Università degli Studi di Messina

Dipartimento di Scienze matematiche e informatiche,

scienze fisiche e scienze della terra

Corso di Laurea Triennale in Informatica

Progetto di

Programmazione 2

Realizzato da

Giovanni Domenico Tripodi

Anno Accademico

2021/2022

1. Introduzione

Il presente documento mira ad illustrare la progettazione di un video player per la navigazione e conferma/declino del pointing, sviluppato mediante linguaggio Java. Per la realizzazione del progetto è stata adottata una strutturazione **OOP**, cioè orientata agli oggetti. Tra gli elementi OOP più importanti implementati nel progetto vi sono **modularità, subtyping, ereditarietà, polimorfismo.**

2. Componenti principali

L'applicazione utilizza come libreria grafica JavaFx. E' utilizzato, inoltre JavaFX FXML, ovvero un formato XML che consente di comporre le User-interface in maniera molto simile a come si compongono sul web.
FXML consente di separare il codice del layout dal resto del codice dell'applicazione, consentendo così di avere un codice più pulito.

1.1. Player-View.fxml

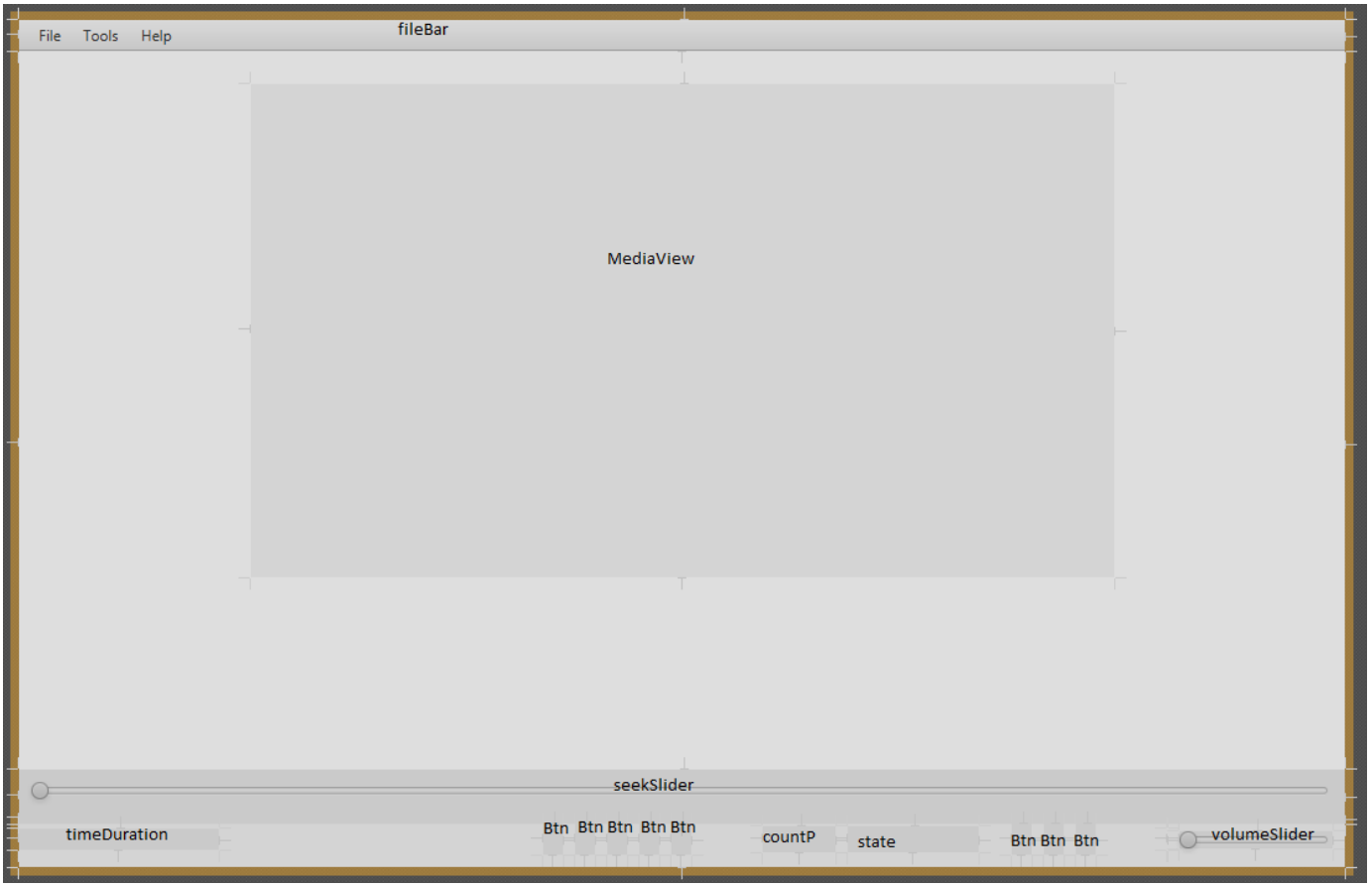
Il file player-view.fxml possiede ogni elemento del layout della user-interface:

Piccola porzione del file player-view.fxml

```
<AnchorPane prefHeight="687.0" prefWidth="1075.0"
xmlns="http://javafx.com/javafx/19"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.pointingplayer.Controller">
  <children>
     <BorderPane layoutX="3.0" maxHeight="-Infinity"
maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
prefHeight="400.0" prefWidth="600.0" AnchorPane.bottomAnchor="0.0"
AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0"
AnchorPane.topAnchor="0.0">
        <bottom>
           <VBox alignment="CENTER" prefHeight="40.0"
prefWidth="600.0" BorderPane.alignment="CENTER">
              <children>
                 <Slider fx:id="seek Slider">
                    <VBox.margin>
                       <Insets />
                    </VBox.margin>
                    <padding>
                       <Insets bottom="20.0" left="10.0"
right="10.0" top="10.0"
```

```
                </padding>
            </Slider>
```

Graficamente, la user-interface avrà questa struttura:



Ogni elemento della user-interface è rappresentato come un oggetto, pertanto avremo dei metodi che ci permettono di manipolare l'elemento. Possiamo definire degli eventi con cui è possibile far comunicare l'utente con l'elemento specifico (es. evento sul click del bottone x).

1.2. Application.java

E' la main class del player, effettua il loading del file player-fxml.
Estende la classe astratta Application, manipola l'oggetto stage, esegue l'override del metodo *start* (usato per caricare il file fxml e settare weight ed height iniziale) e implementa il metodo *handle* dell'interfaccia EventHandler di JavaFx (per gestire l'evento del doppio click sul player, impostando la modalità fullscreen).

```
public class Application extends javafx.application.Application {

   @Override
   public void start(Stage stage) throws Exception {

   FXMLLoader fxmlLoader = new
FXMLLoader(Application.class.getResource("player-view.fxml"));
   Scene scene = new Scene(fxmlLoader.load(), 1400, 800);
   stage.setTitle("Pointing Player");
```

```
scene.setOnMouseClicked(new EventHandler<MouseEvent>() {
   @Override
   public void handle(MouseEvent doubleClicked) {
       if(doubleClicked.getClickCount() == 2) {
           stage.setFullScreen(true);
       }
   }
});
```

Inoltre, viene definito il comportamento in merito all'evento sul click del bottone X per chiudere l'applicazione. In particolare, verrà richiamato il metodo del controller *exitVideo()*:

```
stage.addEventFilter(WindowEvent.WINDOW_CLOSE_REQUEST, event -> {
   Controller ctrl = fxmlLoader.getController();
   ctrl.exitVideo(new ActionEvent());
   event.consume();
});
```

1.3. Controller.java

La classe controller definisce al suo interno gli elementi della user-interface, inoltre definisce i metodi che saranno utilizzati in merito agli eventi sugli elementi della user-interface. Inoltre, implementa l'interfaccia Initializable, per poter utilizzare il metodo initialize.

Un elemento della user-interface verrà definito utilizzato l'annotation @FXML, creando un oggetto della classe dell'elemento.
Ad esempio, un bottone sarà definito così:

```
@FXML
private Button playButton;
```

Un evento, verrà definito sempre utilizzando l'annotation @FXML.

Ad esempio, l'evento in merito al click sul bottone di pausa sarà definito così:

```
@FXML
private void pauseVideo(ActionEvent event){
   mediaPlayer.pause();
}
```

Principi OOP

ASTRAZIONE

L'astrazione permette di esporre solamente le informazioni necessarie, celando quindi i dettagli implementativi.. Nascondendo i dettagli implementativi ed esponendo all'utente solo l'interfaccia richiesta, l'astrazione rende le applicazioni più semplici.

L'astrazione, in Java è implementata tramite la keyword **abstract**, applicabile sia alle classi che ai metodi.

Nel progetto, non è stato necessario creare una classe astratta da zero, bensì è stata utilizzata la classe astratta Application fornita da JavaFx.

```
public abstract class Application {}
```

Contiene al suo interno i metodi e gli attributi che servono per lanciare un'applicazione JavaFx, costituendo quindi l'entry point.

L'astrazione può anche essere implementata tramite l'utilizzo delle **interfacce** che, a differenza delle classi astratte che vanno ad astrarre oggetti troppo generici per poter essere istanziati, le interfacce vanno ad astrarre comportamenti che oggetti diversi potrebbero implementare. Spesso le interfacce hanno nomi che richiamano aggettivi e comportamenti (Comparable, Runnable, Cloneable).
Una classe che utilizza l'interfaccia, utilizzerà la keyword **implements** per indicare l'utilizzo dell'interfaccia.

Nel progetto, si utilizza l'**interfaccia generica** fornita da JavaFx EventHandler per gestire gli eventi sugli elementi della GUI. Si usa l'annotation FunctionalInterface per indicare che l'interfaccia è una interfaccia funzionale, ovvero una interfaccia con solo un metodo definito.

```
@FunctionalInterface
public interface EventHandler<T extends Event> extends EventListener {
   void handle(T var1);
}
```

Si utilizza il tipo Generics indicato con <>, ovvero tipi parametrizzati che servono per creare classi o interfacce in cui il tipo di dato sul quale si opera può essere specificato come parametro.

T extends Event significa che T può essere qualsiasi tipo che è sottoclasse di Event.

In questa maniera, si astrae dal tipo di dato specifico sul quale si opera, in quanto nella nostra applicazione possiamo avere diversi tipi di eventi, come ad esempio MouseEvent, KeyEvent, DragEvent e WindowEvent.

INCAPSULAMENTO

Un principio OOP importante oltre all'astrazione, è l'incapsulamento. L'incapsulamento è il meccanismo che permette di eseguire il wrap (ovvero avvolgere) le variabili e i metodi che agiscono sulle variabili in una singola unità.
Mentre l'astrazione viene utilizzata per esporre solo i dettagli rilevanti all'esterno, l'incapsulamento si occupa principalmente della sicurezza dei dati.
Avendo campi e metodi in un unico oggetto, risulta più facile introdurre l'**information hiding**, che fornisce un ulteriore strato di protezione all'interno del programma.

Un esempio di incapsulamento nel progetto lo abbiamo nella classe Pointing.

```
public class Pointing {

   private ArrayList<Frame> pointing = new ArrayList<>();

   public void setPointing(Path filePath) {

       String[] line;
       int num_line = 0;

       try (Reader reader = Files.newBufferedReader(filePath)) {
           CSVReader csvReader = new CSVReader(reader);
               while((line = csvReader.readNext()) != null) {
                   num_line++;
```

```
                    if (num_line == 1) {
                        line = csvReader.readNext();
                    }
                    pointing.add(new Frame(line[0], line[1], line[2]));
                }
            Alert alert = new Alert(Alert.AlertType.INFORMATION, "Time
series caricate correttamente!");
            alert.showAndWait();
        } catch (Exception e){
            Alert alert = new Alert(Alert.AlertType.WARNING, "Nessun file
csv trovato. Time series non disponibili!");
            alert.showAndWait();
        }
    }

    public String getBeginTime(int index){
        return pointing.get(index).getBeginTime();
    }

    public String getFinishTime(int index) {
        return pointing.get(index).getFinishTime();
    }

    public String getState(int index) {
        return pointing.get(index).getState();
    }

    public void setState(int index, String state) {
        pointing.get(index).setState(state);
    }

    public void modifyRow(int index, Frame element) {
        pointing.set(index, element);
    }

    public Frame getRow(int index) {
        return pointing.get(index);
    }

    public int getSize(){
        return pointing.size();
    }
```

INFORMATION HIDING

Agisce da separatore tra l'interfaccia della classe, che risulta visibile all'utente, e l'implementazione interna, che è resa protetta. In questa maniera, i dati presenti all'interno della classe possono essere modificati solo da metodi interni (della classe), e non da metodi esterni.
L'information hiding permette quindi di avere un codice più sicuro, evitando che un oggetto acceda a informazioni a cui non deve accedervi.
Per applicare la tecnica dell'information hiding, basta utilizzare il modificatore di accesso **private** per la dichiarazione di variabili (e/o metodi) all'interno di una classe.
I metodi e le variabili che sono dichiarati **private** sono accessibili solamente all'interno della classe e non dall'esterno.
Per accedere alle variabili di tipo private al di fuori della classe, si utilizzano i metodi public **getter** e **setter**.

Applicazione di information hiding nel progetto

Frame.java
Vengono definite le variabili private (quindi non possono essere accessibili dall'esterno della classe), mentre i metodi per accedervi sono pubblici (per permettere all'esterno della classe di accedere alle variabili).

```
public class Frame {

   private String beginTime;
   private String finishTime;
   private String state;

   public Frame(String beginTime, String finishTime, String state) {
       this.beginTime = beginTime;
       this.finishTime = finishTime;
       this.state = state;
   }


   public String getBeginTime() {
       return beginTime;
   }

   public void setBeginTime(String beginTime) {
       this.beginTime = beginTime;
   }

   public String getFinishTime() {
       return finishTime;
```

```
    }

    public void setFinishTime(String finishTime) {
        this.finishTime = finishTime;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

}
```

Controller.java

In questo caso, definiamo oggetti e metodi privati, per evitare che una classe esterna a Controller.java possa accedervi.

```
public class Controller implements Initializable {
    @FXML
    private MediaView mediaView;
    private MediaPlayer mediaPlayer;
    private String filePath;
    @FXML
    private Button openFile;
    @FXML
    private Button playButton;

    private boolean isPressed= false;
    private boolean isPaused = false;
    private boolean isSaved = true;
    private String pathCsv;

    public Pointing pointing;
    private String fileName;


    @FXML
    private void pauseVideo(ActionEvent event){ mediaPlayer.pause(); }
```

```
@FXML
private void playVideo(ActionEvent event){
    try{
        MediaPlayer.Status status = mediaPlayer.getStatus();

        if (status == MediaPlayer.Status.PLAYING){
            ///pause...
            mediaPlayer.pause();;
            setIconPlay();
        } else {
            mediaPlayer.play();
            setIconPause();
        }
    } catch (Exception ignored){}

}

@FXML
private void stopVideo(ActionEvent event){
    try{
        mediaPlayer.stop();
        setIconPlay();
        this.pointingCounter = -1;
        lbPointing.setText("0/" + (pointing.getSize()));
        lbState.setText("");
    } catch (Exception ignored){}

}
```

EREDITARIETA’

Con l’ereditarietà, una nuova classe viene creata acquisendo i membri di una classe esistente ed eventualmente estendendo con capacità nuove o modificate.
Permette di risparmiare tempo durante la programmazione, basando le nuove classi su altre già esistenti.
La classe già esistente si chiama superclass. Mentre la classe nuova si chiama subclass.
Con l’ereditarietà si definisce la relazione “is-a” tra due classi: un oggetto di una subclass può essere trattato come oggetto della sua superclass.

Nel progetto, il concetto di ereditarietà è contenuto nella classe *Application*

```
public class Application extends javafx.application.Application { }
```

La classe Application, è una subclass di javafx.application.Application, pertanto ne eredita variabili e metodi.

COMPOSIZIONE

La composizione definisce la relazione “has-a” tra classi. E’ una forma di associazione in cui una classe contiene un’altra classe e la classe contenuta dipende dalla classe contenente in maniera tale che non può esistere indipendentemente.
Facciamo in modo quindi che una classe utilizzi un’istanza di una classe direttamente, invece di ‘estenderla’ da un’altra classe come facciamo in caso di ereditarietà.

Pointing.java

```
public class Pointing {

private ArrayList<Frame> pointing = new ArrayList<>();
pointing.add(new Frame(line[0], line[1], line[2]));
CSVReader csvReader = new CSVReader(reader);
Alert alert = new Alert(Alert.AlertType.INFORMATION, "Time series
caricate correttamente!");
Alert alert = new Alert(Alert.AlertType.WARNING, "Nessun file csv
trovato. Time series non disponibili!");

}
```

Controller.java

```
public class Controller implements Initializable {
      FileChooser fileChooser = new FileChooser();
      Media media = new Media(filePath);
      mediaPlayer = new MediaPlayer(media);
      pointing = new Pointing();
      BigDecimal bd = new BigDecimal(seconds)
      mediaPlayer.seek(new Duration(seekSlider.getValue()));
}
```

POLIMORFISMO

E’ l’abilità di un oggetto di assumere varie forme, consente la scrittura di programmi che elaborano oggetti che condividono la stessa superclasse, come se fossero tutti oggetti della superclasse.

Mentre l'ereditarietà permette di acquisire attributi e metodi di un'altra classe, il polimorfismo è applicato per consentire ai metodi che la classe ha acquisito, di effettuare compiti differenti.

Ci sono tre tipi di polimorfismo:

- Ad-hoc polymorphism (method **overloading**)
- Inclusion polymorphism (method **overriding**)
- Parametric polymorphism (**generics**)

All'interno del progetto, vengono impiegati l'inclusion polymorphism e il parametric polymorphism.

Inclusion Polymorphism

Consiste nella possibilità che una sottoclasse A di una data classe B ridefinisca uno dei metodi della superclasse e che quindi quando viene utilizzata una istanza della classe A le invocazioni al metodo ridefinito (detto "overriden") eseguiranno il codice definito nella sottoclasse.

Nella classe *Frame*, si è effettuato l'override del metodo toString() della classe Object (ereditata automaticamente dalla classe)

```
@Override
public String toString() {
   return
(""+this.getBeginTime()+","+this.getFinishTime()+","+this.getState());
}
```

E' stato utilizzato anche nella classe Application, sottoclasse della classe astratta javafx.application.Application, per definire il metodo astratto start.

```
@Override
public void start(Stage stage) throws Exception {

   FXMLLoader fxmlLoader = new
FXMLLoader(Application.class.getResource("player-view.fxml"));
   Scene scene = new Scene(fxmlLoader.load(), 1400, 800);
   stage.setTitle("Pointing Player");
stage.setScene(scene);
stage.show();
}
```

Parametric polymorphism

I generics sono lo strumento fornito da Java per realizzare il polimorfismo parametrico. Essi permettono di utilizzare un parametro nella definizioni di metodi/classi/interfacce, il cui tipo viene determinato in fase di istanziazione.

Nella classe *Pointing* lo applichiamo per dichiarare un ArrayList di oggetti di tipo Frame.

```
private ArrayList<Frame> pointing = new ArrayList<>();
pointing.add(new Frame(line[0], line[1], line[2]));
```

Subtyping

Il subtyping riguarda l'implementazione di un'interfaccia e quindi la possibilità di sostituire diverse implementazioni di tale interfaccia in fase di esecuzione.
Nell'ereditarietà una subclass potrebbe non comportarsi come la superclass perchè una volta che ha ereditato i metodi e gli attributi, potrebbe cambiare totalmente il proprio comportamento. Inoltre NON può essere utilizzata dove la super class è prevista. La sub class in questo caso, è abilitata a comportarsi come la super class.

Con il subtyping invece, ci si assicura che la classe non abbandoni il comportamento previsto dall'interfaccia. La classe che implementa l'interfaccia è abilitata a cambiare diverse implementazioni di quell'interfaccia in fase di esecuzione, e può essere utilizzata ovunque sia richiesta l'interfaccia. La classe che implementa un'interfaccia è quindi un subtype dell'interfaccia.

Nel progetto, classi anonime implementano l'**interfaccia generica** fornita da JavaFx EventHandler per gestire gli eventi sugli elementi della GUI. Si usa l'annotation FunctionalInterface per indicare che l'interfaccia è una interfaccia funzionale. Ovvero una interfaccia con solo un metodo definito.

```
@FunctionalInterface
public interface EventHandler<T extends Event> extends EventListener {
   void handle(T var1);
}
```

Si utilizza il tipo Generics indicato con <>, ovvero tipi parametrizzati che servono per creare classi o interfacce in cui il tipo di dato sul quale si opera può essere specificato come parametro.

T extends Event significa che T può essere qualsiasi tipo che è sottoclasse di Event.
Ogni qual volta si dovrà gestire un evento, si eseguirà l'override del metodo handle.

In questa maniera, si astrae dal tipo di dato specifico sul quale si opera, in quanto nella nostra applicazione possiamo avere diversi tipi di eventi, come ad esempio MouseEvent, KeyEvent, DragEvent e WindowEvent.

Ad esempio, per gestire l'evento sul click sullo slider del video:

```
seekSlider.setOnMousePressed(new EventHandler<MouseEvent>() {
   @Override
   public void handle(MouseEvent mouseEvent) {
       mediaPlayer.seek(new Duration(seekSlider.getValue()));
   }
});
```

Si utilizza la parola chiave **new**, per creare una classe anonima, ovvero una classe locale senza nome definita e istanziata in un'unica espressione, in questo caso implementa l'interfaccia EventHandler, passando come parametro generico mouseEvent. Il tutto sarà passato come parametro al metodo setOnMousePressed, definito dalla classe Slider.

Nel progetto, la classe Controller implementa l'interfaccia **Initializable,** che permette di inizializzare un Controller.

```
public interface Initializable {
      public void initialize(URL location, ResourceBundle resources);
}
```

Il metodo initialize è chiamato per inizializzare un controller dopo che il relativo root element è stato completamente elaborato. Il parametro location, è usato per risolvere i percorsi relativi al root object, il parametro resources è usato per localizzare il root object.
La classe controller implementa l'interfaccia richiama al suo interno i metodi per settare le icone del player.

```
@Override
public void initialize(URL url, ResourceBundle resourceBundle) {
   setIconPlay();
   setIconStop();
   setIconPrevPointing();
   setIconNextPointing();
   setIconAccept();
   setIconDecline();
   setIconSave();
   setIconVolume();
   setIconRepeat();
```

```
}
```

```
public class Pointing {

   private ArrayList<Frame> pointing = new ArrayList<>();

   public void setPointing(Path filePath) {

       String[] line;
       int num_line = 0;

       try (Reader reader = Files.newBufferedReader(filePath)) {
           CSVReader csvReader = new CSVReader(reader);
               while((line = csvReader.readNext()) != null) {
                   num_line++;
                   if (num_line == 1) {
                       line = csvReader.readNext();
                   }
                   pointing.add(new Frame(line[0], line[1], line[2]));
               }
           Alert alert = new Alert(Alert.AlertType.INFORMATION, "Time
series caricate correttamente!");
           alert.showAndWait();
       } catch (Exception e){
           Alert alert = new Alert(Alert.AlertType.WARNING, "Nessun file
csv trovato. Time series non disponibili!");
           alert.showAndWait();
       }
   }

   public String getBeginTime(int index){
       return pointing.get(index).getBeginTime();
   }

   public String getFinishTime(int index) { return
pointing.get(index).getFinishTime(); }

   public String getState(int index) { return
pointing.get(index).getState(); }

   public void setState(int index, String state)
{pointing.get(index).setState(state); }
```

```
   public void modifyRow(int index, Frame element) {pointing.set(index,
element);}

   public Frame getRow(int index) { return pointing.get(index); }

   public int getSize(){
       return pointing.size();
   }


}
```