# Criterion C

## Advanced techniques use in an application

- Use of fxml files
- Recursion
- Hashmap
- ArrayLists
- Self-made classes
- Aggregation of classes
- Reading from .csv file
- Self-made complex methods
- The switch statement
- Nested “for loops”
- Reading from and writing to .java files
- Try/catch block
- Drag & drop technology

## Importing Java files to the application

A user chooses which Java files or directories are being imported to the application. The process of creating appropriate UML diagrams begins with searching for Java files among the chosen ones.

### Searching for Java files in the inputted directory

```
public void enterDirectory(File file) throws FileNotFoundException {
    if (file.isDirectory()) {
        File[] content = file.listFiles();
        assert content != null;
        for (File value : content) {
            enterDirectory(value);
        }
    } else if (file.toString().contains(".java")) {
        createClass(file);
    }
}
```

*Figure 1. enterDirectory() method from the ImportLayoutController class.*

The enterDirectory() method uses recursion to search for the Java files in the folder primarily inputted by a user. The method is given a parameter – a path to the location of the file inputted.

If it is a directory, then the method creates an array of the files it contains. Then the method goes through them, calling itself, giving the cell as a parameter and starts the algorithm from the beginning. If the parameter is not a directory, the method checks if the file's name contains ".java", which is equivalent to it being a Java file. If it does, the method calls createClass() function.

## Obtaining a name, attributes and methods of the class

```
public void createClass(File file) throws FileNotFoundException {
    Scanner scan = new Scanner(file);
    int depth = 0;
    String name = "";
    List<String> attributes = new ArrayList<>();
    List<String> methods = new ArrayList<>();
    while (scan.hasNext()) {
        String line = scan.nextLine();
        if (depth == 0 && line.contains("class") && !line.contains(";")) {
            String[] parts = line.split( regex: " ");
            name = parts[parts.length - 2];
            depth++;
```

*Figure 3. The beginning of the createClass() method from the ImportLayoutController class and the conditions necessary for the algorithm to know if a given line from the source code is a name of a class.*

```
} else if (depth == 1 && line.contains("{") && line.contains("Timeline")) {
    depth++;
} else if (depth == 1 && line.contains("{")) {
    StringBuilder newText;
    boolean isStatic = false;
    String[] parts = line.split( regex: "\\)");
    String[] nameAndParameters = parts[0].split( regex: "\\(");
    String[] names = nameAndParameters[0].split( regex: " ");
```

*Figure 2. The other conditions needed to be met in order for the createClass() method to know if a given line from the source code is respectively: a beginning of a Timeline or a beginning of a method.*

In order to find all necessary data from the source code, I decided to count a level of brackets' nesting, storing its depth and find appropriate lines in a file basing on it. The expected value for the depth, for which the name could be found, equalled 0 and for the attributes and the methods – 1 (while searching for methods I needed to look out for other statements, with similar syntax, for instance Timeline. Figure 3 presents the distinction I have made in order to prevent the method from confusing them). For every opening bracket the depth value was incremented, for

closing – decremented. The orange rectangle marks all the necessary conditions that needed to be met, to be sure that the line contains a name of the class, and the method of obtaining its name. I use String's method "split()" in order to get every each word of the line separated. The name of the class happens to be the second word from the end in a default format of the Intellij (keyboard shortcut: ctrl+alt+l).

```
else if (depth == 1 && line.contains(";")) {                                    ⚠ 17 ^
  String text = line;
  StringBuilder newText;
  if (line.contains("=")) {
      String[] parts = line.split( regex: "=");
      text = parts[0];
  }
  String[] fragments = text.split( regex: " ");
  if (fragments[fragments.length - 1].contains(";"))
      newText = new StringBuilder(fragments[fragments.length - 1].replace( target: ";",  replacement: ""));
  else newText = new StringBuilder(fragments[fragments.length - 1]);
  List<String> fragmentsList = new ArrayList<>();
  for (int i = 0; i < fragments.length - 1; i++) {
      if (fragments[i].equals("static")) newText.append(";static");
      else if (!fragments[i].equals("")) fragmentsList.add(fragments[i]);
  }
```

*Figure 4. createClass() – obtaining attribute text.*

If the condition, stated at the top of the figure 4, was met then the algorithm knew that the line contains the class's attribute. The next "if" statement checks if the attribute has a value assigned to it. Again, I used String's method split() to separate the line into two parts. Only the first one, before the '=' sign, is needed. Next, the method creates an array containing all the words, from the first part, separated. Then, if necessary, it deletes unimportant signs, for instance ';' and starts analysing the content of the array. If the attribute was static in the code, the method adds

";static" to the text in order to later know if the attribute text should be underscored. Then all parts, except for blank spaces, are placed in the fragmentsList.

```
String accessModifier = "";
if (fragmentsList.size() == 1) newText.insert( offset: 0,  str: "- " + fragmentsList.get(0) + " ");
else {
    for (int i = 0; i < fragmentsList.size(); i++) {
        String temp = fragmentsList.get(i);
        if (temp.equals("private") || temp.equals("public") || temp.equals("protected")) {
            switch (temp) {
                case "private":
                    accessModifier = "-";
                    break;
                case "public":
                    accessModifier = "+";
                    break;
                case "protected":
                    accessModifier = "#";
                    break;
            }
            fragmentsList.remove(i);
            break;
        }
    }
    newText.insert( offset: 0,  str: accessModifier + " " + fragmentsList.get(0) + " ");
}
attributes.add(newText.toString());
```

*Figure 5. The rest of the action needed to obtain an attribute in the createClass() method.*

The figure 5 presents how the appropriate access modifier, that will be written before the attribute type, is being searched. First "if" statement checks if there was an access modifier. If not – the attribute is private, thus the '-' sign. If there was, then every fragmentList is checked if it contains any of the keywords. If it does, then the method establishes appropriate access

modifier and leaves the loop. At the end, the text of a new attribute is made and added to the list of attributes of the class.

An algorithm creating methods works similarly.

```
    sort(attributes);
    sort(methods);
    rec = new sample.RectangleClass(name, attributes, methods);
    imported=true;
    addClass();
}

public void sort(List<String> arr) {
    for (int i = 0; i < arr.size(); i++) {
        String smallest = arr.get(i);
        int index = i;
        for (int j = i; j < arr.size(); j++) {
            if (smallest.compareTo(arr.get(j)) < 0) {
                smallest = arr.get(j);
                index = j;
            }
        }
        String temp = arr.get(i);
        arr.set(i, smallest);
        arr.set(index, temp);
    }
}
```

*Figure 6. The end of the createClass() method and the sort() method.*

At the end, the createClass() method sorts all attributes and methods (using self-written selection sort), basing on the access modifiers, with the hierarchy: private, public, protected,

and overwrites an instance of RectangleClass, later used in the addClass(), responsible for drawing the diagram.

## UML diagrams drawings structure

```
class ClassDrawings{
    int theLongestPhrase=5;
    ClassRows name;
    List<Rectangle> rectangles;
    List<ImageView> images;
    List<ClassRows> attributeRows;
    List<ClassRows> methodRows;
    Rectangle nameSpace;
    Rectangle attributeSpace;
    Rectangle methodSpace;
    Rectangle changingParametersAttribute;
    Rectangle changingParametersMethod;
    ImageView deleteClass;
    String className;
    AnchorPane root;
```

*Figure 7. Class responsible for each drawing of UML diagram in the import and the export window.*

```
public class ClassRows {
    AnchorPane row;
    Text text;
    TextField textField;
    ImageView save;
    ImageView edit;
    ImageView delete;
```

*Figure 8. Class responsible for holding elements of each row.*

In order to have my classes neatly programmed I changed the concept how they should be structured. I created a new “ClassRows” class containing every element of the class diagram’s row (ImageViews serve the function of the buttons present on the UML diagram drawings). Instances of the ClassDrawings class possess these rows with the division on: the name, attribute and method ones.

# Exporting the classes

The process of generating the code out of UML diagrams uses the structure of drawn diagrams. It begins with calling constructClasses() method.

## Creating a code to the preview and to the file exported

```
if (selectedOne != null) {
    for (int i = 0; i < classSpaces.size(); i++) {
        if (classSpaces.get(i) == selectedOne) {
            index = i;
            break;
        }
    }
}
if (selectedOne == null) {
    for (int i = 0; i < nr; i++) {
        if (i > 0) text.append("\n\n-----------------------------------------------------\n\n");
        text.append(createContent(i));
    }
} else {
    text.append(createContent(index));
}
```

*Figure 9. Part of constructClasses() method in the ExportController class.*

There are two parameters given to the constructClasses() method: no of classes to construct and, if there was only one chosen, which class. In the second case the method knows, which class it should construct, but has not got the access to its diagram. The ExportController class contains lists of all drawn diagrams (as AnchorPanes and ClassDrawings, which indexes are stored parallelly). In the beginning, the method searches for the correct AnchorPane and finds its index in the list. Next, it calls a createContent() method, which creates the code. As a parameter, this method obtains an index of the correct instance of the ClassDrawings class in the list.

If a user chose the option to draw all visible classes, then the code is generated for all of them. In the preview they are separated with dashed lines. Then, the constructClasses() method visualises the preview in the distinct modal window.

## Creating content

```
public StringBuilder createContent(int i) {
    StringBuilder text = new StringBuilder();
    List<String> importsToBeWritten = new ArrayList<>();
    text.append("package sample/com.company\n\n");

    StringBuilder attributes = new StringBuilder();
    StringBuilder methods = new StringBuilder();

    List<Text> texts = new ArrayList<>();

    for (int j = 0; j < classDrawings.get(i).attributeRows.size(); j++) {
        texts.add(classDrawings.get(i).attributeRows.get(j).text);
    }
```

*Figure 10. the beginning of the createContent() method in the ExportController class.*

The createContent() method generates a code for a preview and for a file, which is being exported. It creates all necessary parts, such as a package of a user's choice, some of the imports, and the body of the class. It returns the StringBuilder with the code. The process of creating the code for attributes and methods is similar, so I will describe only the first case.

In the beginning, the method collects all texts of attributes inputted into the UML diagram.

```
for (Text item : texts) {
    String attribute = item.getText();
    if (!attribute.isEmpty()) {
        String textToAdd = "";
        attribute = attribute.trim();
        String accessModifier = String.valueOf(attribute.charAt(0));
        if (accessModifier.equals("+")) accessModifier = "\\+";
        attribute = attribute.replaceFirst(accessModifier,  replacement: "");
        switch (accessModifier) {
            case "\\+":
                textToAdd += "public";
                break;
            case "-":
                textToAdd += "private";
                break;
            case "/":
                textToAdd += "protected";
                break;
        }
```

*Figure 11. Part of the createContent() method adding appropriate access modifier to the attribute text.*

Then, the method takes each one of the texts, trims it, and establishes, which access modifier should be added to the particular attribute, exchanging the sign in the text analysed with a blank space.

```
    attribute = attribute.trim();
    if (attribute.contains(":")) {
        String[] parts = attribute.split( regex: ":");
        parts[1] = parts[1].trim();
        if (imports.containsKey(parts[1]) && checkIfImportOccurs(imports.get(parts[1]), importsToBeWritten))
            importsToBeWritten.add(imports.get(parts[1]));
        textToAdd += parts[1] + " " + parts[0] + ";";
    } else {
        String[] parts = attribute.split( regex: " ");
        if (imports.containsKey(parts[0]) && checkIfImportOccurs(imports.get(parts[0]), importsToBeWritten))
            importsToBeWritten.add(imports.get(parts[0]));
        textToAdd += " " + attribute + ";";
    }
    attributes.append("    ").append(textToAdd).append("\n");
}
```

*Figure 12. Part of the createContent() method that creates appropriate parts of attributes depending of the type.*

There are two formats of creating UML diagrams. One is “’access modifier’ ’type of variable’ ’name’”, the other is “’access modifier’ ’name’ ’:’ ’type of variable’”. This part of the method creates appropriate code, basing on the style chosen by a user and adds it to the text that will be

shown in a preview, at the same time checking if for the given type of an attribute by a user it is necessary to include a correct import.

The method at the end appends a package, imports, and the body of the class to the returned String Builder.

## Changing selected ClassDrawing

```
public AnchorPane changeSelected(AnchorPane selectedOne, AnchorPane classSpace) {
    Color paint = new Color( red: 0.0177,  green: 0.2161,  blue: 0.59,  opacity: 1.0);
    DropShadow dropShadow = new DropShadow( radius: 10, paint);
    dropShadow.setSpread(0.7);
    Node rec = classSpace.getChildren().get(2);
    if (selectedOne == null) {
        rec.setEffect(dropShadow);
        selectedOne = classSpace;
        selectedOne.toFront();
    }
    if (selectedOne != classSpace) {
        Node rec0 = selectedOne.getChildren().get(2);
        rec.setEffect(dropShadow);
        rec0.setEffect(null);
        selectedOne = classSpace;
        selectedOne.toFront();
    }
    return selectedOne;
}
```

*Figure 13. The changeSelected() method in the Controller class.*

In order for the application to know, which diagram it should move, when the said action is expected by a user, I created a variable that at all time keeps track, which drawing is selected, used at the moment. It is then marked with a blue shadow. The “selectedOne” variable is also useful when a user wants to preview only one, recently modified diagram. Every time when a user modifies something in a diagram or moves it, the application automatically changes the value of “selectedOne” to that drawing. The method also removes the shadow effect from the

drawing, which was the previously selected one and moves the new “selectedOne” diagram to the front.

Word count: 1196