CUSTOM DATA STRUCTURES

ALEXANDRU C.

Table of Contents

Introduction

The application presented in this document was designed to be as user friendly as possible. Aside from choosing appropriate data structures and algorithms, memory management was also considered. The Java Virtual Machine's memory management is automated (Garbage collector) and unpredictable which come at the cost of speed.

1. Running the applications and test cases

1.1. Loading projects and running the application (for both projects)

For both parts of this project there are 2 separate project folders clearly named to indicate which project belongs to which part of the coursework. These folders are: "Student manager Part 1" and "Student manager Part 2".

Since both projects use Maven, loading it into the editor should be straight forward. Simply open an existing project in your IDE and navigate to where you have either of the project folders saved on your hard disk and load.

1.2. Running tests

The tests are located inside the "Test Packages" item in NetBeans (or in 'test.java' in IntelliJ).

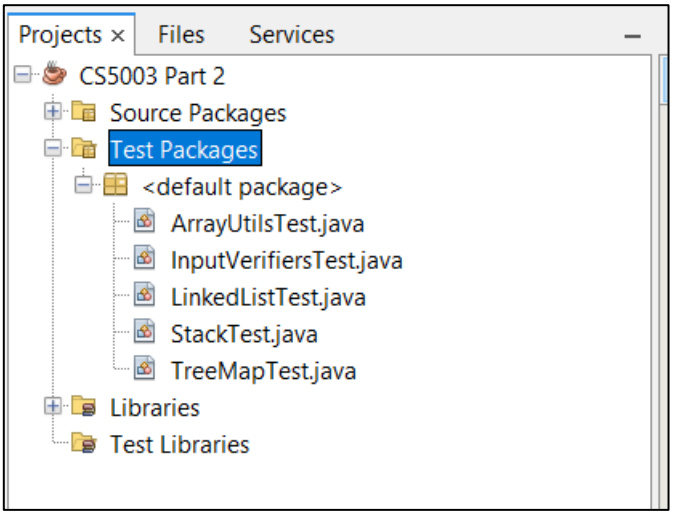

Figure 1. *Location of test cases (NetBeans)*

Right click "<default package>" and click "Test Package" (or 'test.java' and 'Run All Tests' in IntelliJ) to run all the tests at once. Alternatively, each test file can be run individually by right clicking it and selecting "Test File".

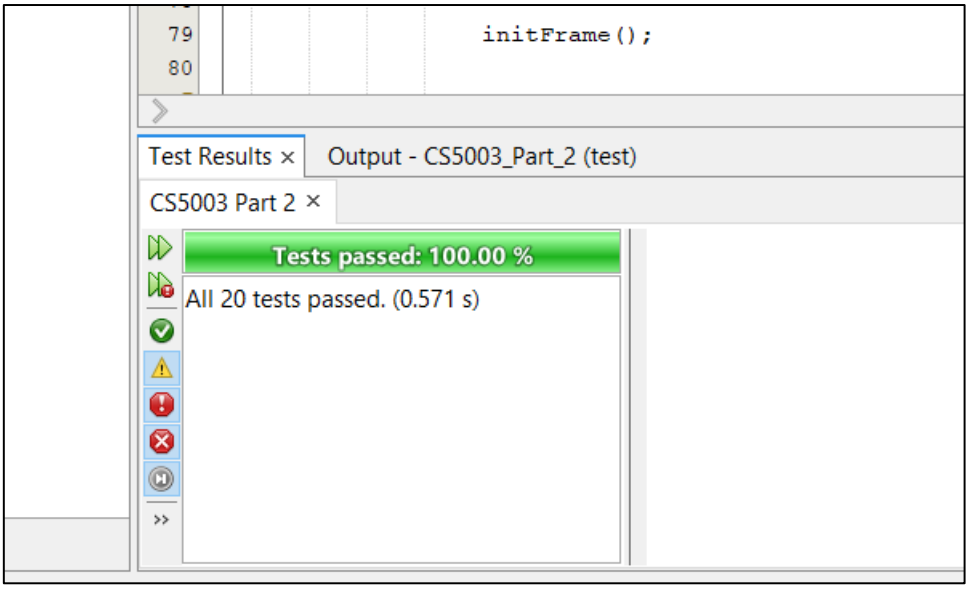

Figure 2. *Test panel (after testing the whole package)*

2. Architecture

For simplicity, the classes of the application can be considered to belong to one of 2 categories: front-end or back-end. The front-end consists of the classes that contain the user interface while the back-end classes are more focused on the functionality “behind the scene”.

Firstly, the front-end classes (the Graphical User Interface) consist of: Window (main class), 2 dialog classes, and 3 other classes which serve as sub-sections of the main window.

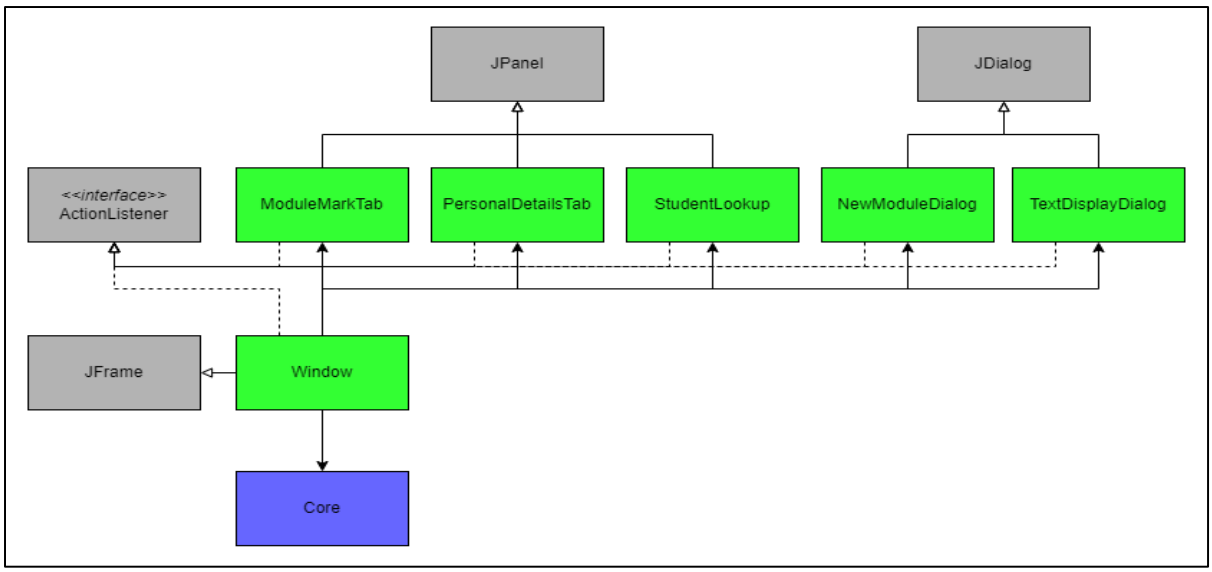

Figure 3. *UML visualisation of front-end (Part 1 and Part 2)*

The classes coloured in grey in **Figure 3** are Java's native classes. The green classes are front-end classes while the blue coloured class is a back-end class.

Before moving on to including the back-end into the diagram, it is important to mention that not all classes can be illustrated in an UML diagram. For example, there is a class named "AppLog" that can be accessed anywhere within the program and is used to log events within the application.

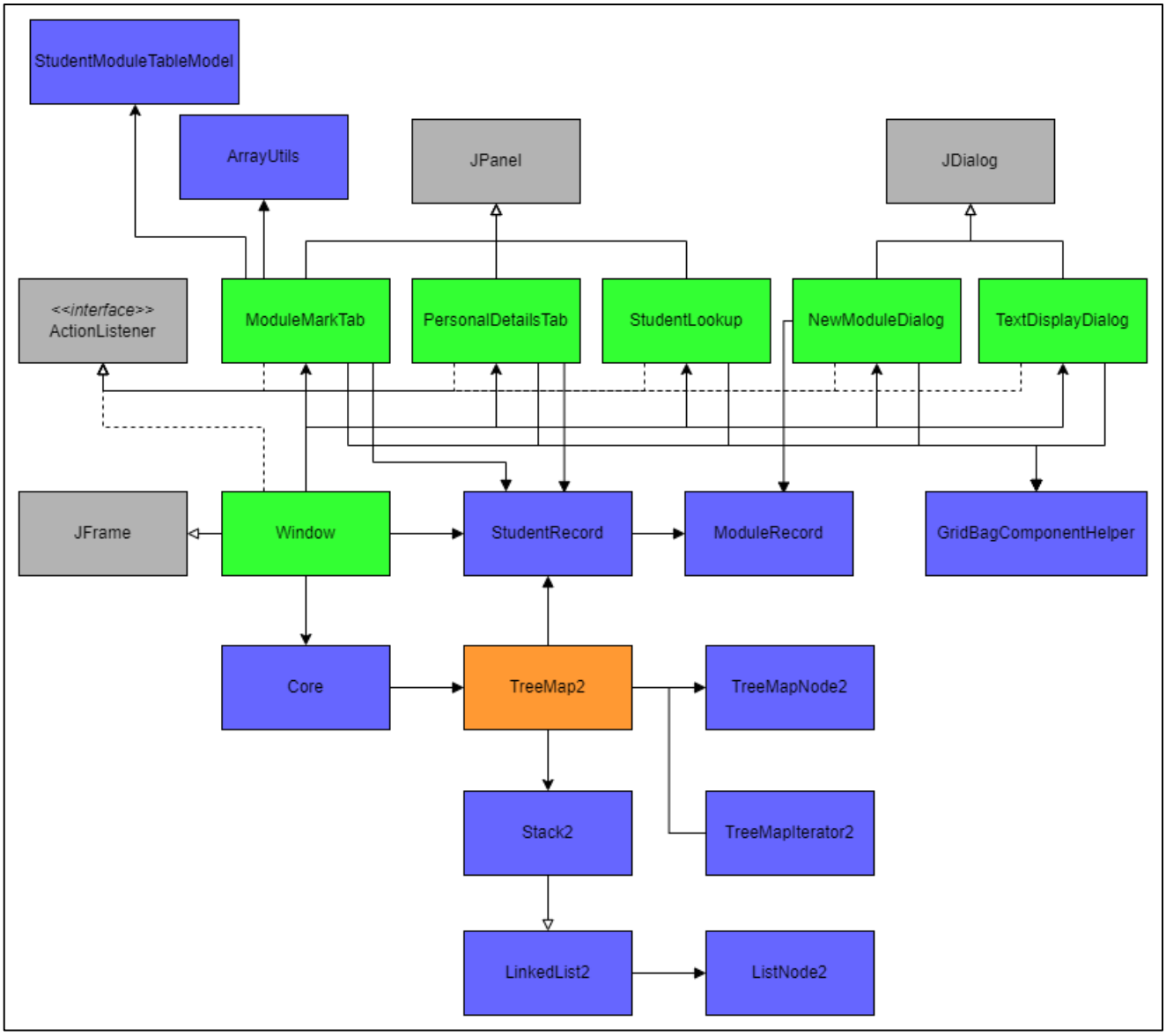

Figure 4. *UML visualisation of both front and back end*

The class coloured in amber is a back-end class as well. It is coloured differently because it was inspired by other sources in its implementation. The blog “Happy Coders” has a comprehensive guide to all sorts of data structure and algorithms. What was of particular interest was the Red-Black tree algorithm as this is what the original “TreeMap” class uses in its implementation (Available at: https://www.happycoders.eu/algorithms/red-black-tree-java/).

The stack and linked list data structures were implemented by looking at the Java documentation and inspecting how the original structures behave. The custom implementations were tested against their original Java counterpart.

The “ArrayUtils” contains 2 utility functions. One is to reverse arrays and the other one is to sort elements of an array. The sorting function implemented is insertion sort as it gives good results for very small arrays.

3. Class and methods description

3.1. AppLog class

This class was implemented to provide the application with basic logging functionality. It outputs a formatted string to the console and to a unique file with events happening in the application.

Method	Description
`public static void initialize()`	Initializes the print writer object of the class for file output.
`public static void close()`	Closes the print writer object of the class at the end of the program.
`public static void err(Object src, String msg)`	Prints to console and file a formatted error message.
`public static void info(Object src, String msg)`	Prints to console and file a formatted information message.
`public static void printStackTrace(Object src, StackTraceElement[] stack_trace)`	Prints to console and file the stack trace (java generated error messages).

3.2. Core class

Contains a data structure to store student records in along with data manipulation functions and file operations.

Method	Description
`public boolean addNewStudent(StudentRecord s)`	Adds a new student to the record. Returns true if the operation was successful.
`public boolean getStudentRecord(String ID, StudentRecord buffer)`	Retrieves a copy of the student record inside the provided buffer, given a student ID. Returns true if the operation was successful.
`public boolean deleteStudentRecord(String ID)`	Deletes a student record. Returns true if the operation was sucesful.
`public boolean updateStudent(String ID, StudentRecord updated_student)`	Updates a student record by copying new data from updated student into the old record. This is to avoid mixing object reffrences. Returns true if the operation was successful.
`public int getStudentCount()`	Returns the current number of registered students.
`public String[] getDisplayStringStudents()`	Returns an array of formatted strings that represent each student record's data.
`public void newFile()`	Clears records and starts a new file.
`public boolean openFile(String path)`	Opens and loads a file.
`public boolean saveFile()`	Saves an already in use file.
`public boolean saveFileTo(String path)`	Saves the current file to a specified location.

3.3. NewModuleDialog class

A custom dialog designed for entering module details.

Method	Description
`public boolean showModuleDataInputDialog( ModuleRecord buffer)`	Displays the dialog. Returns true if the data was successfully input or false otherwise.
`public void actionPerformed(ActionEvent e)`	Handles input events from the interface.

3.4. TextDisplayDialog class

A simple dialog used to display text information.

Method	Description
`public boolean showDialog(String ... strings)`	Displays the dialog.
`public void actionPerformed(ActionEvent e)`	Handles input events from the interface.

3.5. Window class

Method	Description
`public static void main(String args[])`	Entry point of the application.
`private void initFrame()`	Initialises the JFrame with its components.
`private void initMenuBar()`	Initialises the window's menu bar.
`public void addNotify()`	Overridden function. Used to notify when the window was created.
`public void actionPerformed(ActionEvent e)`	Overridden function. Handles input events from the buttons present in the window.
`private void newFile()`	Clears form and starts a new file.
`private void openFile()`	Displays a file chooser dialog and loads the file selected if the operation was not cancelled.
`private void saveFile()`	Updates the previously saved file if there is one or creates a new one if there is no previous file.
`private void saveTo()`	Saves the current file to a location on the hard disk specified by the user.
`private void viewAll()`	Opens a new dialog in which all student records can be viewed.
`public void clearForm()`	Clears the form and unloads any loaded student.
`private void loadStudent(String search_ID)`	Loads a student into the form.
`private void addStudent()`	Adds a new student if all form entries are valid.
`private void deleteStudent()`	Deletes the currently loaded student.
`private void updateStudent()`	Updates the details of the currently loaded student. If the student ID has been changed, then the student will be removed and added with the updated data instead.
`private void newModule()`	Opens a dialog for entering module details and adds the new module if the user does not cancel the action. It also refreshes the module tab.
`private void removeModule()`	Removes a module if there is any selected in the JTable.

3.6. StudentLookup class

This class contains the interface components for the student lookup panel.

Method	Description
`public void addNotify()`	Overridden function. Used to tell when the panel has been successfully added to the window.
`public void clearForm()`	Clears the entries of the this form.
`public String getSearchID()`	Returns the content of the search box.

3.7. PersonalDetailsTab class

This class contains the interface components for the personal details tab of the main window.

Method	Description
`public void addNotify()`	Overridden function. Used to tell when the panel has been successfully added to the window.
`public void clearForm()`	Clears the entries of the this form.
`public void updateForm(StudentRecord buffer)`	Copies data from the buffer into the form.
`private boolean validateEntries()`	Validates entries, checks if they are not null. Entries are already restricted by custom input verifiers.
`public boolean getValidFormEntries(StudentRecord buffer)`	Validates entries and copies them into the given buffer. Returns true if the operation succeeded or false otherwise.

3.8. ModuleMarkTab class

This class contains the interface components for the module details tab of the main window. It makes use of a JTable to display the module information.

Method	Description
`public void addNotify()`	Overridden function. Used to tell when the panel has been successfully added to the window.
`public void clearForm()`	Clears the entries of the this form.
`public void updateForm(StudentRecord buffer)`	Copies modules, sorts them in descending order and adds them to the JTable of this class.
`private void refreshTable()`	Updates the data displayed inside the JTable.
`public String getModuleCodeOfSelected()`	Returns the code of the module that is selected in the JTable.

3.9. StudentModuleTableModel class

This class overrides the “AbstractTableModel” class which is a Java native class. The purpose of this class it to easily create and initialize an empty JTable based on a specified model. All methods present in this class are overridden. This class contains a String array with the column names and a empty 5x2 empty table object.

Method	Description
`public int getColumnCount()`	Returns the number of column in the JTable.
`public int getRowCount()`	Returns the number of rows in the JTable.
`public String getColumnName(int col)`	Returns the name of the column at the specified index.
`public Object getValueAt(int row, int col)`	Returns the value at the specified position in the JTable.
`public boolean isCellEditable(int row, int col)`	Returns true if a cell is editable or not. NOTE: this function was overridden to always return false
`public void setValueAt(Object value, int row, int col)`	Updates a value in the JTable at the specified column.

3.10. GridBagComponentHelper class

A single instance object that can be accessed from anywhere in the program. The GridBagComponentHelper was implemented to help with the tedious and repetitive task of adding components to an interface that uses the GridBag layout.

Method	Description
`public static GridBagComponentHelper getInstance()`	Returns the instance associated with this class. The instance is initialised before return if it was not already.
`private void resetPosition()`	Resets the position at which the next element will be added to the top left corner.
`public void begin(Container parent)`	Used to tell the GridBacComponentHelper where to add the components to.
`public void addComponent(JComponent child, int spanWidth, int spanHeight, double weightX, double weightY)`	Adds a component to the earlier specified parent component with a given span, filling the available space available in its cell according to the given weight values.
`public void addComponent(JComponent child, int spanWidth, int spanHeight)`	Overloaded function. Default value of 1f for weightX and weightY.
`public void addComponent(JComponent child, double weightX, double weightY)`	Overloaded function. Default value of 1 for spanWidth and spanHeight.
`public void addComponent(JComponent child)`	Overloaded function. Default value of 1 for spanWidth, spanHeight, weightX, and weightY.
`public void newRow()`	After this function is called, the next element will be added on the next row (1 row below).
`public void end()`	Forgets (removes) the current parent object and resents the inserting position of the elements.

3.11. Input verifier classes

There are a few input verifiers that all extend Java's native class "InputVerifier". They all override the "verify" function which returns true if the input is valid or false otherwise. The verifiers check for different input patterns however they all share an ability to display an error custom warning message if the input is wrong. This message is set when calling the constructor of any verifier along with whether it should display a warning or not. Only the alphanumeric and alphabetic verifier can also be set to allow for white spaces to be input.

3.12. ArrayUtils class

A class providing static access to useful operations on arrays.

Method	Description
`public static <T extends Comparable<T>> void insertionSort(T[] arr)`	Sorts an array of any kind of comparable elements using insertion sort.
`public static void reverse(Object[] arr)`	Puts the elements of an object array in reverse order.

3.13. StudentRecord class

A class for holding student data. Student's modules are stored in a private fixed size array to. prevent unwanted alterations. The personal details however have public access.

Method	Description
`public void addModule(String code, int mark)`	Adds a module into the student record. If the number of modules added exceeds the capacity of the array specified by MODULES_SIZE then the oldest modules added will be removed to make room.
`public void removeModule(String code)`	Removes the first modules that has its code matching the passed string
`public void clearModules()`	Clears all the registered modules in the modules array.
`public void copyModulesTo(ModuleRecord[] dest)`	Copies the module data from the student array into the passed destination array. Avoids giving direct access to the elements of the array
`public void copyFrom(StudentRecord original)`	Copies in all the data from another student object. Avoids mixing references of member objects.
`public String getDisplayString()`	Returns a formatted string that represents the student record.
`public boolean contains(String code)`	Returns true if there is a module that has a code that matches the passed string.

3.14. ModuleRecord class

A class for holding module data.

Method	Description
`public int compareTo(ModuleRecord m)`	Compares two modules. The comparison is done on marks.

3.15. TreeMap2 class

A “Red-Black tree” based implementation of a Binary Search Tree (BST) where values are mapped to a unique key.

Method	Description
`public int size()`	Returns the number of key-value pairs in the tree map.
`public V get(K key)`	Returns a value associated with the given key.
`private TreeMapNode2<K,V> searchNode(K key)`	Returns the tree node that contains the given key.
`public K putIfAbsent(K key, V value)`	Inserts a value in the tree only if the given key is not present in the tree map (does not have a value associated with it).
`private void insertNode(K key, V value, boolean replaceAllowed)`	Inserts or replaces the key-value pair into the tree.
`private TreeMapNode2<K,V> deleteNode(K key)`	Removes the node that corresponds with the specified key if it exists. (Red-Black tree algorithm function)
`public V remove(K key)`	Removes the given key from the tree (if it exists) along with the associated value.
`public void clear()`	Removes all the nodes of the tree.
`private void fixPropertiesPostInsert( TreeMapNode2<K,V> node)`	Balances the tree and fixes its colouring after node insertion. (Red-Black tree algorithm function)
`private void fixPropertiesPostDelete( TreeMapNode2<K,V> node)`	Balances the tree and fixes its colouring after node deletion. (Red-Black tree algorithm function)
`private TreeMapNode2<K,V> findMinimum(TreeMapNode2<K,V> node)`	Finds the lowest value node in a subtree that starts with the given node.
`private TreeMapNode2<K,V> getSibling(TreeMapNode2<K,V> node)`	Returns the 'sibling' of the given node.
`private TreeMapNode2<K,V> getUncle(TreeMapNode2<K,V> parent)`	Returns the 'uncle' of the given node.
`private TreeMapNode2<K,V> deleteNodeWithMaxOneChild(TreeMapNode2<K,V> node)`	A helper function that is called only on nodes with a missing child node or a leaf node.
`private void replaceParentsChild( TreeMapNode2<K,V> parent, TreeMapNode2<K,V> oldChild, TreeMapNode2<K,V> newChild)`	A helper function that assigns to the parent node a new child in the place of the old one.
`private void rotateRight(TreeMapNode2<K,V> node)`	Rotates a subtree to right.
`private void rotateLeft(TreeMapNode2<K,V> node)`	Rotates a subtree to left.
`private boolean isBlack(TreeMapNode2<K,V> node)`	Returns true if the colour of the given node is black.
`public Iterator<V> iterator()`	Returns an iterator object for this data structure.
`public String toString()`	Returns a string representation of the data structure.

3.16. TreeMapIterator2 class

An iterator class for the TreeMap2 data structure. Iterates over the nodes of the tree in ascending order of the key value of the nodes.

Method	Description
`private void fill(TreeMapNode2<K,V> node)`	Because the iteration is done in ascending order, this function will fill the stack with all the elements on the left branch of any given node because they are the smallest.
`public boolean hasNext()`	Returns true if there is a next element (stack is not empty).
`public V next()`	Returns the next value.
`public TreeMapNode2<K,V> nextNode()`	Returns the next tree map node.

3.17. TreeMapNode2 class

This class is used to store key-value pairs and link them to other key-pairs. There are 3 links appropriately named: parent, left, and right.

Method	Description
`private void fill(TreeMapNode2<K,V> node)`	Because the iteration is done in ascending order, this function will fill the stack with all the elements on the left branch of any given node because they are the smallest.
`public boolean hasNext()`	Returns true if there is a next element (stack is not empty).
`public V next()`	Returns the next value.
`public TreeMapNode2<K,V> nextNode()`	Returns the next tree map node.

3.18. Stack2 class

A reimplementation of Java's stack data structure ("Stack" class). Makes use of a linked list (also reimplemented).

Method	Description
`public boolean empty()`	Returns true if the stack is empty.
`public T peek()`	Returns the top element of the stack (tail of the list).
`public T pop()`	Removes and returns the top of the stack.
`public T push(T data)`	Adds an element on top of the stack.
`public int search(Object o)`	Returns the index in the stack of the passed object if it was found or -1 otherwise.
`public int size()`	Returns an integer representing the size of the stack.
`public String toString()`	Returns a string representation of the stack.

3.19. ListNode2 class

This classed is used to store data of list's item and link it to other items of the list. It has 2 links appropriately named: previous and next. There are no methods for this class.

3.20. LinkedList2 class

A reimplementation of Java's "LinkedList" class.

Method	Description
`public int size()`	Returns the number of items in the list.
`public int indexOf(Object element)`	Searches the list for the given object. Returns the index at which it was found or -1 otherwise.
`public void add(T data)`	Adds a new element at the end of the list.
`public boolean addAll(Collection<? extends T> c)`	Adds all the elements of a collection to the end of the list.
`public void add(int index, T data)`	Adds an element in the list at the given index.
`private void insertNodeToLeft(ListNode2<T> newNode,ListNode2<T> dest)`	A helper function. Inserts a new node the the left of the destination node. If the destinations is the head of the list then the new element becomes the new head of the list.
`public void addFirst(T data)`	Adds an element at the head of the list.
`public void addLast(T data)`	Adds an element at the tail of the list.
`public boolean remove(Object o)`	Removes the first occurrence of the given object. Returns true if the operation was successful or false otherwise.
`private void removeNode(ListNode2<T> node)`	Removes the given node.
`public T remove(int index)`	Removes the element at the specified index and returns it.
`public T removeFirst()`	Removes the element at the head of the list and returns it.
`public T remove()`	Removes the element at the head of the list and returns it.
`public boolean removeFirstOccurrence(Object o)`	Removes the first occurrence of the given object. The search is done from head to tail. Returns true if the value was found and removed or false otherwise.
`public T removeLast()`	Removes and returns the element at the tail of the list.
`public boolean removeLastOccurrence(Object o)`	Removes the last occurrence of the given object. This is done by looking for the first occurence when searching from tail to head. Returns true if the value was found and removed or false otherwise.
`public void clear()`	Removes all elements of the list.
`public boolean contains(T check_data)`	Returns true if the data was found in the list or false otherwise.
`public T get(int index)`	Returns the element at the specified index.
`public T getFirst()`	Returns the element at the head of the list.
`public T getLast()`	Returns the element at the tail of the list.
`public T set(int index, T data)`	Changes the value of the element at the given index.
`public Object[] toArray()`	Returns an array containing the elements of the list.
`public T[] toArray(T[] a)`	Copies te elements of the list into the given array or returns a new one if the given array does not have sufficient space.
`public String toString()`	Returns a string representation of the list.
`public Iterator<T> iterator()`	Returns an ascending iterator object for the list.
`public Iterator<T> descendingIterator()`	Returns a descending iterator object for the list.
`private void checkIndex(int index)`	Checks if the index is within bounds and throws an IndexOutOfBounds exception if the index is not.
`private void copyElementsToArray(Object[] a)`	A helper function that copies the elements of the list into an array.

4. Choice of data structures and algorithms

In order to contain the data of the student and modules they take, the “StudentRecord” and “ModuleRecord” classes were created. They give structure to the data that is being worked on. The instances of these classes are subsequently stored in a “TreeMap” object. A map is great for storing students because the ID will always be known when searching for a student. The tree structure provides a fast way to search an item, which (in this project) is essentially a binary search. The search time for an element is O(h), where h is the height of the tree. Java’s “TreeMap” was implemented using the “Red-Black” algorithm for binary search trees which means the height of the tree is optimised (self-balancing tree). Inserting or removing elements is slow compared to the same operations for other structures like linked lists or arrays. However, it is very unlikely that the program will spend a considerable amount of time on these operations as they are triggered manually by the user per record operation (add, remove, update, delete).

For storing modules, a regular array was used. This was done to avoid allocating new memory for objects only to be disposed moments later. This is important because there will be frequent operations for adding or removing modules associated with a student. If a linked list were to be used, for example, any removed list items will remain in memory until Java’s garbage collector frees their memory space. The garbage collector is often slow and unpredictable. Another reason for choosing arrays is that there is an upper limit (given requirement) to how many modules a student can have in any given year.

The concept of buffers was an influence in implementing a way to move data from the interface to the applications core.

When it come to sorting arrays, insertion sort was chosen because it works best on very small arrays compared to other sorting algorithms despite having a bad time complexity as the number of elements grow. It is also memory efficient.

5. Testing

After implementing all custom data structures, they were tested against their original Java class. This was done with the help of JUnit test cases. Each custom data structure was tested function by function and compared the behaviour with the original class.

First class to be tested was “LinkedList2”. All the tests here were passed on the first run.

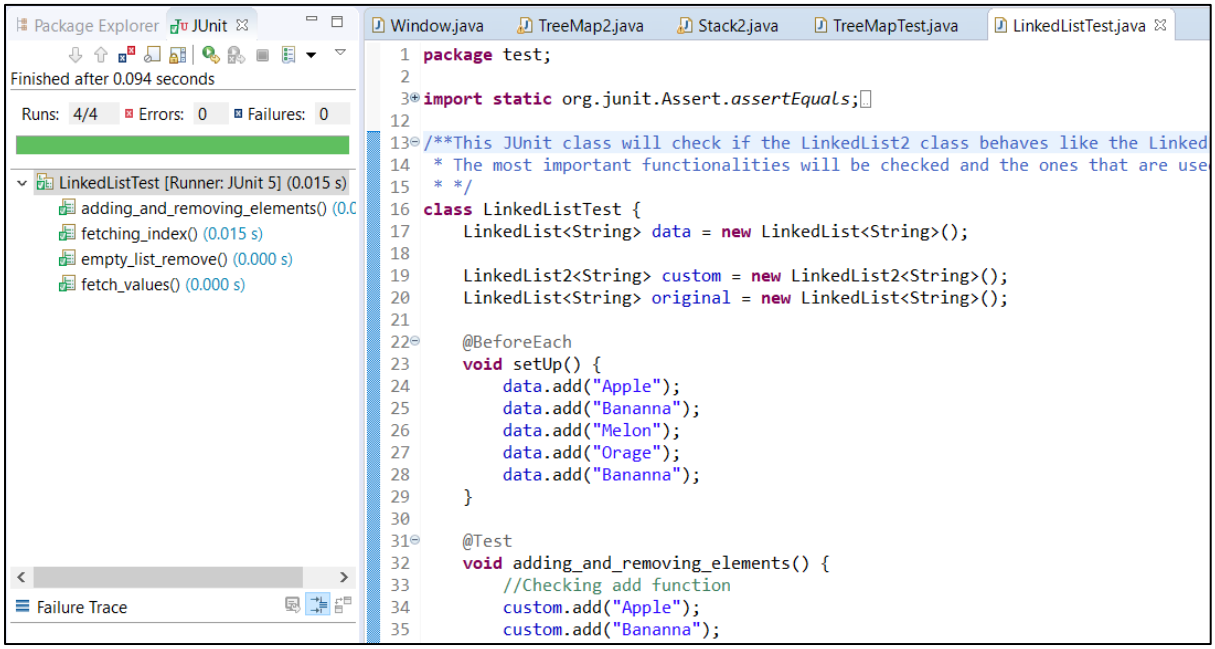

Figure 5. *Testing “LinkedList2” class*

Next was the stack class. Here is where the first issues started to reveal themselves. In most aspects the custom data structure behaved like me original one. When trying to get the index of an element in the stack it came to light that the elements were being added and popped from the stack in reverse order.

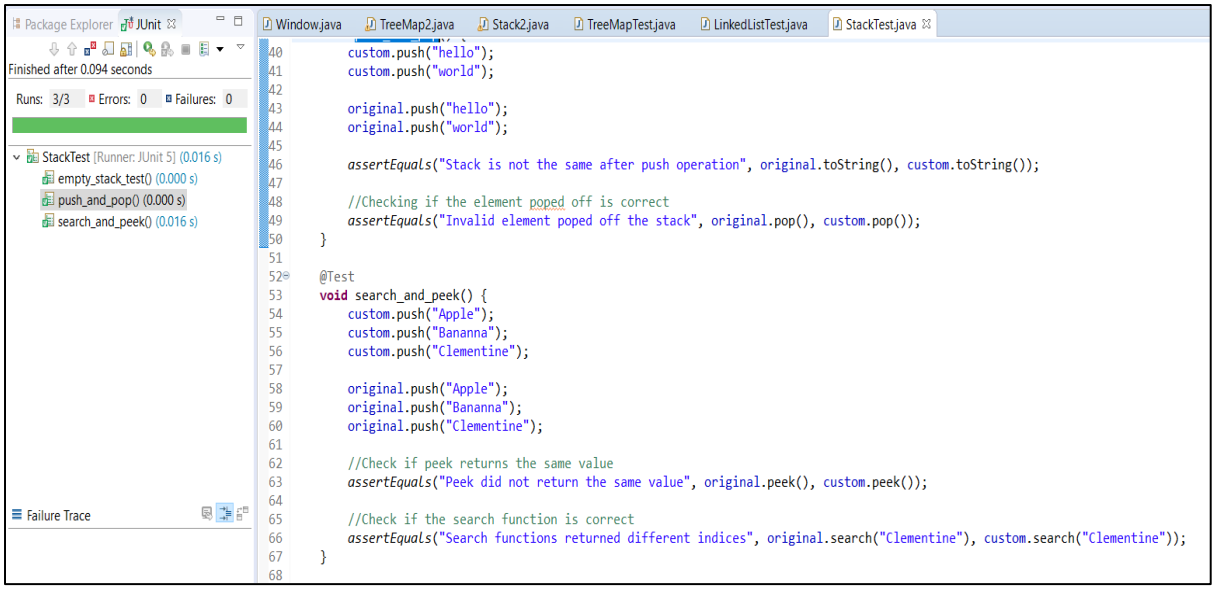

Figure 6. *Testing "Stack2" class*

Everything became more complicated with trees. The issue was with deletion. In the custom class, when deleting a node the key was deleted along with the node but the value got moved and overwrote the value of other keys. After this was fixed the issue became null pointers when the node to be deleted did not exist. The key to fixing this was a simple change in the order in which the instructions were executed.

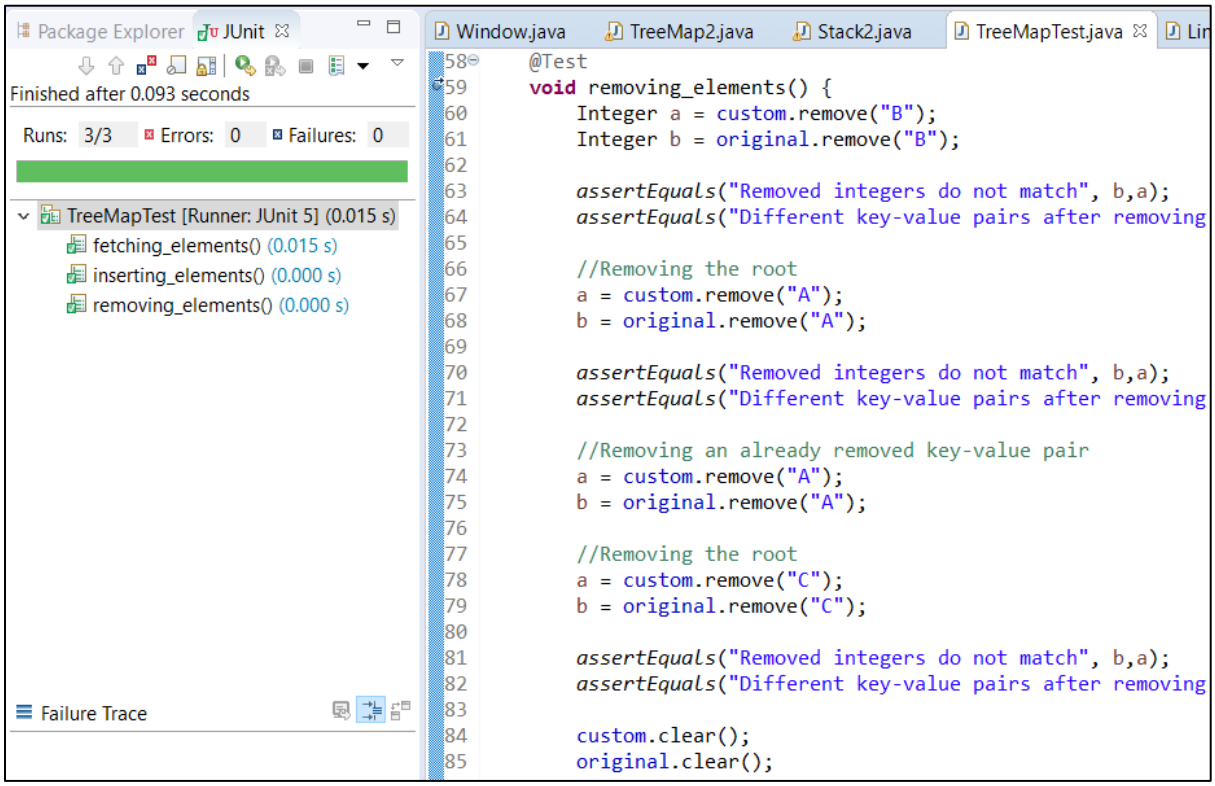

Figure 7. *Testing "TreeMap2" class*

A similar approach was taken when testing the "ArrayUtils" class, specifically the insertion sort and reversing arrays. Just as with the "LinkedList2" class, the "ArrayUtils" tests also run without generating any errors.

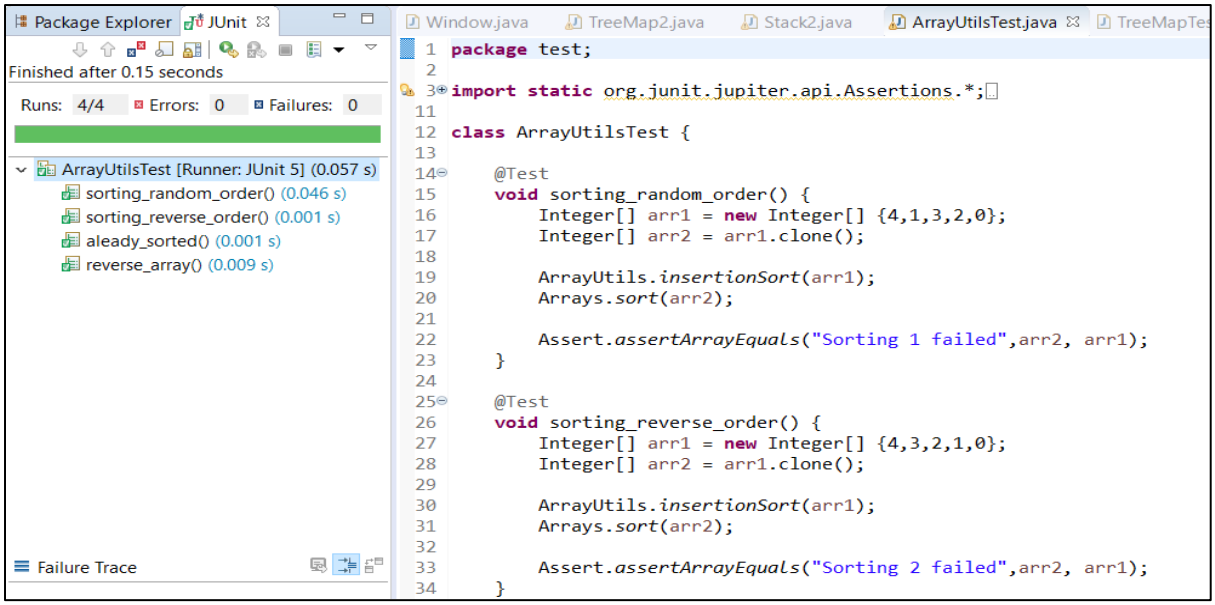

Figure 8. *Testing "ArrayUtils" class*

Test cases were applied to input verifiers too in an attempt to see if there is a string for which the validation fails. Only 1 test case for the email verifier failed when passing just a dot for the email account portion of the address. This was patched and now the email must start with a letter.

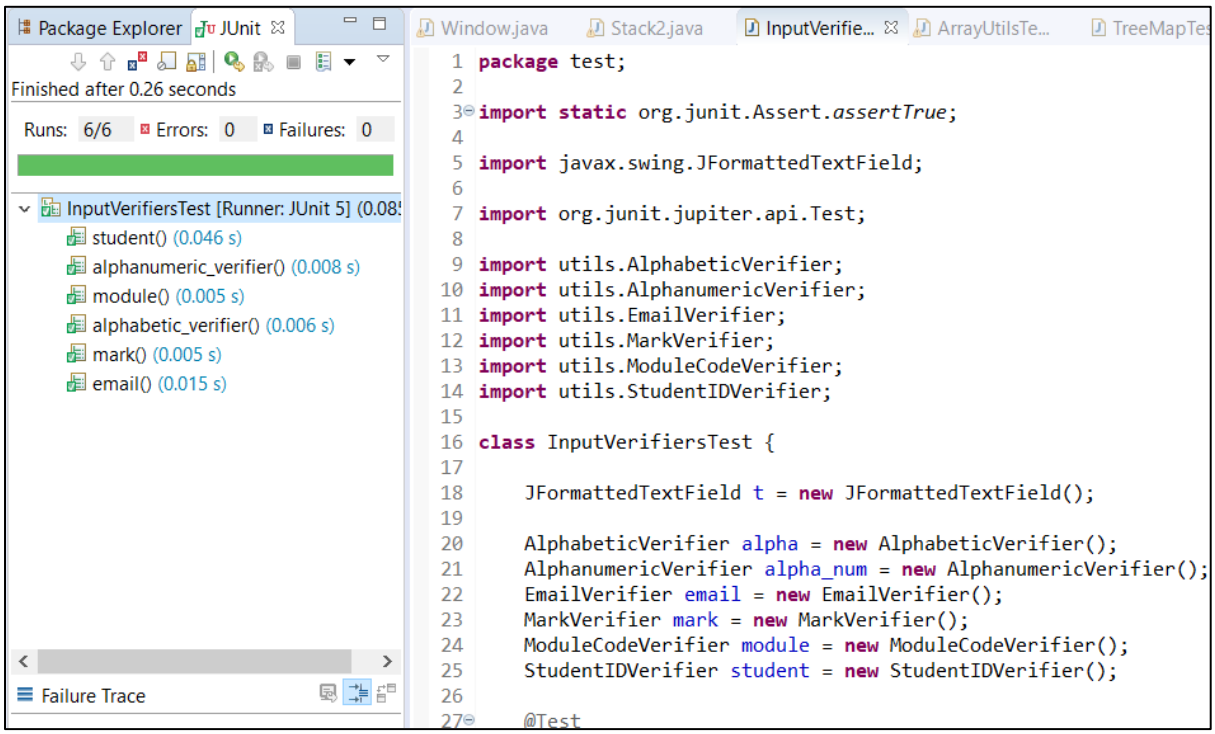

Figure 9. *Testing all implemented input verifiers*

6. Reflection

Trying to rewrite a Collections class is difficult as there are many classes involved in a single data structure and it's easy to get lost in information and code. The greatest challenge, however, was writing the GUI (especially the JTable). Writing code for user interfaces can rapidly pile up and become obscure. It's important to separate everything into different classes and apply good design patterns for such applications. For me, implementing the interface of this application was yet another reminder of how essential proper design and implementation is.

Through this project, I've gained new insight into design patterns and I've seen how many classes in the Collections Framework combine with one another, like pieces of a puzzle, to form something greater.

References

1. Lars V. , Vogella (17.08.2021) “JUnit 5 tutorial – Learn how to write unit tests”. Available at: https://www.vogella.com/tutorials/JUnit/article.html (Accessed: 1 January 2022)
2. PY4U “How to remove extra white space from the bottom of the JTable”. Available at: https://www.py4u.net/discuss/597731 (Accessed: 1 January 2022)
3. Raja, TutorialsPoint (24.07.2019) “How can we disable the cell editing inside a JTable in Java?”. Available at: https://www.tutorialspoint.com/how-can-we-disable-the-cell-editing-inside-a-jtable-in-java (Accessed: 1 January 2022)
4. Oracle “How to Use BoxLayout”. Available at: https://docs.oracle.com/javase/tutorial/uiswing/layout/box.html (Accessed: 1 January 2022)
5. StackOverflow (16.05.2011) “JformattedTextField Question”. Available at: https://stackoverflow.com/questions/6009371/jformattedtextfield-question (Accessed: 1 January 2022)
6. Wikipedia (28.12.2021) “Email address”. Available at: https://en.wikipedia.org/wiki/Email_address (Accessed: 1 January 2022)
7. W3schools “Java Regular Expressions”. Available at: https://www.w3schools.com/java/java_regex.asp (Accessed: 1 January 2022)
8. Lars V. , Vogella (29.07.2021) “Regular expressions in Java”. Available at: https://www.vogella.com/tutorials/JavaRegularExpressions/article.html (Accessed: 1 January 2022)
9. Oracle “How to Use Tables”. Available at: https://docs.oracle.com/javase/tutorial/uiswing/components/table.html#validtext (Accessed: 1 January 2022)
10. StackOverflow (12.09.2012) “Combining InputVerifiers”. Available at: https://stackoverflow.com/questions/12394162/combining-inputverifiers/12394524#12394524 (Accessed: 1 January 2022)
11. javaTpoint “Object Cloning in Java”. Available at: https://www.javatpoint.com/object-cloning (Accessed: 1 January 2022)
12. StackOverflow (11.08.2011) “Why are static variables considered evil?”. Available at: https://stackoverflow.com/questions/7026507/why-are-static-variables-considered-evil (Accessed: 1 January 2022)
13. StackOverflow (25.04.2009) “Java – How to create a custom dialog box?”. Available at: https://stackoverflow.com/questions/789517/java-how-to-create-a-custom-dialog-box/790224 (Accessed: 1 January 2022)
14. Oracle “LinkedList (Java Platform SE 7)”. Available at: https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html (Accessed: 1 January 2022)
15. Anant M. , Dzone (16.08.2019) “Performance Analysis of ArrayList and LinkedList in Java”. Available at: https://dzone.com/articles/performance-analysis-of-arraylist-and-linkedlist-i (Accessed: 1 January 2022)
16. Programming Examples (09.03.2021) “Performance Study – ArrayList vs TreeSet”. Available at: https://coding-examples.com/java/performance-study-arraylist-vs-treeset/ (Accessed: 1 January 2022)

17. Sven W. , HappyCoders (29.09.2021) “Red-Black Tree (Fully Explained, with Java Code)”. Available at: https://www.happycoders.eu/algorithms/red-black-tree-java/ (Accessed: 1 January 2022)