## Refactor 1: Commit 43eb08c53171b041dd9805e59aee71948da737b2

The first refactoring is renaming two duplicate classes from two separate subclasses. The classes are called “setID(String userEmail)”, from classes StudentWindow and FacultyWindow. I use the simple **Rename Method** as the name of the method does not actually reveal its purpose. While the method is USED to set the user’s ID, the actual method itself only **gets** the ID from the text database.

Using IntelliJ, I use the IDE to find all instances of the method and where it is used. Luckily, it is only used in the constructor of the class to set the user’s email, and as such I make sure to change the called method name as well for both student and faculty window. I made sure to compile and test the two classes separately before and after the changes with Junit. For the change, for testing, since I call it several times, when I changed the actual method name, I used IntelliJ’s refactor Rename function to change all instances as well to help save time.

public static String getID(String userEmail) {  
 String result = null;  
 try (BufferedReader in = new BufferedReader(new FileReader("UserDatabase.txt"))) {  
 String line;  
 String[] fields = new String[10]; // UserDatabase contains 10 fields  
  
 while ((line = in.readLine()) != null) {  
 fields = line.split("\\\*");  
 // Looks for the ID of the user we need to check  
 if (fields[4].equals(userEmail)) {  
 result = fields[0];  
 }  
 }  
 } catch (FileNotFoundException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return result;  
}

The method was tested by feeding it different user emails, both that were valid and invalid to ensure the exact same outputs. Hence, I used two valid emails from the database, then 3 invalid emails: “invalidemail”, empty string, and “123”. All of the tests were completed successfully. The code is better structured afterwards due to the fact that someone reading the method name would easily be able to tell what the method does, rather than having to read through the whole method to find out that the method actually **gets** the user’s ID, not **sets** it.

Changed files: StudentWindow.java, FacultyWindow.java

Added files: StudentWindowTest.java, FacultyWindowTest.java, PatronWindowTest.java (Used as a test suite)

## Refactor 2: ddcfc6c3e3961bae1379bb3daf834e9c2fb27423

The second refactoring was to remove methods from both the **StudentWindow** and the **FacultyWindow** classes and store the four methods into the **Utilities** class. The four methods were: **getBooks(String), getBookName(String), getDate(String)** and **listBooks(String)**. There was a lot of duplicate methods in both classes, 6 in total. Of them, these four were the most logical to move to a different class. I used the refactoring method **Move Method**, as these methods are technically dealing with the database and therefore should be in the utilities class. I left a reference in the original classes to call the new ones with the parameters, plus the student/faculty’s email if needed. As such, I was able to remove around 100 lines of code in total for duplicated code.

The process of refactoring was first identifying where the methods were being used. As they were only being used in its class, it was pretty simple. For the sake of being able to test, I left the same named methods inside the Students and Faculty Window classes. I then ensured that all of the references were updated; for example, in some methods, I would call Utilities.xxx. This then just became xxx, as it was moved into the Utilities class. Information about the user that the methods needed were just passed as parameters.

*/\*\*  
 \* I create a copy of listbooks that i transfered to utilities in order to call it through the window  
 \* while testing  
 \** ***@return*** *a string format that is either book, due date, ID OR no books currently out  
 \*/*public static String listBooks(){  
 return Utilities.*listBooks*(*email*);  
}  
  
/\*  
 \* given the book the user has, including the dates, it will take out the date  
 \* from the item we want  
 \*   
 \* @param borrowedItem - all the items the user has out  
 \*/  
public static String getDate(String borrowedItem){  
 return Utilities.*getDate*(*email*, borrowedItem);  
}

As a result, the Utilities class has four new methods, while Students and Faculty Window have lost four, while leaving some referencing middle man methods. I tested the code by testing initially getBookName(), which was the first method I was planning on removing. However, after seeing where it was used, ListBooks(), I decided to just test the two larger methods, getDate and listBooks to ensure functionality. I created instances of the class and called the methods with the necessary parameters.

The code is better structured now since I removed a ton of duplicated code, that could’ve been placed elsewhere. The result of the refactoring enables further refactorings if I want to clean up the two window classes even further.

Changed files: StudentWindow.java, FacultyWindow.java, Utilities.java

Changed Junit files: StudentWindowTest.java, FacultyWindowTest.java

# Refactor 3: 4c43d45a6a34d6b40184428415c8b899ae5a82a4

The third refactoring was to change all the possible public methods and variables to either private or package-private. I did this through IntelliJ’s refactoring ability in which they would tell me if I could change the privacy of certain methods. The refactoring method is called **Encapsulate Field.** All the files that were edited will be listed below. The bad code smell was that too many methods were public. One of the pillars of OO programming is encapsulation, yet the project was wide open for anyone to use the objects. To prevent the modularity of program sections.

Again, I mainly used IntelliJ’s built in refactoring ability to change the privacy fields. In addition, I used the “Find all usages” to find any instance of the methods. This way, I could see if it needed to be public, package-private (Only packages, however since all subclasses are in the same package as their parent, it is equivalent to protected, but more tightly bound), or private (class only). From then on, I changed the fields.

public class StudentWindow extends Style {  
   
 public JFrame frame;  
 public static String *email* = null;  
 public static String *studentID* = null;  
 public static int *numBooks* = 0;  
 public JTextField txtReserve;  
   
 */\*\*  
 \* Launch the application.  
 \** ***@param*** *name - name of student  
 \** ***@param*** *email - email of student  
 \*/* public static void StudentS(String name, String email) {  
 EventQueue.*invokeLater*(new Runnable() {  
 public void run() {  
 try {

public class StudentWindow extends Style {  
   
 private JFrame frame;  
 private static String *email* = null;  
 private static String *studentID* = null;  
 private static int *numBooks* = 0;  
 private JTextField txtReserve;  
   
 */\*\*  
 \* Launch the application.  
 \** ***@param*** *name - name of student  
 \** ***@param*** *email - email of student  
 \*/* static void StudentS(String name, String email) {  
 EventQueue.*invokeLater*(new Runnable() {  
 public void run() {  
 try {

Since I could not test ALL the code and ALL the changed methods, since there was more than 20 changed methods, and also due to the fact that a lot of public classes are now private or protected, my Junit tests which were In a different package would not be able to access the methods. Hence, to do my testing, I ensured perfect functionality externally, and went through all the user tasks to make sure that everything worked as it did beforehand.

The code is better structured due to the extra privacy and security from any possible attackers. They would not be able to just easily access a class’s methods. As I changed all possible fields to encapsulate, this refactoring does not enable refactoring **in this stream**, but I am able to refactor further in other directions.

Change files: StudentWindow.java, FacultyWindow.java, Utilities.java, UserDatabase.java, ResourceDatabase.java, Style.java, RestrictBooks.java, AddUser.java, AdminWindow.java, BookSearch.java, ClerkWindow.java, FindUser.java, ItemAdd.java, MessageBox.java, Io.java, LoginWindow.java

## Refactor 4: 4b21b7bd4afd7eb6809ee7ef1840c77698c9bf9b

The fourth refactoring was **Extract Method**. From the StudentWindow was a method called renewBooks that was a **long method**. As such, I discovered an area in the method that I could extract, that dealt with increasing the date by two weeks. Hence I created a new method, called getTwoWeeks that returned a date object that was two weeks later than the passed initial date.

The refactoring that was applied, **Extract Method**, was done by moving a portion of the code into a separate new method and replaced the old code with a call to the new method. As a result, the renewBooks method was shortened in the StudentWindow, and I added the getTwoWeeks method into the utilities class. I tested the code with Junit with one student and a few books. Due to the fact that the renewBooks is a void function and the only measurable side effect would be to access the database and see if the renew date for the book changed, I chose books that would not actually change the due dates and made sure that the expected and actual outcome matched. The reason that I felt that I could not do a book who’s due date would change was due to the fact that there was no “base date” that after running the program, a book would change to. As such, to test for the “expected outcome”, it would have to change with **each** testing call. I felt that this was too unreasonable, hence I only chose books whose due dates would stay constant.

*/\*\*  
 \* Move a portion of renewBook into this method that will calculate the two weeks and return the new date  
 \** ***@param*** *iniDate - the initial return date  
 \** ***@return*** *returnDate - two weeks later  
 \*/*public static Date twoWeekDate(String iniDate){  
 DateFormat sourceFormat = new SimpleDateFormat("ddMMyyyy");  
 Date date = null;  
 Date currentDate;  
  
 // attempts to convert the date string into a date object of the form ddMMyyyy  
 try {  
 date = sourceFormat.parse(iniDate);  
 } catch (Exception e) {  
 System.*out*.println(e);  
 }  
 //uses a calander object to set the date and then add two weeks, ending up with the currentdate = iniDate + 2 weeks  
 Calendar c = Calendar.*getInstance*();  
 c.setTime(date);  
 c.add(Calendar.*DATE*, 14);  
 currentDate = c.getTime();  
  
 return currentDate;  
}

The code is better structured since the renewBooks method is way too long. I was able to shorten it by 10 lines and make it way more understandable. In addition the class I made is very easy to understand and less crowded. The refactoring suggests further refactoring due to the fact that the method is WAY too huge and could be shortened even further.

Changed files: StudentWindow.java, FacultyWindow.java, Utilities.java

## Refactoring 5:

The fifth refactoring was **substitute algorithm.** I altered code inside Utilities class, getDate method. The bad code smell was that I was doing a lot of work to separate items, but then right afterwards, combining those same variables. While it was good to separate them into the different aspects of dates (day, month, year), it was unnecessary work as I recombined them afterwards.

The refactoring, substitute algorithm, occurred by first examining the existing algorithm and seeing if there were any improvements. Since there was a very easy fix, I exchanged the portion of the algorithm with a single line. Before and after the change, I tested the method with a few Junit tests. The resulting code was just the single line, date = bookField.substring(6).

for (int i = 0; i < fields.length; i++) {  
 if (fields[i].startsWith(borrowedItem)) {  
 bookField = fields[i];  
 day = bookField.substring(6, 8);  
 month = bookField.substring(8, 10);  
 year = bookField.substring(10);  
 // from the 14 digit value, first 6 = item ID, then ddmmyyyy  
 date = day + month + year;  
 // bookName = getBookName(bookID);  
 }  
}

for (int i = 0; i < fields.length; i++) {  
 if (fields[i].startsWith(borrowedItem)) {  
 bookField = fields[i];  
 // from the 14 digit value, first 6 = item ID, then ddmmyyyy  
 date = bookField.substring(6);  
 // bookName = getBookName(bookID);  
 }  
}

The testing was the done with Junit tests where I tested multiple borrowed books and compared the output with my expected string. The code is better structured after the refactoring due to the fact that the algorithm became more simple and easy to understand. The refactoring suggests further refactoring in other classes, as some other algorithms follow the same logic and could most likely be improved upon.

Changed files: Utilities.java