Abstract

Object-oriented programming is a programming paradigm based on the concept of objects, which can contain data and code: data in the form of fields, and code in the form of procedures. In OOP, computer programs are designed by making them out of objects that interact with one another. In this project, three sorting algorithms including Bubble Sort, Insertion Sort and Quick Sort are implemented and visualized using OOP techniques learned through the course. By following the utilization of OOP principles, sorting algorithms are encapsulated within classes, combined with polymorphism and inheritance. The visualization shows how each sorting algorithm actually affects on the pure initial array, enhances comprehension and analysis of algorithmic behavior.

CONTENTS

Chapter 1

Overview

1.1 Members Assignment

Here is the contribution of each member in our project:

- Ngo Duy Dat - 20225480: Group Leader, responsible for the code of both the controller and the view of the application, merging, code refactoring and cleaning.
- Pham Dang Tan Dung - 20225569: handling user input, randomize button, QuickSort, Bar, speed adjustment slider; fixing errors, giving the idea to design the UI; slide and report preparation.

Our project is taken inspiration from [1]

1.2 Project Requirement

Overview: Array is the most basic structure of computer science. Most operations as well as other data structures are built and performed on array. In this project, we make an application in order to explain three sorting algorithms on array: bubble sort, quicksort and insertion sort.

- Design: + On the main menu: title of the application, 3 types of sort algorithms for user to choose, help menu, quit

• User must select a sort type in order to start the demonstration

• Help menu show the basic usage and aim of the program

• Quit exits the program. Remember to ask for confirmation + In the demonstration

• A button for creating the array: User can choose to randomly create an array or input an array for the program

• A button for starting the algorithm with the created array. Remember to show clearly each step of the sorting

• A back button for user to return to main menu at any time

1.3 Usecase Diagram

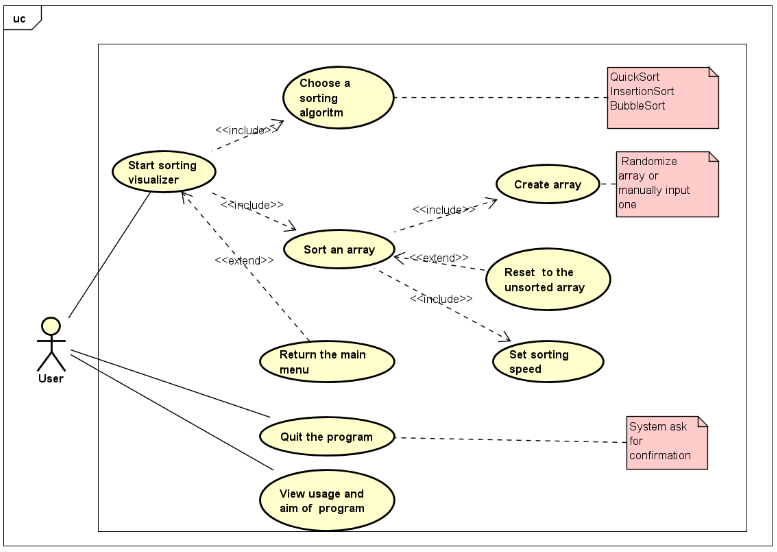


Figure 1.1: Use case diagram

The use case diagram shows how user can interact with the sorting visualizer application, detailing all possible actions and interactions within the system. This visualizer application allows users to choose, visualize, and manipulate different sorting algorithms. There are use cases and components and each serves different purposes.

1.3.1 Actors

- **User**: The individual who interacts with the sorting visualizer application.

1.3.2 Use Cases

- **Start sorting visualizer**: This is the primary and fundamental action when the user interacts with the application. It includes several other actions necessary to perform sorting visualization.
- **Choose a sorting algorithm**: The user selects a sorting algorithm to visualize. The available algorithms include: Quick Sort, Insertion Sort and Bubble Sort.
- **Create an array**: The user creates an integer array to be sorted. The array can be either randomized or manually inputted by the user.
- **Sort an array**: The sorting process will be displayed on the application.

- **Reset to the unsorted array**: This allows the user to revert the array to its original unsorted state.
- **Set sorting speed**: User can adjust the speed at which the sorting algorithm is visualized and the sorting speed will change correspondingly.
- **Return to the main menu**: The user can navigate back to the main menu from any point in the application. This is an extension of the "Start sorting visualizer" use case.
- **Quit the program**: This use case allows the user to exit the application. Before quitting, the system will ask for confirmation.
- **View usage and aim of the program**: The user can view information about how to use the program and the objectives it aims to achieve.

Chapter 2

Design Analysis

2.1 General Class Diagram

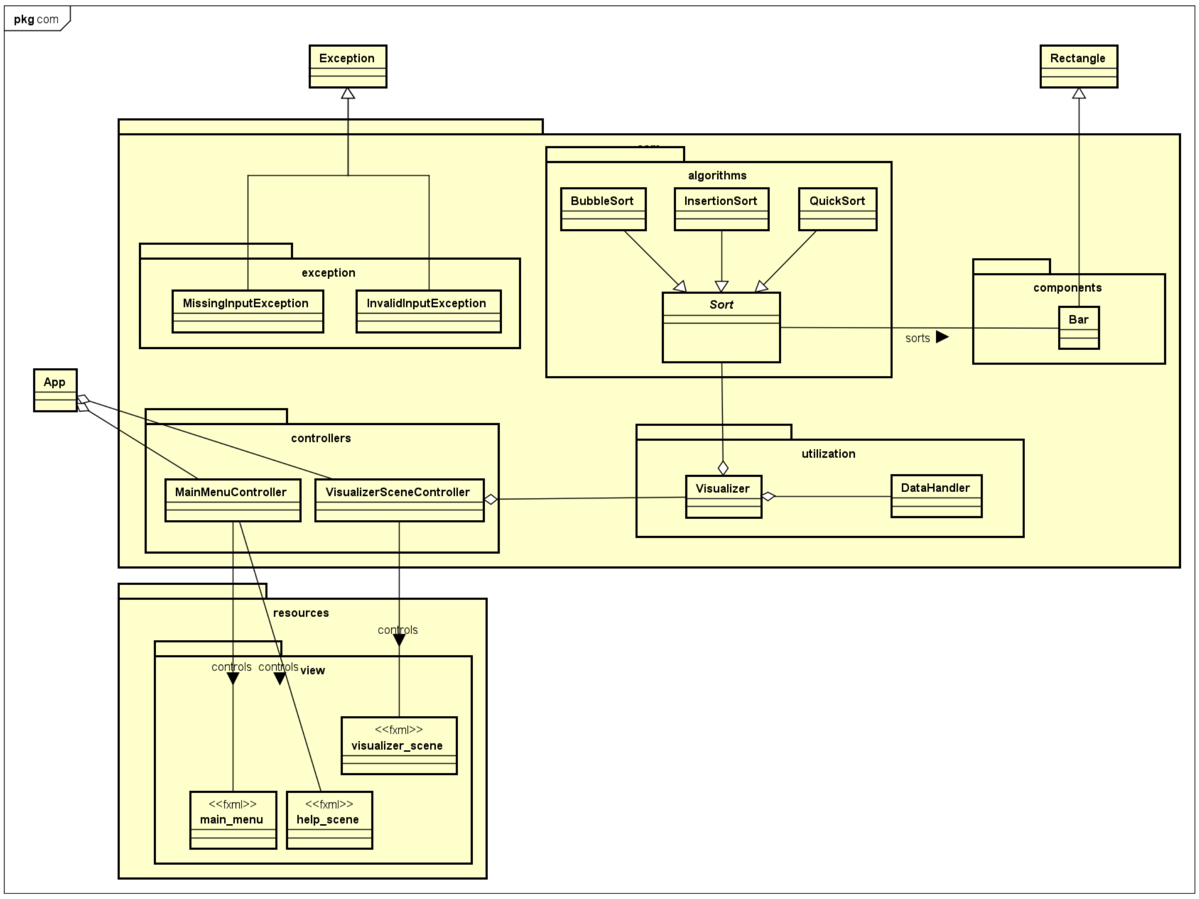


Figure 2.1: General class diagram

The main components in a sorting visualizer here is a Bar. A Bar is an UI representation of an element in an array. Here, class Bar is extended from the Rectangle class of JavaFX.

The package main.com.algorithms is where the sorting algorithms are implemented. We have 3 classes InsertionSort, BubbleSort and QuickSort that all inherited from an abstract class Sort. Those class directly sorts an array of Bars, not pure integer array.

Then, in the package main.com.utilization, we have 2 classes DataHandler and Visualizer. A DataHandler is to generate an integer array or throw exceptions if there is invalid input when the user input or randomize an array. New Exceptions are defined in packages main.com.exceptions.

A Visualizer possesses a DataHandler to get integer array to create Bars and a sorting algorithm to sort those Bars. Package main.com.controllers contains controllers for the FXML files that are responsible for the UI. There are 2 controllers corresponding to 2 scenes: MainMenu and Visualizer. Visualizer is the scene where the main function of our application is performed. In the class VisualizerSceneController, we use appropriate methods of the class Visualizer to control the action of each components in the UI. Here the diagram explaining the flow of our application.

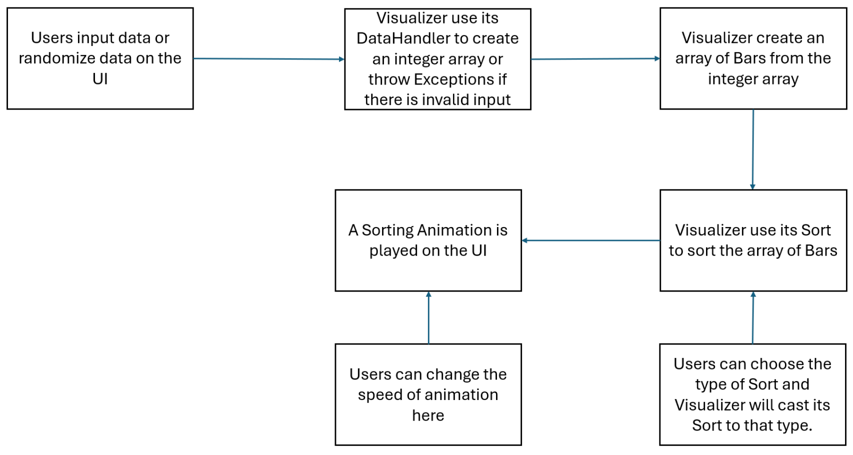

Users input data or randomize data on the UI
Visualizer use its DataHandler to create an integer array or throw Exceptions if there is invalid input
Visualizer create an array of Bars from the integer array
Visualizer use its Sort to sort the array of Bars
A Sorting Animation is played on the UI
Users can change the speed of animation here
Users can choose the type of Sort and Visualizer will cast its Sort to that type.

Figure 2.2: Flow of sorting visualizer

2.2 Detailed Class Diagrams

2.2.1 Bar Class

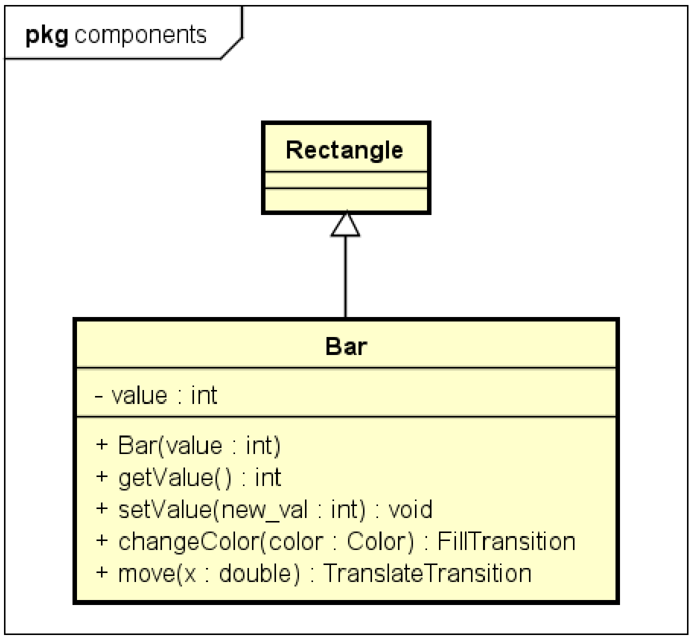

pkg components
Rectangle
Bar
- value : int
+ Bar(value : int)
+ getValue() : int
+ setValue(new_val : int) : void
+ changeColor(color : Color) : FillTransition
+ move(x : double) : TranslateTransition

Figure 2.3: Bar class diagram

As mentioned before, a Bar is also a Rectangle of JavaFX library. A Bar has only 1 attribute value that is corresponding to the value of an integer. It has 2 methods changeColor() and move() to perform necessary operation in a sorting visualizer. The changeColor() method is to highlight a bar when it is selected or sorted and the move() method is to move the bars to do swap between bars. Both methods return a transition of JavaFX.

2.2.2 Package main.com.algorithms

Package algorithms is where we implement all class of sorting algorithms that directly sort an array of Bars. Here is the diagram of this package.

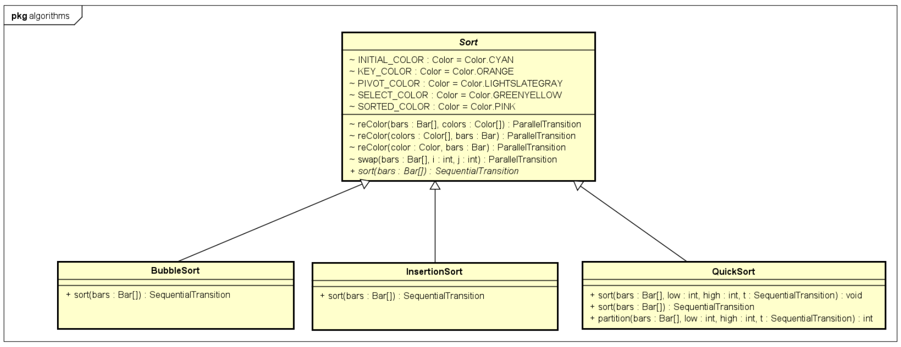
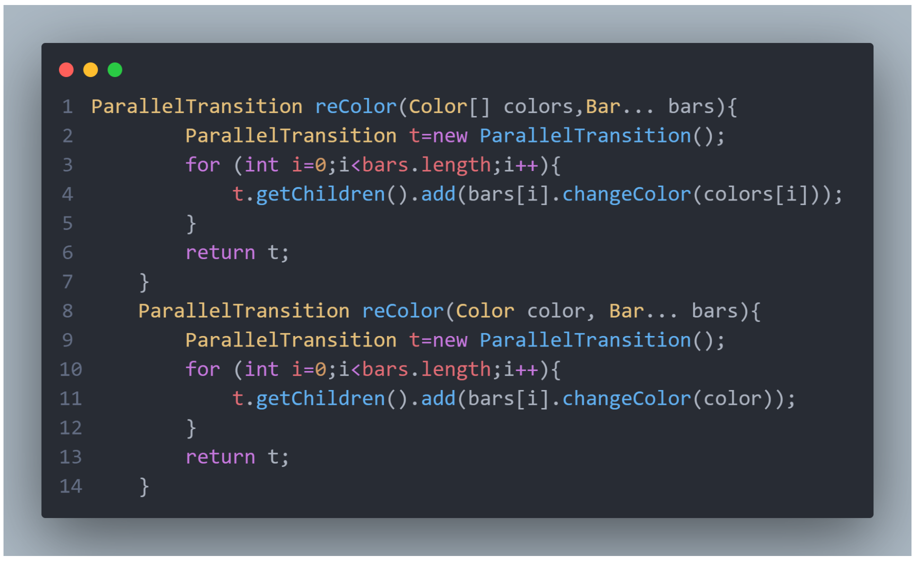

pkg algorithms
Sort
~ INITIAL_COLOR : Color = Color.CYAN
~ KEY_COLOR : Color = Color.ORANGE
~ PIVOT_COLOR : Color = Color.LIGHTSLATEGRAY
~ SELECT_COLOR : Color = Color.GREENYELLOW
~ SORTED_COLOR : Color = Color.PINK
~ reColor(bars : Bar[], colors : Color[]) : ParallelTransition
~ reColor(colors : Color[], bars : Bar) : ParallelTransition
~ reColor(color : Color, bars : Bar) : ParallelTransition
~ swap(bars : Bar[], i : int, j : int) : ParallelTransition
+ sort(bars : Bar[]) : SequentialTransition
BubbleSort
+ sort(bars : Bar[]) : SequentialTransition
InsertionSort
+ sort(bars : Bar[]) : SequentialTransition
QuickSort
+ sort(bars : Bar[], low : int, high : int, t : SequentialTransition) : void
+ sort(bars : Bar[]) : SequentialTransition
+ partition(bars : Bar[], low : int, high : int, t : SequentialTransition) : int

Figure 2.4: Class diagram of package main.com.algorithms

Sort abstract class

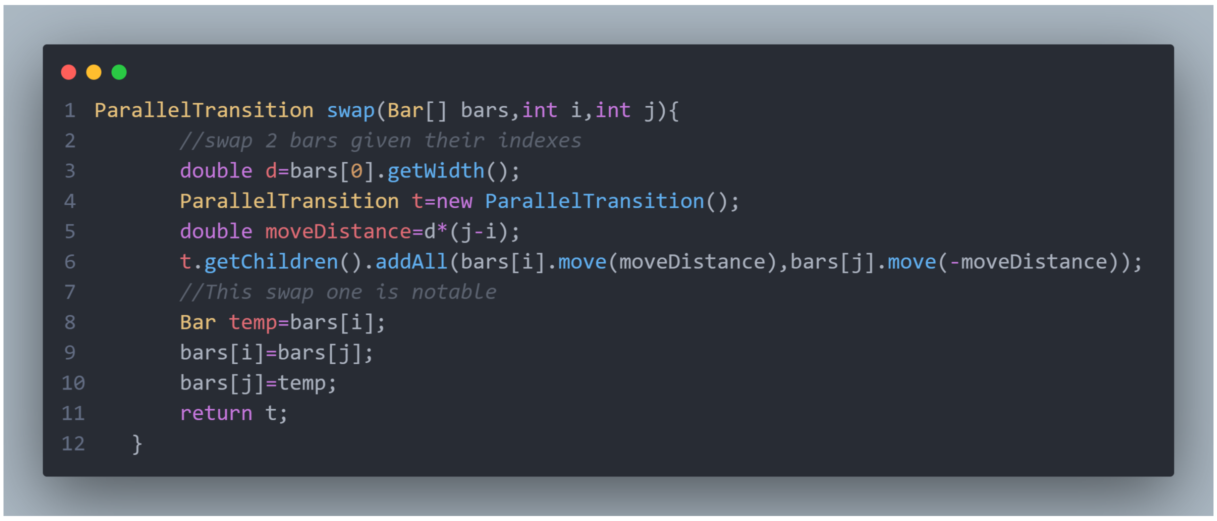
Sort is an abstract class that has some methods that shared by all types of sort. Its attributes are related to the way we color bars. The reColor() method is to change color of a bunch of bars. This method takes advantage of the changeColor() method from class Bars. We also use technique of overloading here to create variations of reColor() with different functionality: one for recoloring all given bars to a same color and one for recoloring each bar to a given color in the color array.



```
ParallelTransition reColor(Color[] colors,Bar... bars){
        ParallelTransition t=new ParallelTransition();
        for (int i=0;i<bars.length;i++){
            t.getChildren().add(bars[i].changeColor(colors[i]));
        }
        return t;
    }
    ParallelTransition reColor(Color color, Bar... bars){
        ParallelTransition t=new ParallelTransition();
        for (int i=0;i<bars.length;i++){
            t.getChildren().add(bars[i].changeColor(color));
        }
        return t;
    }
```

Figure 2.5: reColor() method to recolor bars

The swap() method is to swap between 2 bars which is essential in a sorting algorithm. This method takes advantage of the move() method of class Bar.



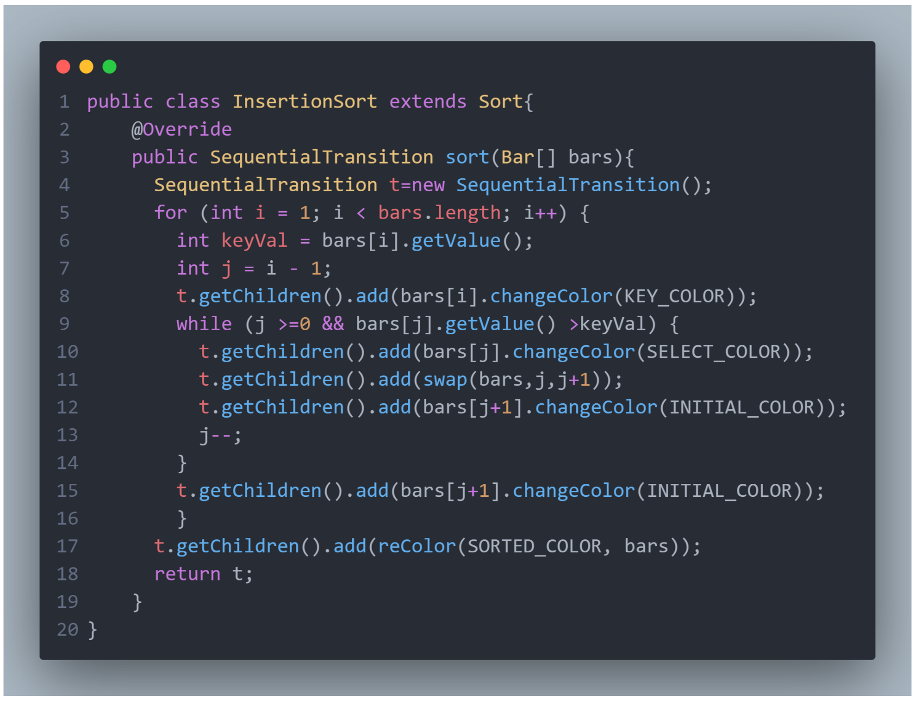
```java
ParallelTransition swap(Bar[] bars,int i,int j){
        //swap 2 bars given their indexes
        double d=bars[0].getWidth();
        ParallelTransition t=new ParallelTransition();
        double moveDistance=d*(j-i);
        t.getChildren().addAll(bars[i].move(moveDistance),bars[j].move(-moveDistance));
        //This swap one is notable
        Bar temp=bars[i];
        bars[i]=bars[j];
        bars[j]=temp;
        return t;
    }
```

Figure 2.6: swap() method to swap bars

The final method sort() has an argument being an array of Bar and has type of SequentialTransition. This method is abstract and will be overrided by classes of specific sorting type.

Inherited class from Sort

3 classes InsertionSort, BubbleSort and QuickSort override the sort() method of their parent class to return a SequentialTransition of the sorting process corresponding to the algorithm. For example, this is how InsertionSort class takes advantage of reColor(), swap() to override the method sort():



```java
public class InsertionSort extends Sort{
    @Override
    public SequentialTransition sort(Bar[] bars){
      SequentialTransition t=new SequentialTransition();
      for (int i = 1; i < bars.length; i++) {
        int keyVal = bars[i].getValue();
        int j = i - 1;
        t.getChildren().add(bars[i].changeColor(KEY_COLOR));
        while (j >=0 && bars[j].getValue() >keyVal) {
          t.getChildren().add(bars[j].changeColor(SELECT_COLOR));
          t.getChildren().add(swap(bars,j,j+1));
          t.getChildren().add(bars[j+1].changeColor(INITIAL_COLOR));
          j--;
        }
        t.getChildren().add(bars[j+1].changeColor(INITIAL_COLOR));
        }
      t.getChildren().add(reColor(SORTED_COLOR, bars));
      return t;
    }
}
```

Figure 2.7: Overrided sort() method of class InsertionSort

2.2.3 DataHandler Class

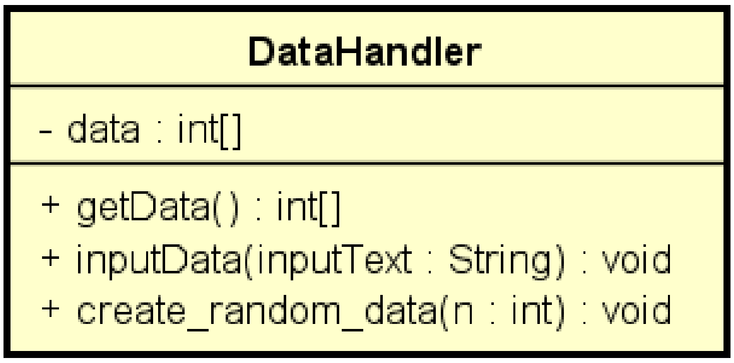
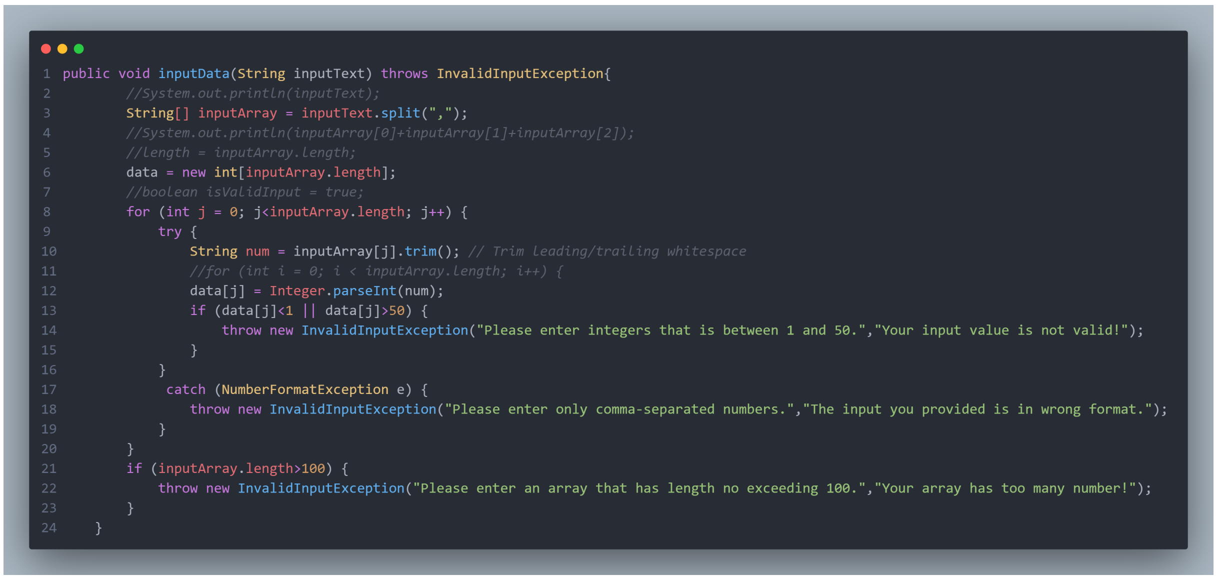

DataHandler
- data : int[]
+ getData( ) : int[]
+ inputData(inputText : String) : void
+ create_random_data(n : int) : void

Figure 2.8: DataHandler class diagram

DataHandler is the class to create an integer array when users input or press the randomize button. It has a data attribute which is an int array. Two methods inputData() and create_random_data() will directly modify this attribute. create_random_data() will generate a random integer array and inputData() will generate an integer array based on user input. It is noted that the only valid user input is a string with comma-seperated numbers like 1,2,3,4,5,...; the user cannot input an array with over 100 elements and each number is between 1 and 50.Thus, if the user enter invalid input, the inputData() method will throw exceptions defined in class main.com.exceptions.



```java
public void inputData(String inputText) throws InvalidInputException{
        //System.out.println(inputText);
        String[] inputArray = inputText.split(",");
        //System.out.println(inputArray[0]+inputArray[1]+inputArray[2]);
        //length = inputArray.length;
        data = new int[inputArray.length];
        //boolean isValidInput = true;
        for (int j = 0; j<inputArray.length; j++) {
            try {
                String num = inputArray[j].trim(); // Trim leading/trailing whitespace
                //for (int i = 0; i < inputArray.length; i++) {
                data[j] = Integer.parseInt(num);
                if (data[j]<1 || data[j]>50) {
                    throw new InvalidInputException("Please enter integers that is between 1 and 50.","Your input value is not valid!");
                }
            }
             catch (NumberFormatException e) {
                throw new InvalidInputException("Please enter only comma-separated numbers.","The input you provided is in wrong format.");
            }
        }
        if (inputArray.length>100) {
            throw new InvalidInputException("Please enter an array that has length no exceeding 100.","Your array has too many number!");
        }
    }
```

Figure 2.9: inputData() method of class DataHandler

2.2.4 Package main.com.exceptions

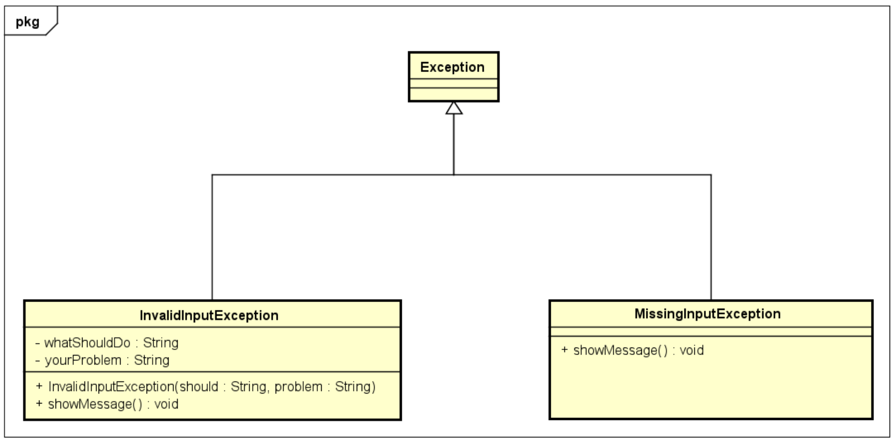


Figure 2.10: Package main.com.exceptions diagram

This package has 2 classes InvalidInputException and MissingInputException that both extend from Exception class of Java. Both classes has a showMessage() method that will pop up an error alert if users type invalid input or press sort without creating data first.

2.2.5 Visualizer Class

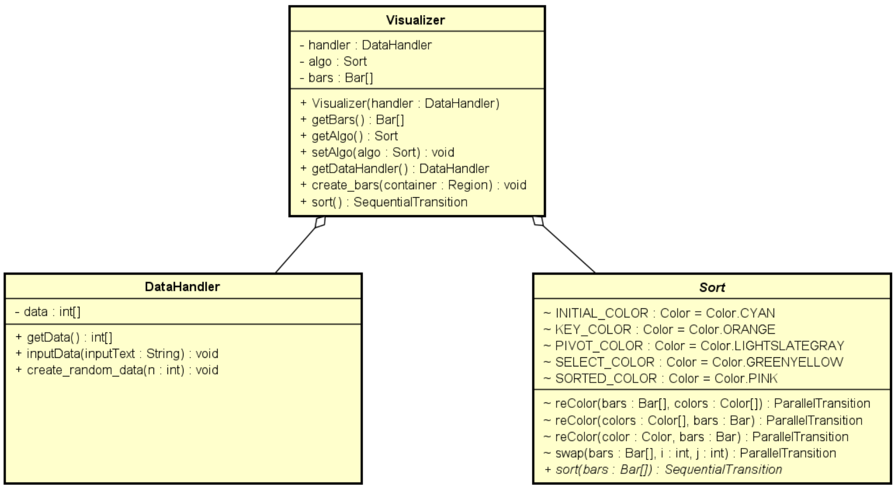
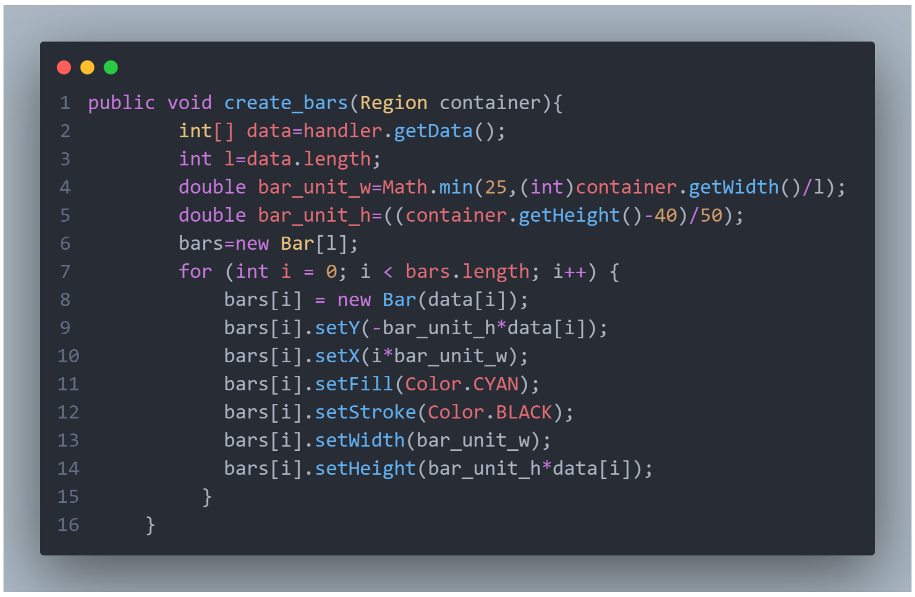

Visualizer
- handler : DataHandler
- algo : Sort
- bars : Bar[]
+ Visualizer(handler : DataHandler)
+ getBars() : Bar[]
+ getAlgo() : Sort
+ setAlgo(algo : Sort) : void
+ getDataHandler() : DataHandler
+ create_bars(container : Region) : void
+ sort() : SequentialTransition
DataHandler
- data : int[]
+ getData() : int[]
+ inputData(inputText : String) : void
+ create_random_data(n : int) : void
Sort
~ INITIAL_COLOR : Color = Color.CYAN
~ KEY_COLOR : Color = Color.ORANGE
~ PIVOT_COLOR : Color = Color.LIGHTSLATEGRAY
~ SELECT_COLOR : Color = Color.GREENYELLOW
~ SORTED_COLOR : Color = Color.PINK
~ reColor(bars : Bar[], colors : Color[]) : ParallelTransition
~ reColor(colors : Color[], bars : Bar) : ParallelTransition
~ reColor(color : Color, bars : Bar) : ParallelTransition
~ swap(bars : Bar[], i : int, j : int) : ParallelTransition
+ sort(bars : Bar[]) : SequentialTransition

Figure 2.11: Visualizer class diagram

A visualizer is the class responsible for all the sorting process. It has a DataHandler and a Sort and an attribute bars. A DataHandler, as described before, is to create an integer array and then the Visualizer will create an array of Bars from that integer array by the method create_bars(). This method has an argument "Region container" to set the height and the width of each bar scaled with the size of the container of those Bars in the UI.



```java
public void create_bars(Region container){
        int[] data=handler.getData();
        int l=data.length;
        double bar_unit_w=Math.min(25,(int)container.getWidth()/l);
        double bar_unit_h=((container.getHeight()-40)/50);
        bars=new Bar[l];
        for (int i = 0; i < bars.length; i++) {
            bars[i] = new Bar(data[i]);
            bars[i].setY(-bar_unit_h*data[i]);
            bars[i].setX(i*bar_unit_w);
            bars[i].setFill(Color.CYAN);
            bars[i].setStroke(Color.BLACK);
            bars[i].setWidth(bar_unit_w);
            bars[i].setHeight(bar_unit_h*data[i]);
          }
     }
```

Figure 2.12: create_bars() method of class Visualizer

Then, a Visualizer can use its Sort to sort the Bars in the sort() method.

2.2.6 VisualizerSceneController Class

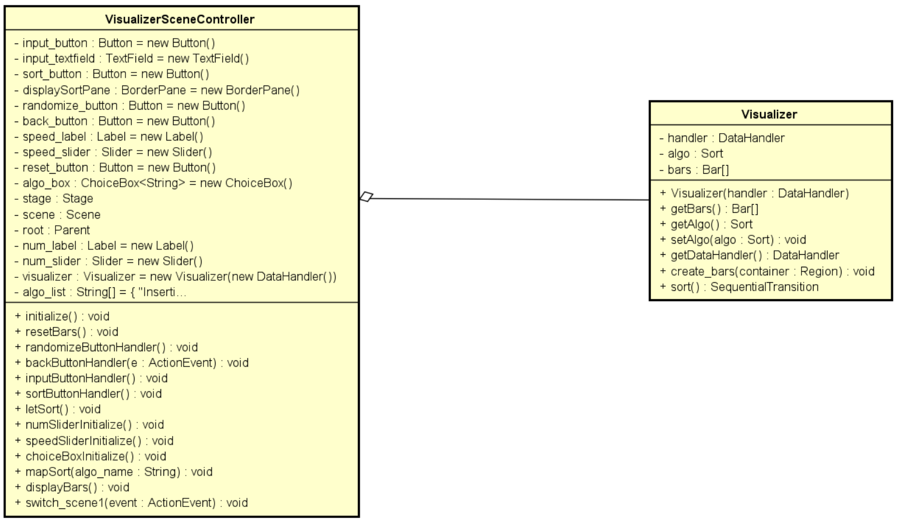
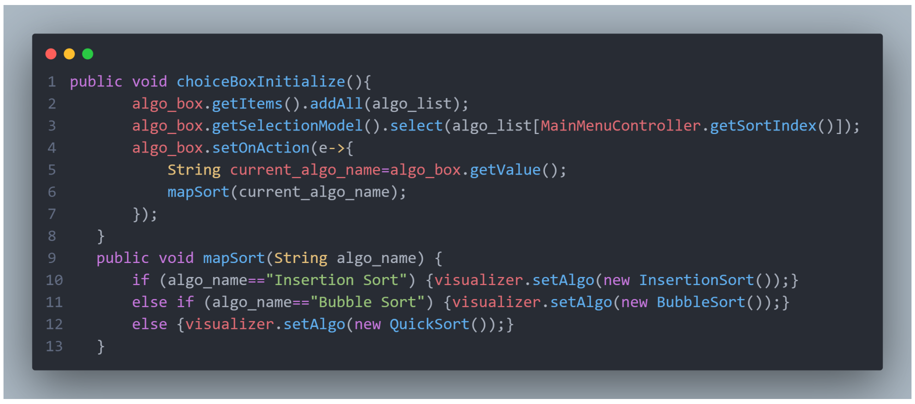

VisualizerSceneController
- input_button : Button = new Button()
- input_textfield : TextField = new TextField()
- sort_button : Button = new Button()
- displaySortPane : BorderPane = new BorderPane()
- randomize_button : Button = new Button()
- back_button : Button = new Button()
- speed_label : Label = new Label()
- speed_slider : Slider = new Slider()
- reset_button : Button = new Button()
- algo_box : ChoiceBox<String> = new ChoiceBox()
- stage : Stage
- scene : Scene
- root : Parent
- num_label : Label = new Label()
- num_slider : Slider = new Slider()
- visualizer : Visualizer = new Visualizer(new DataHandler())
- algo_list : String[] = { "Inserti...
+ initialize() : void
+ resetBars() : void
+ randomizeButtonHandler() : void
+ backButtonHandler(e : ActionEvent) : void
+ inputButtonHandler() : void
+ sortButtonHandler() : void
+ letSort() : void
+ numSliderInitialize() : void
+ speedSliderInitialize() : void
+ choiceBoxInitialize() : void
+ mapSort(algo_name : String) : void
+ displayBars() : void
+ switch_scene1(event : ActionEvent) : void
Visualizer
- handler : DataHandler
- algo : Sort
- bars : Bar[]
+ Visualizer(handler : DataHandler)
+ getBars() : Bar[]
+ getAlgo() : Sort
+ setAlgo(algo : Sort) : void
+ getDataHandler() : DataHandler
+ create_bars(container : Region) : void
+ sort() : SequentialTransition

Figure 2.13: VisualizerSceneController class diagram

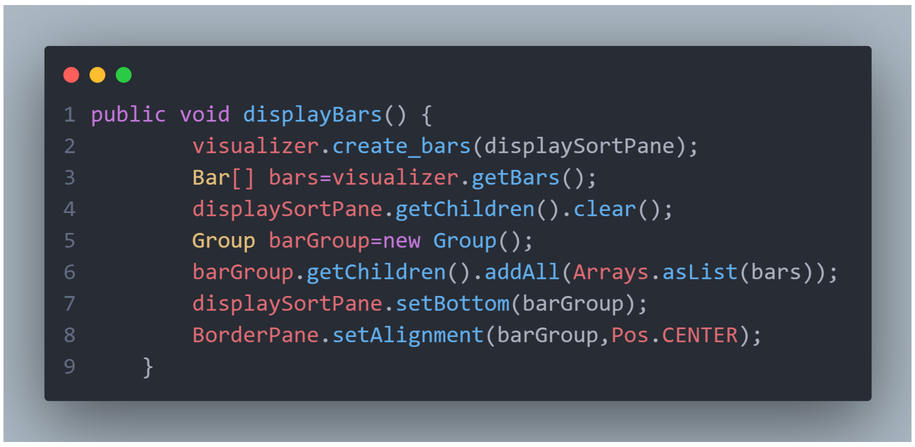
VisualizerSceneController class is where we implement the action of components in the UI. Those actions are implemented based on the methods of class Visualizer.

displayBars() method This method is to add the bars created by class Visualizer to a JavaFX container which is a BorderPane here.



```java
public void choiceBoxInitialize(){
        algo_box.getItems().addAll(algo_list);
        algo_box.getSelectionModel().select(algo_list[MainMenuController.getSortIndex()]);
        algo_box.setOnAction(e->{
            String current_algo_name=algo_box.getValue();
            mapSort(current_algo_name);
        });
    }
    public void mapSort(String algo_name) {
        if (algo_name=="Insertion Sort") {visualizer.setAlgo(new InsertionSort());}
        else if (algo_name=="Bubble Sort") {visualizer.setAlgo(new BubbleSort());}
        else {visualizer.setAlgo(new QuickSort());}
    }
```

Figure 2.15: Handler for algorithm selection choicebox

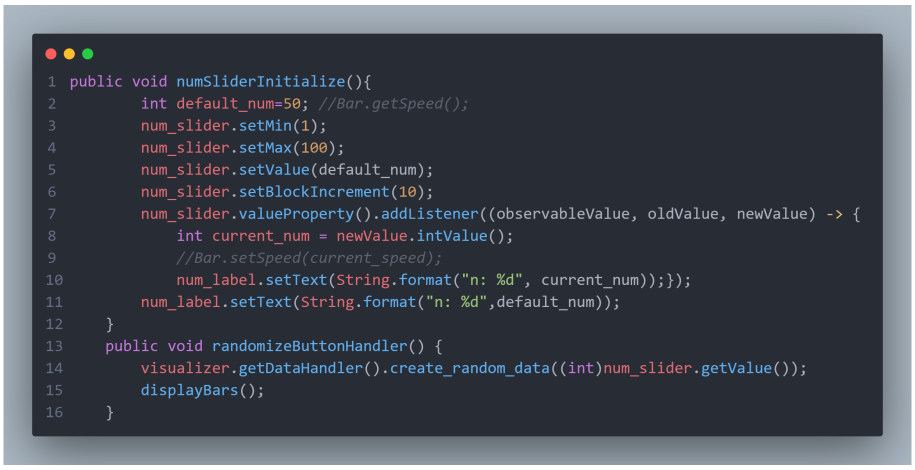


```java
public void displayBars() {
            visualizer.create_bars(displaySortPane);
            Bar[] bars=visualizer.getBars();
            displaySortPane.getChildren().clear();
            Group barGroup=new Group();
            barGroup.getChildren().addAll(Arrays.asList(bars));
            displaySortPane.setBottom(barGroup);
            BorderPane.setAlignment(barGroup,Pos.CENTER);
      }
```

Figure 2.14: displayBars() method

Handler for Choicebox This handler is to set the algorithm used in the Visualizer based on what users choose from the choiceBox in the UI.

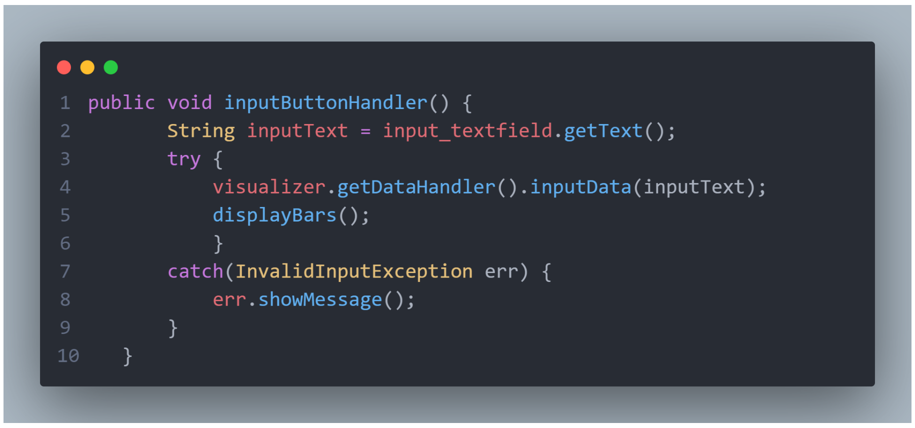
Handler for randomize button When user press "Randomize", an array of random Bars must be displayed to be ready for the sorting proceess.



```java
public void numSliderInitialize(){
        int default_num=50; //Bar.getSpeed();
        num_slider.setMin(1);
        num_slider.setMax(100);
        num_slider.setValue(default_num);
        num_slider.setBlockIncrement(10);
        num_slider.valueProperty().addListener((observableValue, oldValue, newValue) -> {
            int current_num = newValue.intValue();
            //Bar.setSpeed(current_speed);
            num_label.setText(String.format("n: %d", current_num));});
        num_label.setText(String.format("n: %d",default_num));
    }
    public void randomizeButtonHandler() {
        visualizer.getDataHandler().create_random_data((int)num_slider.getValue());
        displayBars();
    }
```

Figure 2.16: Handler for the randomize button

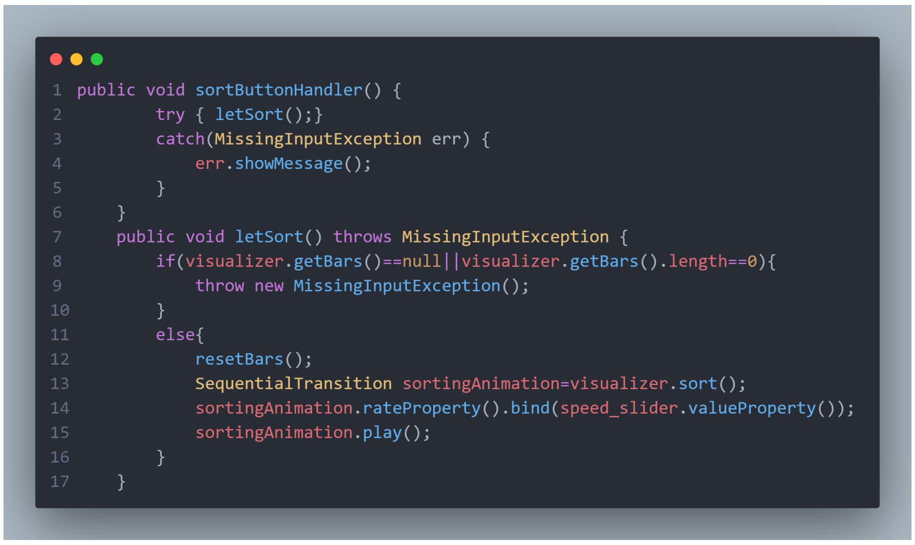
Handler for input button When an user press the "OK" button after typing his/her array in the text field, this handler will catch exception if the input is invalid, otherwise a corresponding array of Bars is displayed.



```java
public void inputButtonHandler() {
        String inputText = input_textfield.getText();
        try {
            visualizer.getDataHandler().inputData(inputText);
            displayBars();
            }
        catch(InvalidInputException err) {
            err.showMessage();
        }
    }
```

Figure 2.17: Handler for the input button

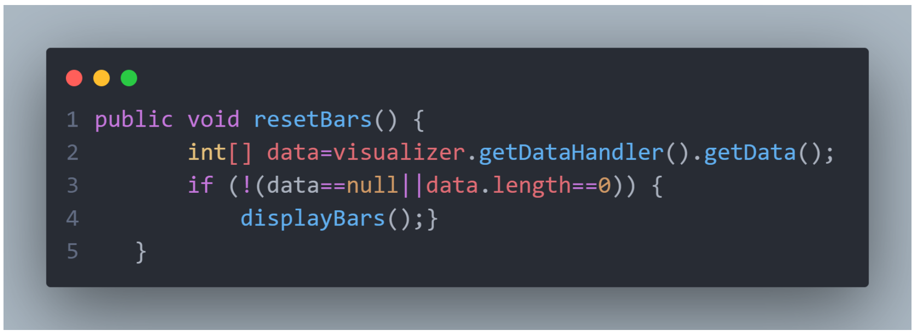
Handler for sort button When an user press the sort button, it will display an alert if no array has been created. Otherwise, if an array of Bars is not sorting, it will start playting sorting animation and if an array is in sorting process and press the sort button again, it will replay the process.



```java
public void sortButtonHandler() {
        try { letSort();}
        catch(MissingInputException err) {
            err.showMessage();
        }
    }
    public void letSort() throws MissingInputException {
        if(visualizer.getBars()==null||visualizer.getBars().length==0){
            throw new MissingInputException();
        }
        else{
            resetBars();
            SequentialTransition sortingAnimation=visualizer.sort();
            sortingAnimation.rateProperty().bind(speed_slider.valueProperty());
            sortingAnimation.play();
        }
    }
```

Figure 2.18: Handler for the sort button

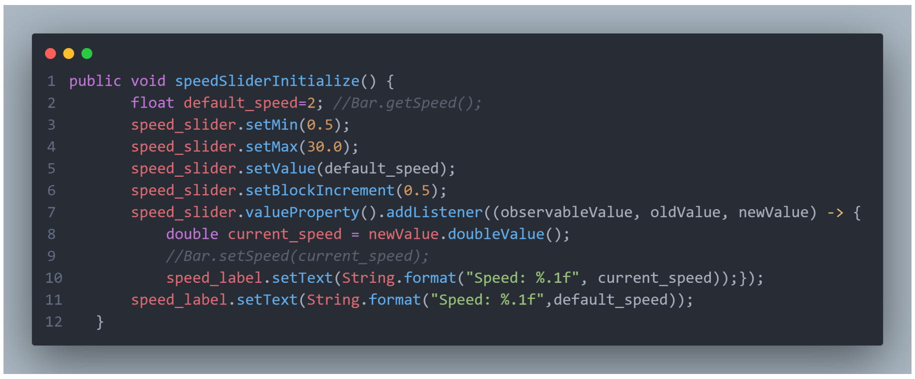
Handler for reset button The reset button is to reset the Bars displayed to its original state before doing any sorting.



```java
public void resetBars() {
        int[] data=visualizer.getDataHandler().getData();
        if (!(data==null||data.length==0)) {
            displayBars();}
    }
```

Figure 2.19: Handler for the reset button

Handler for speed slider The speed slider is control the playing speed of the sorting animation



```java
public void speedSliderInitialize() {
        float default_speed=2; //Bar.getSpeed();
        speed_slider.setMin(0.5);
        speed_slider.setMax(30.0);
        speed_slider.setValue(default_speed);
        speed_slider.setBlockIncrement(0.5);
        speed_slider.valueProperty().addListener((observableValue, oldValue, newValue) -> {
            double current_speed = newValue.doubleValue();
            //Bar.setSpeed(current_speed);
            speed_label.setText(String.format("Speed: %.1f", current_speed));});
        speed_label.setText(String.format("Speed: %.1f",default_speed));
    }
```

Figure 2.20: Handler for the speed slider

2.3 OOP Technique Explaination

2.3.1 Inheritance

In our design, inheritance is a very important property of OOP that we take advantage of. We have 3 classes InsertionSort, BubbleSort and QuickSort that are all inherited from Sort class . This way, we can reuse methods that are used across all classes of sorting algoritms like reColor() or swap().

Additionally, the class Bar is inherited from Rectangle class of JavaFX. This is very useful as Bar can be treated as a Rectangle to be displayed, set color, set height, set width and added to a javafx transition.

We also define some classes that inherit from Exception class(which is a subclass of Throwable class in java) to throw appropriate exception when we need.

2.3.2 Polymorphism

In class Sort, we have an abstract method sort() and all children classes will implement that method in their own way. The visualizer class has an attribute name "algo" of type Sort. By polymorphism in OOP, when users choose a sorting algorithm, we only need to cast appropriate type of Sort to that attribute and then call the method sort which has been overrided.

2.3.3 Association

There are a lot of relationship between different classes in our application. Firstly, the controller class sets action for the UI components in the FXML files. The VisualizerSceneController class has a Visualizer to do that(Aggregation). A Visualizer class has a DataHandler and a Sort(Aggregation). DataHandler class will prepare data for an array of Bars and Sort class sorts that array of Bars.

Aggregation

In our design, a Visualizer has a Datahandler and a Sort. This helps breaking down a Visualizer into smaller components with important and distinct functionality for each component. Additionally, a VisualizerSceneController owns a Visualizer to implement the action of each UI component in the scene for sorting visualizer. Thus, aggeration helps our design to promote code reusability by using existing objects within a class as well as breaking down complex objects into smaller components. This comes in handy for understanding, modifying and testing a complex object like a Visualizer because we can focus on each small components.

Chapter 3

Conclusion

In conclusion, this project demonstrates the implementation and visualization of three fundamental sorting algorithms: Bubble Sort, Insertion Sort, and Quick Sort. By utilizing object-oriented programming principles such as encapsulation, inheritance, and polymorphism, we are able to work on a structured and efficient codebase. The visualizer, created by JavaFX, provides an intuitive understanding of how each algorithm operates, offering a clear view of the sorting process and its effects on each element of the data. This project has not only solidified the understanding of sorting algorithms but also showcases the practical application of OOP concepts in software development generally.

Chapter 4

Reference

[1] https://github.com/chr12c/VisualSortingAlgorithms