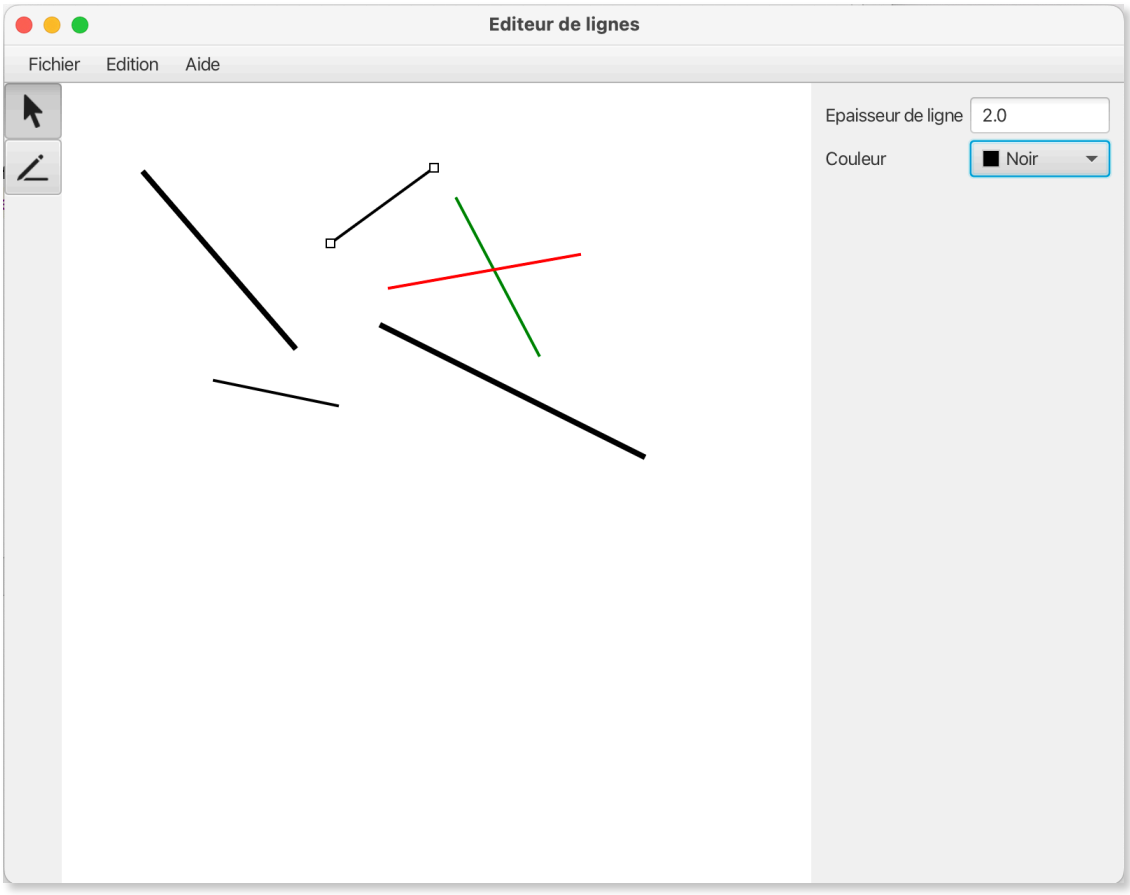
2023

ARCHITECTURE DES INTERFACES HUMAIN-MACHINE

JAVAFX

P. STUDER
ENSISA

UNE APPLICATION DE DESSIN VECTORIEL



1. INTRODUCTION

SUJET

L'application à créer est une application de dessin vectoriel à manipulation directe. L'utilisateur pourra dessiner des segments de droite à l'écran, les déplacer, les supprimer et modifier leurs propriétés.

2. CRÉATION DU PROJET

✍ Dans IntelliJ, créez un nouveau projet JavaFX nommé **lines**, group : **ensisa**

Exécutez l'application pour vérifier que le template ne comporte pas d'erreurs. S'il y a des erreurs liées à JavaFX, essayez de modifier la version de JavaFX comme indiqué dans l'étape suivante :

✍ Editez **pom.xml**

Changez la version de JavaFX en 21 (ou +) :

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>21</version>
</dependency>
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>21</version>
</dependency>
```

Clic-droit sur **pom.xml** > **Maven** > **Reload project**

✍ Renommez la classe HelloApplication en LinesApplication en utilisant la fonction de renommage d'IntelliJ

✍ Dans **resources.ensisa.lines**, créez un fichier fxml nommé **main-view.fxml**

Changez le nom du contrôleur en : ensisa.lines.MainController

✍ Dans **java.ensisa.lines**, créez la classe ensisa.lines.MainController

✍ Dans la méthode start de LinesApplication, remplacez le nom du fichier fxml à charger et modifiez quelques propriétés :

```
public void start(Stage stage) throws IOException {
    FXMLLoader fxmlLoader = new FXMLLoader(
        LinesApplication.class.getResource("main-view.fxml"));
    Scene scene = new Scene(fxmlLoader.load(), 800, 600);
    stage.setTitle("Editeur de lignes");
    stage.setScene(scene);
    stage.show();
}
```

✍ Supprimez les fichiers **hello-view.fxml** et **HelloController.java**

✍ Lancez l'application pour vérifier qu'il n'y a pas d'erreurs.

3. COMPOSITION DE LA FENÊTRE PRINCIPALE

La fenêtre principale de l'application comporte une barre de menus déroulants en haut, une palette d'outils à gauche, l'éditeur de lignes au centre et un inspecteur de propriétés à droite. Le composant conteneur approprié est un **BorderPane**.

✍ Ouvrez **main-view.fxml** dans Scene Builder

✍ Supprimez le composant **AnchorPane**

✍ Glissez un composant **BorderPane**

✍ Comme la racine de la scène a été détruite, associez la classe ensisa.lines.MainController à **BorderPane** (dans la section "Controller" en bas à droite de la fenêtre de Scene Builder)

4. MENUS DÉROULANTS

✍ Glissez un composant **MenuBar** dans la partie haute de **BorderPane**

Des menus sont déjà présents.

✍ Dans la classe MainController, ajoutez une méthode pour traiter l'appui sur un article de menu "Quit"

```
public class MainController {

    @FXML
    private void quitMenuAction() {
        Platform.exit();
    }
}
```

✍ Dans Scene Builder, sélectionnez le menu **File**, renommez-le en **Fichier**, sélectionnez l'article de menu **Close**, renommez-le en **Quitter**, spécifiez un raccourci clavier et associez-le au traitant d'événement quitMenuAction

✍ Lancez l'application et vérifiez que l'application s'arrête lorsque l'article de menu **Quitter** est sélectionné.

5. LE MODÈLE

Avant de poursuivre la construction de l'IHM, il faut définir le modèle, au sens MVC, de l'application. Nous définissons un modèle actif qui pourra être observé.

Pour modéliser une ligne, nous définissons une classe nommée StraightLine (*StraightLine* pour ne pas provoquer de confusion avec la classe *Line* définie par JavaFX).

✍ Dans **ensisa.lines**, créez un paquetage **model**

✍ Dans **ensisa.lines.model**, créez une classe StraightLine. Une ligne est caractérisée par les coordonnées des points à ses extrémités, une épaisseur et une couleur. La couleur est considérée comme une information du modèle et non un composant de l'IHM.

```
public class StraightLine {
   private final DoubleProperty startX;
   private final DoubleProperty startY;
   private final DoubleProperty endX;
   private final DoubleProperty endY;
   private final DoubleProperty strokeWidth;
   private final ObjectProperty<Color> color;

   public StraightLine() {
      startX = new SimpleDoubleProperty(0);
      startY = new SimpleDoubleProperty(0);
      endX = new SimpleDoubleProperty(0);
      endY = new SimpleDoubleProperty(0);
      strokeWidth = new SimpleDoubleProperty(2.0);
      color = new SimpleObjectProperty<>(Color.BLACK);
   }

}
```

✍ A l'aide d'Intellij, créez les accesseurs et les mutateurs correspondant aux propriétés.

```
public double getStartX() {
   return startX.get();
}

public DoubleProperty startXProperty() {
   return startX;
}

public void setStartX(double startX) {
   this.startX.set(startX);
}
```

```
public double getStartY() {
    return startY.get();
}

public DoubleProperty startYProperty() {
    return startY;
}

public void setStartY(double startY) {
    this.startY.set(startY);
}

public double getEndX() {
    return endX.get();
}

public DoubleProperty endXProperty() {
    return endX;
}

public void setEndX(double endX) {
    this.endX.set(endX);
}

public double getEndY() {
    return endY.get();
}

public DoubleProperty endYProperty() {
    return endY;
}

public void setEndY(double endY) {
    this.endY.set(endY);
}

public double getStrokeWidth() {
```

```
    return strokeWidth.get();
}

public DoubleProperty strokeWidthProperty() {
    return strokeWidth;
}

public void setStrokeWidth(double strokeWidth) {
    this.strokeWidth.set(strokeWidth);
}

public Color getColor() {
    return color.get();
}

public ObjectProperty<Color> colorProperty() {
    return color;
}

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

Pour modéliser le document qui représente l'ensemble des tracés réalisés par l'utilisateur, nous créons une classe … Document. Un document définit un état et un comportement indépendant de toute interface utilisateur. Typiquement, le contenu d'un document est sauvegardé dans un fichier et relu à partir de celui-ci.

Le document comporte une collection de StraightLine. Or si le contenu de cette collection change parce que l'utilisateur crée ou détruit une ligne, il faut notifier ce changement. Nous utilisons une collection observable de JavaFX.

✍ Dans **ensisa.lines.model**, créez une classe Document :

```
public class Document {
    private final ObservableList<StraightLine> lines;
```

```
    public Document() {
        lines = FXCollections.observableArrayList();
    }

    public ObservableList<StraightLine> getLines() {
        return lines;
    }
}
```

✍ Instanciez le document dans le contrôleur MainController :

```
public class MainController {
    private final Document document;

    public MainController() {
        document = new Document();
    }

    public Document getDocument() { return document; }
```

6. LA VUE

Il faut dessiner à l'écran les lignes qui figurent dans le modèle. Deux possibilités s'offrent à nous :

- dessiner les lignes à l'aide d'opérations définies dans un contexte de dessin
- instancier des composants qui représentent des formes géométriques.

C'est la deuxième possibilité que nous mettons en œuvre pour simplifier (un peu) le codage.

Pour pouvoir positionner des composants enfants en absolu dans un conteneur, nous utilisons un composant **Pane**.

✍ Ajoutez une variable d'instance annotée dans MainController pour référencer le composant **Pane** :

```
@FXML
public Pane editorPane;
```

✍ Dans le fxml, placez un composant **Pane** dans la partie centrale du **BorderPane** avec **fx:id** à editorPane

✍ Dans **Style**, sélectionnez le style CSS **-fx-background-color** et indiquez la valeur **white**

Pour dessiner une ligne, nous utilisons un composant Line de JavaFX qu'il faudra insérer dans editorPane. Il faudra également mémoriser les associations entre une ligne du modèle, instance de la classe StraightLine et cette même ligne représentée à l'écran, instance de la classe Line. Pour réduire la taille du contrôleur, nous définissons une classe liée à editorPane nommée LinesEditor qui gérera ces associations. Nous définissons aussi des liaisons de données entre les coordonnées des extrémités de la ligne JavaFX et les propriétés de StraightLine.

✍ Créez une classe LinesEditor avec la table d'association :

```
public class LinesEditor {
   private final Pane editorPane;
   private final Map<StraightLine, Line> lines;

   public LinesEditor(Pane editorPane) {
      this.editorPane = editorPane;
      lines = new HashMap<>();
   }

}
```

✍ Ecrivez une méthode pour lier les propriétés d'un ligne JavaFX aux propriétés d'une ligne du modèle :

```
private void bind(Line line, StraightLine straightLine) {
   line.startXProperty().bind(straightLine.startXProperty());
```

```
        line.startYProperty().bind(straightLine.startYProperty());
        line.endXProperty().bind(straightLine.endXProperty());
        line.endYProperty().bind(straightLine.endYProperty());
        line.strokeWidthProperty().bind(straightLine.strokeWidthProperty());
        line.strokeProperty().bind(straightLine.colorProperty());
    }
```

✍ Ecrivez une méthode pour ajouter une ligne nouvellement créée à la vue :

```
    public void createLine(StraightLine straightLine) {
        Line line = new Line();
        lines.put(straightLine, line);
        bind(line, straightLine);
        editorPane.getChildren().add(line);
    }
```

✍ Ecrivez une méthode qui retire une ligne de la vue :

```
    public void removeLine(StraightLine straightLine) {
        Line line = lines.remove(straightLine);
        editorPane.getChildren().remove(line);
    }
```

L'utilisateur pourra librement ajouter ou supprimer des lignes ; ces opérations modifient le modèle. Il faut donc observer le modèle pour insérer ou retirer des composants enfants d'editorPane selon les opérations réalisées par l'utilisateur.

✍ Modifiez MainController :

```
    public class MainController {
        private final Document document;
        private LinesEditor linesEditor;
        @FXML
        public Pane editorPane;

        public MainController() {
            document = new Document();
```

```
    }

    public void initialize() {
        linesEditor = new LinesEditor(editorPane);
        observeDocument();
    }

    private void observeDocument() {
        document.getLines().addListener(new ListChangeListener<StraightLine>() {
            public void onChanged(ListChangeListener.
                Change<? extends StraightLine> c) {
                while (c.next()) {
                    // Des lignes ont été supprimées du modèle
                    for (StraightLine line : c.getRemoved()) {
                        linesEditor.removeLine(line);
                    }
                    // Des lignes ont été ajoutées au modèle
                    for (StraightLine line : c.getAddedSubList()) {
                        linesEditor.createLine(line);
                    }
                }
            }
        });
    }

public LinesEditor getLinesEditor() {
    return linesEditor;
}
```

✍ Pour tester l'application, créez par code une ligne dans la méthode initialize() de MainController. Lancez l'application, une ligne doit apparaitre à l'écran.

```
public void initialize() {
    linesEditor = new LinesEditor(editorPane);
    observeDocument();

    StraightLine l = new StraightLine();
```

```
        l.setStartX(10);
        l.setStartY(20);
        l.setEndX(300);
        l.setEndY(60);
        document.getLines().add(l);
    }
```

7. OUTIL DE DESSIN

L'utilisateur utilise un outil de dessin pour créer une nouvelle ligne. Cet outil a besoin de détecter l'appui sur le bouton gauche de la souris lorsque le pointeur est dans l'éditeur (événement **Mouse Pressed**), le déplacement du pointeur bouton enfoncé (événement **Mouse Dragged**), le relâchement du bouton de la souris (événement **Mouse Released**).

L'application comportera plusieurs outils. Nous définissons une interface Tool pour modéliser le comportement général de ces outils.

7.1. GÉNÉRALISATION DES OUTILS

✍ Créez un paquetage **tools** dans **ensisa.lines**

✍ Créez une interface Tool dans ce paquetage :

```
    public interface Tool {

        default void mousePressed(MouseEvent event) {}
        default void mouseDragged(MouseEvent event) {}
        default void mouseReleased(MouseEvent event) {}
    }
```

7.2. CRÉATION DE L'OUTIL DE DESSIN

✍ Créez une classe qui modélise l'outil de dessin :

```
    public class DrawTool implements Tool {
        public MainController mainController;
```

```
    public DrawTool(MainController controller) {
        this.mainController = controller;
    }
}
```

✍ Dans la classe MainController, déclarez une propriété qui mémorise l'outil courant :

```
private final ObjectProperty<Tool> currentTool;

public MainController() {
    document = new Document();
    currentTool = new SimpleObjectProperty<>(new DrawTool(this));
}

public ObjectProperty<Tool> currentToolProperty() {
    return currentTool;
}

public Tool getCurrentTool() {
    return currentTool.get();
}

public void setCurrentTool(Tool currentTool) {
    this.currentTool.set(currentTool);
}
```

Il faut déléguer le traitement des événements souris émis par editorPane à l'outil courant.

✍ Dans la classe MainController, déclarez les traitants d'événements :

```
@FXML
private void mousePressedInEditor(MouseEvent event) {
    getCurrentTool().mousePressed(event);
}

@FXML
private void mouseDraggedInEditor(MouseEvent event) {
```

```
    getCurrentTool().mouseDragged(event);
}

@FXML
private void mouseReleasedInEditor(MouseEvent event) {
    getCurrentTool().mouseReleased(event);
}
```

✍ Dans **main-view.fxml**, liez les événements émis par editorPane aux méthodes de traitement.

Il reste à écrire le traitement effectué par l'outil de dessin.

Le principe est le suivant :

- quand l'utilisateur appuie sur le bouton gauche de la souris, nous ajoutons une nouvelle ligne au modèle
- quand l'utilisateur déplace le pointeur bouton enfoncé, nous faisons suivre le pointeur par la seconde extrémité de la ligne
- quand l'utilisateur relâche le bouton, l'outil revient dans son état initial

✍ Ecrivez le code de la classe DrawTool :

```
public class DrawTool implements Tool {
    enum State { initial, drawing }
    private State state;
    private StraightLine currentLine;
    public MainController mainController;

    public DrawTool(MainController controller) {
        state = State.initial;
        this.mainController = controller;
    }

    @Override
    public void mousePressed(MouseEvent event) {
```

```
            if (event.isPrimaryButtonDown()) {
                currentLine = new StraightLine();
                currentLine.setStartX(event.getX());
                currentLine.setStartY(event.getY());
                currentLine.setEndX(event.getX());
                currentLine.setEndY(event.getY());
                mainController.getDocument().getLines().add(currentLine);
                state = State.drawing;
            }
        }

        @Override
        public void mouseDragged(MouseEvent event) {
            if (state == State.drawing && event.isPrimaryButtonDown()) {
                currentLine.setEndX(event.getX());
                currentLine.setEndY(event.getY());
            }
        }

        @Override
        public void mouseReleased(MouseEvent event) {
            state = State.initial;
        }
    }
```

✍ Lancez l'application. Remarquez que les lignes peuvent se dessiner hors des limites du panneau de dessin. Par défaut, le composant Pane ne contraint pas le dessin. Nous allons contraindre le dessin aux limites de la zone de dessin.

✍ Dans la classe MainController :

```
    public void initialize() {
        linesEditor = new LinesEditor(editorPane);
        setClipping();
        observeDocument();
    }
```

```
private void setClipping() {
    final Rectangle clip = new Rectangle();
    editorPane.setClip(clip);

    editorPane.layoutBoundsProperty().addListener((v, oldValue, newValue) -> {
        clip.setWidth(newValue.getWidth());
        clip.setHeight(newValue.getHeight());
    });
}
```

7.3. MODIFICATION DE L'APPARENCE DU POINTEUR

✍ Dans l'interface Tool, déclarez des traitants des événements d'entrée et de sortie de la zone de dessin

```
default void mouseEntered(MouseEvent event) {}
default void mouseExited(MouseEvent event) {}
```

✍ Dans la classe MainController, déclarez les traitants d'événements correspondants :

```
@FXML
private void mouseEntered(MouseEvent event) {
    getCurrentTool().mouseEntered(event);
}

@FXML
void mouseExited(MouseEvent event) {
    getCurrentTool().mouseExited(event);
}
```

✍ Dans **main-view.fxml**, liez ces méthodes aux événements

✍ Ecrivez les traitants dans la classe DrawTool :

```
@Override
```

```
    public void mouseEntered(MouseEvent event) {
        mainController.editorPane.setCursor(Cursor.CROSSHAIR);
    }

    @Override
    public void mouseExited(MouseEvent event) {
        mainController.editorPane.setCursor(Cursor.DEFAULT);
    }
```

✍ Lancez l'application

8. PALETTE D'OUTILS

L'application comporte deux outils : un outil de sélection et un outil de dessin. Un seul outil peut être actif à la fois. Ces outils sont regroupés dans une palette d'outils à gauche de la zone de dessin.

8.1. VUE

✍ Dans **resources.ensisa.lines**, créez un dossier **icons**

✍ Dans **icons**, copiez les images **cursorarrow.png** et **pencil.line.png**

✍ Dans la classe MainController, ajoutez des variables d'instance pour référencer deux boutons radio :

```
@FXML
private RadioButton selectToolButton;
@FXML
private RadioButton drawToolButton;

public void initialize() {
   linesEditor = new LinesEditor(editorPane);
   setClipping();
   initializeToolPalette();
   observeDocument();
}

private void initializeToolPalette() {
   // Change style class to not paint the round button
   selectToolButton.getStyleClass().remove("radio-button");
   selectToolButton.getStyleClass().add("toggle-button");
   drawToolButton.getStyleClass().remove("radio-button");
   drawToolButton.getStyleClass().add("toggle-button");
}
```

✍ Dans la classe MainController, ajoutez des traitants d'événement pour les boutons radio :

```
@FXML
private void selectToolAction() {
}

@FXML
private void drawToolAction() {
}
```

✍ Editez **main-view.fxml** dans Scene Builder

✍ Glissez un **VBox** dans la partie gauche de **BorderPane**

✍ Glissez un **RadioButton** dans le **Vbox** avec **fx:id** à **selectToolButton**, **Pref Width** et **Pref Height** à 40, **Selected** coché, **Content Display** à **GRAPHIC_ONLY**, **Toggle Group** à **toolGroup**, **On Action** à **selectToolAction**

✍ Glissez un composant **ImageView** dans le **RadioButton** avec **Image** à **icons/cursorarrow.png**, **Fit Width** et **Fit Height** à **24**

✍ De même, glissez un **RadioButton** dans le **Vbox** avec **fx:id** à **drawToolButton**, **Pref Width** et **Pref Height** à 40, **Content Display** à **GRAPHIC_ONLY**, **Toggle Group** à **toolGroup**, **On Action** à **drawToolAction**

✍ Glissez un composant **ImageView** dans le **RadioButton** avec **Image** à **icons/pencil.line.png**, **Fit Width** et **Fit Height** à **24**

✍ Pour la **VBox**, mettez **Pref Width** à **USE_COMPUTED_SIZE**, **Pref Height** à **USE_COMPUTED_SIZE**

8.2.EBAUCHE DE L'OUTIL DE SÉLECTION

✍ Dans **ensisa.lines.tools**, créez une classe SelectTool :

```
public class SelectTool implements Tool {
    private MainController mainController;
```

```
    public SelectTool(MainController controller) {
      this.mainController = controller;
    }
}
```

✍ Dans MainController, créez et mémorisez les outils :

```
private final ObjectProperty<Tool> currentTool;
private final DrawTool drawTool;
private final SelectTool selectTool;

public MainController() {
    document = new Document();
    selectTool = new SelectTool(this);
    drawTool = new DrawTool(this);
    currentTool = new SimpleObjectProperty<>(selectTool);
}
```

8.3. TRAITEMENT DU CHANGEMENT D'OUTIL

```
@FXML
private void selectToolAction() {
    setCurrentTool(selectTool);
}

@FXML
private void drawToolAction() {
    setCurrentTool(drawTool);
}
```

9. OUTIL DE SÉLECTION

L'utilisateur peut sélectionner plusieurs lignes à la fois. Nous mémorisons les lignes sélectionnées dans un ensemble. Une ligne sélectionné sera dessinée différemment pour donner de la rétroaction à l'utilisateur : nous dessinons des petits carrés aux

extrémités de la ligne sélectionnée. La sélection est une information du contrôleur et non du modèle.

✍ Dans MainController, déclarez une variable qui mémorise l'ensemble des lignes sélectionnées :

```
private final ObservableSet<StraightLine> selectedLines;

public MainController() {
   document = new Document();
   selectTool = new SelectTool(this);
   drawTool = new DrawTool(this);
   currentTool = new SimpleObjectProperty<>(selectTool);
   selectedLines = FXCollections.observableSet();
}

public  ObservableSet<StraightLine> getSelectedLines() {
   return selectedLines;
}
```

✍ Ajoutez une méthode pour sélectionner une ligne :

```
public void selectLine(StraightLine line, boolean keepSelection) {
   if (!keepSelection)
      getSelectedLines().clear();
   getSelectedLines().add(line);
}
```

✍ Ajoutez une méthode pour dé-sélectionner une ligne :

```
public void deselectLine(StraightLine line) {
   getSelectedLines().remove(line);
}
```

✍ Ajoutez une méthode pour tout dé-sélectionner :

```
public void deselectAll() {
```

```
        getSelectedLines().clear();
    }
```

✍ Modifiez le traitant d'événement de modification du modèle pour synchroniser la sélection :

```
    private void observeDocument() {
        document.getLines().addListener(new ListChangeListener<StraightLine>() {
            public void onChanged(ListChangeListener.
                Change<? extends StraightLine> c) {
                while (c.next()) {
                    for (StraightLine line : c.getRemoved()) {
                        deselectLine(line);
                        linesEditor.removeLine(line);
                    }
                    for (StraightLine line : c.getAddedSubList()) {
                        linesEditor.createLine(line);
                    }
                }
            }
        });
    }
```

9.1. MISE EN ÉVIDENCE DE LA SÉLECTION

Nous modifions LinesEditor pour dessiner les carrés en cas de sélection de lignes. Pour dessiner les carrés, nous instancions la classe Rectangle de JavaFX et ajoutons les instances au panneau qui représente la zone de dessin. Lorsque la sélection disparait, nous retirons ces instances du panneau.

✍ Dans LinesEditor, ajoutez des méthodes pour dessiner ou non les carrés de sélection :

```
    private final Map<StraightLine, Rectangle> startSelectionSquares;
    private final Map<StraightLine, Rectangle> endSelectionSquares;
```

```
    private final static int selectionSquareWidth = 6;

    public LinesEditor(Pane editorPane) {
        this.editorPane = editorPane;
        lines = new HashMap<>();
        startSelectionSquares = new HashMap<>();
        endSelectionSquares = new HashMap<>();
    }

    // Crée un carré de sélection
    private Rectangle createSelectionSquare() {
        var square = new Rectangle();
        square.setWidth(selectionSquareWidth);
        square.setHeight(selectionSquareWidth);
        square.setFill(Color.WHITE);
        square.setStroke(Color.BLACK);
        return square;
    }

    // Lie le centre des carrés de sélection aux extrémités de la ligne
    private void bind(Rectangle startSelectionSquare, Rectangle endSelectionSquare,
    StraightLine straightLine) {
        startSelectionSquare.xProperty().bind(straightLine.startXProperty().
                subtract(selectionSquareWidth / 2));
        startSelectionSquare.yProperty().bind(straightLine.startYProperty().
                subtract(selectionSquareWidth / 2));
        endSelectionSquare.xProperty().bind(straightLine.endXProperty().
                subtract(selectionSquareWidth / 2));
        endSelectionSquare.yProperty().bind(straightLine.endYProperty().
                subtract(selectionSquareWidth / 2));
    }

    public void selectLine(StraightLine straightLine) {
        var startSelectionSquare = createSelectionSquare();
        var endSelectionSquare = createSelectionSquare();
        startSelectionSquares.put(straightLine, startSelectionSquare);
        endSelectionSquares.put(straightLine, endSelectionSquare);
        bind(startSelectionSquare, endSelectionSquare, straightLine);
```

```
editorPane.getChildren().add(startSelectionSquare);
editorPane.getChildren().add(endSelectionSquare);
}

public void deselectLine(StraightLine straightLine) {
   var selectionSquare = startSelectionSquares.get(straightLine);
   editorPane.getChildren().remove(selectionSquare);
   selectionSquare = endSelectionSquares.get(straightLine);
   editorPane.getChildren().remove(selectionSquare);
}
```

9.2. SÉLECTION SIMPLE

Pour commencer, nous ne traitons que la sélection simple. Lorsque l'utilisateur clique près d'une ligne, cette ligne est sélectionnée, les autres sont dé-sélectionnées. S'il ne clique sur aucune ligne, la sélection est vide.

Il nous faut écrire une méthode qui teste si un point est dans l'espace occupé par une ligne en tenant d'une marge d'erreur de quelques pixels.

✍ Dans la classe LinesEditor, écrivez ces méthodes :

```
public boolean isPointInLine(double x, double y,
   StraightLine straightLine
){
   return squaredDistanceToSegment(x, y, straightLine.getStartX(),
      straightLine.getStartY(), straightLine.getEndX(), straightLine.getEndY()) < 16;
}

/**
 * return the squared distance between a point and
 * a segment
 * p point for which the squared distance is
 * evaluated
 * ps start of segment
 * pe end of segment
 */
```

```
public static double squaredDistanceToSegment(double x, double y, double
startX, double startY, double endX, double endY) {
    if (startX == endX && startY == endY) return squaredDistance(startX, startY, x, y);

    double sx = endX - startX;
    double sy = endY - startY;

    double ux = x - startX;
    double uy = y - startY;

    double dp = sx * ux + sy * uy;
    if (dp < 0) return squaredDistance(startX, startY, x, y);

    double sn2 = sx * sx + sy * sy;
    if (sn2 <= dp) return squaredDistance(endX, endY, x, y);

    double b = dp / sn2;
    return squaredDistance(startX + b * sx, startY + b * sy, x, y);
}

/**
 * return the squared distance between two points
 */
public static double squaredDistance(double p1X, double p1Y, double p2X,
double p2Y) {
    return (p2X - p1X) * (p2X - p1X) + (p2Y - p1Y) * (p2Y - p1Y);
}
```

✍ Ecrivez des méthodes pour déterminer si un point est dans un carré de sélection :

```
public boolean isPointInStartSelectionSquare(double x, double y, StraightLine
straightLine) {
    var selectionSquare = startSelectionSquares.get(straightLine);
    if (selectionSquare != null) {
        return selectionSquare.contains(x, y);
    }
    return false;
}
```

```
public boolean isPointInEndSelectionSquare(double x, double y, StraightLine
straightLine) {
   var selectionSquare = endSelectionSquares.get(straightLine);
   if (selectionSquare != null) {
      return selectionSquare.contains(x, y);
   }
   return false;
}
```

✍ Dans la classe MainController, écrivez une méthode pour rechercher la ligne qui contient un point parmi l'ensemble des lignes du document :

```
public StraightLine findLineForPoint(double x, double y) {
   for (var straightLine : getDocument().getLines()) {
      if (linesEditor.isPointInStartSelectionSquare(x, y, straightLine) ||
            linesEditor.isPointInEndSelectionSquare(x, y, straightLine) ||
            linesEditor.isPointInLine(x, y, straightLine))
         return straightLine;
   }
   return null;
}
```

✍ Dans la classe SelectTool, définissez un type énuméré pour indiquer l'état de l'outil et une variable pour mémoriser la ligne sur laquelle l'utilisateur a éventuellement cliqué:

```
enum State { initial, selection }
private State state;
private MainController mainController;
private StraightLine straightLine;

public SelectTool(MainController controller) {
   this.mainController = controller;
   state = State.initial;
}
```

✍ La sélection est traitée lorsque l'utilisateur appuie sur le bouton gauche de la souris :

```
@Override
public void mousePressed(MouseEvent event) {
  if (event.isPrimaryButtonDown()) {
      state = State.selection;
      straightLine = mainController.findLineForPoint(event.getX(), event.getY());
      if (straightLine != null) {
         mainController.selectLine(straightLine, false);
      } else {
         mainController.deselectAll();
      }
   }
}
```

✍ L'outil de sélection revient dans l'état initial lorsque l'utilisateur relâche le bouton de la souris :

```
@Override
public void mouseReleased(MouseEvent event) {
   state = State.initial;
}
```

✍ Dans la classe MainController, écrivez un écouteur d'événements liés à la modification de la sélection pour mettre à jour la vue :

```
public void initialize() {
   linesEditor = new LinesEditor(editorPane);
   setClipping();
   initializeToolPalette();
   observeDocument();
   observeSelection();
}

private void observeSelection() {
   selectedLines.addListener(new SetChangeListener<StraightLine>() {
      @Override
      public void onChanged(Change<? extends StraightLine> change) {
         if (change.wasRemoved()) {
            linesEditor.deselectLine(change.getElementRemoved());
```

```
            }
            if (change.wasAdded()) {
                linesEditor.selectLine(change.getElementAdded());
            }
        }
    });
}
```

✍ Lancez l'application. Une ligne peut être sélectionnée à la fois. La sélection disparait si l'utilisateur ne clique pas sur une ligne.

9.3.SÉLECTION MULTIPLE

Le principe de la sélection multiple est le suivant :

- lorsque l'utilisateur place le pointeur de la souris sur une ligne non sélectionnée et appuie uniquement sur le bouton gauche, la ligne est sélectionnée et toutes les autres sont désélectionnées
- lorsque l'utilisateur place le pointeur de la souris sur une ligne sélectionnée et appuie uniquement sur le bouton gauche, la sélection ne change pas
- lorsque l'utilisateur place le pointeur de la souris sur une ligne non sélectionnée et appuie sur le bouton gauche tout en maintenant la touche SHIFT enfoncée, la ligne est sélectionnée et toutes celles qui étaient sélectionnées le restent
- lorsque l'utilisateur place le pointeur de la souris sur une ligne sélectionnée et appuie sur le bouton gauche tout en maintenant la touche SHIFT enfoncée, la ligne est désélectionnée et toutes celles qui étaient sélectionnées le restent
- lorsque l'utilisateur ne place pas le pointeur de la souris sur une ligne et appuie uniquement sur le bouton gauche, les lignes précédemment sélectionnées ne le sont plus
- lorsque l'utilisateur ne place pas le pointeur de la souris sur une ligne et appuie sur le bouton gauche tout en maintenant la touche SHIFT enfoncée, la sélection ne change pas

✍ Modifiez la méthode mousePressed de SelectTool pour mettre en œuvre la sélection multiple

```
public void mousePressed(MouseEvent event) {
  if (event.isPrimaryButtonDown()) {
    state = State.selection;
    straightLine = mainController.findLineForPoint(event.getX(), event.getY());
    if (straightLine != null) {
      var isSelected = mainController.getSelectedLines().contains(straightLine);
      if (!event.isShiftDown()) {
        if (!isSelected)
          mainController.selectLine(straightLine, false);
      } else if (isSelected)
        mainController.deselectLine(straightLine);
      else
        mainController.selectLine(straightLine, true);
    } else {
      if (!event.isShiftDown())
        mainController.deselectAll();
    }
  }
}
```

✍ Lancez l'application et vérifiez le comportement de la sélection

9.4. DÉPLACEMENT DES LIGNES

Dans la plupart des logiciels de dessin, l'outil de sélection permet aussi le déplacement des éléments sélectionnés.

Nous allons intégrer le déplacement à l'outil selon le principe suivant :

- suite à l'opération de sélection, tant que l'utilisateur laisse le bouton gauche de la souris enfoncé, les lignes sélectionnées suivent de façon relative les mouvements du pointeur de la souris

- lorsque l'utilisateur relâche le bouton gauche, les lignes sélectionnées restent à leurs nouvelles positions

✍ Dans la classe StraightLine, ajoutez une méthode offset qui ajoute un déplacement aux extrémités de la ligne :

```
public void offset(double dx, double dy) {
    setStartX(getStartX() + dx);
    setStartY(getStartY() + dy);
    setEndX(getEndX() + dx);
    setEndY(getEndY() + dy);
}
```

✍ Dans la classe SelectTool, ajoutez des variables d'instance nommée lastX et lastY. Je vous laisse le soin de penser à leur utilité.

```
enum State { initial, selection }
private State state;
private MainController mainController;
private double lastX;
private double lastY;
```

✍ Initialisez-les dans la méthode mousePressed :

```
public void mousePressed(MouseEvent event) {
    if (event.isPrimaryButtonDown()) {
        state = State.selection;
        lastX = event.getX();
        lastY = event.getY();
        straightLine = mainController.findLineForPoint(event.getX(), event.getY());
```

✍ Ecrivez la méthode mouseDragged qui modifie les coordonnées des lignes sélectionnées relativement aux mouvements de la souris

```
@Override
public void mouseDragged(MouseEvent event) {
```

```
        if (event.isPrimaryButtonDown()) {
            double deltaX = event.getX() - lastX;
            double deltaY = event.getY() - lastY;
            switch (state) {
                case selection:
                    for (var line : mainController.getSelectedLines()) {
                        line.offset(deltaX, deltaY);
                    }
                    break;
            }
            lastX = event.getX();
            lastY = event.getY();
        }
    }
```

✍ Lancez l'application

9.5. MODIFICATION DES LIGNES

L'utilisateur peut placer le pointeur de la souris dans un carré de sélection, appuyer sur le bouton gauche et déplacer la souris pour modifier les coordonnées de l'extrémité de la ligne correspondante. La sélection ne change pas si l'utilisateur agit sur un carré de sélection.

✍ Dans la classe SelectTool :

```
    enum State { initial, selection, selectionAtStart, selectionAtEnd }

    public void mousePressed(MouseEvent event) {
        if (event.isPrimaryButtonDown()) {
            state = State.selection;
            lastX = event.getX();
            lastY = event.getY();
            straightLine = mainController.findLineForPoint(event.getX(), event.getY());
            if (straightLine != null) {
```

```
var isSelected = mainController.getSelectedLines().contains(straightLine);
if (isSelected && mainController.getLinesEditor().
   isPointInStartSelectionSquare(event.getX(), event.getY(), straightLine)) {
   state = State.selectionAtStart;
} else if (isSelected && mainController.getLinesEditor().
    isPointInEndSelectionSquare(event.getX(), event.getY(), straightLine)) {
   state = State.selectionAtEnd;
} else {
   if (!event.isShiftDown()) {
      if (!isSelected)
         mainController.selectLine(straightLine, false);
   } else if (isSelected)
      mainController.deselectLine(straightLine);
   else
      mainController.selectLine(straightLine, true);
}
```

✍ Modifiez mouseDragged :

```
switch (state) {
   case selection:
      for (var line : mainController.getSelectedLines()) {
         line.offset(deltaX, deltaY);
      }
      break;
      case selectionAtStart: {
         straightLine.setStartX(straightLine.getStartX() + deltaX);
         straightLine.setStartY(straightLine.getStartY() + deltaY);
      }
      break;
      case selectionAtEnd: {
         straightLine.setEndX(straightLine.getEndX() + deltaX);
         straightLine.setEndY(straightLine.getEndY() + deltaY);
      }
      break;
}
```

✍ Lancez l'application