INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Interaction Homme-Machine

TP – IHM, 3INFO

1. Objectif

La finalité de ces 3 TP d'IHM sur JavaFX est la mise en pratique d'un certain nombre de concepts d'IHM que partagent une grande partie des boîtes à outils. Ces concepts sont : la séparation donnée – IHM, le patron d'architecture modèle-vue-contrôleur/presentateur (MVC / MVP), la description d'une IU dans un formalisme dédié et généralement graphique ; l'utilisation de widgets ; la programmation évènementielle.

L'application à développer est un éditeur de dessins vectoriel dans lequel il est possible d'ajouter, de supprimer et de déplacer des formes (rectangles et ellipses).

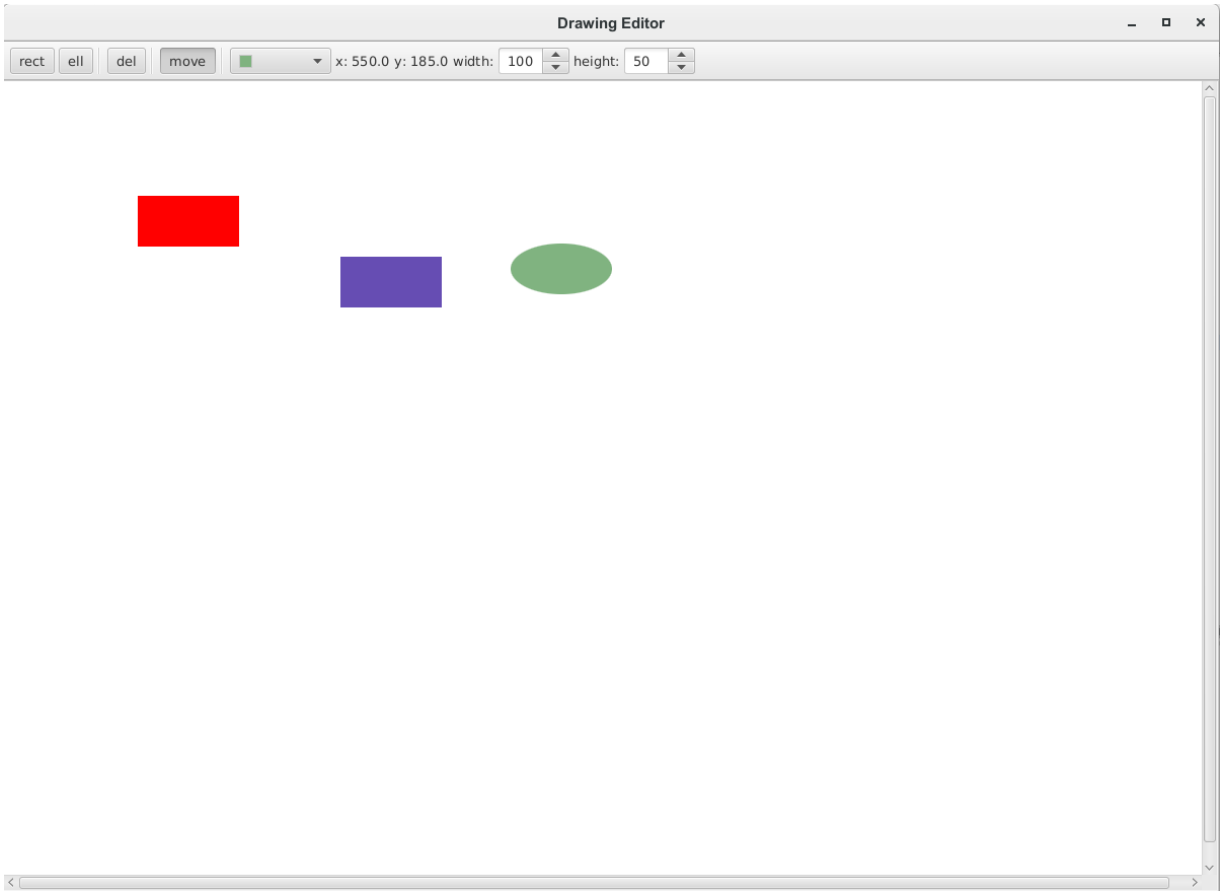


Figure 1: Interface utilisateur de l'application à développer avec JavaFX

Pensez à utiliser la doc JavaFX ainsi que le polycopié du cours d'IHM :

https://docs.oracle.com/javase/8/javase-clienttechnologies.htm

L'IU fonctionnera de la manière suivante : un clic sur le bouton à bascule (*toggle button*) *rect* sélectionne le mode de création de rectangles (*ell* pour les ellipses). Une fois sélectionnée une clic sur la zone de dessin ajoute la forme voulue à la position du clic. La couleur de la palette ainsi que la largeur et la hauteur indiquées dans les widgets seront utilisés pour personnaliser la forme à créer.

Le bouton à bascule *del* active le mode suppression de formes. Une fois activé, un clic sur une forme la supprime du dessin.

Le bouton à bascule *move* active le mode déplacement de formes. Une fois activé, un glisser-déposer (drag-and-drop) d'une forme permet de la déplacer.

L'utilisation de la palette de couleurs et des boutons fléchés (*spinners*) n'ont pas d'effet sur les formes déjà créées mais sur celle à venir.

Les labels *x* et *y* ne sont visibles que lors du déplacement d'un forme et indiquent la position de celle-ci.

Les ascenseurs (*Scroll bars*) permettent de déplacer la vue du dessin lorsque des formes sortent du champ de vision de l'interface graphique.

La création de cette application se fera pas à pas. Suivez donc bien chaque question.

Le code de l'application sera à rendre sous moodle.

2. Création du projet et des packages

Q1. Dans IntelliJ, créez un projet Java nommé TP-JavaFX-*vosNomsDeBinomes*. Assurez-vous bien que la JDK utilisée est la JDK d'oracle dans sa version 8 (*cf.* le poly du cours). En effet, JavaFX n'est pas encore fourni avec OpenJDK, le pendant open-source de la JDK d'Oracle. Assurez-vous également que le compilateur du projet est paramétré pour compiler du code Java 8.

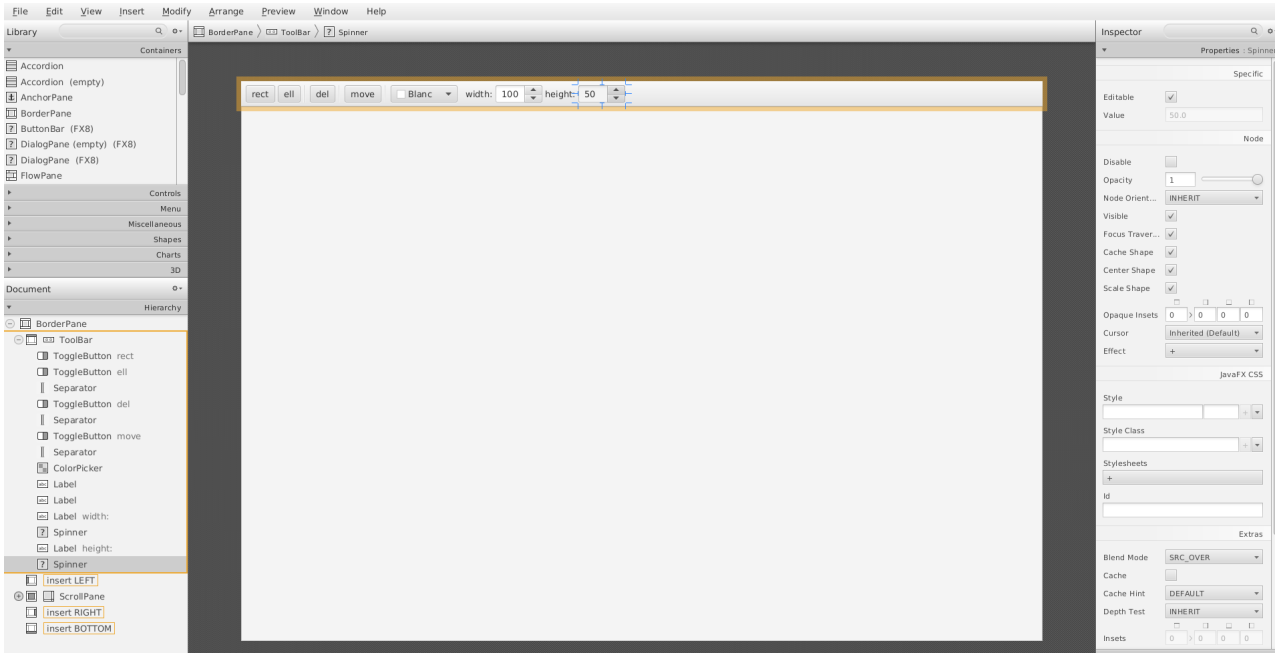
Q2. Créez un package *drawingEditor*. Puis créez ajoutez-y les sous-packages *model*, *view* et *controller*.

3. Création de l'interface graphique en FXML

En JavaFX, les interfaces graphiques sont décrites dans un formalisme XML particulier nommé FXML. Ce document permet de décrire à gros grain la structure de l'interface. Le comportement sera défini dans le code Java, en particulier dans les contrôleurs.

Q3. Créez un document FXML dans le package view (cf. le cours). Choisissez le layout *BorderPane* comme layout de la racine. Vous pouvez ouvrir ce document soit en mode XML, soit en mode 2D.

Q4. En utilisant SceneBuilder, créez l'interface graphique telle que visible dans la figure 1. Vous aurez besoin de *Toggle Buttons*, de *Spinners*, de *Labels*, de *Separator*, d'un *ColorPicker*, d'un *ScrollPane* qui contiendra un *Pane* (si SceneBuilder vous ajoute un *AnchorPane*, remplacer le par un *Pane*). Vous devrez obtenir quelque chose de cet ordre (certains labels ne sont pas visible car ils ne contiennent pas de texte pour l'instant) :



Q5. Toujours via SceneBuilder, donnez des identifiants à chaque widget (le champ *fx:id* se trouvant dans l'accordéon « Code » à droite en bas). Ces attributs permettront d'utiliser ces widgets dans le code Java. Cochez « *editable* » pour les *spinners*. Fermez SceneBuilder. Vous utiliserez désormais l'éditeur FXML pour modifier l'interface.

Q6. Pour tester l'interface, créez un classe principale *DrawingEditor* dans le package *drawingEditor* (adaptez pour cela le code de la diapo 35 du cours). Testez l'application en exécutant cette classe comme une application JavaFX.

Q7. En cliquant sur les boutons rect, ell, del et move, vous constatez que tous peuvent être cochés en même temps alors qu'un seul mode est possible à la fois. Il faut donc créer ce que l'on appelle un groupe.

Ouvrez le document FXML avec l'éditeur FXML, et ajoutez le code XML suivant juste après l'ouverture de la balise racine *BorderPane* :

```
<fx:define>
      <ToggleGroup fx:id="groupEditing" />
</fx:define>
```

La balise *ToggleGroup* définit un groupe de boutons à bascule du nom de *groupEditing*. Dans la balise de chaque widget concerné par ce groupe, ajoutez l'attribut suivant :

```
toggleGroup="$groupEditing"
```

Cela marque le widget comme faisant partie de ce groupe.

Testez le résultat.

Q8. Les spinners doivent permettent de changer la largeur et la hauteur des formes à créer. Actuellement vous pouvez constater que les spinners permet de saisir des valeurs négatives. Ouvrez le document FXML avec l'éditeur FXML et ajoutez les attributs *min*, *max*, *initialValue*, et *amountToStepBy* aux spinners (avec des valeurs du type double, exemples : 1.0 au lieu de 1). Testez le résultat.

Q9. Dernières modifications de l'interface. Définissez la taille du *ScrollPane* à 1200 x 800 à l'aide des attributs *prefWidth* et *prefHeight*.

4. Modèle

Q10. Téléchargez depuis moodle et placez les interfaces *Dessin* et *Forme* dans le package *model*. Implémentez l'interface *Forme* dans une classe abstraite *FormeImpl*. Cette classe devra avoir un constructeur prenant en paramètre un x, un y, une largeur, une hauteur (type *double*) et une couleur (du type *javafx.scene.paint.Paint* pour simplifier les choses plus tard, même s'il existe la classe *javafx.scene.paint.Color*). Les attributs de cette classe (correspondant aux paramètres du constructeur) seront des *Property* Java (*cf.* à partir de la diapo 56 du cours) : *DoubleProperty* et *ObjectProperty* pour la couleur. Ces *Properties* permettront plus tard de synchroniser la vue avec le modèle à l'aide du *data binding* (*cf.* à partir de la diapo 59).

Q10. Définissez deux classes concrètes héritant de *FormeImpl* : *Rect* et *Ell* (pour rectangle et ellipse). Ces classes définiront juste leur constructeur.

Q11. Implémentez l'interface *Dessin* dans une classe *DessinImpl*. Dans le constructeur, l'initialisation de la liste se fera en utilisant ce code :

```
FXCollections.observableArrayList();
```

Une *ObservableList* est une liste qui notifie ses observateurs en cas de modifications (ajouts, suppressions, *etc.*).

5. Contrôleur / Presentateur

Nous avons vu en cours que dans le patron d'architecture MVC le contrôleur reçoit les évènements produits par l'interface graphique, les traite pour réaliser des actions en conséquence. Le patron MVP (modèle-vue-présentateur) ressemble très fortement au MVC. La différence : dans MVC, la vue connaît son modèle et la synchronisation est réalisée par le modèle qui notifie sa vue. Dans MVP, la vue ne connaît pas son modèle car c'est le présentateur qui gère la synchronisation entre le modèle et la vue. Beaucoup de développeurs disent faire du MVC alors qu'ils font du MVP, ou un mixe des deux. C'est pourquoi certains frameworks disent qu'ils permettent de faire du MV*, i.e. où * signifie contrôleur, présentateur, ou modelview (e.g. AngularJS).

Dans ce TP, notre contrôleur sera en fait un mix entre contrôleur et présentateur, mais on parlera de contrôleur.

Q12. Créer une classe *ControleurDessin* dans le package *controller*. Dans le document FXML, ajoutez un attribut à la balise racine définissant le contrôleur (cf. diapo 82). Le principe : la vue FXML est chargée dans la classe principale, puis le contrôleur de ce FXML est automatiquement créé.

Q13. Faites que *ControleurDessin* implémente l'interface *Initializable*. L'opération implémentée de cette interface est automatiquement appelée juste après la création du contrôleur. Toute l'initialisation du contrôleur doit être réalisée dans cette opération et non dans son constructeur (que l'on ne définit généralement pas en JavaFX pour cette raison).

Q14. Ajoutez tous les widgets de l'interface graphique comme attribut du contrôleur. N'oubliez pas l'annotation *@FXML* et de bien utiliser les *fx:id* comme noms (*cf.* diapo 93). Testez dans l'opération *initialize* que ces attributs sont bien automatiquement initialisés sans votre intervention (c'est ce que l'on appelle de l'injection de dépendance). Pour les *spinners*, le type générique doit être *Double*.

Q15. Dans l'opération *initialize*, sélectionnez le bouton *rect* (pour choisir le mode création de rectangle par défaut, opération *setSelected*). Choisissez la couleur rouge (*javafx.scene.paint.Color.RED*) dans la palette (*setValue*). Masquez les labels *x* et *y* (*setVisible*). Définissez la couleur de fond du pane (qui affichera les formes) :

```
pane.setBackground(new Background(new BackgroundFill(Color.WHITE, null, null)));
```
Testez.

Q16. Ajoutez un attribut au contrôleur du type *Dessin*. Initialisez cet attribut dans l'opération *initialize*. Il s'agit du modèle.

Q17. Il existe plusieurs manières pour récupérer les évènements produits par les widgets et les traiter : la première est décrite dans le cours (cf. diapos 79 et 85) ; nous allons utiliser l'autre ici. Dans l'opération *initialize*, appeler l'opération *setOnMouseClicked* sur le widget *pane*. Cet appel requière un *EventHandler* en paramètre. Passez alors en paramètre une classe anonyme du type *EventHandler* :

```
pane.setOnMouseClicked(new EventHandler<MouseEvent>() {
      @Override
      public void handle(MouseEvent event) {
            // TODO Auto-generated method stub

      }
});
```
Ce traitement d'un clic sur le pane sera utilisé pour ajouter une forme ou en supprimer une. Ajoutez une condition dans l'opération *handle* testant si le bouton *rect* est sélectionné. Si c'est le cas créez un rectangle et ajoutez-le au dessin. Faites de même pour l'ellipse.

Q17. Les formes sont donc ajouter dans le modèle, mais leurs vues ne sont pas créées et affichées. Toujours dans *initialize*, ajoutez un écouteur (*listener*) à la liste de formes du modèle pour être notifié des ajouts et retraits faits dans cette liste :

```
  modele.getFormes().addListener(new ListChangeListener<Forme>() {
    @Override
    public void onChanged(javafx.collections.ListChangeListener.Change<? extends Forme> event) {
     if(event.next())
        if(event.wasAdded()) {
...
        } else if(event.wasRemoved()) {
...
        }
    }
  });
```
Oui, celui-là est compliqué. Utilisez et complétez le code ci-dessus (vous aurez besoin d'utiliser les opérations *getAddedSubList* et *getRemoved* de l'objet *event*). Vous devrez également créer une opération *private Shape createViewShapeFromShape(final Forme forme)* pour créer une vue JavaFX *Shape* à partir d'une *Forme*. Le principe : *javafx.scene.shape.Shape* sera la vue de *Forme*, *javafx.scene.shape.Rectangle*, celle de *Rect* et *javafx.scene.shape.Ellipse* celle de *Ell*. Ces *Shapes* seront ajoutés au / supprimés du *pane* : *pane.getChildren().add(shape)*. Utilisez la Javadoc pour trouver les opérations à appeler.

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Rectangle.html

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Ellipse.html

Q18. Dans la question précédente, vous avez certainement utiliser les paramètres des constructeurs de *Rectangle* / *Ellipse* ou bien les opérations *setCentreX*, *setX*, *etc.* pour définir les caractéristiques des vues. Vous allez maintenant utiliser le data binding pour faire cela et synchroniser automatiquement le modèle avec sa vue (cf. slide 93). Par exemple, pour synchroniser la position x d'une forme Rect avec la position de sa vue Rectangle :

```
recVue.xProperty().bind(forme.positionXProperty());
```
Faites de même pour tous les autres attributs des formes. N'y a-t-il pas un problème pour le binding entre *Ell* et *Ellipse* au niveau des radius ? La synchronisation entre ces valeurs n'est pas automatique car l'un

prend des rayons et l'autre des diamètres. Il faut donc convertir manuellement à l'aide d'un binding personnalisé à l'aide de Bindings.divide(DoubleProperty, double) comme vu en cours.

Faites de même pour *radiusY*. Testez alors l'ajout de rectangles et d'ellipses.

Q19. On veut maintenant pouvoir supprimer des formes lorsque l'on clique dessus et que le bouton *del* est activé. L'interface *Node* (dont héritent *Shape* et donc *Ellipse* et *Rectangle*) possède une opération très pratique : *setUserData*.

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html#setUserData-java.lang.Object-

Utilisez cette opération pour associer le modèle à sa vue lors de la création de la vue (opération *createViewShapeFromShape*).

Q20. Dans *createViewShapeFromShape*, appelez l'opération *setOnMouseClicked* sur la vue que vous avez créée afin de supprimer la forme visée du dessin. Attention, ne supprimez pas la vue mais la forme du modèle (en utilisant *getUserData* sur la vue pour récupérer la forme). Si tout se passe bien, la synchronisation que vous avez faite (Q17) supprimera automatiquement la vue.

Q21. On veut maintenant pourvoir déplacer des formes à l'aide d'une interaction glisser-déposer (*drag-and-drop*). Cette interaction est plus compliquée que le simple clic car plusieurs évènements interviennent (pression, déplacement, relâchement).

Créez une classe interne *DnDToMoveShape* dans *ControleurDessin*. Le principe de cette classe : elle récupère les évènements souris effectués sur une *Shape*. Lors d'une pression d'un bouton de la souris, la position de la pression est sauvegardée dans des attributs de la classe interne. Lors d'un déplacement de la souris, la forme (et non sa vue) est déplacée (ce qui déplace automatiquement la vue grâce aux bindings de la question Q18).

Q22. Ajoutez à cette classe deux attributs : *pressPositionX* et *pressPositionY* (du type *double*). Créez un constructeur prenant en paramètre un objet *view* du type *Shape*. Dans *createViewShapeFromShape*, créez un objet *DnDToMoveShape* en lui passant en paramètre la vue créée. Le principe : on écoute les évènements souris effectués sur chaque vue.

Q23. Toujours dans le constructeur de *DnDToMoveShape*, il est nécessaire de capturer à partir de la vue *view* certains évènements souris. Il faut pour cela utiliser trois listeners : `setOnMousePressed, setOnMouseDragged, setOnMouseReleased` (ce sont des opérations fournies par *Shape*, cf le cours). Ces opérations prennent en paramètre un listerner que vous pouvez écrire soit comme une classe anonyme, soit comme un lambda (cf le cours) :

```
view.setOnMousePressed(evt -> {
 //..
});
```

Complétez ces trois listeners pour déplacer la forme visée. Pour récupérer la position du pointeur de la souris, utilisez *evt.getSceneX()* et *evt.getSceneY()*. Testez.

Q24. Afin de mettre à jour les ascenseurs, le *Pane* et le *BorderPane* ne doivent plus avoir de taille définie dans le FXML. Dans le contrôleur ajoutez une méthode privée *updateSizePane* qui calcule la taille du *Pane* : la largeur du *Pane* est Math.max(largeur du scroll pane, coordonnée x maximale des formes [tenez compte de la largeur des formes]) ; idem pour la hauteur. Utilisez cette méthode pour définir la taille du *Pane*. Il faut que le déplacement d'une forme puisse mettre à jour (agrandir, enlever) les ascenseurs.

Q25. On veut maintenant afficher les labels *x* et *y* lorsqu'une forme est en cours de déplacement. On veut également que la valeur de ces labels corresponde à la position courante de la forme en cours de déplacement. Complétez l'opération *handle* pour gérer cette fonctionnalité en utilisant l'opération setVisible et un data binding spécifique :

```
x.textProperty().bind(Bindings.createStringBinding(…) ;
```

Testez.

Q26. S'il vous reste du temps, complétez l'application avec d'autres fonctionnalités de votre choix. Exemples : gestion de polygones, agrandissement des formes, etc.

N'oubliez pas de rendre votre code sous Moodle.