Mini-Projet 3 (TME 7–9) : éditeur de dessins vectoriels

Objectifs pédagogiques :

- développer une interface graphique avec l'API JavaFX ;
- développer une architecture d'application extensible ;
- mettre en pratique le principe de délégation ;
- mettre en pratique les *design patterns* observateur, composite, singleton, commande, stratégie.

But du projet

Le but du projet est de développer un éditeur de dessins vectoriels, sur le modèle d'Adobe Illustrator ou Inkscape. Bien sûr, notre éditeur ne possédera qu'un nombre très réduit de fonctionnalités. Nous compensons cela en mettant l'accent sur une conception extensible, permettant d'ajouter facilement des fonctionnalités. La conception de l'éditeur sera basée sur les principes de délégation, de programmation vis-à-vis d'interfaces et l'emploi de quelques *design patterns*. Pour la partie interface graphique (GUI), nous utiliserons l'API JavaFX.

Quelques points clés de notre éditeur :

- séparation entre modèle de document (représentation du dessin sous forme d'objets Java) et interface graphique d'édition ;
- possibilité d'ouvrir simultanément plusieurs fenêtres, contenant chacune un dessin différent ;
- barre de boutons, menus déroulants, palette de couleurs (en bonus) ;
- ajout de rectangles, d'ellipses, d'images (en bonus) ;
- outil de sélection, simple ou multiple ;
- déplacer, grouper, dégrouper des éléments ;
- copier et coller avec un presse-papiers partagé entre les fenêtres ;
- annuler et refaire une action jusqu'à un niveau arbitraire ;
- sauvegarde du dessin dans un fichier et relecture du fichier (en bonus).

Notre application se nommera *PinBoard*, signifiant en anglais « planche à épingler », puisqu'elle permet de positionner et déplacer des éléments graphiques (appelés *clips*) sur une surface (appelée *board*).

L'application sera développée dans un nouveau projet Eclipse nommé `PinBoard`, dans le package `pobj.pinboard` et ses sous-packages. Ce projet, et ce package, nous occupera durant les TME 7 à 9. Nous n'allons plus découper le package en sous-packages correspondant à chaque TME (ce que nous faisions avec les TME précédents), mais en sous-packages correspondant à des fonctionnalités de l'application.

Comme pour les projets précédents, vous trouverez sur le serveur GitLab, dans le groupe `POBJ-XXYY`, un projet `PinBoard` en lecture seule contenant un squelette d'application. Vous devrez commencer par en faire un *fork* pour votre binôme, puis ajouter à ce projet personnel votre camarade de binôme et votre chargé de TME, et enfin importer ce projet sous Eclipse avec « File > Import... > Git > Projects from Git ».

TME 7 : modèle de document, dessiner des rectangles

7.1 Modèle de document

Dans un premier temps, nous nous intéressons au modèle de document et pas encore à l'interface graphique d'édition. Le modèle est un ensemble de classes Java donnant la représentation interne de nos dessins et offrant des méthodes pour lire leurs attributs, les modifier et les dessiner à l'écran. Nous pouvons imaginer que le modèle sera commun à plusieurs applications ; il est donc bien indépendant de l'interface d'édition. Les classes du modèle seront placées dans le package `pobj.pinboard.document`.

7.1.1 Interface des éléments graphiques

Un dessin est composé d'une planche (*board*) sur laquelle sont posés des éléments graphiques (*clips*). Il existera plusieurs types d'éléments graphiques : rectangles, ellipses, images, etc. Une caractéristique commune de ces éléments est qu'ils occupent un certain espace sur la planche, et qu'ils peuvent être dessinés. Chaque élément est inscrit dans un rectangle : son *rectangle englobant* (même s'il ne recouvre pas tous les pixels du rectangle, comme dans le cas d'une ellipse). Le rectangle englobant définit la position et la taille d'un élément. Déplacer ou redimensionner un élément se fait en changeant ce rectangle. Nous utiliserons l'interface `pobj.pinboard.document.Clip` commune à tous les éléments graphiques :

tme/src/tme7/src/pobj/pinboard/document/Clip.java

```
package pobj.pinboard.document;

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

public interface Clip {
   // Drawing
   public void draw(GraphicsContext ctx);

   // Geometry
   public double getTop();
   public double getLeft();
   public double getBottom();
   public double getRight();
   public void setGeometry(double left, double top, double right, double bottom);
   public void move(double x, double y);
   public boolean isSelected(double x, double y);

   // Colors
   public void setColor(Color c);
   public Color getColor();

   // Cloning
   public Clip copy();
}
```

- `draw` dessine l'élément en utilisant le contexte graphique JavaFX passé en argument.
- `getTop`, `getLeft`, `getBottom`, `getRight` donnent les coordonnées haut, gauche, bas et droite du rectangle englobant. Notez que, suivant les conventions graphiques en informatique, les coordonnées $(0, 0)$ correspondent au coin en haut à gauche. L'axe des abscisses croît vers la droite, et l'axe des ordonnées croît vers le bas. La figure 1 illustre le système de coordonnées et les valeurs des coordonnées haut, gauche, bas et droite d'un rectangle. Pour faciliter les calculs géométriques, les coordonnées sont de type `double`.

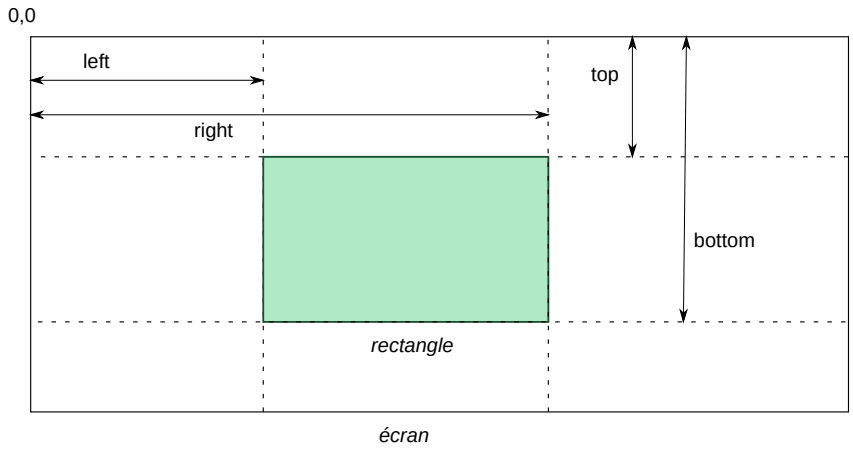

Figure 1: Coordonnées d'un rectangle.

- `setGeometry` modifie le rectangle englobant, ce qui a pour effet de déplacer ou redimensioner l'élément.
- `move` est un cas particulier de `setGeometry` qui se contente de déplacer l'élément sans changer sa taille (i.e., ajouter `x` aux coordonnées `left` et `right`, et ajouter `y` aux coordonnées `top` et `bottom`).
- `getColor` et `setColor` permettent de retrouver et de modifier la couleur de l'objet.
- `isSelected` indique si le point de coordonnées `x`, `y` est recouvert par l'élément graphique ou non (cela sera utile pour l'opération de sélection à la souris, au TME suivant).
- `copy` permet de créer une copie de l'objet, modifiable indépendemment de l'original.

Le fichier `Clip.java` est fourni dans le projet `PinBoard` de GitLab.

JavaFX sous Eclipse. L'utilisation de JavaFX nécessite la version de Java 8 d'Oracle. Si ce n'est pas déjà fait, assurez-vous que le projet est bien configuré pour Java 8 : clic droit sur le projet « Properties > Java Compiler » (ou menu « Window > Preferences > Java > Compiler » pour les préférences globales).
De même, assurez-vous que la VM est bien configurée : « Window > Preferences > Java > Installed JREs » et choisir une version en 1.8 d'Oracle. JavaFX ne fonctionnera pas avec OpenJDK.
Malgré cela, Eclipse peut indiquer une erreur de la forme : « *Access restriction: The type [...] is not accessible due to restriction on required library [...]/lib/ext/jfxrt.jar* ». Une solution consiste à changer la configuration de la bibliothèque système avec la manipulation suivante : clic droit sur le projet, « Properties > Java Build Path > Libraries > JRE System Library > Edit... > Workspace default JRE > Finish ».

7.1.2 Premier élément graphique : le rectangle (`ClipRect`)

Objectif pédagogique : programmer une implantation d'une interface, dessiner.

Notre premier élément graphique sera le **rectangle plein**, implanté par une classe `ClipRect` obéissant à l'interface `Clip`.

Un rectangle est défini par les coordonnées de son rectangle englobant et une couleur de remplissage, qui sont donc des attributs de l'objet. La classe possède un constructeur fixant la valeur initiale de ces attributs : `ClipRect(double left, double top, double right, double bottom, Color color)`.

⇒ **Travail demandé :** Écrivez la classe `pobj.pinboard.document.ClipRect`. Vous pourrez utiliser la classe de test `pobj.pinboard.document.test.ClipRectTest` fournie dans le GitLab pour valider votre implantation (celle-ci teste toutes les méthodes, excepté `draw` qui requiert d'ouvrir une fenêtre JavaFX, ce que nous ferons par la suite).

Dessiner dans un `GraphicsContext`. Dessiner notre élément consiste à peindre l'intérieur du rectangle englobant avec la couleur spécifiée. La classe `GraphicsContext`, dont une instance sera passée en argument à notre méthode `draw`, possède les méthodes nécessaires pour dessiner :
- `setFill(Color.RED)` spécifie la couleur de remplissage (ici, la couleur prédéfinie rouge) ;
- `fillRect(left,top,width,height)` dessine un rectangle plein ;
- `fillOval(left,top,width,height)` dessine une ellipse pleine ;
- `setStroke(Color.BLUE)` spécifie la couleur de contour ;
- `strokeRect(left,top,width,height)` dessine le contour d'un rectangle ;
- `strokeOval(left,top,width,height)` dessine le contour d'une ellipse ;
- `drawImage(image,left,top)` dessine une image (`javafx.scene.image.Image`).

Nous ne fournissons pas ici le détail de toutes les méthodes utiles au projet. C'est à vous de consulter la documentation de `GraphicsContext` sur le site d'Oracle pour plus d'informations.
Attention : JavaFX définit un rectangle par son coin supérieur gauche et ses dimensions (`left`, `top`, `width`, `height`), alors que nous utilisons le coin supérieur gauche et le coin inférieur droit (`left`, `top`, `bottom`, `right`).

7.1.3 Les planches (`Board`)

Objectif pédagogique : programmer vis-à-vis d'une interface, revoir les collections Java.

Une planche, implantée par la classe `Board`, est un conteneur pour un ensemble d'éléments graphiques (`List<Clip>`). Bien entendu, la classe `Board` est programmée vis-à-vis de l'interface `Clip`, et non vis-à-vis d'une classe concrète comme `ClipRect`. La classe `Board` fournira :

- `public Board()`
  un constructeur sans argument construisant une planche vide.
- `public List<Clip> getContents()`
  une méthode pour retourner la liste des éléments de la planche.
- `public void addClip(Clip clip)`
  `public void addClip(List<Clip> clip)`
  des méthodes pour ajouter un élément, ou toute une liste d'éléments.
- `public void removeClip(Clip clip)`
  `public void removeClip(List<Clip> clip)`
  des méthodes pour supprimer un élément, ou toute une liste d'éléments.
- `public void draw(GraphicsContext gc)`
  une méthode pour dessiner le contenu de la planche. Pour dessiner la planche, il est d'abord nécessaire d'effacer la zone d'affichage, en dessinant un rectangle blanc de coordonnées (0,0) (i.e., en haut à gauche) et de la taille du contexte graphique (qui peut être déterminée par `gc.getCanvas().getWidth()` et `getHeight()`). Il faut ensuite appeler la méthode `draw` de chaque élément de la planche.

⇒ **Travail demandé :** Écrivez la classe `pobj.pinboard.document.Board`. Nous fournissons dans le projet GitLab une classe de test `pobj.pinboard.document.test.BoardTest`.

7.1.4 Nouvel élément graphique : l'ellipse (`ClipEllipse`) ; factorisation (`AbstractClip`)

Objectif pédagogique : factoriser des implantations dans une classe abstraite.

Nous ajoutons maintenant un nouvel élément graphique : l'**ellipse pleine**, décrite par la classe `ClipEllipse`. Une ellipse est construite par un constructeur similaire à celui des rectangles :

```
public ClipEllipse(double left, double top, double right, double bottom, Color color);
```

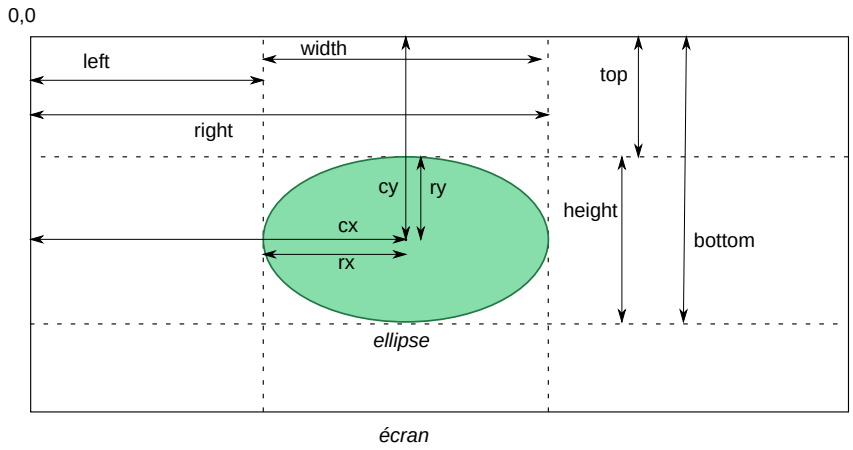

Figure 2: Coordonnées d'une ellipse.

et sera dessinée grâce à la méthode `fillOval` de JavaFX. Concernant l'implantation de `isSelected`, si nous notons $cx = (\texttt{left} + \texttt{right})/2$ et $cy = (\texttt{top} + \texttt{bottom})/2$ le centre de l'ellipse, et $rx = (\texttt{right} - \texttt{left})/2$ et $ry = (\texttt{bottom} - \texttt{top})/2$ les deux rayons de l'ellipse, alors déterminer si un point x, y est dans l'ellipse revient à tester si :

$$\left(\frac{x - cx}{rx}\right)^2 + \left(\frac{y - cy}{ry}\right)^2 \leq 1 \ .$$

Les coordonnées utilisées par notre classe (`left`, `top`, `right`, `bottom`), celles utilisées par `fillOval` (`left`, `top`, `width` ,`height`), le centre (cx, cy) et les rayons (rx, ry) sont précisés en figure 2.

Avant de se lancer dans la programmation de la classe, nous observons que les ellipses et les rectangles partagent des fonctionnalités communes. Plutôt que de dupliquer le code, nous allons isoler dans une classe `AbstractClip` les fonctionnalités réutilisables, auxquelles nos éléments graphiques accéderont par héritage. Plus précisément, `AbstractClip` :

- fournira les méthodes `getTop`, `getRight`, `getBottom`, `getLeft`, `setGeometry`, `move`, `getColor` et `setColor` et définira les attributs (privés) nécessaires au support de ces méthodes ;
- ne fournira **pas** les méthodes `draw` et `copy`, spécifiques à chaque type d'élément graphique ;
- fournira une implantation par défaut de `isSelected` correspondant à celle de `ClipRect` (un point est accepté s'il est dans le rectangle englobant), qui pourra donc être utilisée directement par `ClipRect` (et d'autres éléments rectangulaires, comme l'élément « image » proposé en extension) mais sera redéfinie par `ClipEllipse`.

Pour indiquer qu'`AbstractClip` est une implantation partielle de l'interface `Clip`, nous la déclarons comme classe abstraite (mot-clé Java `abstract`).

⇒ **Travail demandé :**

1. Programmez une classe abstraite `pobj.pinboard.document.AbstractClip`, et déplacez les attributs et méthodes nécessaires de `ClipRect` vers `AbstractClip`.
2. Modifiez `ClipRect` pour qu'elle hérite d'`AbstractClip`. Attention : les attributs (privés) ayant été déplacés dans `AbstractClip`, il devient nécessaire d'utiliser des *getters* et *setters* (publiques) pour y accéder depuis `ClipRect` (`getWidth`, `setGeometry`, etc.).
3. Écrivez la classe `pobj.pinboard.document.ClipEllipse` héritant de `AbstractClip` et implantant l'interface `Clip`.
4. Testez vos implantations avec `pobj.pinboard.document.test.ClipRectTest` et `pobj.pinboard.document.test.ClipEllipseTest` fournis.

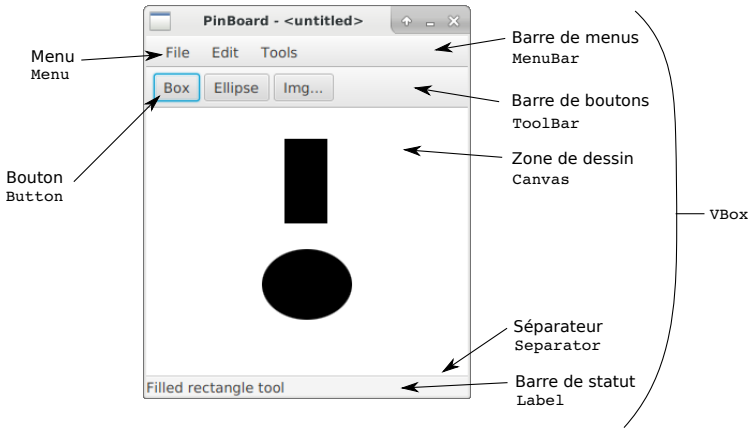

Figure 3: Interface graphique de l'éditeur.

7.1.5 Premier test d'affichage

Objectif pédagogique : application JavaFX minimale.

Nous avons déjà effectué des tests unitaires pour nos méthodes qui ne nécessitaient pas d'affichage graphique. Afin de tester la méthode `draw`, JUnit ne suffit plus ; il est nécessaire de créer une fenêtre affichant le résultat. Nous n'avons pas de méthode de test automatique pour l'affichage, cela se fait par observation par l'humain ! Pour cela, nous fournissons dans le projet GitLab une application minimale, `pobj.pinboard.test.TestDocument`, qui crée un dessin simple et l'affiche dans une fenêtre JavaFX.

⇒ **Travail demandé :** Exécutez cette classe. Elle doit afficher une fenêtre contenant un rectangle bleu et une ellipse rouge.

7.2 Interface graphique d'édition

Dans cette partie, nous commençons notre interface graphique pour visualiser et éditer nos documents. Pour l'instant, les fonctionnalités sont limitées à créer des nouvelles fenêtres, les fermer et placer dans ces fenêtres les éléments graphiques définis dans la partie précédente. Les classes de l'interface seront placées dans le package `pobj.pinboard.editor` et ses sous-packages.

7.2.1 Fenêtres d'édition (`EditorWindow`)

Objectif pédagogique : créer des fenêtres et des contrôles GUI JavaFX.

Une fenêtre de notre application se présente comme décrit en figure 3. Elle comprend les contrôles suivants :

- une barre de menu (classe JavaFX `MenuBar`), contenant des menus (`Menu`) ;
- une barre de boutons (`ToolBar`), contenant des boutons (`Button`) servant à choisir le type d'élément graphique à ajouter ;
- une zone de dessin (classe `Canvas`), dans laquelle la planche sera affichée (notre classe `Board`) ;
- une barre de statut (`Label`), séparée de la zone de dessin par un séparateur (`Separator`).

Ces éléments sont agencés verticalement dans un conteneur (*layout*) `VBox`.

Une fenêtre de l'éditeur et le dessin édité seront encapsulés dans la classe `EditorWindow`, de constructeur `EditorWindow(Stage stage)`. Nous fournissons dans `pobj.pinboard.editor.EditorMain` le

point d'entrée de notre application. Celui-ci se contente de créer une nouvelle `EditorWindow` en lui transmettant l'objet `Stage` passé à sa méthode `start`. L'avantage de séparer ainsi le point d'entrée et la classe fenêtre est qu'il sera très facile de créer des fenêtres supplémentaires, avec `new EditorWindow (new Stage())`.

L'essentiel du travail de `EditorWindow` est pour l'instant concentré dans son constructeur, qui doit :

- créer une nouvelle planche vide, `Board`, stockée dans un attribut ;
- choisir le titre de la fenêtre ;
- y associer une scène, contenant les différents contrôles demandés, et organisés comme montré en figure 3.

Pour l'instant, aucun comportement n'est associé aux contrôles ni aux événements de la souris.

⇒ **Travail demandé :** Écrivez la classe `pobj.pinboard.editor.EditorWindow`. Vous pourrez vous inspirer de la classe `pobj.pinboard.test.TestDocument` fournie.

Fenêtre principale d'une application JavaFX. Rappelons qu'une fenêtre principale d'une application JavaFX est donnée par une instance de `javafx.stage.Stage`, et qu'une instance est créée automatiquement par JavaFX lors du démarrage de l'application et passée en argument à `start`. Une fenêtre contient une scène (`javafx.scene.Scene`), qui contient à son tour différents contrôles et layouts. La configuration d'une fenêtre `stage` comporte généralement les étapes suivantes :

- choix d'un titre : `stage.setTitle("Titre")` ;
- création d'un layout, dans notre cas, une simple `VBox` ;
- création et ajout à cette boîte de contrôles (ou d'autres layouts imbriqués) ;
- création d'un objet scène contenant notre boîte principale : `new Scene(vbox)` ;
- association de la scène à la fenêtre : `stage.setScene(scene)` ;
- finalement, affichage de la fenêtre : `stage.show()`.

Contrôles JavaFX. Parmi les éléments d'interface du package `javafx.scene.control`, les éléments suivants seront particulièrement utiles pour notre application :

- `Button` : bouton cliquable, dont le texte est spécifié dans le constructeur.
- `Label` : ligne de texte simple, non éditable, sans décoration. Le texte est spécifié dans le constructeur et peut être modifié par `label.textProperty().set("texte")`.
- `Toolbar` : un conteneur de boutons. La liste des boutons peut être spécifiée dans le constructeur : `new Toolbar(new Button("A"), new Button ("B"), ...)`.
- `Separator` : un ligne séparatrice, utile à ajouter dans une boîte ou une barre d'outils pour améliorer la présentation.
- `Canvas` : une zone de dessin libre. La taille est spécifiée dans le constructeur : `new Canvas(width,height)`. La méthode `getGraphicsContext2D` permet à tout moment d'obtenir un contexte graphique pour dessiner dans la zone.
- `MenuBar` : une barre de menu. La liste des menus (`Menu`) peut être spécifiée lors de la construction, avec `new MenuBar(menu1, menu2, ...)`.
- `Menu` : un menu déroulant, dont le titre est passé en argument au constructeur. Des entrées de menu (`MenuItem`) et des séparateurs (`MenuSeparator`) peuvent être ajoutés par `menu.getItems().addAll(...)`.
- `MenuItem` : une entrée dans un menu ; l'intitulé est passé en argument au constructeur.
- `MenuSeparateur` : une ligne séparatrice pour grouper des entrées de menu.

Pour plus d'informations, consultez la documentation du package `javafx.scene.control`.

Layout JavaFX. Les classes de layout permettent d'organiser les composants de l'interface graphique et se situent dans le package `javafx.scene.layout`. Ici, nous utiliserons uniquement la boîte verticale, créée avec `new VBox()`. Rappelons que, pour ajouter un ou plusieurs éléments (contrôles ou layouts imbriqués) à une boîte `box`, il faut utiliser la méthode `box.getChildren().addAll(...)`, où `...` dénote une liste arbitrairement longue d'éléments. Note : il est inutile de créer explicitement une liste avant de la passer au constructeur ; il suffit de spécifier les éléments comme autant d'arguments au constructeur (Java créera implicitement une liste pour vous).
Pour plus d'informations, consultez la documentation du package `javafx.scene.layout`.

7.2.2 Menus déroulants

Objectif pédagogique : associer un comportement à des contrôles GUI JavaFX.

Nous allons commencer à peupler nos menus déroulants avec des entrées (`MenuItem`), et associer des actions à effectuer quand l'utilisateur clique sur ces entrées. Plus précisément, nous ajoutons deux entrées au menu « File » :

- « New », qui crée une nouvelle fenêtre avec `new EditorWindow(new Stage())`.
- « Close », qui ferme la fenêtre courante avec `stage.close()`. L'application JavaFX se termine d'elle-même quand toutes les fenêtres sont fermées.

⇒ **Travail demandé :** implantez les entrées « New » et « Close » dans le menu « File » et leurs actions.

Associer une action à un événement. Pour ajouter une action à une entrée de menu, `item`, de classe `MenuItem`, il faut appeler la méthode `item.setOnAction(action)`, où `action` obéit à l'interface `EventHandler<ActionEvent>`. Cette interface requiert d'implanter une seule méthode : `public void handle(ActionEvent)`. Celle-ci sera appelée par JavaFX à chaque fois que l'entrée du menu sera sélectionnée. L'objet passé en argument, ici `ActionEvent`, fournit des informations supplémentaires sur l'événement et sa cible, mais elles ne nous seront pas utiles dans cette question.
Une méthode, vue en cours, pour créer un objet d'interface `EventHandler<ActionEvent>` consiste à utiliser une classe anonyme, par exemple :

```
item.setOnAction(
  new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) { System.out.println("coucou"); }
  });
```

Une solution équivalente, mais plus concise, consiste à utiliser les *lambdas*, un trait ajouté à Java 8 que nous verrons en détails dans les derniers cours mais que nous pouvons dès à présent utiliser. Cela donnerait simplement :
`item.setOnAction( (e)-> { System.out.println("coucou"); } );`

7.2.3 Outils de dessin

Objectif pédagogique : événements souris en JavaFX, motif stratégie.

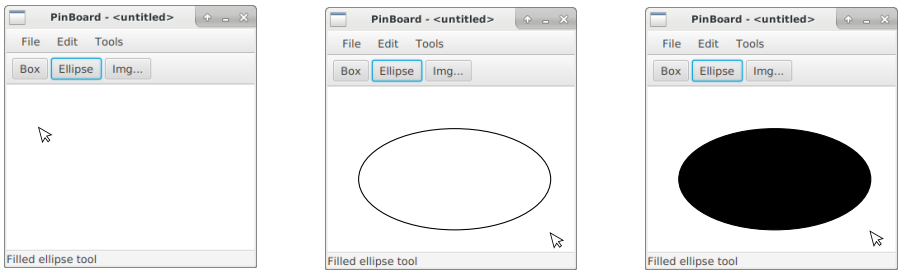

Figure 4: Étapes pour le dessin d'une ellipse.

Nous allons maintenant permettre à l'utilisateur d'ajouter des éléments graphiques à une planche. Chaque fenêtre `EditorWindow` possède un *outil courant*, qui indique si on ajoute des rectangles ou des ellipses. L'outil courant est changé en cliquant sur un des boutons de la barre de boutons.

Supposons l'outil « ellipse » sélectionné. L'utilisateur clique sur la zone de dessin (`Canvas`) pour sélectionner le premier coin du rectangle englobant, déplace la souris en maintenant le bouton de la souris enfoncé, puis relâche le bouton pour définir le second coin du rectangle englobant. Tant que le bouton de la souris est enfoncé, un retour d'affichage (*feedback*) dessine le *contour seul* de l'ellipse, permettant à l'utilisateur d'ajuster sa taille en déplaçant la souris. Ce n'est qu'au moment où le bouton de la souris est relâché qu'un élément graphique `ClipEllipse` est effectivement ajouté à la planche, et l'*ellipse pleine* apparaît alors. La figure 4 illustre ces étapes. L'outil « rectangle » est similaire, mais affiche le contour du rectangle tant que le bouton reste appuyé et ajoute un objet `ClipRect` quand le bouton est relâché. Pour l'instant, nous n'offrons pas à l'utilisateur le moyen de choisir la couleur des éléments graphiques, ce qui n'empêche pas les outils de choisir eux-même une couleur fixe (par exemple, les rectangles seront bleus et les ellipses rouges) ou aléatoire (voir la classe `Random` de l'API standard Java).

Événements souris. Tout nœud JavaFX (en particulier, tout contrôle, comme notre objet `Canvas`) peut réagir aux divers événements de la souris (déplacement, clic, etc.) quand le pointeur est au-dessus de ce nœud. De manière similaire aux entrées de menu, il faut associer une action à ces événements. Plus précisément, nous utiliserons les méthodes :

- `canvas.setOnMousePressed(EventHandler<MouseEvent>)` pour un appui de bouton ;
- `canvas.setOnMouseDragged(EventHandler<MouseEvent>)` pour un déplacement de souris avec un bouton appuyé ;
- `canvas.setOnMouseReleased(EventHandler<MouseEvent>)` pour un bouton relâché.

L'événement passé en argument à la méthode `handle` de notre `EventHandler` est maintenant de classe `MouseEvent`. Il fournit des informations utiles sur l'état de la souris au moment où l'événement s'est produit, en particulier :

- `getX()` et `getY()` permettent de retrouver les coordonnées de la souris (relatives au nœud qui reçoit l'événement) ;
- `getButton()` permet d'identifier le bouton pressé ou relâché.

Nous pourrions coder directement le comportement de tous nos outils dans des méthodes `press`, `drag` et `release` de `EditorWindow`, et les associer à notre `Canvas` par `setOnMousePressed`, etc. Cependant, nous souhaitons éviter de surcharger `EditorWindow` avec la gestion de tous ces cas afin de gagner en lisibilité et en facilité d'extension. Nous allons donc encapsuler le comportement de chaque outil dans une classe dédiée, et demander à `EditorWindow` de déléguer les actions de la souris à la classe de l'outil en cours. Un outil obéit à l'interface `pobj.pinboard.editor.tools.Tool` :

tme/src/tme7/src/pobj/pinboard/editor/tools/Tool.java

```java
package pobj.pinboard.editor.tools;

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.input.MouseEvent;
import pobj.pinboard.editor.EditorInterface;

public interface Tool {
   public void press(EditorInterface i, MouseEvent e);
   public void drag(EditorInterface i, MouseEvent e);
   public void release(EditorInterface i, MouseEvent e);
   public void drawFeedback(EditorInterface i, GraphicsContext gc);
   public String getName();
}
```

Nous ajoutons à la classe `EditorWindow` un attribut `Tool` dénotant l'« outil courant ». Les méthodes `press`, `drag` et `release` de `EditorWindow` se contentent maintenant d'appeler la méthode correspondante dans l'outil courant. Néanmoins, ces méthodes doivent pouvoir accéder à certains attributs de l'objet `EditorWindow` appelant : en particulier, `release` doit pouvoir accéder à l'attribut `Board` de la fenêtre appelante pour y ajouter un élément graphique quand le bouton de la souris est relâché ! Pour résoudre cette dépendance, tout en limitant au maximum le couplage entre `EditorWindow` et `Tool`, nous définissons l'interface `pobj.pinboard.editor.EditorInterface` :

tme/src/tme7/src/pobj/pinboard/editor/EditorInterface.java

```java
package pobj.pinboard.editor;

import pobj.pinboard.document.Board;

public interface EditorInterface {
   public Board getBoard();
   // ...
}
```

Il s'agit des *getters* pour les attributs qu'une fenêtre d'édition rend visible aux outils. Pour l'instant, elle se contente d'exposer sa planche à dessin `Board`. Cependant, nous serons amenés par la suite à ajouter à `EditorWindow` des fonctionnalités à exposer via `EditorInterface`. Pour faciliter ces ajouts, nous avons inclus dans `EditorInterface` des *getters* pour des classes `Selection` et `CommandStack` qui seront définies dans les TME suivants. Pour l'instant, ces classes ont une implantation vide, et les *getters* correspondants de `EditorWindow` se contentent de retourner `null`. Plus précisément :

- `EditorWindow` est modifié pour implanter l'interface `EditorInterface` et offre donc un *getter* pour son attribut `Board` (ainsi que des *getters* `null` pour `Selection` et `CommandStack`) ;
- lors de la délégation aux méthodes de `Tool`, la classe `EditorWindow` passe en argument `this` en guise d'interface `EditorInterface`, exportant ainsi le *getter* sur la planche ;
- la méthode `release` des `Tool` se sert de l'argument `EditorInterface` pour trouver la planche où ajouter un élément graphique.

En plus des méthodes `press`, `drag` et `release`, l'interface `Tool` possède les méthodes suivantes :

- `drawFeedback`, qui sera appelée par `EditorWindow` après chaque événement pour le retour d'affichage (affichage du contour du rectangle ou de l'ellipse lors de l'action `drag`) ;
- `getName`, donnant le nom de l'outil. Ce nom sera affiché dans la barre de statut.

⇒ **Travail demandé :** Implantez des classes concrètes `ToolRect` et `ToolEllipse` implantant `Tool`. Vous pourrez les tester grâce aux classes de test `pobj.pinboard.editor.tools.test.ToolRectTest` et `pobj.pinboard.editor.tools.test.ToolEllipseTest`.

Note : les classes de test se basent sur la classe utilitaire fournie `ToolTest`. Cette dernière crée un objet obéissant à l'interface `EditorInterface` pour fournir aux outils, avec une classe interne spéciale

`MockEditor` : cet objet « factice » (*mock* en anglais) se contente de définir les attributs nécessaires à `EditorInterface`, sans créer l'interface graphique d'édition autour.

⇒ **Travail demandé :** Modifiez `EditorWindow` pour y ajouter un attribut `Tool` ; ajoutez des actions aux événements de la souris sur le `Canvas` (ces actions délèguent leur travail à l'attribut `Tool`) ; ajoutez des actions aux boutons de la barre de boutons (pour sélectionner l'outil courant et mettre à jour la barre de statut) ; ajoutez une méthode `draw` pour mettre à jour le `Canvas` en y dessinant la planche et le retour d'affichage de l'outil, et appelez-la après chaque action susceptible de modifier l'affichage (ajout d'un élément à la planche, action de la souris sur le `Canvas`, etc.).

En option, vous pouvez ajouter, dans le menu déroulant « Tools », des options « Rectangle » et « Ellipse », offrant à l'utilisateur une méthode alternative pour choisir ces outils.

7.3 Bonus : les images

Pour tester les capacités d'extension de notre architecture, nous ajoutons un nouveau type d'élément graphique : les images. Il sera donc nécessaire d'ajouter :

1. Une classe `ClipImage`.
   Le constructeur précisera uniquement le coin supérieur gauche et le nom du fichier contenant l'image :

   ```
   public ClipImage(double left, double top, File filename);
   ```

   Le coin inférieur droit sera déduit automatiquement grâce à la taille de l'image. Le chargement effectif de l'image à partir du nom du fichier se fait en construisant un objet `Image`, par : `new Image("file://" + filename.getAbsolutePath())`. Il est ensuite possible de récupérer sa taille (`getWidth` et `getHeight`) et de la dessiner (`drawImage`). Le test `isSelected` testera si le point est dans le rectangle englobant.
2. Un outil `ToolImage`.
   Déplacer la souris ne fait que modifier le coin supérieur gauche de l'image ; le coin inférieur droit reste défini par la taille de l'image (contrairement à un rectangle, l'outil ne permet que de positionner une image, pas de la redimensionner). Le retour d'affichage pourra afficher l'image complète (ou juste son rectangle englobant) pendant le déplacement de la souris avec un bouton appuyé.
3. Un bouton `Img...` dans la barre de boutons.
   Lorsque l'outil est sélectionné en cliquant sur le bouton `Img...`, une boîte de dialogue s'affiche pour permettre à l'utilisateur de choisir un fichier contenant l'image. Vous utiliserez la classe JavaFX prédéfinie `FileChooser` pour cela.

Les modifications à apporter à la classe `EditorWindow` sont minimes, et aucune autre classe n'est modifiée. L'extension doit se faire essentiellement en ajoutant deux classes: `ClipImage` et `ToolImage`.

⇒ **Travail demandé :** Ajoutez le support pour les images dans le modèle de document et dans l'éditeur. Vous pouvez également ajouter des classes de test pour le modèle et l'outil, basées sur `ClipRectTest` et `ToolRectTest`. N'oubliez pas de les ajouter au script d'intégration continue `.gitlab-ci.yml`.

7.4 Rendu du TME (OBLIGATOIRE)

Le rendu se fait, comme d'habitude, à l'aide d'un *push* sur le serveur GitLab, suivi de la création d'un *tag*. Vous vous assurerez que l'intégration continue sous GitLab passe avec succès les tests spécifiques du TME 7.

Vous attacherez à la « Release notes » de votre *tag* deux captures d'écrans :

– l'application `TestDocument` donnée en 7.1.5 ;

– l'application `EditorMain` réalisée en 7.2 et montrant un dessin que vous avez réalisé.

Vous préciserez également si vous avez développé l'extension proposée en bonus, ou une autre extension de votre invention.