.addDetail("Associate Name: " + associate.getName(), 0);

billingInfo.addDetail("Associate Email: " + associate.getEmail(), 0);

for (Magazine magazine : associate.getMagazines()) {

billingInfo.addDetail(" Magazine: " + magazine.getMagazineName(), magazine.getMagazineCost());

for (Supplement supplement : magazine.getMagazineSupplement()) {

billingInfo.addDetail(" Supplement: " + supplement.getSupplementName(), supplement.getSupplementCost());

}

}

for (Supplement supplement : associate.getSupplements()) {

billingInfo.addDetail(" Supplement: " + supplement.getSupplementName(), supplement.getSupplementCost());

}

billingInfo.addDetail("--------", 0); // Visual separator for each associate

}

}

// Total Cost: This line aggregates all individual costs added above.

//billingInfo.addDetail("Total Cost", billingInfo.getTotalCost());

return billingInfo;

}

/\*\*

\* Displays the details of the paying customer including associated

\* magazines, supplements, and customers.

\*/

@Override

public void display() {

System.out.println();

System.out.println();

System.out.println();

System.out.println("Paying Customer Details:");

System.out.println("Customer Name: " + this.getName());

System.out.println("Customer Email: " + this.getEmail());

System.out.println("Selected Payment Method: " + this.selectedPaymentMethod);

System.out.println("Bank: " + this.bank);

System.out.println();

// Display associated magazines

if (this.getMagazines() != null && !this.getMagazines().isEmpty()) {

System.out.println("-----------------------------------------");

System.out.println("Magazines and their associated supplements:");

System.out.println("-----------------------------------------");

for (Magazine mag : this.getMagazines()) {

mag.display();

}

} else {

System.out.println("No magazines associated with this customer.");

}

System.out.println("---------------------------------");

// Display associated supplements

if (this.getSupplements() != null && !this.getSupplements().isEmpty()) {

System.out.println("---Supplement Details---:");

for (Supplement supp : this.getSupplements()) {

supp.display();

}

} else {

System.out.println("No supplements associated with this customer.");

}

// Display associated customers

if (this.associateCustomers != null && !this.associateCustomers.isEmpty()) {

System.out.println();

System.out.println("-----------------------------------------------");

System.out.println("Associated Customers:");

for (Customer customer : this.associateCustomers) {

customer.display();

}

} else {

System.out.println("No associated customers.");

}

}

/\*\*

\* Displays the graphical user interface representation of the paying

\* customer including associated magazines, supplements, and customers.

\*

\* @return A Node representing the GUI of the paying customer.

\*/

@Override

public Node displayGUI() {

VBox vbox = new VBox(5); // Create a VBox with a spacing of 5

// Paying Customer Details Section

vbox.getChildren().add(new Separator());

vbox.getChildren().add(new Label("Paying Customer Details:"));

vbox.getChildren().add(new Label("Customer Name: " + this.getName()));

vbox.getChildren().add(new Label("Customer Email: " + this.getEmail()));

vbox.getChildren().add(new Label("Selected Payment Method: " + this.selectedPaymentMethod));

vbox.getChildren().add(new Label("Bank: " + this.bank));

vbox.getChildren().add(new Separator());

// Magazines and their associated supplements Section

if (this.getMagazines() != null && !this.getMagazines().isEmpty()) {

vbox.getChildren().add(new Label("Magazines and their associated supplements:"));

for (Magazine mag : this.getMagazines()) {

vbox.getChildren().add(mag.displayGUI()); // assuming Magazine class has a displayGUI() method that returns a Node

}

} else {

vbox.getChildren().add(new Label("No magazines associated with this customer."));

}

vbox.getChildren().add(new Separator());

// Supplement Details Section

if (this.getSupplements() != null && !this.getSupplements().isEmpty()) {

vbox.getChildren().add(new Label("Supplement Details:"));

for (Supplement supp : this.getSupplements()) {

vbox.getChildren().add(supp.displayGUI()); // assuming Supplement class has a displayGUI() method that returns a Node

}

} else {

vbox.getChildren().add(new Label("No supplements associated with this customer."));

}

// Associated Customers Section

if (this.associateCustomers != null && !this.associateCustomers.isEmpty()) {

vbox.getChildren().add(new Separator());

vbox.getChildren().add(new Label("Associated Customers:"));

for (Customer customer : this.associateCustomers) {

vbox.getChildren().add(customer.displayGUI()); // assuming Customer class has a displayGUI() method that returns a Node

}

} else {

vbox.getChildren().add(new Label("No associated customers."));

}

return vbox; // Return the VBox as a Node

}

/\*\*

\* Returns the name of the paying customer.

\*

\* @return The name of the paying customer.

\*/

@Override

public String toString() {

return this.getName(); // Assuming you have a getName() method that returns the customer's name

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.util.List;

import java.util.Scanner;

/\*\*

\* The RemoveCustomerFromMagazineService class provides functionality to remove

\* a customer from the magazine service.

\*/

public class RemoveCustomerFromMagazineService {

/\*\*

\* Runs the process to remove a customer from the magazine service.

\*

\* @param database The database containing customer information.

\*/

public static void run(Database database) {

Scanner scanner = new Scanner(System.in);

// List out all customers (both paying and associate)

System.out.println("List of All Customers:");

int index = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

System.out.println(index + ". " + payingCustomer.getName() + " (Paying Customer)");

index++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

System.out.println(index + ". " + associateCustomer.getName() + " (Associate Customer)");

index++;

}

}

// Select customer

int selectedCustomerIndex = getUserSelection("Enter the index of the customer: ", index);

// Find selected customer

Customer selectedCustomer = findCustomerByIndex(database, selectedCustomerIndex);

if (selectedCustomer != null) {

// List out magazines

System.out.println("List of Magazines for " + selectedCustomer.getName() + ":");

List<Magazine> magazines = selectedCustomer.getMagazines();

if (magazines != null && !magazines.isEmpty()) {

for (int i = 0; i < magazines.size(); i++) {

System.out.println((i + 1) + ". " + magazines.get(i).getMagazineName());

}

// Select magazine to remove

int selectedMagazineIndex = getUserSelection("Enter the index of the magazine to remove: ", magazines.size());

// Remove magazine from selected customer

selectedCustomer.removeMagazine(magazines.get(selectedMagazineIndex - 1));

System.out.println("Magazine removed successfully from " + selectedCustomer.getName());

} else {

System.out.println("No magazines associated with this customer.");

}

} else {

System.out.println("Invalid selection. Customer not found.");

}

}

/\*\*

\* Gets user selection from the console input.

\*

\* @param prompt The prompt to display to the user.

\* @param size The size of the selection range.

\* @return The user's selection.

\*/

private static int getUserSelection(String prompt, int size) {

Scanner scanner = new Scanner(System.in);

int selection = 0;

boolean validInput = false;

while (!validInput) {

try {

System.out.print(prompt);

selection = scanner.nextInt();

scanner.nextLine(); // Consume newline

if (selection >= 1 && selection <= size) {

validInput = true;

} else {

System.out.println("Invalid selection. Please enter a number between 1 and " + size + ".");

}

} catch (Exception e) {

scanner.nextLine(); // Consume invalid input

System.out.println("Invalid input. Please enter a number.");

}

}

return selection;

}

/\*\*

\* Finds a customer by the given index.

\*

\* @param database The database containing customer information.

\* @param index The index of the customer.

\* @return The customer found at the given index, or null if not found.

\*/

private static Customer findCustomerByIndex(Database database, int index) {

int currentIndex = 1;

for (PayingCustomer payingCustomer : database.getPayingCustomers()) {

if (currentIndex == index) {

return payingCustomer;

}

currentIndex++;

for (Customer associateCustomer : payingCustomer.getAssociateCustomers()) {

if (currentIndex == index) {

return associateCustomer;

}

currentIndex++;

}

}

return null;

}

}

package com.mycompany.assignment2v2;

/\*\*

\* Title: Assignment2 Author: Yin Zhanpeng Date: 29/3/2024 File Name:

\* Assignment2

\*

\* <p>

\* Program Description:</p>

\* <p>

\* This program simulates the weekly and monthly notifications for customers

\* subscribed to magazines and supplements. It allows user interactions for

\* managing subscriptions and customer information.</p>

\*

\* <p>

\* Assumptions/Conditions:</p>

\* <ol>

\* <li>The customers have a list of Supplements and Magazines.</li>

\* <li>Supplements can be individual or included within magazines.</li>

\* <li>Magazines contain their own list of Supplements.</li>

\* <li>The program allows the user to add supplements to both magazines and

\* customers.</li>

\* <li>Paying customers have a list of associate customers.</li>

\* <li>Both the paying and associate customers have their own

\* subscriptions.</li>

\* <li>The program only simulates weekly and monthly notifications.</li>

\* <li>Associate customers of a paying customer can be removed.</li>

\* <li>Removing a paying customer also removes its associated customers.</li>

\* <li>Customers contain magazines, not the other way around.</li>

\* <li>There should be at least one paying customer.</li>

\* <li>Textual data inputs for customer names, addresses, email addresses,

\* supplement names, etc., are handled.</li>

\* <li>Payment methods are represented as strings or simple identifiers.</li>

\* <li>The GUI is designed for desktop use and is not optimized for mobile

\* devices.</li>

\* <li>Billing history is displayed in a simple tabular format.</li>

\* <li>Customer address details are limited to basic information.</li>

\* <li>The program handles a reasonable number of entities without significant

\* performance degradation.</li>

\* <li>Basic error handling is implemented for scenarios such as invalid input

\* formats and file I/O errors.</li>

\* <li>The GUI layout is implemented using JavaFX controls and layouts.</li>

\* <li>The program is developed and tested on the Java SE 8 platform using

\* NetBeans IDE.</li>

\* <li>Data persistence is achieved through serialization.</li>

\* </ol>

\*/

import java.io.Serializable;

import javafx.scene.Node;

import javafx.scene.control.Label;

import javafx.scene.layout.VBox;

/\*\*

\* The Supplement class represents a supplement associated with a magazine. It

\* contains information about the supplement's name and cost.

\*/

public class Supplement implements Serializable {

private String supplementName;

private float supplementCost;

/\*\*

\* Constructs a new Supplement object with the specified name and cost.

\*

\* @param supplementName The name of the supplement.

\* @param supplementCost The cost of the supplement.

\*/

public Supplement(String supplementName, float supplementCost) {

this.supplementName = supplementName;

this.supplementCost = supplementCost;

}

/\*\*

\* Retrieves the name of the supplement.

\*

\* @return The name of the supplement.

\*/

public String getSupplementName() {

return supplementName;

}

/\*\*

\* Sets the name of the supplement.

\*

\* @param supplementName The name of the supplement.

\*/

public void setSupplementName(String supplementName) {

this.supplementName = supplementName;

}

/\*\*

\* Retrieves the cost of the supplement.

\*

\* @return The cost of the supplement.

\*/

public float getSupplementCost() {

return supplementCost;

}

/\*\*

\* Sets the cost of the supplement.

\*

\* @param supplementCost The cost of the supplement.

\*/

public void setSupplementCost(float supplementCost) {

this.supplementCost = supplementCost;

}

/\*\*

\* Displays the details of the supplement in the console.

\*/

public void display() {

System.out.println();

System.out.println("Supplement Name: " + supplementName);

System.out.println("Supplement Cost: " + supplementCost);

}

/\*\*

\* Displays the details of the supplement as a JavaFX Node.

\*

\* @return A JavaFX Node containing the details of the supplement.

\*/

public Node displayGUI() {

VBox vbox = new VBox(5); // VBox with a spacing of 5

vbox.getChildren().add(new Label("Supplement Name: " + supplementName));

vbox.getChildren().add(new Label("Supplement Cost: $" + String.format("%.2f", supplementCost)));

return vbox; // Return the VBox as a Node

}

/\*\*

\* Returns the name of the supplement.

\*

\* @return The name of the supplement.

\*/

@Override

public String toString() {

return supplementName; // This will be used by ListView to display the supplement's name.

}

}