

Abstract

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

1	Overview	2
1.1	Members Assignment	2
1.2	Project Requirement	2
1.3	Usecase Diagram	3
1.3.1	Actors	3
1.3.2	Use Cases	3
2	Design Analysis	5
2.1	General Class Diagram	5
2.2	Detailed Class Diagrams	7
2.2.1	Bar Class	7
2.2.2	Package main.com.algorithms	7
2.2.3	DataHandler Class	10
2.2.4	Package main.com.exceptions	11
2.2.5	Visualizer Class	11
2.2.6	VisualizerSceneController Class	12
2.3	OOP Technique Explanation	15
2.3.1	Inheritance	15
2.3.2	Polymorphism	16
2.3.3	Association	16
3	Conclusion	17
4	Reference	18

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.

Pham Dang Tan Dung - 20225569: coding in some parts like handling user input, randomize button, QuickSort, speed adjustment slider; fixing errors, giving the idea to design the program; 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

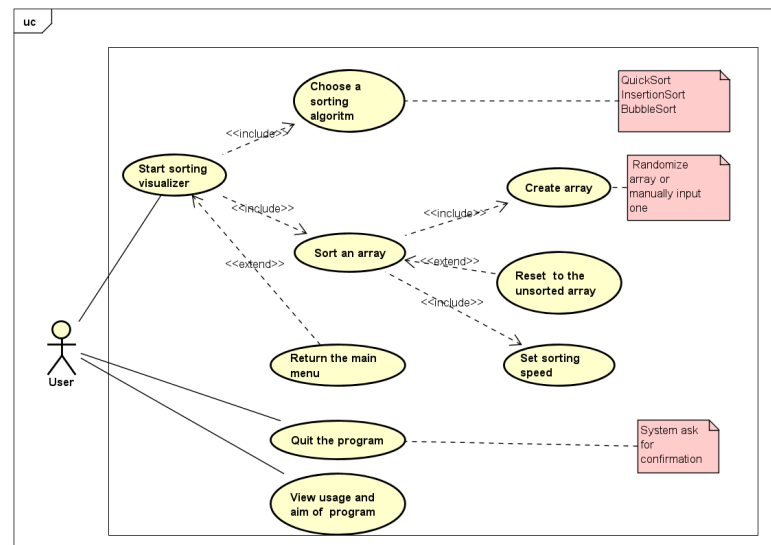


Figure 1.1: Use case diagram

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

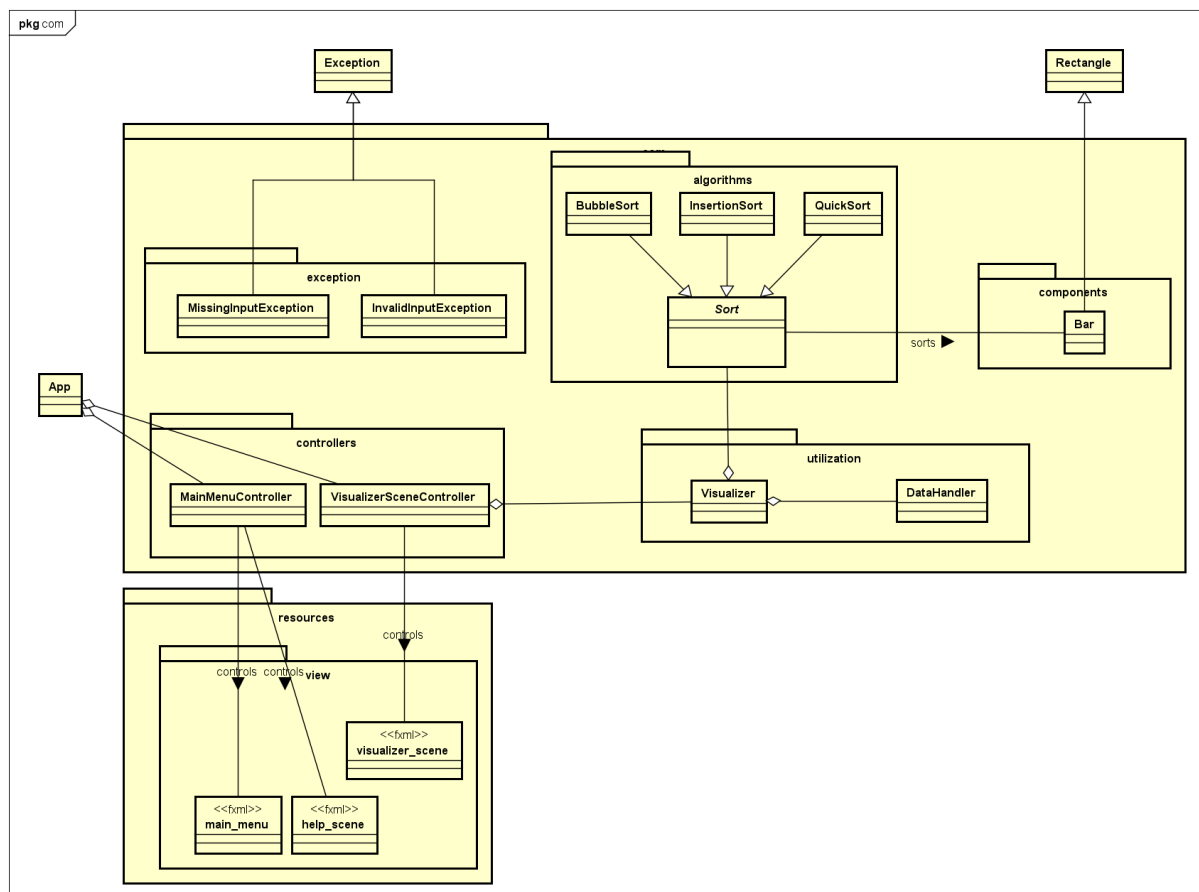


Figure 2.1: General class diagram

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` possessé 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 is 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.

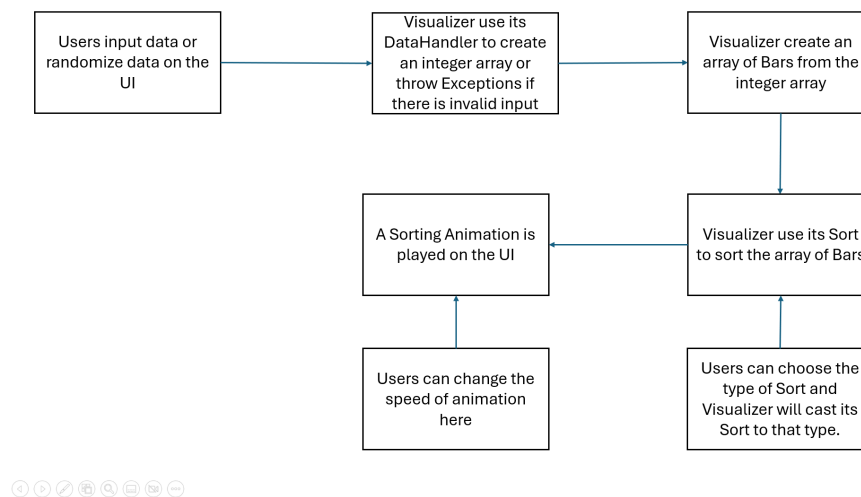


Figure 2.2: Flow of sorting visualizer

2.2 Detailed Class Diagrams

2.2.1 Bar Class

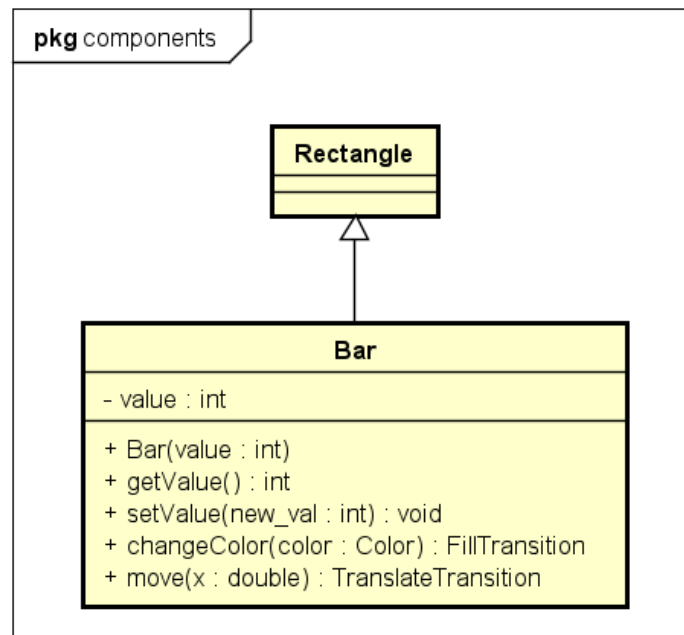


Figure 2.3: Bar class diagram

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.

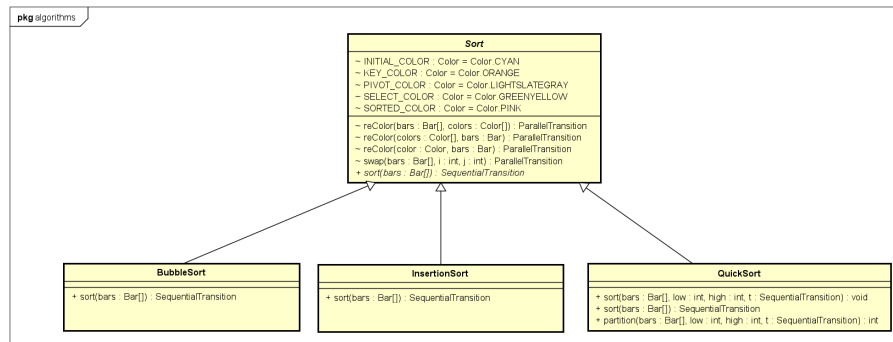


Figure 2.4: Class diagram of package main.com.algorithms

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 some variation 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.

```

1  ParallelTransition reColor(Color[] colors, Bar... bars){
2      ParallelTransition t=new ParallelTransition();
3      for (int i=0;i<bars.length;i++){
4          t.getChildren().add(bars[i].changeColor(colors[i]));
5      }
6      return t;
7  }
8  ParallelTransition reColor(Color color, Bar... bars){
9      ParallelTransition t=new ParallelTransition();
10     for (int i=0;i<bars.length;i++){
11         t.getChildren().add(bars[i].changeColor(color));
12     }
13     return t;
14 }
  
```

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.

```

1 ParallelTransition swap(Bar[] bars,int i,int j){
2     //swap 2 bars given their indexes
3     double d=bars[0].getWidth();
4     ParallelTransition t=new ParallelTransition();
5     double moveDistance=d*(j-i);
6     t.getChildren().addAll(bars[i].move(moveDistance),bars[j].move(-moveDistance));
7     //This swap one is notable
8     Bar temp=bars[i];
9     bars[i]=bars[j];
10    bars[j]=temp;
11    return t;
12 }

```

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 overridden by classes of specific sorting type.

Inherited class from Sort

3 classé 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():

```

1 public class InsertionSort extends Sort{
2     @Override
3     public SequentialTransition sort(Bar[] bars){
4         SequentialTransition t=new SequentialTransition();
5         for (int i = 1; i < bars.length; i++) {
6             int keyVal = bars[i].getValue();
7             int j = i - 1;
8             t.getChildren().add(bars[i].changeColor(KEY_COLOR));
9             while (j >=0 && bars[j].getValue() >keyVal) {
10                 t.getChildren().add(bars[j].changeColor(SELECT_COLOR));
11                 t.getChildren().add(swap(bars,j,j+1));
12                 t.getChildren().add(bars[j+1].changeColor(INITIAL_COLOR));
13                 j--;
14             }
15             t.getChildren().add(bars[j+1].changeColor(INITIAL_COLOR));
16         }
17         t.getChildren().add(reColor(SORTED_COLOR, bars));
18         return t;
19     }
20 }

```

Figure 2.7: Overridden sort() method of class InsertionSort

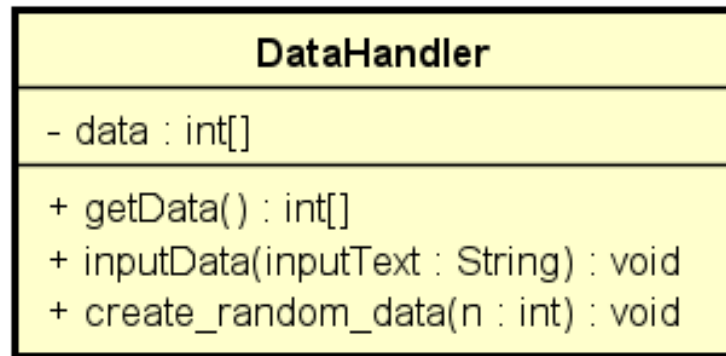


Figure 2.8: DataHandler class diagram

2.2.3 DataHandler Class

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 some invalid input, the inputData() method will throw exceptions defined in class main.com.exceptions.

```

1 public void inputData(String inputText) throws InvalidInputException{
2     //System.out.println(inputText);
3     String[] inputArray = inputText.split(",");
4     //System.out.println(inputArray[0]+inputArray[1]+inputArray[2]);
5     //length = inputArray.length;
6     data = new int[inputArray.length];
7     //boolean isValidInput = true;
8     for (int j = 0; j<inputArray.length; j++) {
9         try {
10             String num = inputArray[j].trim(); // Trim leading/trailing whitespace
11             //for (int i = 0; i < inputArray.length; i++) {
12             data[j] = Integer.parseInt(num);
13             if (data[j]<1 || data[j]>50) {
14                 throw new InvalidInputException("Please enter integers that is between 1 and 50.", "Your input value is not valid!");
15             }
16         }
17         catch (NumberFormatException e) {
18             throw new InvalidInputException("Please enter only comma-separated numbers.", "The input you provided is in wrong format.");
19         }
20     }
21     if (inputArray.length>100) {
22         throw new InvalidInputException("Please enter an array that has length no exceeding 100.", "Your array has too many number!");
23     }
24 }

```

Figure 2.9: inputData() method of class DataHandler

2.2.4 Package main.com.exceptions

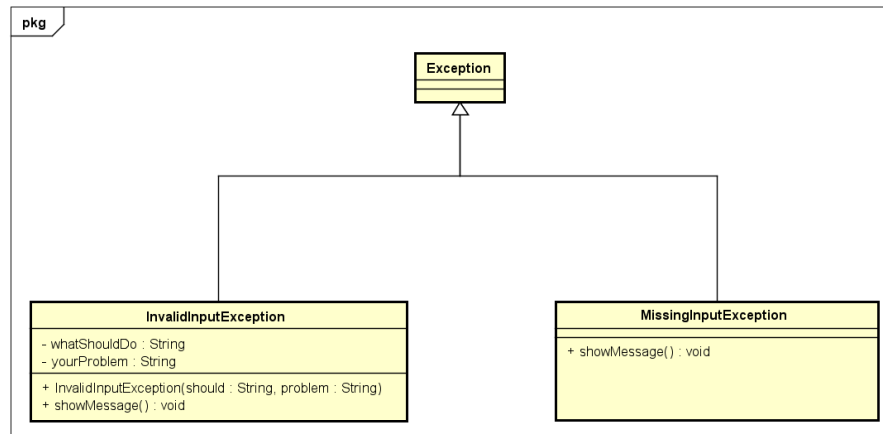


Figure 2.10: Package main.com.exceptions diagram

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

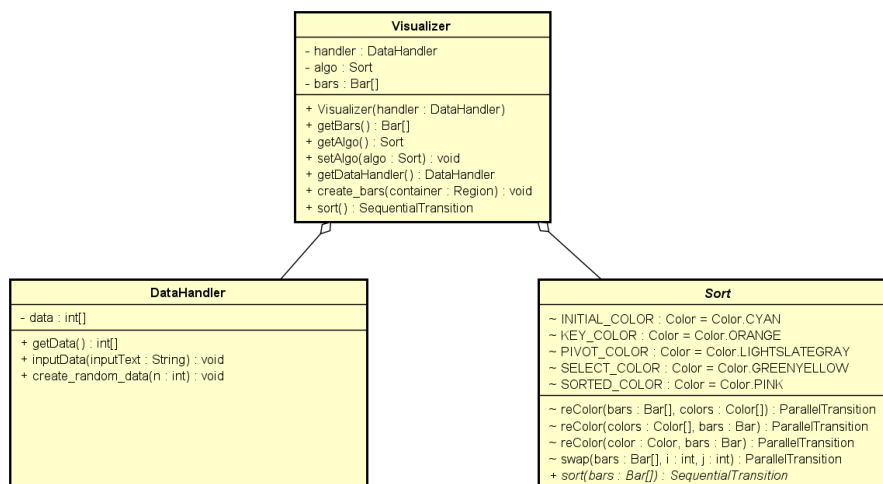


Figure 2.11: Visualizer class diagram

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.

```

1 public void create_bars(Region container){
2     int[] data=handler.getData();
3     int l=data.length;
4     double bar_unit_w=Math.min(25,(int)container.getWidth()/l);
5     double bar_unit_h=((container.getHeight()-40)/50);
6     bars=new Bar[l];
7     for (int i = 0; i < bars.length; i++) {
8         bars[i] = new Bar(data[i]);
9         bars[i].setY(-bar_unit_h*data[i]);
10        bars[i].setX(i*bar_unit_w);
11        bars[i].setFill(Color.CYAN);
12        bars[i].setStroke(Color.BLACK);
13        bars[i].setWidth(bar_unit_w);
14        bars[i].setHeight(bar_unit_h*data[i]);
15    }
16 }

```

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

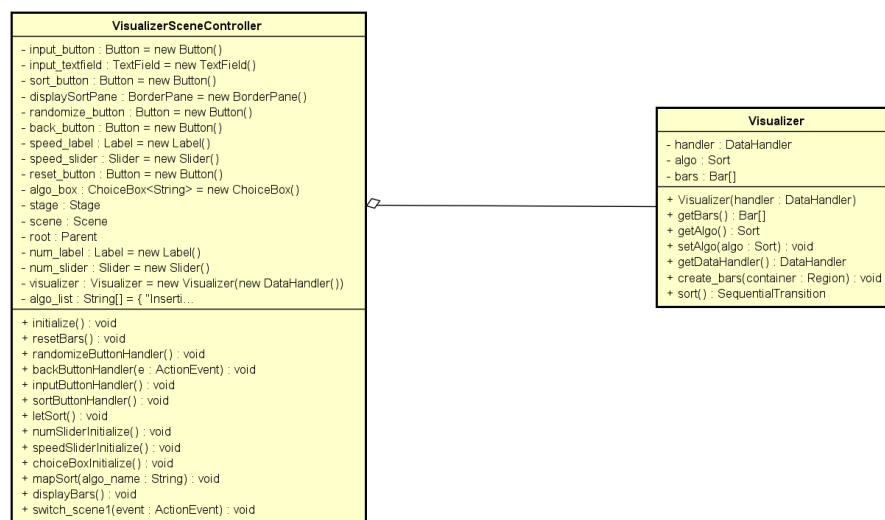


Figure 2.13: VisualizerSceneController class diagram

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.

```

1 public void choiceBoxInitialize(){
2     algo_box.getItems().addAll(algo_list);
3     algo_box.getSelectionModel().select(algo_list[MainMenuController.getSortIndex()]);
4     algo_box.setOnAction(e->{
5         String current_algo_name=algo_box.getValue();
6         mapSort(current_algo_name);
7     });
8 }
9 public void mapSort(String algo_name) {
10     if (algo_name=="Insertion Sort") {visualizer.setAlgo(new InsertionSort());}
11     else if (algo_name=="Bubble Sort") {visualizer.setAlgo(new BubbleSort());}
12     else {visualizer.setAlgo(new QuickSort());}
13 }

```

Figure 2.15: Handler for algorithm selection choicebox

```

1 public void displayBars() {
2     visualizer.create_bars(displaySortPane);
3     Bar[] bars=visualizer.getBars();
4     displaySortPane.getChildren().clear();
5     Group barGroup=new Group();
6     barGroup.getChildren().addAll(Arrays.asList(bars));
7     displaySortPane.setBottom(barGroup);
8     BorderPane.setAlignment(barGroup,Pos.CENTER);
9 }

```

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 process.

```

1 public void numSliderInitialize(){
2     int default_num=50; //Bar.getSpeed();
3     num_slider.setMin(1);
4     num_slider.setMax(100);
5     num_slider.setValue(default_num);
6     num_slider.setBlockIncrement(10);
7     num_slider.valueProperty().addListener((observableValue, oldValue, newValue) -> {
8         int current_num = newValue.intValue();
9         //Bar.setSpeed(current_speed);
10        num_label.setText(String.format("n: %d", current_num));});
11    num_label.setText(String.format("n: %d",default_num));
12 }
13 public void randomizeButtonHandler() {
14     visualizer.getDataHandler().create_random_data((int)num_slider.getValue());
15     displayBars();
16 }

```

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.

```
1 public void inputButtonHandler() {
2     String inputText = input_textfield.getText();
3     try {
4         visualizer.getDataHandler().inputData(inputText);
5         displayBars();
6     }
7     catch(InvalidInputException err) {
8         err.showMessageDialog();
9     }
10 }
```

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 playing sorting animation and if an array is in sorting process and press the sort button again, it will replay the process.

```
1 public void sortButtonHandler() {
2     try { letSort();}
3     catch(MissingInputException err) {
4         err.showMessageDialog();
5     }
6 }
7 public void letSort() throws MissingInputException {
8     if(visualizer.getBars()==null||visualizer.getBars().length==0){
9         throw new MissingInputException();
10    }
11    else{
12        resetBars();
13        SequentialTransition sortingAnimation=visualizer.sort();
14        sortingAnimation.rateProperty().bind(speed_slider.valueProperty());
15        sortingAnimation.play();
16    }
17 }
```

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.

```

1 public void resetBars() {
2     int[] data=visualizer.getDataHandler().getData();
3     if (!(data==null||data.length==0)) {
4         displayBars();
5     }
}

```

Figure 2.19: Handler for the reset button

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

```

1 public void speedSliderInitialize() {
2     float default_speed=2; //Bar.getSpeed();
3     speed_slider.setMin(0.5);
4     speed_slider.setMax(30.0);
5     speed_slider.setValue(default_speed);
6     speed_slider.setBlockIncrement(0.5);
7     speed_slider.valueProperty().addListener((observableValue, oldValue, newValue) -> {
8         double current_speed = newValue.doubleValue();
9         //Bar.setSpeed(current_speed);
10        speed_label.setText(String.format("Speed: %.1f", current_speed));
11        speed_label.setText(String.format("Speed: %.1f",default_speed));
12    }
}

```

Figure 2.20: Handler for the speed slider

2.3 OOP Technique Explanation

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 algorithms 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 inherited 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 a user choose his wanted sorting algorithm, we only need to cast appropriate type of Sort to that attribute and then call the method sort which has been overridden.

2.3.3 Association

There are a lot of relationship between different classes in our application. Firstly, the controller class set action for the UI components in the fxml files. The Visualizer-SceneController class has a Visualizer to do that(Aggregation). A Visualizer class has a DataHandler and a Sort(Aggregation). DataHandler 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 break 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, aggregation helps our design promote code reusability by using existing objects within a class as well as breaking down complex objects into smaller components. This is very helpful in understand, modify and test 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. Utilizing object-oriented programming principles such as encapsulation, inheritance, and polymorphism allowed for a structured and efficient codebase. The visualizer provides an intuitive understanding of how each algorithm operates, offering a clear view of the sorting process and its effects on data. This project not only solidifies the understanding of sorting algorithms but also showcases the practical application of OOP concepts in software development.

Chapter 4

Reference

- [1] <https://github.com/chr12c/VisualSortingAlgorithms>