

# **CPT111**

## **Course Work 3**

### Report

Yichuan.Zhang

2251668

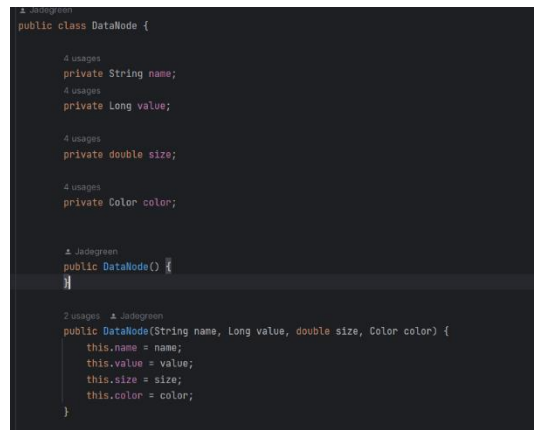
## Sankey Plot Report

This report aims to provide a comprehensive explanation of the developed Java code, focusing on code structure, algorithms, file and error handling, display features, and the application of Object-Oriented Programming (OOP) principles. The code provided in the report fully satisfies the basic requirements of the Sankey diagram. With the help of JavaFX packages, the diagram successfully links components of different entities provided in the source files. Additionally, this diagram undergoes dynamic processing for its individual components, allowing the shapes of the elements to change with varying window sizes. With the inclusion of special functions, users can simply drag and drop the .txt file into the program window, enabling the program to process the file and plot the diagram with vibrant colors and animation. The report is structured into seven chapters, each dedicated to specific aspects ranging from OOP principles to the detailed implementation of the code.

### Chapter 1 - Object-oriented Principles

#### 1. Encapsulation

There are four classes encapsulated in this project. As shown in the picture. The data within each entity is encapsulated within a class called “DataNode” for storage. There are some other attributes that is also added to the class to help the program plot the diagram. The variables inside this

A screenshot of a code editor showing the Java code for the DataNode class. The code is as follows:

```
1. Jsdagreen
public class DataNode {

    4 usages
    private String name;
    4 usages
    private Long value;

    4 usages
    private double size;

    4 usages
    private Color color;

    1 Jsdagreen
    public DataNode() {}

    2 usages  1 Jsdagreen
    public DataNode(String name, Long value, double size, Color color) {
        this.name = name;
        this.value = value;
        this.size = size;
        this.color = color;
    }
}
```

class are all marked with the 'private' modifier and can be accessed and modified through the following get and set methods. The “SankeyPlot” class mainly deal with the algorithms of the coordinates and invoke the “PlotItem” class to plot the diagrams including the rectangles and the curves. The “main” class is designed to implement the drag and drop functional event and instantiate the scene, pane and other necessary components to run the program.

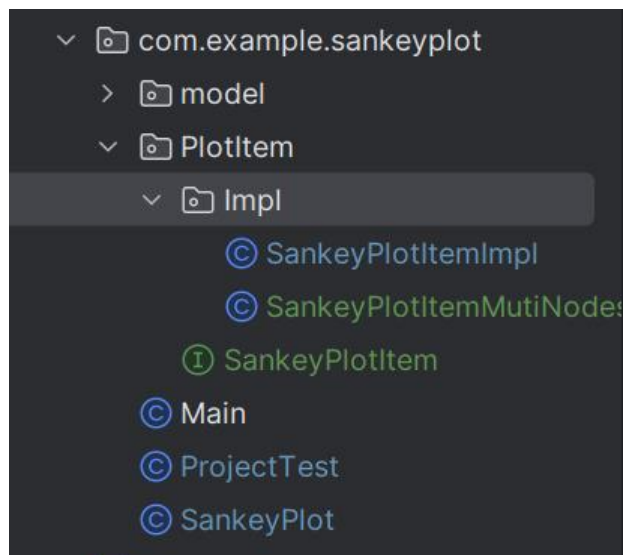
#### 2. Inheritance

The “Main” class extends the “application” class of the JavaFX package so that it can display the scene. The “Main” class override the “start” method to display the Scene at the beginning of the program.

```
public class Main extends Application {  
    3 usages  
    private SankeyPlot sankeyPlot;  
    1 usage  
    private double recRightUp = 0.15;  
    1 Jadedgreen  
    @Override  
    public void start(Stage primaryStage) {  
  
        VBox root = new VBox();  
        root.setAlignment(Pos.CENTER);  
  
        // Set drag and drop event listener  
        root.setOnDragOver(event -> {  
            if (event.getDragboard().hasFiles()) {  
                event.acceptTransferModes(TransferMode.COPY_OR_MOVE);  
            }  
            event.consume();  
        });  
    }  
}
```

### 3. polymorphism

The “SankeyPlotItemImpl” and the “SankeyPlotItemMutiNodes” are all implementations of the Interface called “SankeyPlotItem”. These two classes both implement the same interface, so their types can be interchangeable, allowing them to be placed within the same data structure of the interface type for subsequent management and maintenance. Due to the limitation



of the datasets the “SankeyPlotItemMutiNodes” is not implemented but to show the polymorphism and extensibility of the program.

### 4. Abstraction

Abstraction in programming refers to the process of simplifying complex real-world problems into a manageable representation suitable for computer processing. It involves focusing on the essential aspects while ignoring unnecessary details. As shown in the polymorphism, the program abstracted an interface called "SankeyPlotItem" and then implemented it in two classes. This enables them to have similar functionalities, such as the ability to draw similar types of graphs. However, they can modify their respective code according to different scenarios to achieve distinct functionalities. The code of the interface is shown below.

```

public interface SankeyPlotItem {
    1 usage 1 implementation new *
    void plotRectangles(Scene scene, ArrayList<Rectangle> rectangleList, double recRightUp, double rectangleX);

    1 usage 1 implementation new *
    void plotCurve(double leftUpY1, double rightUpY2, double rightDownY2, double leftDownY1, double rectangleX);

    1 usage 1 implementation new *
    void plotText(ArrayList<Text> textList);

    1 usage 1 implementation new *
    void plotAnimation();
}

```

## 5. Modularity

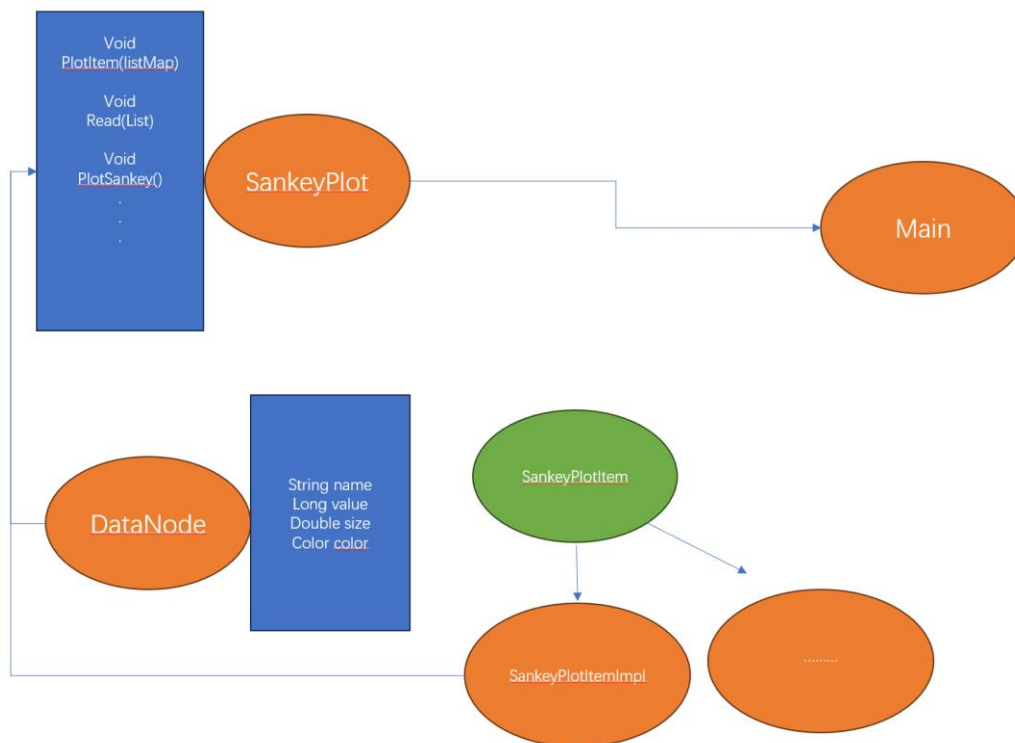
The project hierarchy was previously presented. I've created two packages to store different Java source files. The first package is named "model," intended for storing various data types. In the future, when the project needs to handle new data, newly abstracted data storage classes can be uniformly managed within this package. The classes in the other package are primarily used to store the overall logic for drawing. Interfaces will be placed in the first layer, while their implementation classes will be placed in the "Impl" package. This approach will result in a clearer overall package structure.

## 6. Dependency

To enhance the project's engineering and scalability, I've decoupled the continuous logic of drawing and algorithm implementation. I segregated the logic responsible for drawing into a separate interface, subsequently encapsulating it as an independent interface. This was achieved by implementing classes for image rendering. Upon program initiation, an instance of the "SankeyPlot" class is created. This object selectively instantiates different "PlotItem" implementations based on requirements for executing the drawing logic. This establishes a dependency between the two classes, reducing the likelihood of extensive code occurrences and enhancing code maintainability and extensibility.

## 7. Class Design

For the class design of this project, there has been a general exposition within the framework of the entire project as outlined earlier. The following presentation focuses on some classes and properties I designed throughout the entire design process.



## Chapter 2 - Diagram Display Algorithm

In this part of the report, the detailed explanation of the design and implementation details of the diagram display algorithm, providing insight into the logic and execution of the display algorithm is provided.

Firstly, to create the entire Sankey diagram, we need to obtain all the nodes to be drawn (encapsulated as `DataNodes` in this program). All these nodes are placed

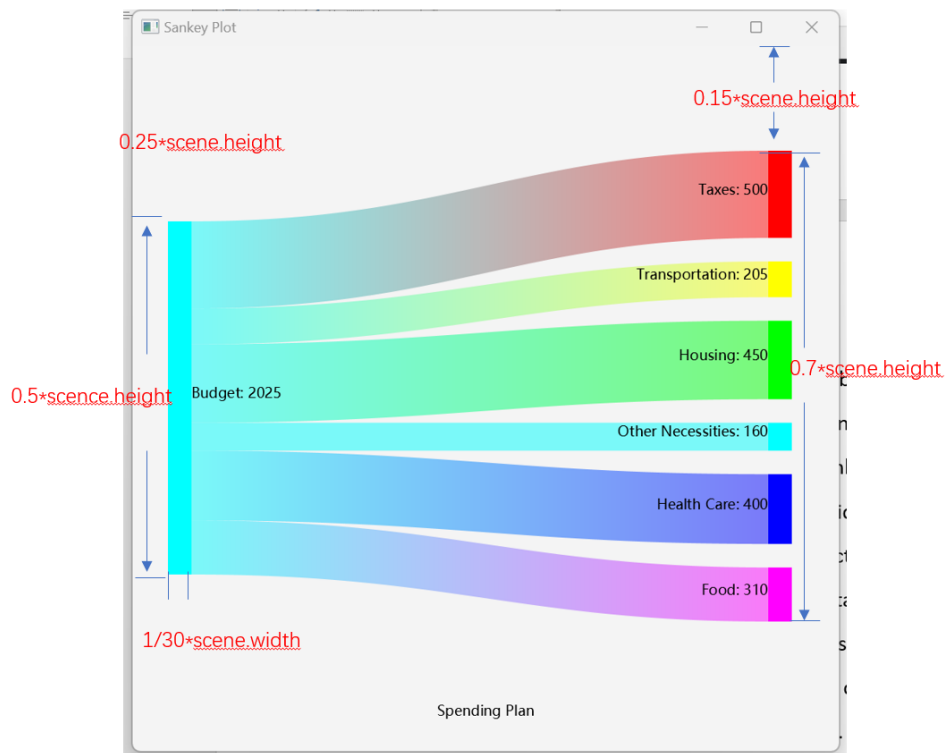
```

//plot the rectangles and curves
for (DataNode node : nodeList) {

    SankeyPlotItemImpl sankeyPlotItem = createSankeyPlotItem(node);
    sankeyPlotItem.plotRectangles(scene, rectangleList, recRightUp, rectangleX);
    recRightUp += (0.4 / nodeList.size()) * 0.5 + node.getSize() * 0.5;
    double rightDownY2 = rightUpY2 + (0.5 * node.getSize());
    double leftDownY1 = leftUpY1 + node.getSize() * 0.5;
    sankeyPlotItem.plotCurve(leftUpY1, rightUpY2, rightDownY2, leftDownY1, rectangleX);
    leftUpY1 += (node.getSize()) * 0.5;
    rightUpY2 += (0.4 / nodeList.size()) * 0.5 + node.getSize() * 0.5;
    sankeyPlotItem.plotText(textList);
    sankeyPlotItem.plotAnimation();
}

```

within a mutable ArrayList named 'nodeList.' Subsequently, we iterate through the entire nodeList, fetching the data for each node in the list and individually proceeding with the drawing process. In this Sankey diagram, the height of the left rectangles is half the window's height, and the width of all rectangles is one-thirtieth of the window's width. The height of the right rectangles is determined based on the ratio of each node's representation in the left rectangles. The right section is 1.4 times the left section, implying that the combined gaps on the right side amount to 0.4 times the left rectangles.



Next, we need to abstract six pointers, used respectively for drawing the left and right graphics. We need to calculate the specific coordinates for each node and the increments required for the drawing pointers. They are named as rectangleX, recRightUp, rightUpY2, leftUpY1, rightDownY2, leftDownY1 (all of type double), where their values denote the ratio of these coordinates to the corresponding window coordinates. Among these, 'rectangleX' represents the horizontal coordinate

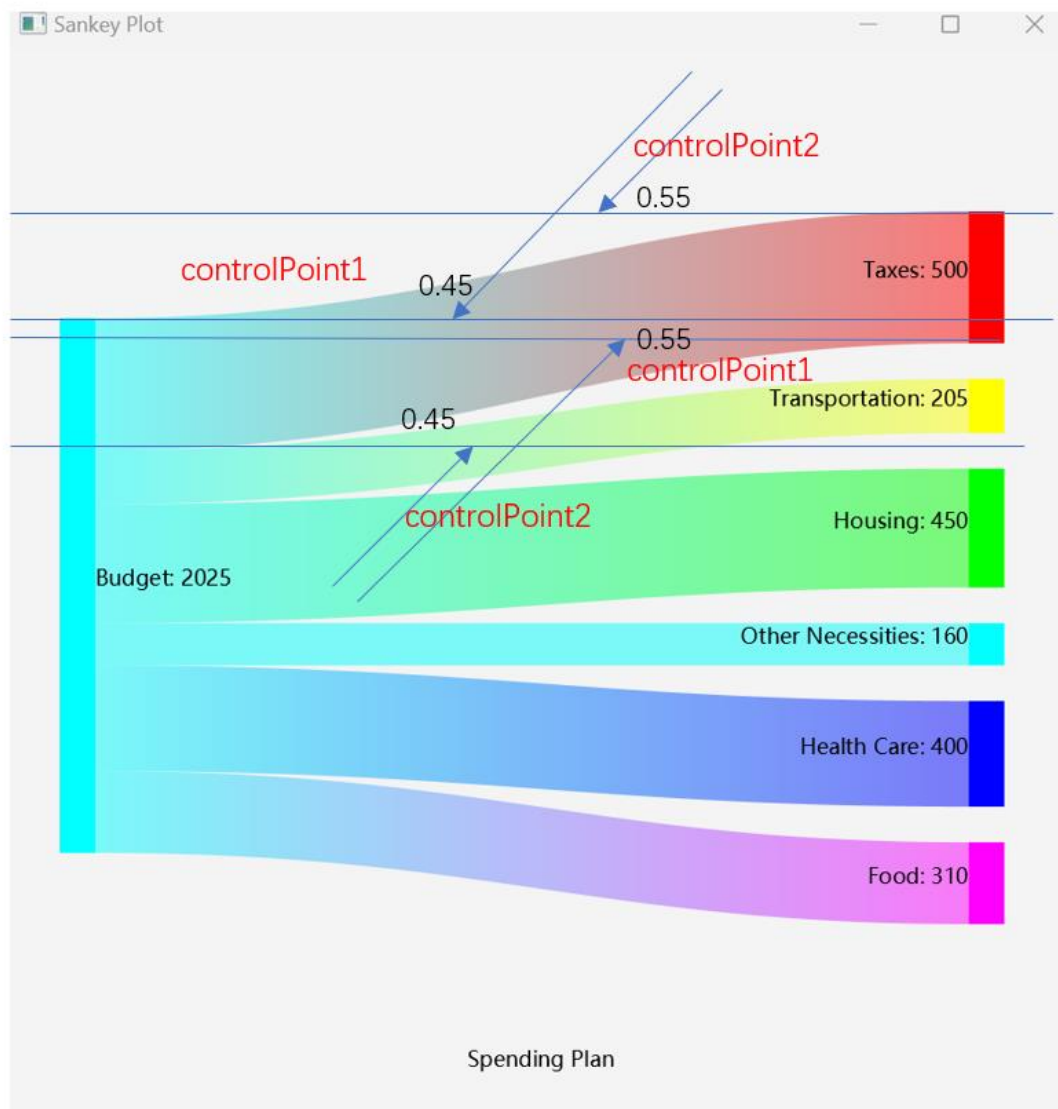
of all the rectangles on the right side. 'recRightUp' signifies the vertical coordinate of the pointer positioned at the upper edge when drawing the right-side rectangle. 'rightUpY2' refers to the pointer situated at the upper edge of the right-side rectangle when drawing curves. 'leftUpY1' denotes the pointer located at the upper edge of the left-side rectangle when drawing curves. 'rightDownY2' indicates the pointer at the lower edge of the right-side rectangle when drawing curves. 'leftDownY1' represents the pointer at the lower edge of the left-side rectangle when drawing curves. The pointer will change its size during the loop, thus affecting the rendering of the entire figure. The pointer below will move in conjunction with the upper pointer, which follows the logic outlined below:

```
leftUpY1 += (node.getSize()) * 0.5;  
rightUpY2 += (0.4 / nodeList.size()) * 0.5 + node.getSize() * 0.5;
```

The sizes here represent proportions relative to the total count, hence relative to the left rectangles. Therefore, when transforming into window coordinates, they need to be multiplied by the ratio of the left rectangles to the window, which is 0.5. When calculating the upper pointers of the right-side rectangles, it's essential to consider the gap's size. As mentioned earlier, the total size of the gaps is 0.4 times the size of the left rectangles. Therefore, dividing the total size by the number of nodes gives us the size of each gap. Of course, this value also needs to be multiplied by the ratio of the left rectangles to the window.

All the images utilize standard graphics from the JavaFX library. Rectangles are created using 'Rectangle', while curves are made using 'Path', 'CubicCurveTo', 'MoveTo', and 'LineTo'. The coordinates for curves can be a bit complex. Using 'Path' to draw block curves requires specifying parameters for two separate curves: the coordinates of the starting point and the parameters for the line connecting the two curves. Each curve has three components: the coordinates of control point one, the coordinates of control point two, and the coordinates of the end point. The end point positions and the coordinates of the control points are inversely related for the two curves. Upon measurement, the control points need to be aligned horizontally with both the starting and ending points. Therefore, the vertical coordinate of the control

point is equal to the coordinate of a parallel pointer (usually the closest to the control point). After measurement, the curve line appears smoothest when the horizontal coordinates of the control points are respectively at 0.45 and 0.55 times the window width.



Finally, we need to add text to the entire image, which is also done within the loop. In this project, we need to create all text objects and then associate each one with a rectangle. Here, we've utilized bindings, which can be used to link the properties between two objects. These text objects will be placed in an ArrayList and collectively added to the entire pane.



```

// Usage: JadeGreen
public void plotText(ArrayList<Text> textList){
    Text text = new Text( s: node.getName() + ": " + node.getValue());
    //The text is placed near the rectangle
    text.yProperty().bind(Bindings.add(rectangle.yProperty(), rectangle.heightProperty().multiply(v: 0.5)));
    text.xProperty().bind(Bindings.subtract(rectangle.xProperty(), text.getLayoutBounds().getWidth()));
    textList.add(text);
}

```

## Chapter 3 - Display while Resizing Algorithm

In this part of report, we will explore the algorithm employed for dynamic display adjustment when resizing the window, illustrating how the image adapts to window size changes.

This project, we've employed 'bind' and the 'Bindings' class to establish connections with the window. As mentioned earlier, all coordinates have been converted to ratios relative to the window. Therefore, by binding all coordinates to their respective window coordinates, we achieve the functionality of resizing.

```

CubicCurveTo cubicCurveTo1 = new CubicCurveTo();
double controlX1 = 0.45;
cubicCurveTo1.controlX1Property().bind(Bindings.multiply(scene.widthProperty(), controlX1));
cubicCurveTo1.controlY1Property().bind(Bindings.multiply(scene.heightProperty(), leftUpY1));
double controlX2 = 0.55;
cubicCurveTo1.controlX2Property().bind(Bindings.multiply(scene.widthProperty(), controlX2));
cubicCurveTo1.controlY2Property().bind(Bindings.multiply(scene.heightProperty(), rightUpY2));
cubicCurveTo1.xProperty().bind(Bindings.multiply(scene.widthProperty(), rectangleX));
cubicCurveTo1.yProperty().bind(Bindings.multiply(scene.heightProperty(), rightUpY2));
CubicCurveTo cubicCurveTo2 = new CubicCurveTo();
cubicCurveTo2.controlX1Property().bind(Bindings.multiply(scene.widthProperty(), controlX2));
cubicCurveTo2.controlY1Property().bind(Bindings.multiply(scene.heightProperty(), rightDownY2));
cubicCurveTo2.controlX2Property().bind(Bindings.multiply(scene.widthProperty(), controlX1));
cubicCurveTo2.controlY2Property().bind(Bindings.multiply(scene.heightProperty(), leftDownY1));
cubicCurveTo2.xProperty().bind(Bindings.add(rectangle1.xProperty(), rectangle1.widthProperty()));
cubicCurveTo2.yProperty().bind(Bindings.multiply(scene.heightProperty(), leftDownY1));
MoveTo moveTo = new MoveTo();
moveTo.xProperty().bind(Bindings.add(rectangle1.xProperty(), rectangle1.widthProperty()));
moveTo.yProperty().bind(Bindings.multiply(scene.heightProperty(), leftUpY1));
filledPath.getElements().add(moveTo);
filledPath.getElements().add(cubicCurveTo1);
LineTo lineTo = new LineTo();
lineTo.xProperty().bind(rectangle.xProperty());
lineTo.yProperty().bind(Bindings.multiply(scene.heightProperty(), rightDownY2));
filledPath.getElements().add(lineTo);
filledPath.getElements().add(cubicCurveTo2);
filledPath.getElements().add(new ClosePath());
// Set the stroke color to transparent
filledPath.setStroke(Color.TRANSPARENT);

```

## Chapter 4 - Additional Features

This project not only possesses strong engineering and scalability, but also harbors

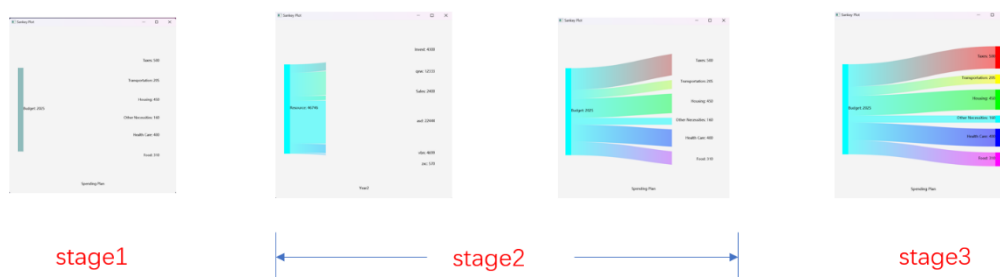
several distinctive features.

## 1. Gradually changing colors

The Sankey diagram in this project optimizes color matching, making the transition of colors more consistent. The leftmost side is cyan, and the color of the middle curve gradually changes from the color of the left rectangle to the color of the right rectangle. The color of the right rectangle changes with the amount of data. The implementation principle is to encode colors using HSB. The hue increases from 0 to 360, and for each additional color, the hue increases by the value of 360 divided by the number of data, so that it always starts from 0 and ends at 360. The brightness of the middle curve is reduced by half using HSB encoding.

## 2. Carefully crafted animated transformations.

To enhance the fun and smoothness of this Sankey diagram, I added a beautiful rendering animation, which is divided into three parts and plays from left to right. The first part renders the leftmost rectangle, which will change from transparent to cyan color and render for 0.5 seconds. The second part draws the entire curve, which



will render for 2.5 seconds until it links to the right array of rectangles. The third part renders the entire rectangle from left to right, rendering for 0.5 seconds.

The animation function of rectangles is implemented through the TimeLine built-in to JavaFX. In the TimeLine, you can set the initial and end states of the graph, and then set the change time to achieve the animation effect.

### 3. Sort the data

```
1 usage: new *
@
public static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map) {
    return map.entrySet().stream()
        .sorted(Map.Entry.comparingByValue())
        .collect(Collectors.toMap(
            Map.Entry::getKey,
            Map.Entry::getValue,
            (oldValue, newValue) -> oldValue,
            LinkedHashMap::new
        ));
}
```

This code snippet defines a method named “sortByValue” that sorts a provided map based on its values in ascending order. It employs Java streams to process the map entries. By obtaining a stream of the map's entries and utilizing the “sorted” operation with “Map.Entry.comparingByValue()”, it arranges the entries according to their values. The sorted entries are then collected back into a new map using “Collectors.toMap()”. To maintain the original insertion order, it utilizes a “LinkedHashMap”. Ultimately, this method returns a new map where the entries are sorted in ascending order based on their values.

## Chapter 5 - File Handling

This report will introduce the methods used by this project to process file data in this section.

This program mainly uses JavaFX encapsulated events to receive and process files. In it, I use a VBox to handle different controls, and then bind a drag-over event on the VBox to detect whether any files are dragged and uploaded. Then, the program will start reading the files in it and read all the data contained in this txt file line by line into a

```
VBox root = new VBox();
root.setAlignment(Pos.CENTER);

// Set drag and drop event listener
root.setOnDragOver(event -> {
    if (event.getDragboard().hasFiles()) {
        event.acceptTransferModes(TransferMode.COPY_OR_MOVE);
    }
    event.consume();
});

ArrayList<String> list = new ArrayList<>();
// Handle drag and drop events
root.setOnDragDropped(event -> {
    var db = event.getDragboard();
    boolean success = false;
    list.clear();
    if (db.hasFiles()) {
        success = true;
        for (File file : db.getFiles()) {
            try {
                BufferedReader reader = new BufferedReader(new FileReader(file));
                String line;
                while ((line = reader.readLine()) != null) {
                    list.add(line);
                }
                reader.close();
                if (!checkFileValidity(list)) {
                    showAlert("Invalid file, Please drag and drop a standard text file.");
                    event.consume();
                    return;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    // Create and display the next scene
    openNextScene(primaryStage, list);
    event.setDropCompleted(success);
    event.consume();
});
```

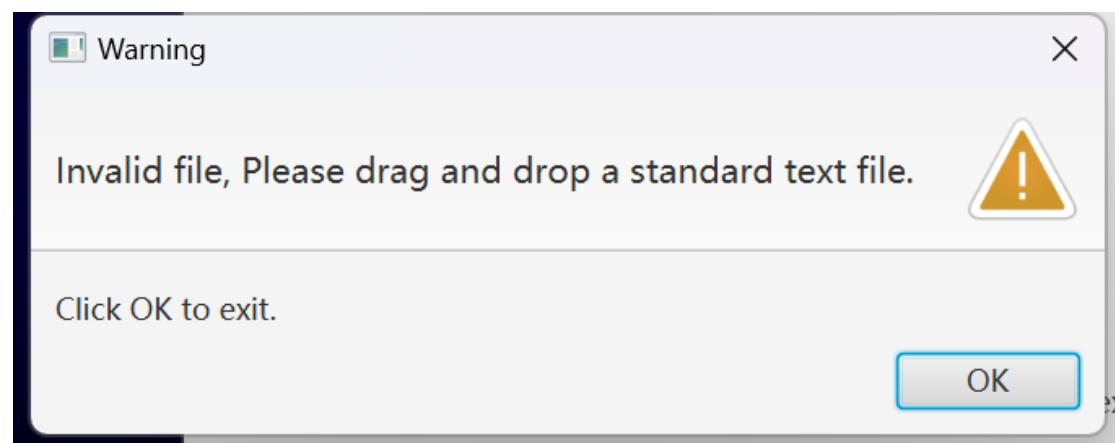
mutable array List. Here, FileReader and BufferedReader will be used to read the data, and then it will be processed according to the specified data format. The first line is the name of the Sankey diagram, the second line is the name of the left rectangle, and the rest are entities that need to be encapsulated. I encapsulate them one by one into DataNode as shown above.

## Chapter 6 - Exception Handling

The report will show the approaches and techniques for handling and preventing potential exceptions that may arise during execution in this section.

There are two main exceptions that may occur in this program: one is caused by the quantity, and the other is caused by incorrect data format in the provided file. In this program, I have set a limit on the maximum value of data, which cannot exceed the maximum value of Long type, and all values must be greater than 0, otherwise it will jump out of the window, issue a warning and thus rejected the file

If the data format in this program is incorrect, such as lacking numbers or categories, or having duplicate keys, the program will also issue a window warning and terminate the entire program.



Here, a hashmap is used to detect duplicate keys and try-catch statements are used for exception detection. For more details, please refer to the code below.

## Chapter 7 - My Java Code

The final chapter showcases the Java code developed, demonstrating how it incorporates the concepts and functionalities discussed in the preceding chapters.

## Main:

```
package com.example.sankeyplot;
```

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.*;
```

```
public class Main extends Application {
    private SankeyPlot sankeyPlot;
    private double recRightUp = 0.15;
```

```
    @Override
```

```
    public void start(Stage primaryStage) {
```

```
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
```

```
        // Set drag and drop event listener
```

```
        root.setOnDragOver(event -> {
            if (event.getDragboard().hasFiles()) {
                event.acceptTransferModes(javafx.scene.input.TransferMode.COPY_OR_MOVE);
            }
            event.consume();
        });
```

```
        ArrayList<String> list = new ArrayList<>();
```

```
        // Handle drag and drop events
```

```
        root.setOnDragDropped(event -> {
            var db = event.getDragboard();
```

```

        boolean success = false;
        list.clear();
        if (db.hasFiles()) {
            success = true;
            for (File file : db.getFiles()) {
                try {
                    BufferedReader reader = new BufferedReader(new FileReader(file));
                    String line;
                    while ((line = reader.readLine()) != null) {
                        list.add(line);
                    }
                    reader.close();
                    if (!checkFileValidity(list)) {
                        showAlert("Invalid file, Please drag and drop a standard text file.");
                        event.consume();
                        return;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            // Create and display the next scene
        }
        openNextScene(primaryStage, list);
        event.setDropCompleted(success);
        event.consume();
    });
    Scene scene = new Scene(root, 600, 600);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Sankey Plot");
    Label label = new Label("Please drag and drop a standard text here.");
    root.getChildren().add(label);
    primaryStage.show();
}

private void openNextScene(Stage primaryStage, ArrayList<String> list) {
    VBox root = new VBox();
    // Set drag and drop event listener
    root.setOnDragOver(event -> {
        if (event.getDragboard().hasFiles()) {
            event.acceptTransferModes(TransferMode.COPY_OR_MOVE);
        }
    });
}

```

```

        event.consume();
    });

    ArrayList<String> list1 = new ArrayList<>();
    // Handle drag and drop events
    root.setOnDragDropped(event -> {
        var db = event.getDragboard();
        boolean success = false;
        list1.clear();
        if (db.hasFiles()) {
            success = true;

            for (File file : db.getFiles()) {
                try {
                    BufferedReader reader = new BufferedReader(new FileReader(file));
                    String line;
                    while ((line = reader.readLine()) != null) {
                        list1.add(line);
                    }
                    reader.close();
                    if (!checkFileValidity(list1)) {
                        showAlert("Invalid file, Please drag and drop a standard text file.");
                        event.consume();
                        return;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            // Reset the recRightUp
            recRightUp = 0.15;
            // Create and display the next scene
            openNextScene(primaryStage, list1);
        }
        event.setDropCompleted(success);
        event.consume();
    });

    Rectangle rectangle1 = new Rectangle();
    Pane pane = new Pane(rectangle1);
    StackPane stackPane = new StackPane();
    stackPane.getChildren().addAll(pane, root);
    Scene scene = new Scene(stackPane, 600, 600);
    primaryStage.setScene(scene);
    primaryStage.show();

```

```

sankeyPlot = new SankeyPlot(scene, pane, rectangle1);
sankeyPlot.ReadData(list);
sankeyPlot.plotSankey();

}

public static void main(String[] args) {
    launch(args);
}

public boolean checkFileValidity(List<String> list) {
    boolean check = true;
    Map<String, Long> map = new HashMap<>();
    for (int i = 0; i < list.size(); i++) {
        switch (i) {
            case 0:
                break;
            case 1:
                break;
            default:
                String[] split = list.get(i).split(" ");
                Long test = 0L;
                try {
                    test = Long.parseLong(split[split.length - 1]);
                } catch (Exception e) {
                    return false;
                }
                if (test <= 0L) {
                    return false;
                }
                if (split.length == 1) {
                    return false;
                }
                } else if (split.length == 2) {
                    try {
                        var value = map.put(split[0], Long.parseLong(split[1]));
                        if (value != null) {
                            return false;
                        }
                    }
                    break;
                } catch (Exception e) {

```



```

        return false;

    }

}

String key = "";
for (int i1 = 0; i1 < split.length - 1; i1++) {
    key += " " + split[i1];
}
try {
    map.put(key, Long.parseLong(split[split.length - 1]));
    break;
} catch (Exception e) {
    return false;

}

}

System.out.println(list.get(i));
}

return check;
}

public void showAlert(String headerText) {
    Alert alert = new Alert(Alert.AlertType.WARNING);
    alert.setTitle("Warning");
    alert.setHeaderText(headerText);
    alert.setContentText("Click OK to exit the application.");

    ButtonType okButton = new ButtonType("OK");
    alert.getButtonTypes().setAll(okButton);
    alert.setOnCloseRequest(event -> {
        if (alert.getResult() != null && alert.getResult().equals(okButton)) {
            alert.close();
        }
    });
    alert.show();
}

}

```

## SankeyPlot:

```
package com.example.sankeyplot;
```

```

import com.example.sankeyplot.PlotItem.Impl.SankeyPlotItemImpl;
import com.example.sankeyplot.model.DataNode;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.beans.binding.Bindings;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.scene.text.Text;
import javafx.util.Duration;

import java.util.*;
import java.util.stream.Collectors;

public class SankeyPlot {
    private List<DataNode> nodeList = new ArrayList<>();
    private Scene scene;
    private Pane pane;
    private Rectangle rectangle1;
    private String chartName = "";
    private String chartTotalName = "";
    private Long total;

    private Map<String, String> map = new HashMap<>();

    public SankeyPlot(List<DataNode> nodeList, Scene scene, Pane pane, Rectangle rectangle1, String chartName,
String chartTotalName, Long total, Map<String, String> map) {
        this.nodeList = nodeList;
        this.scene = scene;
        this.pane = pane;
        this.rectangle1 = rectangle1;
        this.chartName = chartName;
        this.chartTotalName = chartTotalName;
        this.total = total;
        this.map = map;
    }

    public SankeyPlot(Scene scene, Pane pane, Rectangle rectangle1) {

```

```

this.scene = scene;
this.pane = pane;
this.rectangle1 = rectangle1;
}

public void plotSankey() {
    //plot the whole rectangle
    plotRectangle1();
    double rectangleX = 18.0 / 20;
    ArrayList<Rectangle> rectangleList = new ArrayList<>();
    double recRightUp = 0.15;
    double rightUpY2 = 0.15;
    double leftUpY1 = 0.25;
    ArrayList<Text> textList = new ArrayList<>();

    //plot the rectangles and curves
    for (DataNode node : nodeList) {

        SankeyPlotItemImpl sankeyPlotItem = createSankeyPlotItem(node);
        sankeyPlotItem.plotRectangles(scene, rectangleList, recRightUp, rectangleX);
        recRightUp += (0.4 / nodeList.size()) * 0.5 + node.getSize() * 0.5;
        double rightDownY2 = rightUpY2 + (0.5 * node.getSize());
        double leftDownY1 = leftUpY1 + node.getSize() * 0.5;
        sankeyPlotItem.plotCurve(leftUpY1, rightUpY2, rightDownY2, leftDownY1, rectangleX);
        leftUpY1 += (node.getSize()) * 0.5;
        rightUpY2 += (0.4 / nodeList.size()) * 0.5 + node.getSize() * 0.5;
        sankeyPlotItem.plotText(textList);
        sankeyPlotItem.plotAnimation();
    }

    //plot the text
    plotTheText(textList);
    pane.getChildren().addAll(textList);
    pane.getChildren().addAll(rectangleList);
}

private void plotTheText(ArrayList<Text> textList) {
    Text text = new Text(chartTotalName + ": " + total);
    text.xProperty().bind(Bindings.add(rectangle1.xProperty(), rectangle1.widthProperty()));
    text.yProperty().bind(Bindings.add(rectangle1.yProperty(), rectangle1.heightProperty().multiply(0.5)));

    //Set the name of the chart

```

```

        Text text1 = new Text(chartName);
        text1.xProperty().bind(Bindings.subtract(scene.widthProperty(),
text1.getLayoutBounds().getWidth()).divide(2));
        text1.yProperty().bind(Bindings.multiply(scene.heightProperty(), 19.0 / 20));
        textList.add(text);
        textList.add(text1);
    }

    private void plotRectangle1() {

        double rectangle1Width = 1.0 / 30;
        rectangle1.widthProperty().bind(Bindings.multiply(scene.widthProperty(), rectangle1Width));
        double rectangle1Height = 1.0 / 2;
        rectangle1.heightProperty().bind(Bindings.multiply(scene.heightProperty(), rectangle1Height));
        // Set the position of the rectangle
        //The x-axis of the left rectangle is 1/20 of the width of the scene
        double rectangle1X = 1.0 / 20;
        rectangle1.xProperty().bind(Bindings.multiply(scene.widthProperty(), rectangle1X));
        rectangle1.yProperty().bind(Bindings.subtract(
            Bindings.divide(scene.heightProperty(), 2),
            rectangle1.heightProperty().divide(2)
        ));
        rectangle1.setFill(Color.TRANSPARENT);
        Duration duration = Duration.seconds(0.5);
        Color startColor = Color.TRANSPARENT;
        Color endColor = Color.CYAN;
        Timeline timeline = new Timeline(
            new KeyFrame(Duration.ZERO, new KeyValue(rectangle1.fillProperty(), startColor)),
            new KeyFrame(duration, new KeyValue(rectangle1.fillProperty(), endColor))
        );
        timeline.setCycleCount(1); // Only play once
        timeline.play();
    }

    private SankeyPlotItemImpl createSankeyPlotItem(DataNode node) {
        Rectangle rectangle = new Rectangle();
        Path filledPath = new Path();
        return new SankeyPlotItemImpl(node, rectangle, scene, pane, rectangle1, filledPath);
    }

    public void ReadData(ArrayList<String> list) {
        Map<String, Long> map = new HashMap<>();
        int count = 0;

```

```

for (int i = 0; i < list.size(); i++) {
    switch (i) {
        case 0:
            chartName = list.get(i);
            break;
        case 1:
            chartTotalName = list.get(i);
            break;
        default:
            String[] split = list.get(i).split(" ");
            if (split.length == 1) {
                break;
            } else if (split.length == 2) {
                map.put(split[0], Long.parseLong(split[1]));
                break;
            }
            String key = "";
            for (int i1 = 0; i1 < split.length - 1; i1++) {
                key += " " + split[i1];
            }
            map.put(key, Long.parseLong(split[split.length - 1]));
            break;
    }
    System.out.println(list.get(i));
    count++;
}

total = map.values().stream().mapToLong(Long::longValue).sum();

Map<String, Long> sortedMap = sortByValue(map);
Set<String> keys = sortedMap.keySet();
int i = 0;
double startHue = 0.0;
double endHue = 360;
double hueStep = (endHue - startHue) / keys.size();

for (String key : keys) {
    Long amount = map.get(key);
    double saturation = 1.0;
    double brightness = 1.0;
    nodeList.add(new DataNode(key, amount, ((double) amount / total), Color.hsb(startHue + i * hueStep,
saturation, brightness)));
    i++;
}

```

```

    }
}

public static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map) {
    return map.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByValue())
        .collect(Collectors.toMap(
            Map.Entry::getKey,
            Map.Entry::getValue,
            (oldValue, newValue) -> oldValue,
            LinkedHashMap::new
        ));
}

}

}

```

## DataNode:

```

package com.example.sankeyplot.model;
import javafx.scene.paint.Color;
public class DataNode {
    private String name;
    private Long value;
    private double size;
    private Color color;
    public DataNode() {
    }

    public DataNode(String name, Long value, double size, Color color) {
        this.name = name;
        this.value = value;
        this.size = size;
        this.color = color;
    }

    public String getName() {
        return name;
    }
}

```

```

    }

    public void setName(String name) {
        this.name = name;
    }

    public Long getValue() {
        return value;
    }

    public void setValue(Long value) {
        this.value = value;
    }

    public double getSize() {
        return size;
    }

    public void setSize(double size) {
        this.size = size;
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public String toString() {
        return "Node{name = " + name + ", value = " + value + ", size = " + size + ", color = " + color + "}";
    }
}

```

### SankeyPlotItem:

```

package com.example.sankeyplot.PlotItem;
import javafx.scene.Scene;

```

```

import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import java.util.ArrayList;
public interface SankeyPlotItem {
    void plotRectangles(Scene scene, ArrayList<Rectangle> rectangleList, double recRightUp, double rectangleX);

    void plotCurve(double leftUpY1, double rightUpY2, double rightDownY2, double leftDownY1, double
rectangleX);

    void plotText(ArrayList<Text> textList);

    void plotAnimation();
}

```

### SankeyPlotItemImpl:

```

package com.example.sankeyplot.PlotItem.Impl;
import com.example.sankeyplot.PlotItem.SankeyPlotItem;
import com.example.sankeyplot.model.DataNode;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.PauseTransition;
import javafx.animation.Timeline;
import javafx.beans.binding.Bindings;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.*;
import javafx.scene.text.Text;
import javafx.util.Duration;
import java.util.ArrayList;

public class SankeyPlotItemImpl implements SankeyPlotItem {
    private DataNode node;
    private Rectangle rectangle;
    private Scene scene;
    private Pane pane;

```



```

private Rectangle rectangle1;
private Path filledPath;

public SankeyPlotItemImpl(DataNode node, Rectangle rectangle, Scene scene, Pane pane,
Rectangle rectangle1, Path filledPath) {
    this.node = node;
    this.rectangle = rectangle;
    this.scene = scene;
    this.pane = pane;
    this.rectangle1 = rectangle1;
    this.filledPath = filledPath;
}

public void plotRectangles(Scene scene, ArrayList<Rectangle> rectangleList, double recRightUp,
double rectangleX) {

    //The width of the rectangle
    double rectangleWidth = 1.0 / 30;
    //Set Binding
    rectangle.widthProperty().bind(Bindings.multiply(scene.widthProperty(), rectangleWidth));
    rectangle.heightProperty().bind(Bindings.multiply(rectangle1.heightProperty(),
node.getSize()));
    rectangle.setStroke(Color.TRANSPARENT);
    rectangle.setFill(Color.TRANSPARENT);
    //The x-axis of the rectangle
    rectangle.xProperty().bind(Bindings.multiply(scene.widthProperty(), rectangleX));
    rectangle.yProperty().bind(Bindings.multiply(scene.heightProperty(), recRightUp));
    //Set the color of the rectangle and set the animation
    Color startColor = Color.TRANSPARENT;
    Color endColor = node.getColor();
    Timeline timeline = new Timeline(
        new KeyFrame(Duration.ZERO, new KeyValue(rectangle.fillProperty(), startColor)),
        new KeyFrame(Duration.seconds(0.5), new KeyValue(rectangle.fillProperty(), endColor))
    );
    timeline.setCycleCount(1);
    PauseTransition delay = new PauseTransition(Duration.seconds(2.5)); // 2 秒延迟

    // Disable the timeline when the delay finishes
    delay.setOnFinished(event -> timeline.play());

    //Activate the delay
    delay.play();
}

```

```

//The move ratio of the rectangle

rectangleList.add(rectangle);

}

public void plotCurve( double leftUpY1, double rightUpY2, double rightDownY2, double
leftDonwY1, double rectangleX) {
    CubicCurveTo cubicCurveTo1 = new CubicCurveTo();
    double controlX1 = 0.45;
    cubicCurveTo1.controlX1Property().bind(Bindings.multiply(scene.widthProperty(),
controlX1));
    cubicCurveTo1.controlY1Property().bind(Bindings.multiply(scene.heightProperty(),
leftUpY1));
    double controlX2 = 0.55;
    cubicCurveTo1.controlX2Property().bind(Bindings.multiply(scene.widthProperty(),
controlX2));
    cubicCurveTo1.controlY2Property().bind(Bindings.multiply(scene.heightProperty(),
rightUpY2));
    cubicCurveTo1.xProperty().bind(Bindings.multiply(scene.widthProperty(), rectangleX));
    cubicCurveTo1.yProperty().bind(Bindings.multiply(scene.heightProperty(), rightUpY2));
    CubicCurveTo cubicCurveTo2 = new CubicCurveTo();
    cubicCurveTo2.controlX1Property().bind(Bindings.multiply(scene.widthProperty(),
controlX2));
    cubicCurveTo2.controlY1Property().bind(Bindings.multiply(scene.heightProperty(),
rightDownY2));
    cubicCurveTo2.controlX2Property().bind(Bindings.multiply(scene.widthProperty(),
controlX1));
    cubicCurveTo2.controlY2Property().bind(Bindings.multiply(scene.heightProperty(),
leftDonwY1));
    cubicCurveTo2.xProperty().bind(Bindings.add(rectangle1.xProperty(),
rectangle1.widthProperty()));
    cubicCurveTo2.yProperty().bind(Bindings.multiply(scene.heightProperty(), leftDonwY1));
    MoveTo moveTo = new MoveTo();
    moveTo.xProperty().bind(Bindings.add(rectangle1.xProperty(), rectangle1.widthProperty()));
    moveTo.yProperty().bind(Bindings.multiply(scene.heightProperty(), leftUpY1));
    filledPath.getElements().add(moveTo);
    filledPath.getElements().add(cubicCurveTo1);
    LineTo lineTo = new LineTo();
    lineTo.xProperty().bind(rectangle.xProperty());
    lineTo.yProperty().bind(Bindings.multiply(scene.heightProperty(), rightDownY2));
    filledPath.getElements().add(lineTo);
    filledPath.getElements().add(cubicCurveTo2);
    filledPath.getElements().add(new ClosePath());
    // Set the stroke color to transparent

```

```

filledPath.setStroke(Color.TRANSPARENT);
// Get the hue value of the certain color and change the transparency
Color hsbCyan = Color.CYAN.deriveColor(0, 1, 1, 0.5);
Color hsbNodeColor = node.getColor().deriveColor(0, 1, 1, 0.5);

//create a linear gradient
LinearGradient linearGradient = new LinearGradient(
    0, 0, 1, 0, true, CycleMethod.NO_CYCLE,
    new Stop(0, hsbCyan),
    new Stop(1, hsbNodeColor)
);
filledPath.setFill(linearGradient);
pane.getChildren().add(filledPath);
}

public void plotText(ArrayList<Text> textList){
    Text text = new Text(node.getName() + ": " + node.getValue());
    //The text is placed near the rectangle
    text.yProperty().bind(Bindings.add(rectangle.yProperty(),
rectangle.heightProperty().multiply(0.5)));
    text.xProperty().bind(Bindings.subtract(rectangle.xProperty(),
text.getLayoutBounds().getWidth()));
    textList.add(text);
}

public void plotAnimation() {

    // Create a Timeline animation to change the clipRect's width from 0 to scene's width

    //set clip with the original width 0
    Rectangle clipRect = new Rectangle(0, 0, 0, scene.getHeight());
    filledPath.setClip(clipRect);

    // Create a Timeline animation to change the clipRect's width from 0 to scene's width
    double animationDuration = 3.0;
    Timeline clipTimeline = new Timeline(
        new KeyFrame(Duration.ZERO, new KeyValue(clipRect.widthProperty(), 0)),
        new KeyFrame(Duration.seconds(animationDuration), new
    KeyValue(clipRect.widthProperty(), scene.getWidth()))
    );
    //Listener to change the clipRect's width and height when the window's width and height
    change
    scene.widthProperty().addListener((observable, oldValue, newValue) -> {
        clipRect.setWidth(newValue.doubleValue()); // 根据新的窗口宽度调整裁剪矩形的宽度
    });
}

```

```
scene.heightProperty().addListener((observable, oldValue, newValue) -> {  
    clipRect.setHeight(newValue.doubleValue()); // 根据新的窗口宽度调整裁剪矩形的宽度  
});  
  
clipTimeline.play();  
  
}  
  
}
```