

Criterion C: Development

Details of computer system used for development:

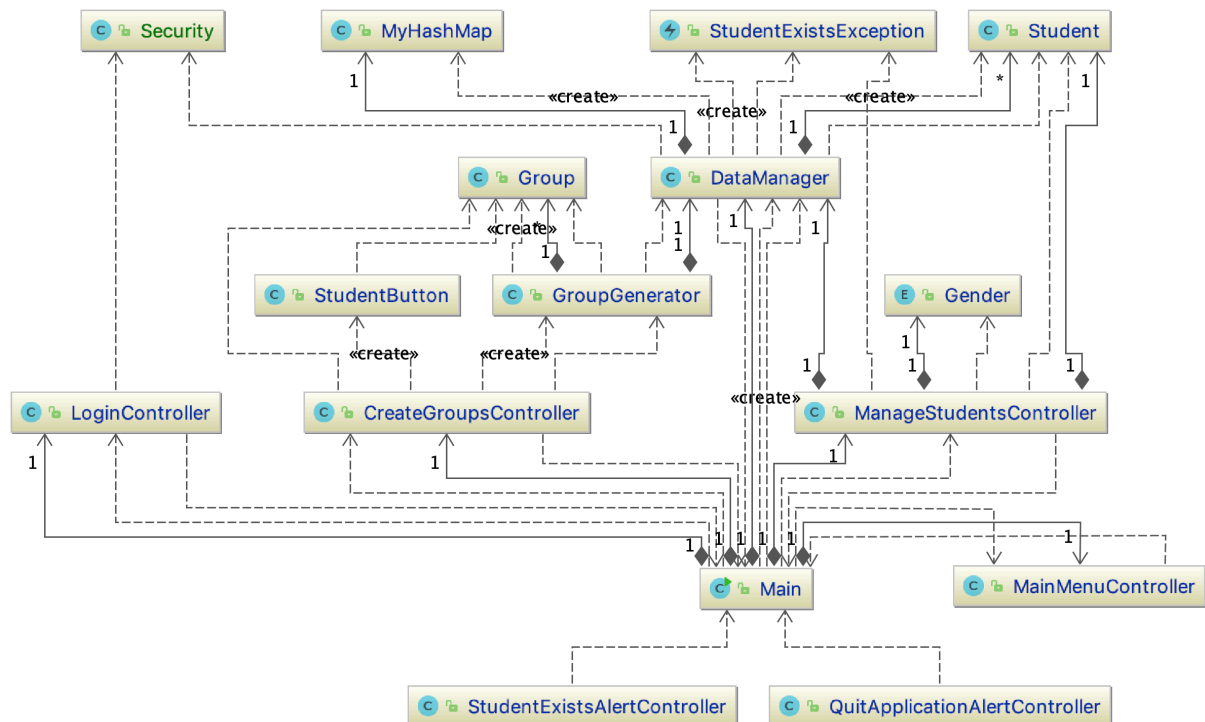
JDK version: javac 11.0.1

JRE version: java version 11.0.1

OS: macOS Mojave 10.14.3

IDE: JetBrains IntelliJ IDEA 2018.3

Diagram of class dependencies



List of techniques used

1. Group-generation algorithms (dictionary and dynamic array)
2. Implementing a custom data structure
3. Encryption
4. Cascade stylesheets
5. Bidirectional updating of elements
6. Boolean binding
7. Handling commits of editable JavaFX TableView cells
8. Nested combo box in table cell
9. Creating a custom class
10. UTF-8 encoding
11. Enum type
12. GitHub

1. Group-generation algorithms (dictionary and dynamic array)

If a specific group size is requested, the number of groups is calculated accordingly.

```
public ArrayList<Group> generate (DataManager dataManager, int num, boolean isGroupSize,
                                boolean filterGender, boolean filterClass) {
    this.dataManager = dataManager;
    if (isGroupSize) {
        numOfGroups = (int) dataManager.getNumOfStudents() / num;
    } else {
        numOfGroups = num;
    }

    studentList = new ArrayList<>(dataManager.getNumOfStudents());
    if (!filterGender) {
        if (!filterClass) {
            algorithmA();
        } else {
            algorithmB();
        }
    } else {
        if (!filterClass) {
            algorithmC();
        } else {
            algorithmD();
        }
    }
    return groupify();
}
```

Figure 1: Function that triggers a relevant group-generation algorithm based on input criteria

```
private ArrayList<Group> groupify() {
    groups = new ArrayList<Group>();

    int groupNum = 0,
        studentNum = 0;

    for (int i = 0; i < numOfGroups; i++) groups.add(new Group(i+1));
    while (studentNum < dataManager.getNumOfStudents()) {
        groups.get(groupNum).add(studentList.get(studentNum));
        studentNum += 1;
        groupNum = (groupNum + 1) % numOfGroups;
    }
    return groups;
}
```

Figure 2: Function that creates groups after a group-generation algorithm was executed

```
private void algorithmB() {
    HashMap<String, ArrayList<String>> hm = new HashMap<>();
    dataManager.getStudents().forEach(s -> {
        // Initialize new array list
        if (!hm.containsKey(s.getPreviousClass())) {
            hm.put(s.getPreviousClass(), new ArrayList<>());
        }
        hm.get(s.getPreviousClass()).add(s.getName());
    });
    for (String key: hm.keySet()) {
        Collections.shuffle(hm.get(key));
        studentList.addAll(hm.get(key));
    }
}
```

Figure 3: Group-generation algorithm that disperses students from the same previous class

```

private void algorithmD() {
    HashMap<String, ArrayList<ArrayList<String>>> hm = new HashMap<>();
    DataManager.getStudents().forEach(s -> {
        // Initialize new array list
        if (!hm.containsKey(s.getPreviousClass())) {
            hm.put(s.getPreviousClass(), new ArrayList<>(2));
            hm.get(s.getPreviousClass()).add(new ArrayList<>());
            hm.get(s.getPreviousClass()).add(new ArrayList<>());
        }
        if (s.isMale()) {
            hm.get(s.getPreviousClass()).get(0).add(s.getName());
        } else {
            hm.get(s.getPreviousClass()).get(1).add(s.getName());
        }
    });

    for (String key: hm.keySet()) {
        ArrayList<String> males = hm.get(key).get(0);
        ArrayList<String> females = hm.get(key).get(1);
        Collections.shuffle(males);
        studentList.addAll(males);
        Collections.shuffle(females);
        studentList.addAll(females);
    }
}

```

Figure 4: Group-generation algorithm that disperses both, same-gender and same-class students

Both group-generation algorithms implement the method shuffle to ensure that the created groups are generated non-deterministically. The use of a dictionary (HashMap) allowed me to access a value using a key (e.g. a student's class) rather than an index. The use of a dynamic array (ArrayList) as value in the HashMap allowed me to store various numbers of students without having to predefine the capacity.

This technique ensures that the user can create groups based on a number and that the user can create groups based on group size or number of groups; gender and/or previously attended class, or neither, as requested by the success criteria.

2. Implementing a custom data structure: MyHashMap

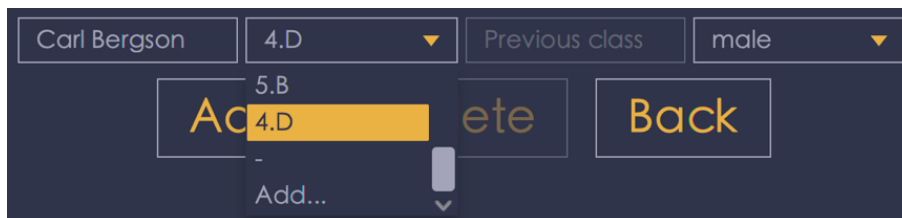


Figure 5: Selecting a class from the class list in a combo box when creating students

When adding a student, the user can select a class from a list of classes used in the student list to prevent redundant class text input. Thus, I aimed to make the UI straightforward.

To prevent unnecessary re-loading of the class list whenever the list of students is updated, I implemented a dictionary (HashMap) storing the number of students for each class. My implementation provides custom functionality that indicates changes in the class list.

```

public boolean classAdded(String className) {
    if (containsKey(className)) {
        put(className, get(className) + 1);
        return false;
    } else {
        put(className, 1);
        return true;
    }
}

public boolean classRemoved(String className) {
    if (get(className) == 1) {
        remove(className);
        return true;
    } else {
        this.put(className, this.get(className)-1);
        return false;
    }
}

public boolean updateClass(String oldVal, String newVal) {
    if (classAdded(newVal) | classRemoved(oldVal)) {
        return true;
    }
    return false;
}

```

Figure 6: Increasing, decreasing and updating the number of students belonging to a specific class

```

public List<String> getClassList() {
    List<String> classes;
    if (this.size() == 0) {
        classes = new ArrayList<>();
    } else {
        classes = new ArrayList<>(keySet());
    }
    classes.add("Add...");
    return classes;
}

```

Figure 7: Exporting the class list to be used by the combo box

3. Encryption

```

public static String encrypt(String str) {
    String shifted = "";
    for(int i = 0; i < str.length(); i++) {
        char original = str.charAt(i);
        char shiftedChar = (char) ((original + _key) % Integer.MAX_VALUE);
        shifted += shiftedChar;
    }
    return shifted;
}

public static String decrypt(String str) {
    String shifted = "";
    for(int i = 0; i < str.length(); i++) {
        char original = str.charAt(i);
        char shiftedChar = (char) ((original - _key) % Integer.MAX_VALUE);
        shifted += shiftedChar;
    }
    return shifted;
}

```

Figure 8: Implementation of the Caesar cypher

The data taken as input is encrypted or decrypted using Caesar cipher. Although breakable, this technique ensures that the data stored (name, previously attended class and gender)

are protected. The data, however, are not delicate and of potential interest, so no attacks are expected.

ymxq	Ye\	bq~zm,Y{~mz{
rqymxq	>:P	Nm~uq,W~{sq~

Figure 9: An unencrypted line and its decrypted version.¹

Clearly, this technique addresses the criterion: “Data are protected from unauthorised access”.

4. Cascade stylesheets (CSS)

The use of CSS, the current application presentation standard, ensured thorough customisability of the UI. Thus, I was able to address the criterion: “UI is straightforward for the user”.

```
.table-view .column-header .arrow
{
    -fx-background-color: #E2B258;
}

.table-view .column-header-background .label{
    -fx-text-fill: #E2B258;
    -fx-font-family: STHeiti;
    -fx-font-size: 20px;
}

.table-view .table-cell {
    -fx-background-color: #414559;
    -fx-border-color: transparent;
    -fx-text-fill: #A3A3B7;
    -fx-font-family: STHeiti;
    -fx-border-width: 1px;
    -fx-min-height: 30px;
}
```

Figure 10: A CSS excerpt used to customise JavaFX TableView

5. Bidirectional updating of elements²

```
// Modify TextField properties
// Update the text in numberOutput TextField as the valueProperty of Slider
changes
numberOutput.textProperty().bindBidirectional(slider.valueProperty(),
NumberFormat.getNumberInstance());
```

Figure 12: The currently selected value is displayed in the top-right corne

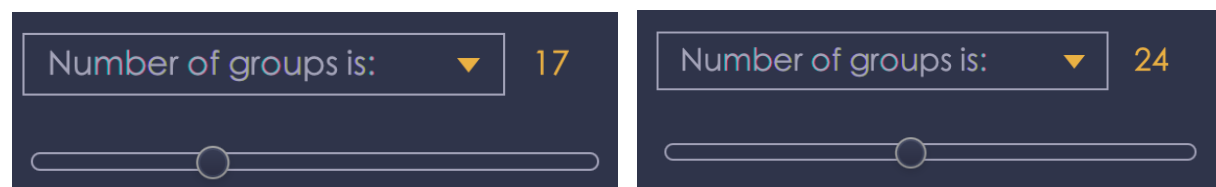


Figure 11: The value displayed in the text field in the upper-right corner updates with the slider value

This technique ensures that the user can view the value currently selected on the slider by connecting its value and the value displayed in a text field and addresses the criterion “UI is straightforward for the user”.

¹ (“Kevin”, 2018)

² (Roland, 2016)

6. Boolean binding

A UI form with four input fields: 'Name' (empty), 'Add...' (dropdown menu), 'Previous class' (empty), and 'female' (dropdown menu). Below the fields are three buttons: 'Add', 'Delete', and 'Back'. The 'Add' button is disabled (greyed out).

Figure 13: Button add is unclickable if the name, previously attended class are not identified

A UI form with four input fields: 'Anne Newman' (filled), 'Add...' (dropdown menu), '5.D' (filled), and 'female' (dropdown menu). Below the fields are three buttons: 'Add', 'Delete', and 'Back'. The 'Add' button is enabled (yellow).

Figure 14: Button add is clickable if the name, previously attended class are inputted

A UI form with four input fields: 'Anne Newman' (filled), 'MYP' (dropdown menu), 'Previous class' (empty), and 'female' (dropdown menu). Below the fields are three buttons: 'Add', 'Delete', and 'Back'. The 'Add' button is enabled (yellow).

Figure 15: Button add is clickable if the name is inputted and the previously attended class is selected

```
BooleanBinding nameValid = Bindings.createBooleanBinding(() -> {  
    return nameInput.textProperty().getValue().matches("\\s*");  
}, nameInput.textProperty());  
  
BooleanBinding addingClass = Bindings.createBooleanBinding(() -> {  
    return existingClasses.getValue().equals("Add...");  
}, existingClasses.valueProperty());  
  
BooleanBinding classValid = Bindings.createBooleanBinding(() -> {  
    return classInput.textProperty().getValue().matches("\\s*");  
}, classInput.textProperty());
```

Figure 16: Boolean Binding objects

```
butAdd.disableProperty().bind(  
    nameValid.or(addingClass.and(classValid))  
);
```

Figure 17: Disabling the add button based on the relevant Boolean Binding objects³

This technique reduces the need for custom alerts (e.g. “Student name is empty!”) because the buttons that should not be accessible (e.g. because input in a field is not valid) are made unclickable. **This provides a clean and simple way to interact with the application**, addressing the criterion: “UI is straightforward for the user”.

³ (“James_D”, 2015)

7. Handling commits of editable JavaFX TableView cells

Name	Previous class	Gender
Adam Gibbon	MYP	male
Anibal Darrington	2.C	male
Barrie Kroger	2.D	female
Barton Faver	MYP	male
Berry Gullo	MYP	male
Berta Torrez	-	female
Bessie Romano	MYP	female
Bethanie Dragoo	MYP	female
Brent Lichty	2.C	male
Britney Furrow	MYP	female

Figure 18: Editing the cell containing the student's name in the table^{4,5}

```
nameColumn.setOnEditCommit(e -> {
    if (!isEmpty(e.getNewValue())) {
        Student s = e.getTableView().getItems().get(e.getTablePosition().getRow());
        s.setName(e.getNewValue());
    } else {
        Student s = e.getTableView().getItems().get(e.getTablePosition().getRow());
        s.setName(e.getOldValue());
        nameColumn.setVisible(false);
        nameColumn.setVisible(true);
    }
});
```

Figure 19: Commit listener for the column containing student names⁶

```
classColumn.setOnEditCommit(e -> {
    if (!isEmpty(e.getNewValue())) {
        dataManager.updateClassCount(e.getOldValue(), e.getNewValue());
        Student s = e.getTableView().getItems().get(e.getTablePosition().getRow());
        s.setPreviousClass(e.getNewValue());
    } else {
        Student s = e.getTableView().getItems().get(e.getTablePosition().getRow());
        s.setPreviousClass(e.getOldValue());
        classColumn.setVisible(false);
        classColumn.setVisible(true);
    }
});
```

Figure 20: Commit listener for the column containing students' previously attended classes

```
genderColumn.setOnEditCommit((TableColumn.CellEditEvent<Student, Gender> e) -> {
    Student s = e.getTableView().getItems().get(e.getTablePosition().getRow());
    s.setGender(e.getNewValue().getText());
});
```

Figure 21: Commit listener for the column containing students' genders

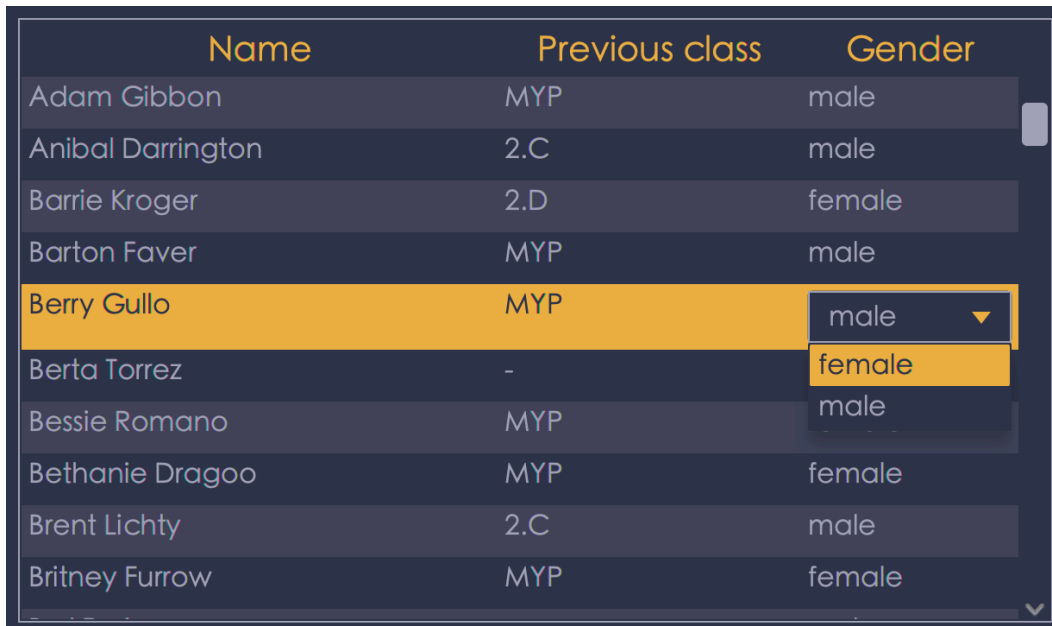
⁴ ("thenewboston, 2015a)

⁵ (Wright, 2017a)

⁶ (code.makery, 2014)

After the user commits changes, the program validates the new input. If valid, the relevant value is updated. Otherwise, the old value is kept, and the table is refreshed. **This technique ensures that the value displayed matches the data stored in the underlying data structures. Since the values are accessed directly based on their position in the table, element-wise iteration of the table's elements is circumvented, thus saving computation time.** Due to the clear interface provided by TableView, this technique also addresses the criterion: "User can manage students".

8. Nested combo box in table cell



Name	Previous class	Gender
Adam Gibbon	MYP	male
Anibal Darrington	2.C	male
Barrie Kroger	2.D	female
Barton Faver	MYP	male
Berry Gullo	MYP	male
Berta Torrez	-	female
Bessie Romano	MYP	male
Bethanie Dragoo	MYP	female
Brent Lichty	2.C	male
Britney Furrow	MYP	female

Figure 22: Editing the cell containing gender using a combo box

```
ObservableList<Gender> genderList = FXCollections.observableArrayList(Gender.values());
genderColumn.setCellFactory(ComboBoxTableCell.forTableColumn(genderList));
```

Figure 23: Loading a list of genders to choose from and displaying it

All available genders are set as combo box values. **This technique prevents unnecessary typing and saves the user's time while avoiding the need for input checking.**

9. Creating a custom class: Group

This technique allows the management of all relevant data such as *group number*, *note*, *captain* and *member list*. The respective access methods result in maintainable code and find their use especially when exporting the groups to a file.

```
for (Group group : groups) {
    output.println("Group " + group.getGroupNum());
    output.println("\t" + group.getLeader());
    output.println("\t" + group.getNote());
    output.println();

    output.println("\tStudents:");
    for (String name: group.getMembers()) {
        output.println("\t" + name);
    }
    output.println();
}
```

Figure 24: The core loop used in storing generated groups to a file, using access methods of class Group


```

public class Group {
    private ArrayList<String> members;
    private String leader;
    private TextField note;
    private int groupNum;
    private boolean leaderSet;

    public Group(int groupNum) {
        this.groupNum = groupNum;
        members = new ArrayList<>();
        leader = "Leader not set";
    }

    public void add(String name) {
        members.add(name);
    }

    public int getGroupNum() {
        return groupNum;
    }

    public ArrayList<String> getMembers() {
        return members;
    }

    public void setLeader(String leader) {
        this.leader = leader;
        leaderSet = true;
    }

    public String getLeader() {
        if (!leaderSet) return leader;
        return "Leader: " + leader;
    }

    public void setNote(TextField note) {
        this.note = note;
    }

    public String getNote() {
        if (note.getText().isEmpty()) return "No note added";
        return "Note: " + note.getText();
    }
}

```

Figure 25: Implementation of the group class

10. UTF-8 encoding

Dragoş Neagu	-	male
Evañ Bertucci	2.D	male
Faust Chû-ang	MYP	male

Figure 26: Special characters shown in the table containing student details

```

public void loadData() {
    try {
        BufferedReader input = new BufferedReader(new InputStreamReader(
            new FileInputStream("src/resources/students.txt"), "UTF-8"));
        input.mark(4);
        // Handle BOM
        if ('\ufeff' != input.read()) input.reset();
        String line, name, previousClass, gender;
        while((line = input.readLine()) != null){
            String[] temp = line.split("\t");
            gender = Security.decrypt(temp[0]);
            previousClass = Security.decrypt(temp[1]);
            name = Security.decrypt(temp[2]);
            try {
                addStudent(name, previousClass, gender);
            } catch (StudentExistsException e) {
            }
        }
        Main.getCreateGroupsController().updateSliderRange();
        input.close();
        dataLoaded = true;
    } catch (IOException e){
        e.printStackTrace();
    }
}

```

Figure 27: Creating a buffer and loading data from a text file

```

public void storeData() {
    if (!dataLoaded) return;
    try{
        PrintWriter output = new PrintWriter(new OutputStreamWriter(
            new FileOutputStream("src/resources/students.txt"), "UTF-8"), false);
        students.forEach(student -> {
            String line = Security.encrypt(student.getGender()) + "\t" +
                Security.encrypt(student.getPreviousClass()) + "\t" +
                Security.encrypt(student.getName());
            output.println(line);
        });
        output.close();
    } catch (Exception e){
        e.printStackTrace();
    }
}

```

Figure 28: Creating an output stream and storing data to a text file

This technique ensures that the students' names are loaded and stored correctly thanks to using an encoding that supports numerous special characters and is widely supported. This also applies to the files that can be generated on user's request after groups were created.

11. Enum type

```
public enum Gender {  
    FEMALE("female"), MALE("male");  
  
    String text;  
  
    Gender(String text) {  
        this.text = text;  
    }  
  
    public String getText() {  
        return text;  
    }  
  
    @Override  
    public String toString() {  
        return this.text;  
    }  
}
```

Figure 29: Implementing an Enum Type to store a list of all available genders as constants

This technique ensures that the product is extensible, because data has to be modified in one place and the change propagates automatically to e.g. the combo boxes in TableView. As an ethical consideration, future developers could see added an option for when a student does not want to disclose their gender.

12. GitHub

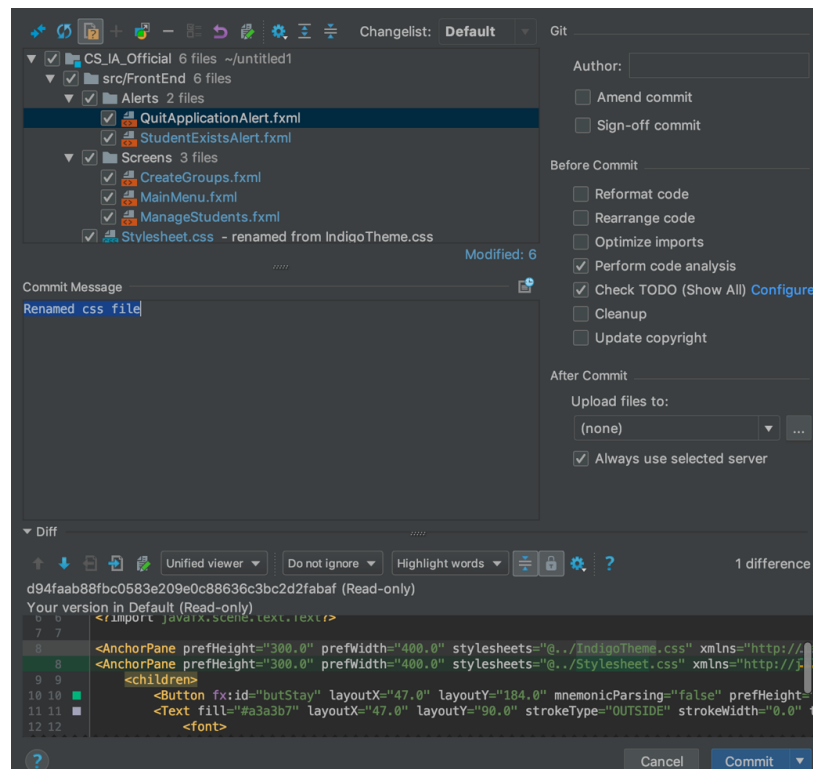


Figure 30: Committing a change to the GitHub repository⁷

⁷ (Michalski, 2016)

This technique addresses the criterion: “Future users can download the application from an online repository” by making it available on GitHub, a major online repository for individual projects. It also ensures that the project is safely stored.

Word count: 669 words

Sources relevant to this file

code.makery. (2014). *JavaFX 8 Event Handling Examples*. [online] Available at:

<https://code.makery.ch/blog/javafx-8-event-handling-examples/> [Accessed 3 Dec. 2018]

“James_D”. (2015). *How to disable a button until some text field are filled?* [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/32526620/how-to-disable-a-button-until-some-text-field-are-filled> [Accessed 17 Dec. 2018]

“Kevin”. (2018). *Keep Spaces, Punctuation, and char Cases in Java Caesar Cipher*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/33022420/keep-spaces-punctuation-and-char-cases-in-java-caesar-cipher> [Accessed 31 Dec. 2018]

Michalski, D. (2016). *IntelliJ IDEA GitHub Integration (export project to GitHub)*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=NVTRJDk8de0&t=141s> [Accessed 18 Dec. 2018]

Roland. (2015). *JavaFx Tutorial For Beginners 19 - JavaFX Bidirectional Binding and using Slider*. Available at: <https://stackoverflow.com/questions/34478174/disable-column-rearrangement-of-tableview-javafx> [Accessed 5 Dec. 2018]

“thenewboston”. (2015a). *JavaFX Java GUI Tutorial - 19 - Editable Tables*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=q1LEN2assfU&t=7s> [Accessed 7 Dec. 2018]

Wright, J. (2017a). *Edit TableView Object using JavaFX*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=z4LVolG6ch0> [Accessed 7 Dec. 2018]

Other sources

“Jhonny007”. (2016). *What is the relationship between Java FX Application, Scene and Parent?* [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/39873287/what-is-the-relationship-between-java-fx-application-scene-and-parent> [Accessed 3 Dec. 2018]

“jns”. (2016). *ScrollPane not scrolling with VBox*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/36579609/scrollpane-not-scrolling-with-vbox> [Accessed 14 Dec. 2018]

“ka4eli”. (2016). *How to convert hash map keys into list?* [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/31820049/how-to-convert-hash-map-keys-into-list> [Accessed 10 Dec. 2018]

- Oracle. (n.d.). *Titled Pane and Accordion*. [online] Available at: https://docs.oracle.com/javafx/2/ui_controls/accordion-titledpane.htm [Accessed 4 Dec. 2018]
- O7planning. (n.d.). *Disable column rearrangement of tableview, javafx*. [online] Available at: <https://o7planning.org/en/11079/javafx-tableview-tutorial> [Accessed 18 Dec. 2018]
- ProgrammingKnowledge. (2015). *Adding and populating a TableView object using SceneBuilder*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=x3jhTJTDlwQ> [Accessed 7 Dec. 2018]
- “thenewboston”. (2015b). *JavaFX Java GUI Tutorial - 4 - Switching Scenes*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=7LxWQID0zyE> [Accessed 3 Dec. 2018]
- “thenewboston”. (2015c). *JavaFX Java GUI Tutorial - 7 - Closing the Program Properly*. [online] YouTube. Available at: <https://www.youtube.com/watch?v=ZuHcl5MmRck> [Accessed 5 Dec. 2018]
- Wright, J. (2017b). *Adding and populating a TableView object using SceneBuilder*. [online] YouTube. Available at: https://www.youtube.com/watch?v=uh5R7D_vFto [Accessed 7 Dec. 2018]