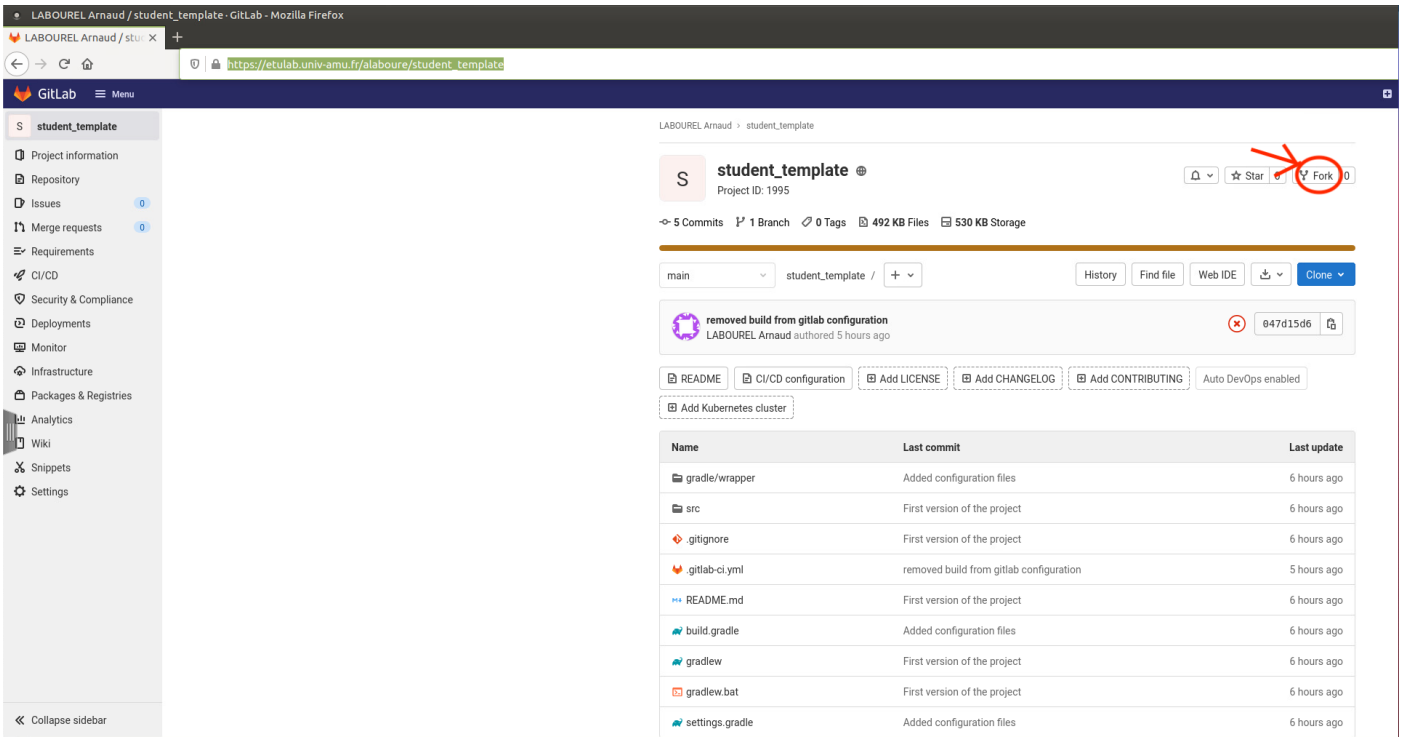
1 Dépôt pour application de dessin

Le but de ce projet est de rajouter des fonctionnalités à une application graphique permettant de dessiner des figures géométriques, de sauvegarder les dessins dans des fichiers `.daff` et de les exporter au format SVG. L'application qui vous est fournie permet seulement de dessiner des rectangles et on vous demande donc de rajouter

1.1 Fork d'un projet

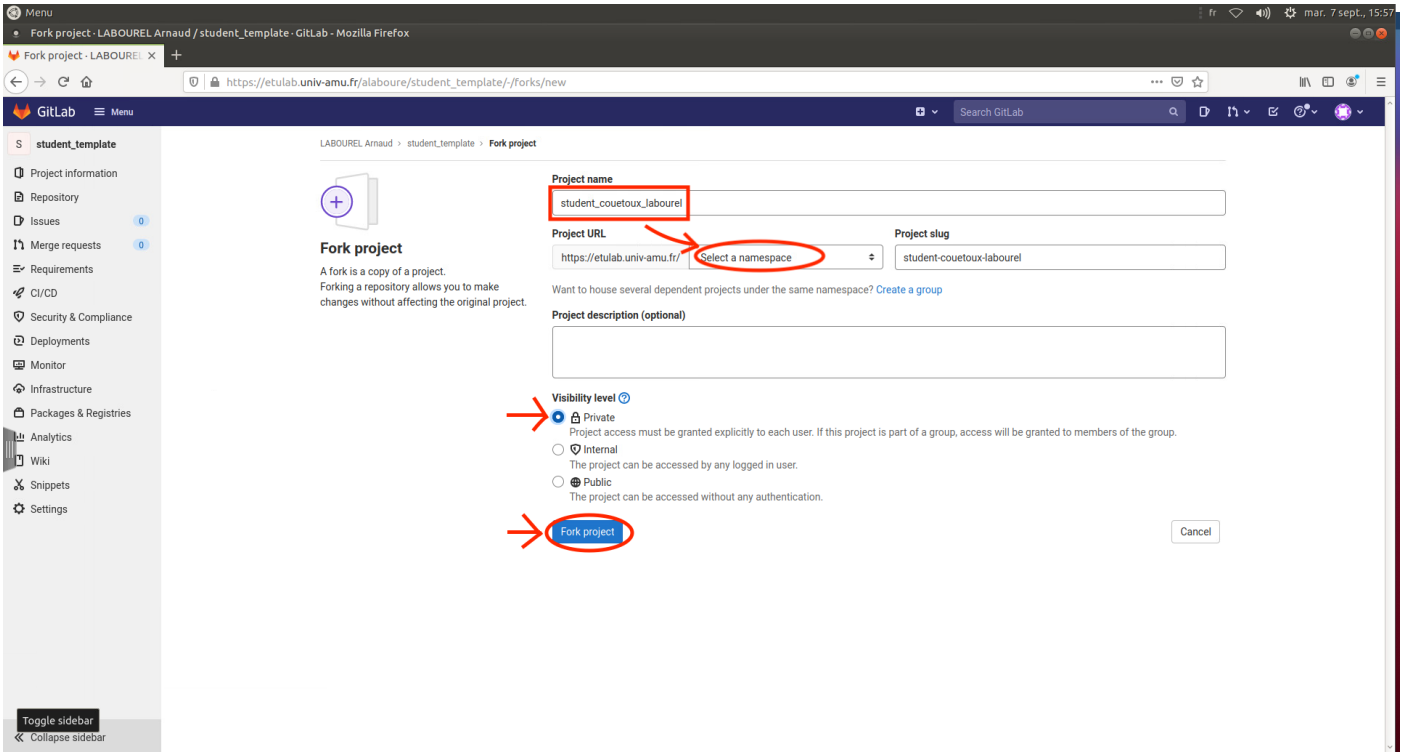
Vous allez maintenant créer votre projet en utilisant un projet déjà existant. Pour cela, il faut :

1. Aller sur le projet `drawing-app-template` qui servira de base pour ce TP en accédant à l'adresse suivante : https://etulab.univ-amu.fr/alaboure/drawing-app-template
2. Cliquer sur le bouton *fork*.

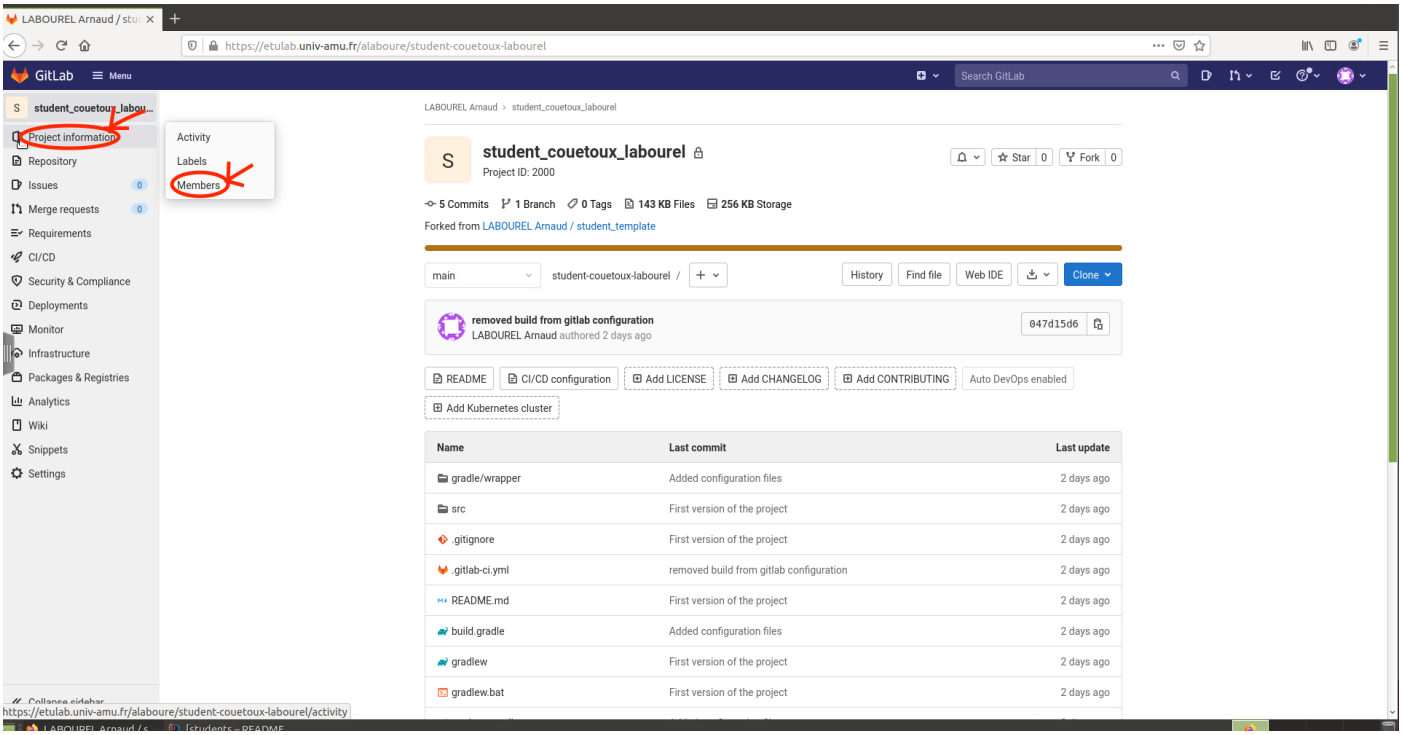


1. Changer le nom du projet pour le changer `drawing-app-xxxxx-yyyyy` avec `xxxxx` le nom de famille du premier (par ordre alphabétique) étudiant du binôme et `yyyyy` le nom du deuxième étudiant du binôme. Si vous faites le projet seul, mettez `drawing-app-xxxxx` avec `xxxxx` votre nom de famille.

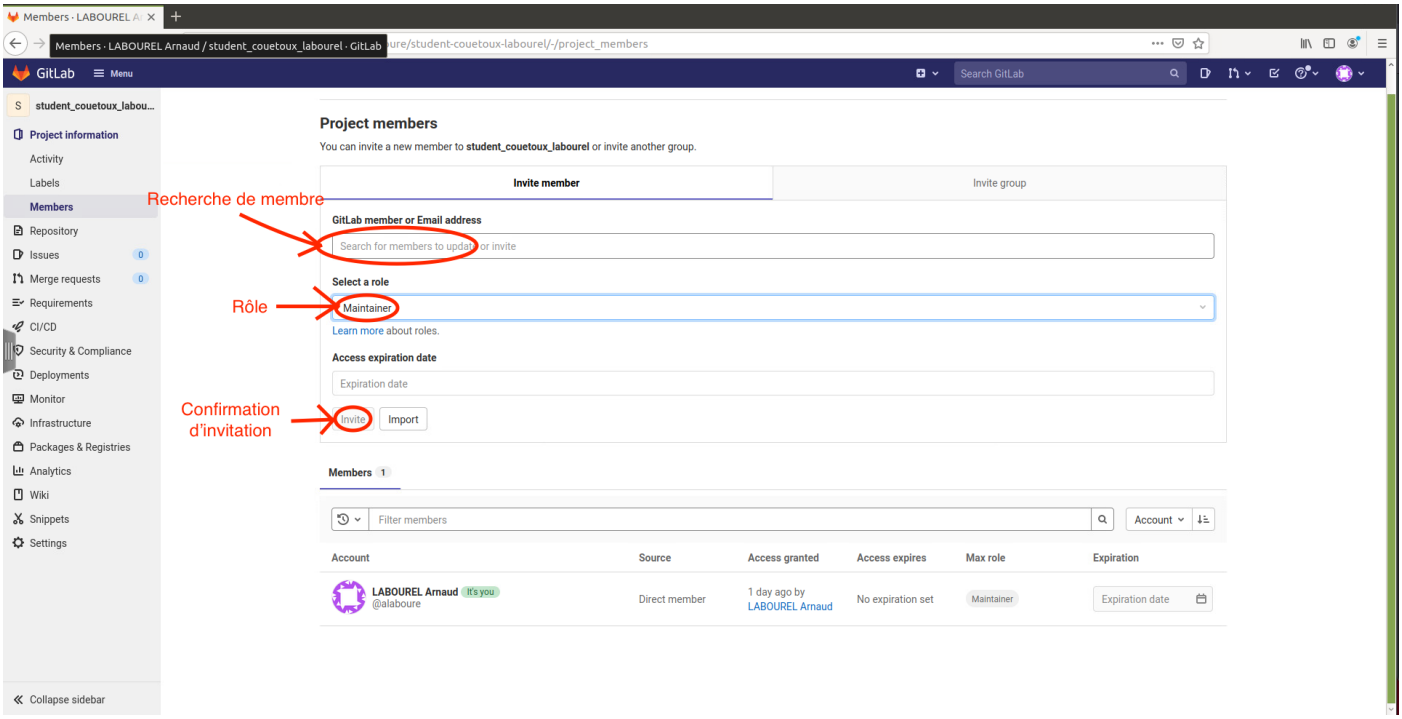
2. Sélectionner comme espace de nom (*spacename*) votre propre compte. Mettez la visibilité du projet en *private* afin que vos camarades en dehors du projet n'y aient pas accès et validez le fork en cliquant sur le bouton `fork project`.



1. Une fois le projet créé, vous pouvez rajouter des membres en cliquant sur `project information` dans le menu de gauche puis `members`.

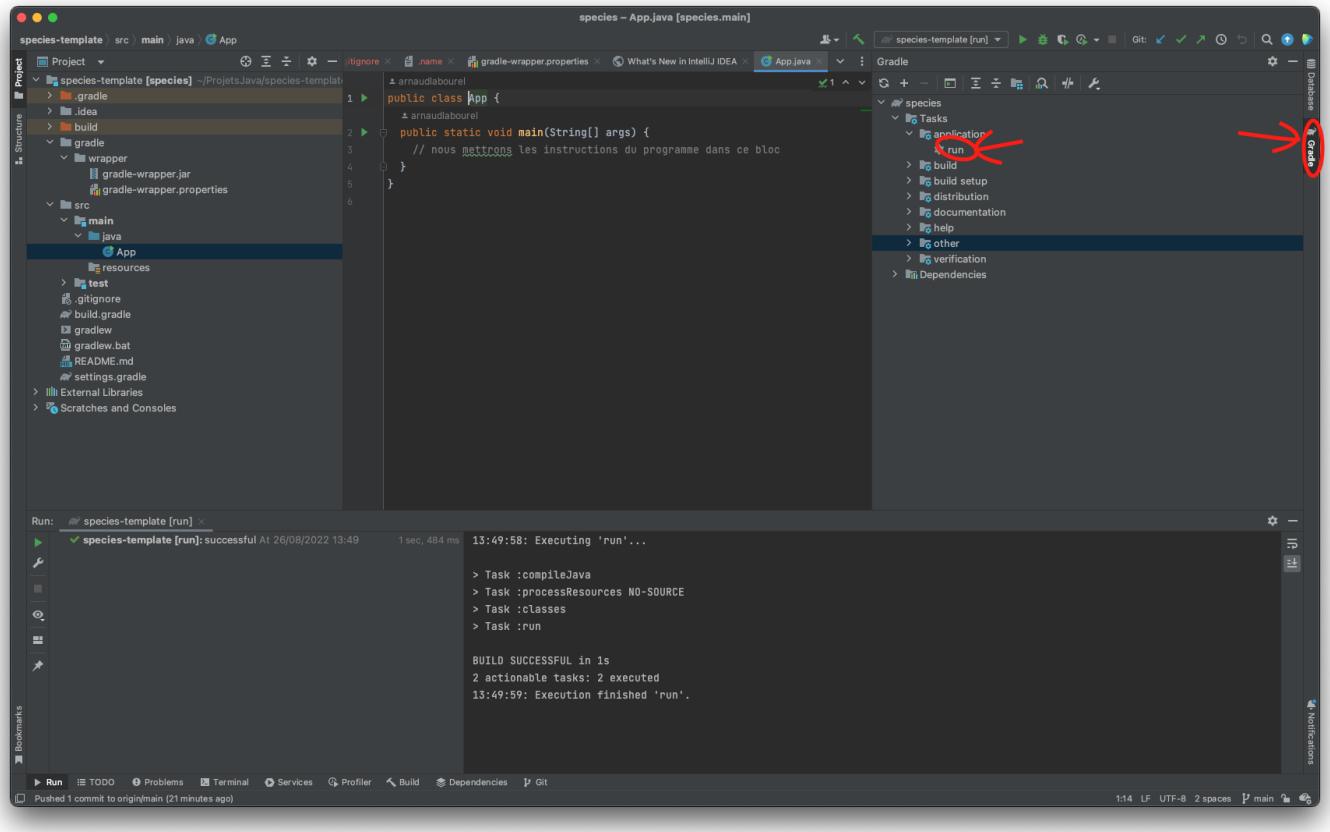


5. Ensuite vous pouvez rechercher une personne dans la barre dédiée. Une fois celle-ci trouvé vous pouvez lui donner un rôle (au moins `reporter` pour donner l'accès en lecture du code, `maintainer` pour l'accès en lecture et écriture sur la branche principale *master* ou *main* et *owner* pour donner approximativement les mêmes droits que le créateur du projet), puis confirmer son invitation en cliquant sur le bouton `invite`.



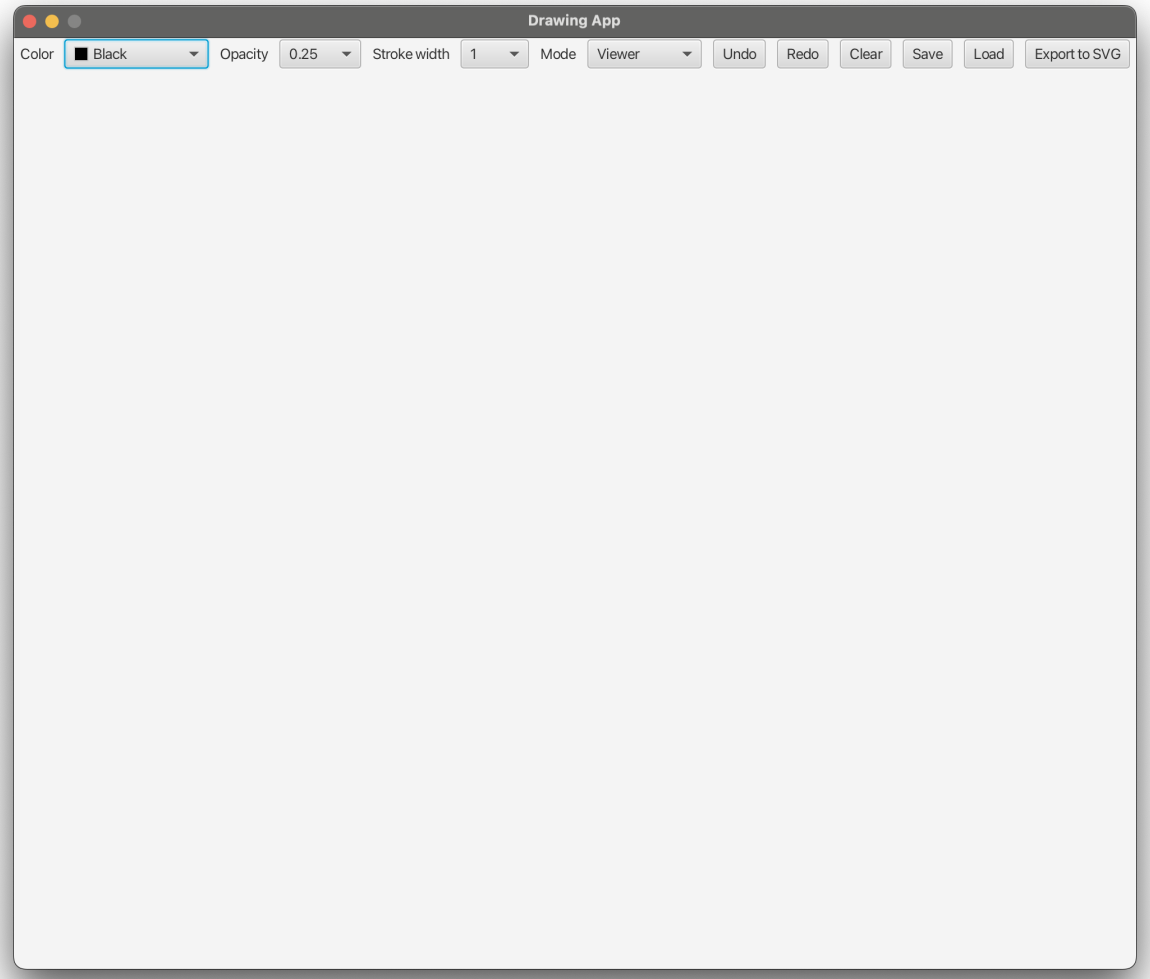
1.2 Exécuter le projet du dépôt

Pour compiler et exécuter votre programme, il faut passer par l'onglet gradle à droite.



— pour les tests, il faut cliquer deux fois sur `drawing_app` -> `Tasks` -> `verification` -> `test`. Pour le moment, les tests ne passeront pas car certaines classes sont incomplètes.

— pour l'affichage, il faut cliquer deux fois sur `regulation-network` -> `Tasks` -> `application` -> `run`. Vous devriez obtenir un affichage similaire à l'affichage suivant.



2 Explication application de dessin

L'application à laquelle ont vous donne l'accès permet de dessiner des rectangles (l'interface prends déjà en compte la possibilité de tracer des cercles et des polygones, mais ces deux types de figures ne sont pas encore implémentées), de sauvegarder et charger des dessins avec un format `.daff` ainsi que d'exporter les dessins au format SVG. Les fonctionnalités de l'application dans l'ordre de leur apparition dans la barre de menu sont les suivantes :

— Un sélecteur de couleur `Color` qui permet de choisir la couleur des figures créées ;
— Un sélecteur d'opacité `Opacity` qui permet de choisir l'opacité comprise entre 0 et 1 de l'intérieur des figures créées, 0 correspondant à un intérieur totalement transparent alors que 1 correspond à un intérieur de la couleur de base.
— Un sélecteur de taille de bord qui permet de choisir la largeur du trait délimitant les figures créées en pixels ;
— Un sélecteur de mode permettant de choisir le mode d'édition du dessin parmi 6 modes possibles, les

modes peuvent être changés via le menu ou en appuyant sur la première lettre du mode :
  - *Viewer* : ne permet aucune édition et sert juste à visualiser la figure ;
  - *Rectangle* : permet de créer des figures rectangulaires en faisant un clic gauche pour définir le premier coin et en relâchant le clic gauche pour définir le deuxième coin ;
  - *Circle* (non-implémenté) : permet de créer des figures circulaires en faisant un clic gauche pour définir le centre et en relâchant le clic gauche pour définir un point sur le cercle ;
  - *Polygon* (non-implémenté) : permet de créer des figures polygonales en faisant des clics gauches pour définir tous les points du polygone sauf le dernier et en faisant un clic droit pour définir le dernier point du polygone ;
  - *Move* (non-implémenté) : permet de bouger des figures en faisant un clic gauche dessus, puis en bougeant la souris et finalement relâchant le clic gauche pour finaliser le déplacement ;
  - *Delete* (non-implémenté) : permet de supprimer des figures en faisant un clic gauche dessus ;
- Un bouton *undo* (non-implémenté) qui permet d'annuler la dernière modification effectuée (sauf *undo* et *redo*) par l'utilisateur ;
- Un bouton *redo* (non-implémenté) qui permet d'annuler le dernier *undo* non encore annulé ;
- Un bouton *Clear* qui permet d'enlever toutes les figures du dessin en cours ;
- Un bouton *Save* qui ouvre une fenêtre permettant de sauvegarder le dessin en cours dans un fichier au format `.daff` ;
- Un bouton *Load* qui une fenêtre permettant de charger un fichier `.daff` et de remplacer le dessin en cours par le contenu du fichier ;
- Un bouton *Export to SVG* qui ouvre une fenêtre permettant d'exporter le dessin en cours dans un fichier au format `.svg`.

3 Tâches à effectuer

3.1 Réécriture classe `CanvasControllerContext`

Votre première tâche sera de réécrire la classe `CanvasControllerContext` du package `fr.univ_amu.l3mi.drawing_app.controller.canvas` afin d'utiliser le patron de conception *state*.

Le but sera de créer des classes correspondant aux trois états (pour le moment) possibles de l'interface :

- `ViewerMode` : mode visualisation
- `RectangleEdition` : mode édition rectangle avec un curseur en forme de croix qui permet de commencer la création d'un rectangle;
- `RectangleEditionClicked` : mode édition rectangle après un clic gauche de l'utilisateur qui affiche un rectangle entre la position du clic et le curseur de la souris et crée puis ajoute un rectangle au dessin lorsque l'utilisateur relâche le clic gauche.

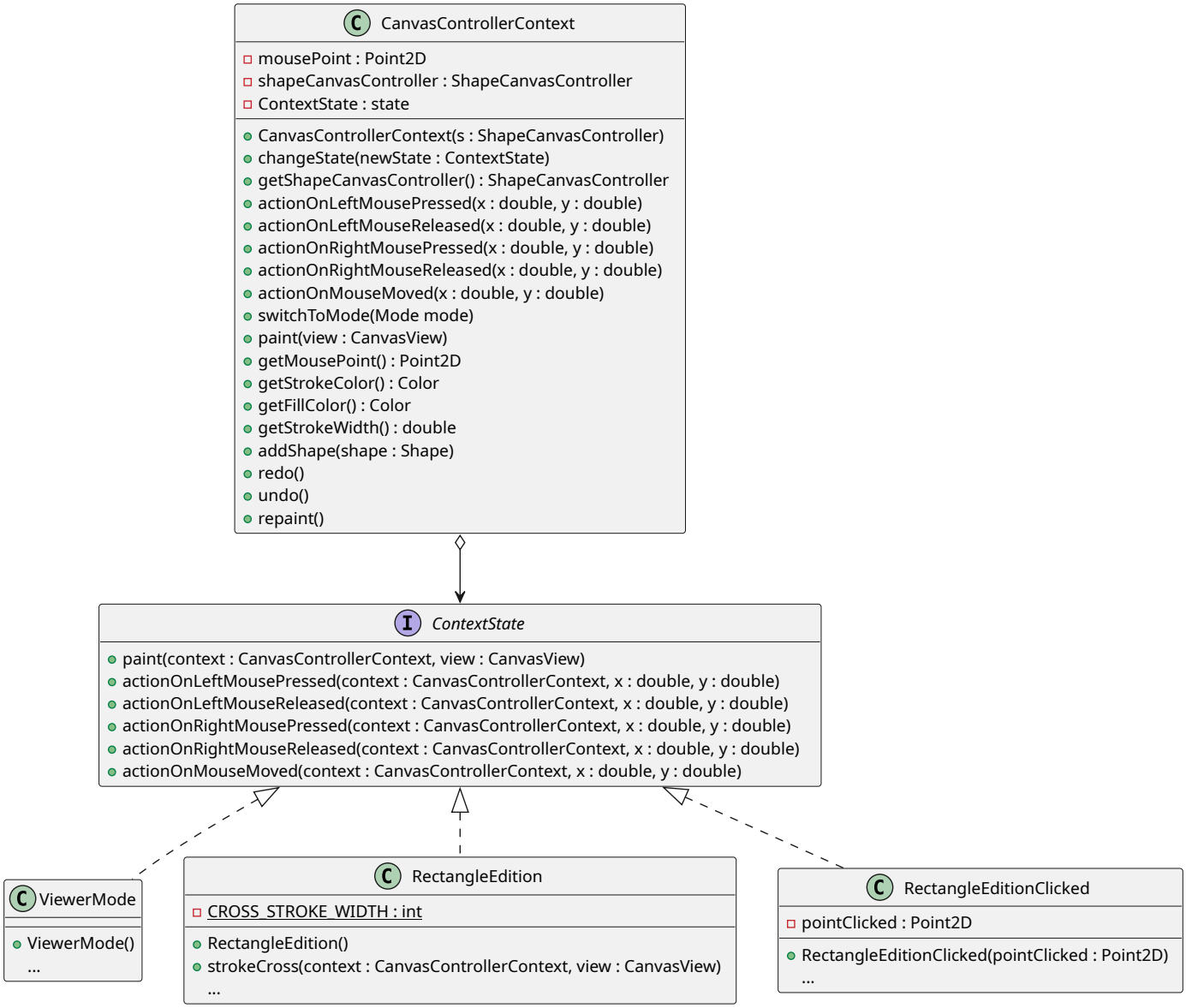
Un état devra implémenter l'interface suivante :

```
public interface ContextState{
    void paint(CanvasControllerContext context, CanvasView view);
    void actionOnLeftMousePressed(CanvasControllerContext context, double x, double y);
```

```
    void actionOnLeftMouseReleased(CanvasControllerContext context, double x, double y);
    void actionOnRightMousePressed(CanvasControllerContext context, double x, double y);
    void actionOnRightMouseReleased(CanvasControllerContext context, double x, double y);
    void actionOnMouseMoved(CanvasControllerContext context, double x, double y);
}
```

La méthode `paint` correspond à l'affichage particulier de l'état (affichage de la croix sur le curseur de la souris pour `RectangleEdition` et affichage du rectangle en cours de construction pour `RectangleEditionClicked`). Les autres méthodes correspondent aux actions à effectuer lorsque l'événement correspondant de la souris survient. Pour le moment, vous n'avez rien à faire pour les méthodes `actionOnRightMousePressed` et `actionOnRightMouseReleased` car le bouton droit de la souris n'est pas encore utilisé.

Le but est d'obtenir le même comportement que le code initial avec l'architecture suivante (les méthodes de l'interface `ContextState` ne sont pas explicitement écrite dans les classes implémentant l'interface afin de ne pas surcharger le diagramme de classe) :



Initialement, l'attribut `state` de `CanvasControllerContext` contiendra une instance de `ViewerMode`.

Tâche 1 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/controller/canvas` l'interface `ContextState` et les classes `ViewerMode`, `RectangleEdition` et `RectangleEditionClicked`, et modifiez la classe `CanvasControllerContext` afin d'obtenir l'architecture décrite par le diagramme.

3.2 Ajout de l'édition des cercles

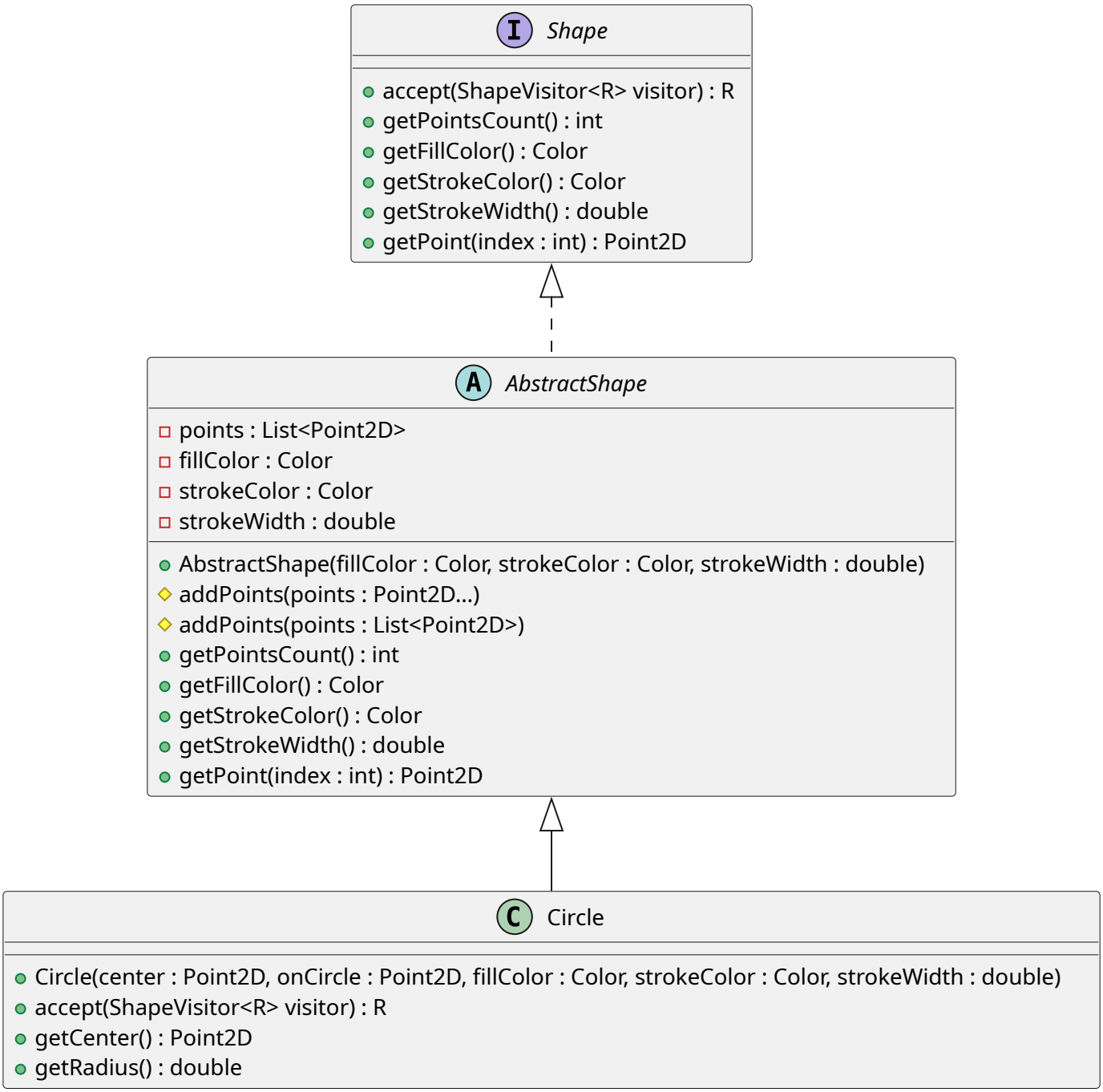
On souhaite pouvoir créer des cercles en plus des rectangles à l'aide d'une mode d'édition de cercle. Cette modification demande de faire plusieurs ajouts :

1. Ajouter une classe `Circle` dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model` similaire à la classe `Rectangle` mais permettant de représenter des cercles.
2. Ajouter des états (classes implémentant l'interface `ContextState`) permettant de gérer les états de l'interface pour les cercles.

3. Ajouter dans les visiteurs de `Shape` permettant l'écriture des fichiers des méthodes de visite pour les cercles.
4. Ajouter du code pour charger des cercles à partir d'un fichier.

3.2.1 Classe `Circle`

La première étape est de créer une classe `Circle` étendant la classe `AbstractShape` dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model`. Le but est d'avoir la classe `Circle` correspondant au diagramme suivant :



Pour le moment, mettez comme code de la méthode `accept` l'instruction suivante : `return null`.

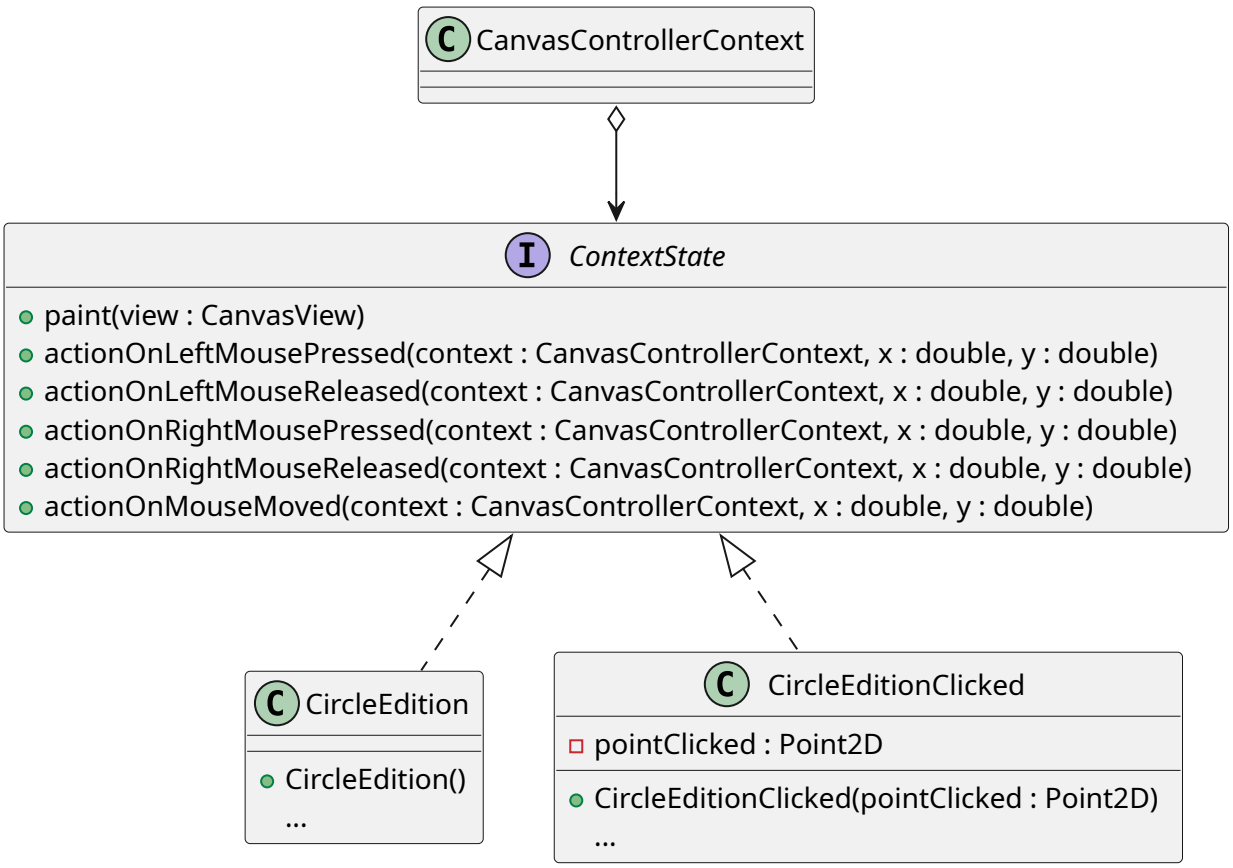
Tâche 2 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model` la classe `Circle` afin d'obtenir l'architecture décrite par le diagramme.

3.2.2 Classes `CircleEdition` et `CircleEditionClicked`

La deuxième étape est de rajouter des états afin d'avoir une interface utilisateur permettant d'éditer des cercles de manière similaire aux rectangles. Le mode *Circle* devra permettre de créer des cercles. Le mode de création des cercles sera similaire à celui des rectangles. Une croix devra être affichée avant le clic gauche. Lorsque l'utilisateur fait un clic gauche, il définit le centre du cercle et le point sur le cercle est défini lorsqu'il relâche le clic gauche créant ainsi le cercle. Un cercle est affiché tant que l'utilisateur ne relâche pas le clic gauche de manière similaire a ce qu'il se passe pour les rectangles. Pour dessiner les cercles, vous devez utiliser la méthode suivante de l'interface `CanvasView` :

```
/**
   * Draws a circle on the canvas.
   *
   * @param center the center point of the circle.
   * @param radius the radius of the circle.
   * @param fillColor the color used to fill the circle.
   * @param strokeColor the color of the circle's border.
   * @param strokeWidth the thickness of the circle's border.
   */
  void drawCircle(Point2D center, double radius, Color fillColor, Color strokeColor, double
  ↪  strokeWidth);
```

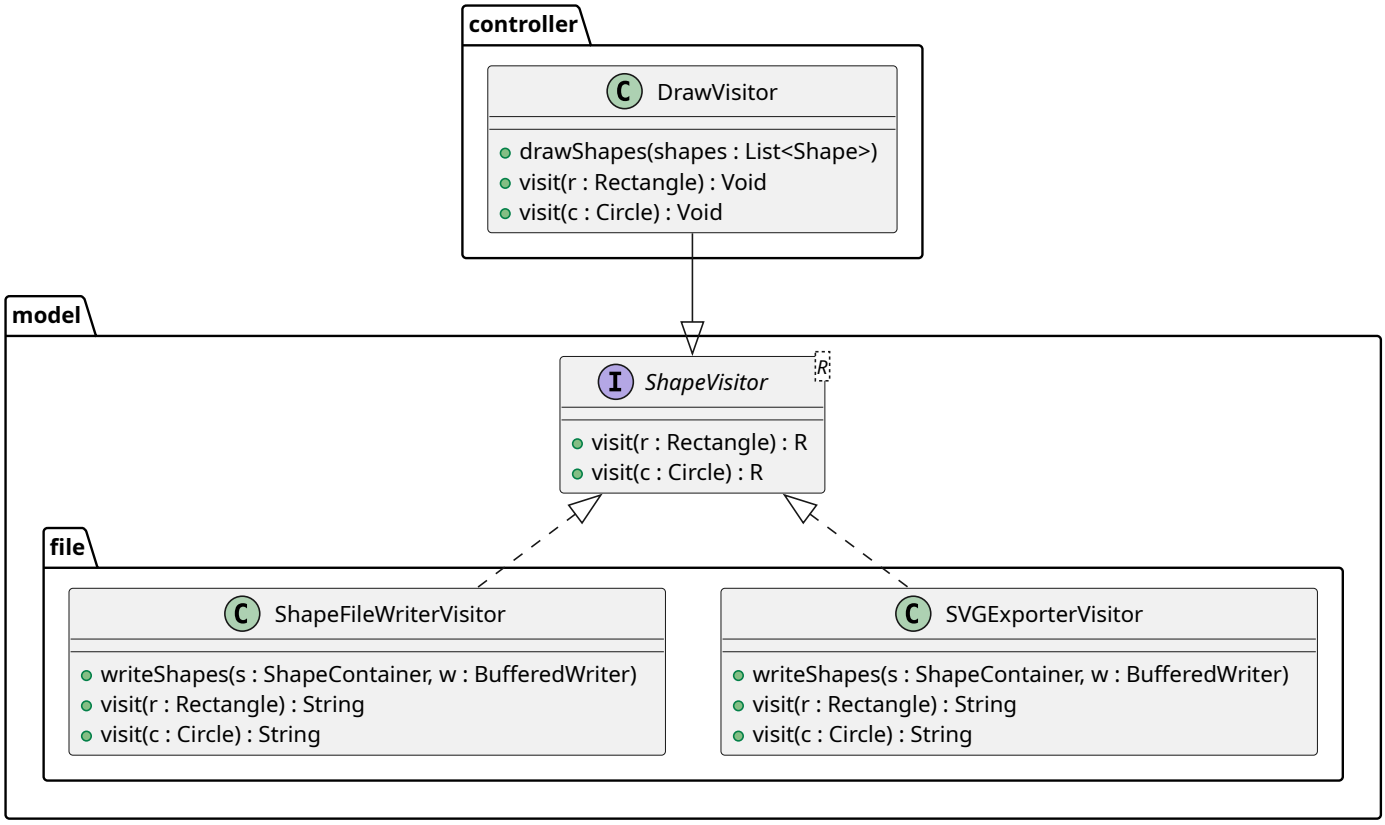
Le but est d'obtenir le diagramme de classe suivant :



Tâche 3 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/controller/canvas` les classes `CircleEdition` et `CircleEditionClicked` afin d'obtenir l'architecture décrite par le diagramme.

3.2.3 Modification des `ShapeVisitor`

La prochaine étape est de modifier les visiteurs implémentant `ShapeVisitor` dans `src/main/java/fr/univ_amu/l3mi/drawing_app/model` afin d'obtenir le diagramme suivant :



Il faudra aussi modifier la méthode `accept` dans `Circle` afin qu'elle appelle la méthode `visit(Circle c)`.

Pour la méthode `String visit(Circle c)` de `ShapeFileWriterVisitor`, vous devez renvoyer une chaîne de caractères commençant par `"Circle"` suivi des informations du cercle séparées par des espaces : x et y du centre, la valeur du rayon, les deux couleurs *fill* et *stroke* et la largeur du trait. Pour la méthode `String visit(Circle c)` de `SVGExporterVisitor`, vous renvoyer une chaîne de caractère correspondant à la balise `<circle>` du format SVG.

Tâche 4 : Rajoutez dans l'interface `ShapeVisitor` la méthode `R visit(Circle c)`, rajoutez dans les classes `ShapeFileWriterVisitor` et `SVGExporterVisitor` la méthode `String visit(Circle c)` et modifiez la méthode `accept` de `Circle` afin qu'elle appelle `visit(Circle c)`.

3.2.4 Modification de `NaiveShapeFileReader`

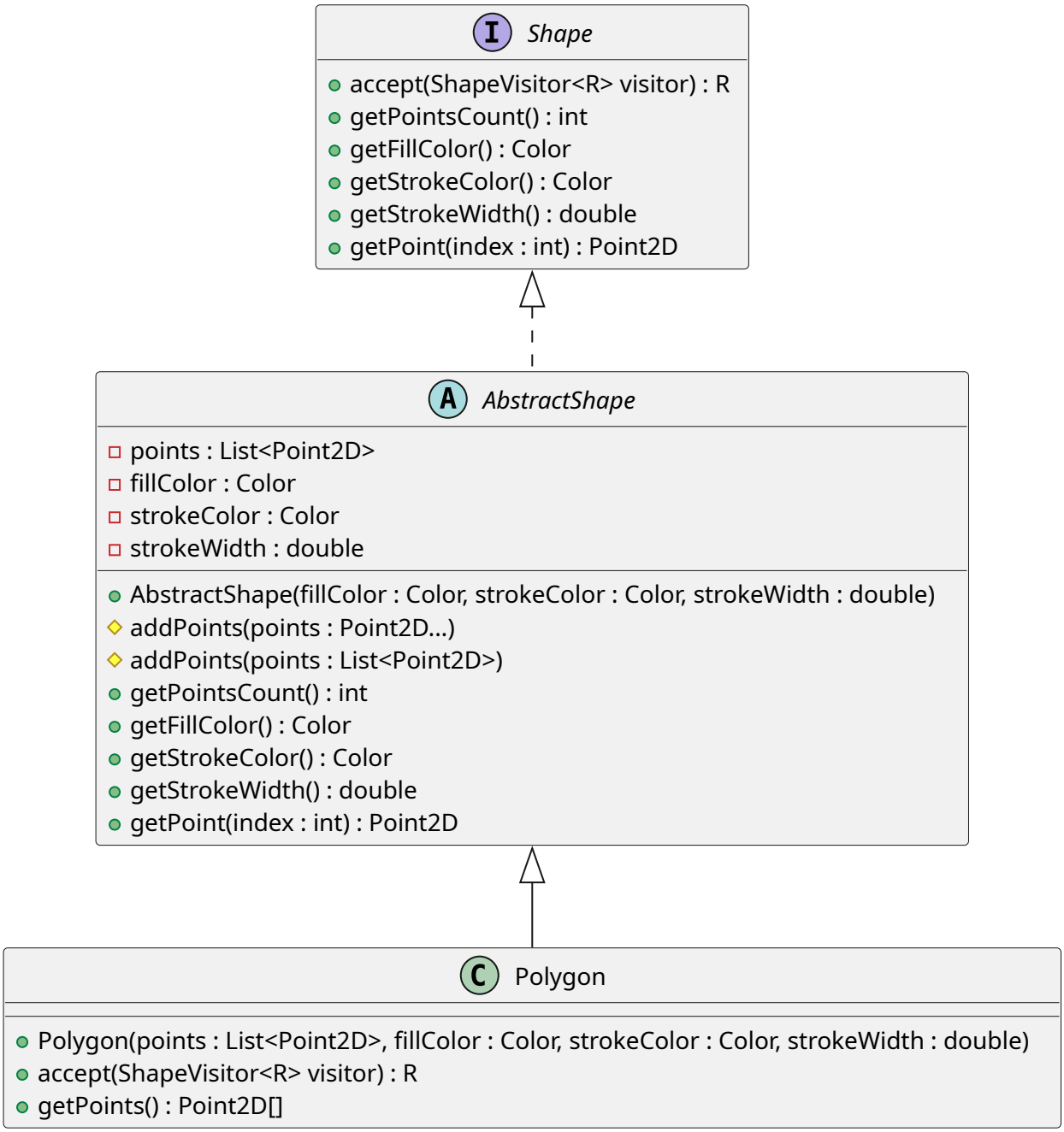
Il ne reste plus qu'à modifier la classe `NaiveShapeFileReader` dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model/file` afin de prendre en compte les cercles. Il vous faut modifier la méthode `readShapes` afin de faire un traitement spécifique pour les lignes commençant par `"Circle"`. La création et le rajout du cercle sera à faire dans une méthode `void readCircle(String[] tokens, ShapeContainer shapeContainer)` similaire à `readRectangle`.

Tâche 5 : Modifiez la classe `NaiveShapeFileReader` afin de prendre en compte les cercles.

3.3 Ajout de l'édition de polygones

De manière similaire à ce que vous avez fait pour les cercles, vous allez rajouter du code pour permettre l'édition de polygone.

La première étape est de créer une classe `Polygon` correspondant au diagramme suivant :



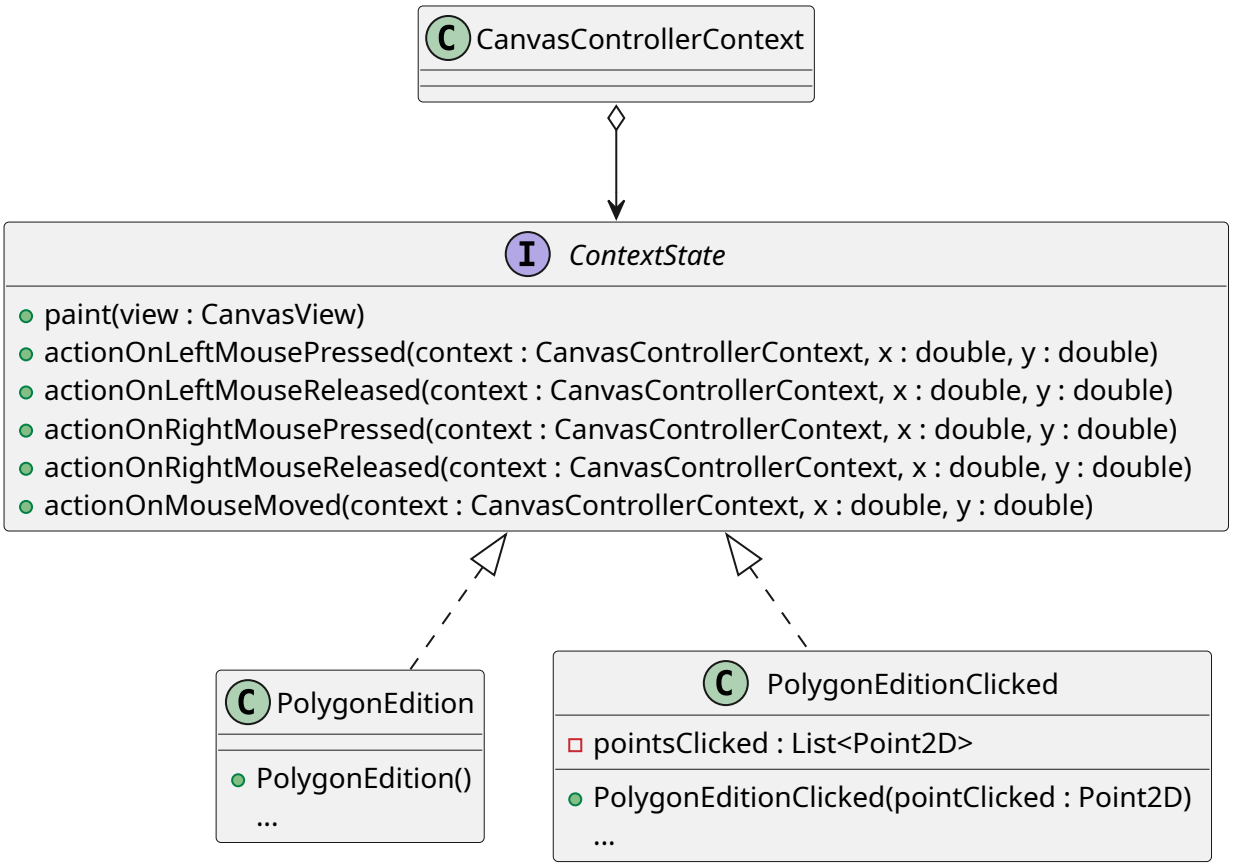
Tâche 6 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model` la classe `Polygon` étendant la classe `AbstractShape` (mettez `return null` comme code de la méthode `accept` de `Polygon`).

La deuxième étape est de rajouter des classes pour les états de l'interface pour la création de polygone. L'interface utilisateur sera la suivante : chaque clic gauche créera un point du polygone et un clic droit finira la création du

polygone en définissant le dernier point. Une croix devra être affichée durant l'édition et le polygone en cours de construction devra aussi être affiché. Pour dessiner les polygones, vous devez utiliser la méthode suivante de l'interface `CanvasView` :

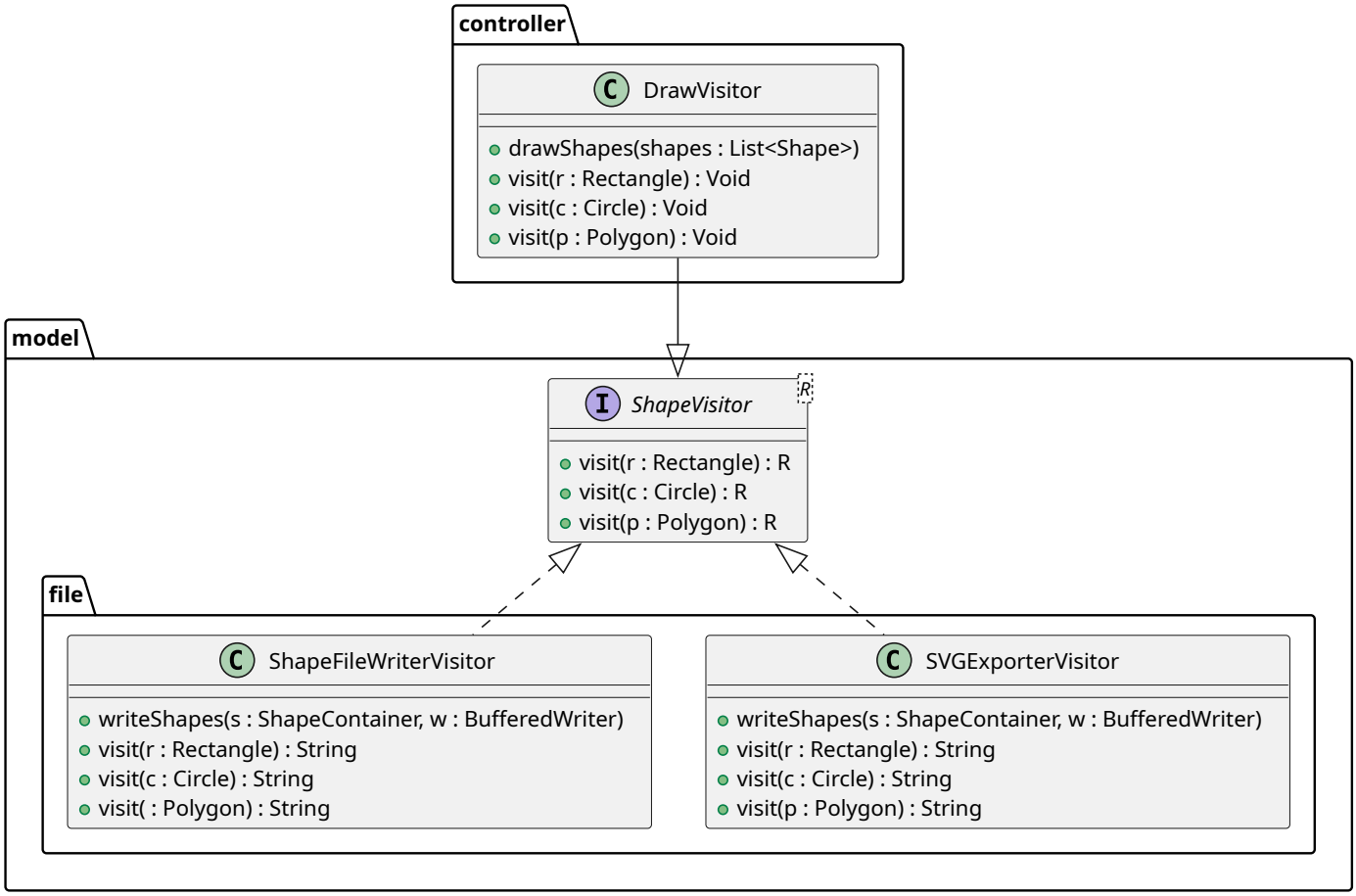
```
/**
 * Draws a polygon on the canvas.
 *
 * @param points an array of points representing the vertices of the polygon.
 * @param fillColor the color used to fill the polygon.
 * @param strokeColor the color of the polygon's border.
 * @param strokeWidth the thickness of the polygon's border.
 */
void drawPolygon(Point2D[] points, Color fillColor, Color strokeColor, double strokeWidth);
```

Le but est d'obtenir le diagramme de classe suivant :



Tâche 7 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/controller/canvas` les classes `PolygonEdition` et `PolygonEditionClicked` correspondant au diagramme ci-dessus.

La prochaine étape est de modifier les visiteurs implémentant `ShapeVisitor` dans `src/main/java/fr/univ_amu/l3mi/drawing_app/model` afin d'obtenir le diagramme suivant :



Il faudra aussi modifier la méthode `accept` dans `Polygon` afin qu'elle appelle la méthode `visit(Polygon p)`.

Pour la méthode `String visit(Polygon p)` de `ShapeFileWriterVisitor`, vous devez renvoyer une chaîne de caractères commençant par `"Polygon"` suivi des informations du polygone séparées par des espaces : les coordonnées x et y des points du polygone, les deux couleurs *fill* et *stroke* et la largeur du trait. Pour la méthode `String visit(Polygon p)` de `SVGExporterVisitor`, vous renvoyer une chaîne de caractère correspondant à la balise `<polygon>` du format SVG.

Tâche 8 : Rajoutez dans l'interface `ShapeVisitor` la méthode `R visit(Polygon p)`, rajoutez dans les classes `ShapeFileWriterVisitor` et `SVGExporterVisitor` la méthode `String visit(Polygon p)` et modifiez la méthode `accept` de `Polygone` afin qu'elle appelle `visit(Polygon p)`.

Il ne reste plus qu'à modifier la classe `NaiveShapeFileReader` dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model/file` afin de prendre en compte les polygones. Il vous faut modifier la méthode `readShapes` afin de faire un traitement spécifique pour les lignes commençant par `"Polygon"`. La création et le rajout du polygone sera à faire dans une méthode `void readPolygon(String[] tokens, ShapeContainer shapeContainer)` similaire à `readRectangle`.

Tâche 9 : Modifiez la classe `NaiveShapeFileReader` afin de prendre en compte les polygones.

3.4 Ajout du mode de suppression de formes

Le but de cette partie est de permettre la suppression de forme. Pour cela, il faut :

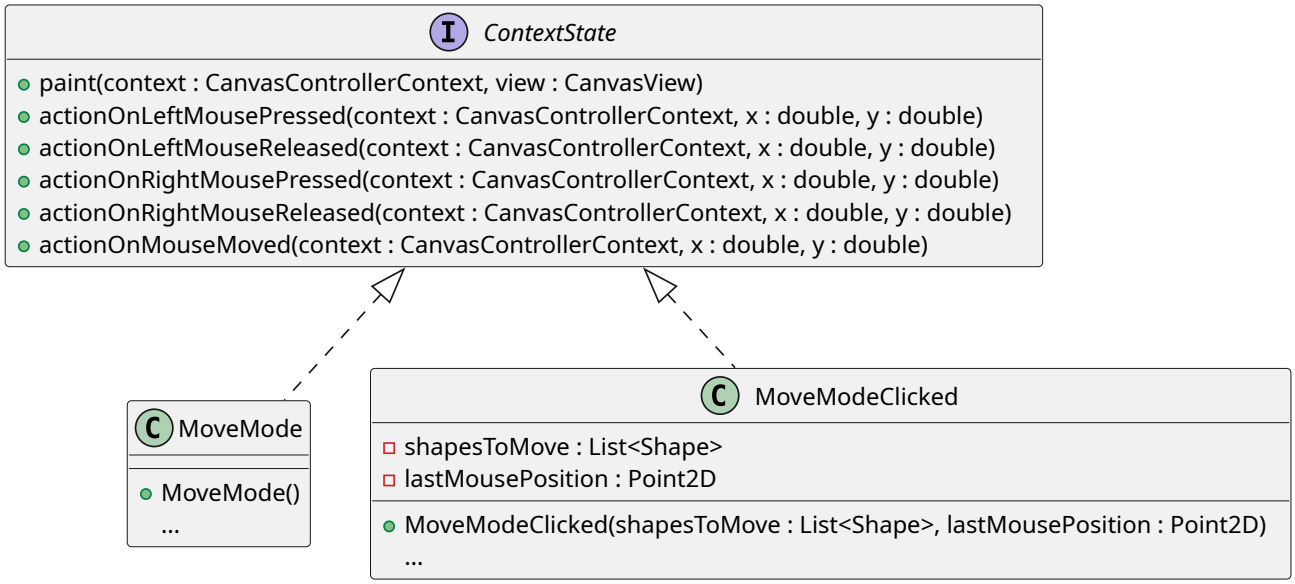
1. Rajoutez une méthode `boolean contains(Point2D point)` dans l'interface `Shape` ;
2. Implémentez la méthode `public boolean contains(Point2D point)` dans les classes `Circle`, `Rectangle` et `Polygon` afin qu'elle renvoie `true` si et seulement si `point` est à l'intérieur de la forme. On pourra utiliser la méthode `contains` de la classe `javafx.scene.shape.Polygon` pour implémenter `contains` de votre classe `Polygon`.
3. Ajoutez dans la classe `ShapeContainer` du package `model` une méthode `public List<Shape> shapesContaining(Point2D point)` renvoyant toutes les formes de l'attribut `shapes` qui contiennent `point`.
4. Ajoutez dans la classe `ShapeContainer` du package `model` une méthode `public void remove(Shape shape)` supprimant la forme `shape` du `ShapeContainer`.
5. Ajoutez dans la classe `ShapeCanvasController` du package `controller.canvas` les méthodes `public List<Shape> shapesContaining(Point2D point)` et `public void remove(Shape shape)` appelant les méthodes correspondante sur son attribut `shapeContainer`.
6. Ajoutez dans la classe `ShapeCanvasControllerContext` du package `controller.canvas` les méthodes `public List<Shape> shapesContaining(Point2D point)` et `public void remove(Shape shape)` appelant les méthodes correspondante sur son attribut `shapeCanvasController`.
7. Ajoutez une classe `DeleteMode` implémentant `ContextState` correspondant à un mode ou tout clic gauche supprime les formes contenant le point cliqué.
8. Modifiez la classe `CanvasControllerContext` afin d'utiliser la classe `DeleteMode`.

Tâche 10 : Effectuez les modifications décrites ci-dessus.

3.5 Ajout du mode de déplacement de formes

Le but de cette partie est de permettre le déplacement de forme. Pour cela, il faut :

1. Ajouter la méthode `void translate(dx : double, dy : double)` à l'interface `Shape` ;
2. Implémentez la méthode `public void translate(dx : double, dy : double)` dans la classe `AbstractShape` afin qu'elle modifie la forme en lui appliquant une translation de vecteur (`dx`, `dy`) ;
3. Ajoutez deux classes `MoveMode` et `MoveModeClicked` implémentant `ContextState` correspondant à un mode ou tout clic gauche permettre à l'utilisateur de déplacer une forme en la faisant glisser avec la souris tant que le clic gauche est appuyé. Lorsque le clic gauche est relâché le déplacement est finalisé. L'affichage devra permettre de voir les formes se déplacer tant que le clic est maintenu.



4. Modifiez la classe `CanvasControllerContext` afin d'utiliser la classe `MoveMode`.

Tâche 11 : Effectuez les modifications décrites ci-dessus.

3.6 Réécriture lecture/écriture de fichiers

La lecture/écriture de fichier n'est pas très satisfaisante, car le format d'enregistrement de chaque forme est dans deux classes distinctes (`ShapeFileWriterVisitor` et `NaiveShapeFileReader`) ce qui demande donc de modifier les deux classes lorsqu'on souhaite faire un changement. Vous allez donc remédier à cela en créant des classes de sérialisation pour chaque classe de forme. On va tout d'abord créer une interface `ShapeSerializer` qui va nous permettre de regrouper, pour chaque type d'objet, son écriture (sérialisation) et sa lecture (désérialisation) dans une même classe.

L'interface `ShapeSerializer<S extends Shape>` est une interface paramétrée par un type `S` qui correspond au type de l'objet à sérialiser ou désérialiser. Cette interface contient les trois méthodes suivantes :

- `getCode` : qui renvoie un identifiant de l'objet à sérialiser ou désérialiser, c'est-à-dire le nom de sa classe (`"Circle"`, `"Rectangle"` ou `"Polygon"`) ;
- `serialize` : retourne une chaîne qui décrit l'objet à sérialiser ;
- `deserialize` : reconstruit un objet à partir d'une chaîne de caractères qui le décrit (généralement une ligne du fichier qui a été produite par la méthode `serialize`).

Plus précisément, l'interface est la suivante :

```
public interface ShapeSerializer<S extends Shape> {
  String getCode();
  String serialize(S shape);
  S deserialize(String[] tokens);
}
```

Cette interface sera implémentée par les classes suivantes :

— `RectangleSerializer extends ShapeSerializer<Rectangle>`
— `CircleSerializer extends ShapeSerializer<Circle>`
— `PolygonSerializer extends ShapeSerializer<Polygon>`

Chacune de ces classes utilisera le code écrit dans le visiteur `ShapeFileWriterVisitor` pour la méthode `serialize` et le code de la classe `NaiveShapeFileReader` pour la méthode `deserialize`.

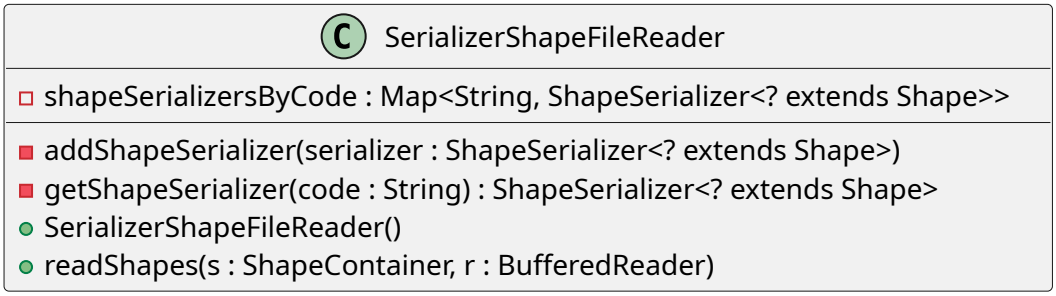
Chacune des classes `RectangleSerializer`, `CircleSerializer` et `PolygonSerializer` devra implémenter le patron de conception *singleton*, c'est-à-dire avoir :

— un constructeur privé (dans notre cas il n'aura pas de paramètres) ;
— un attribut de classe (`static`) de type de la classe non instancié ;
— une méthode de classe (`static`) `getInstance()` qui instancie l'attribut de la classe s'il est `null` et le renvoie dans tous les cas.

Tâche 12 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/model/file` l'interface `ShapeSerializer` et les classes `RectangleSerializer`, `CircleSerializer` et `PolygonSerializer`.

Il faut maintenant modifier les deux classes `ShapeFileWriterVisitor` et `NaiveShapeFileReader` afin d'utiliser les `Serializer` :

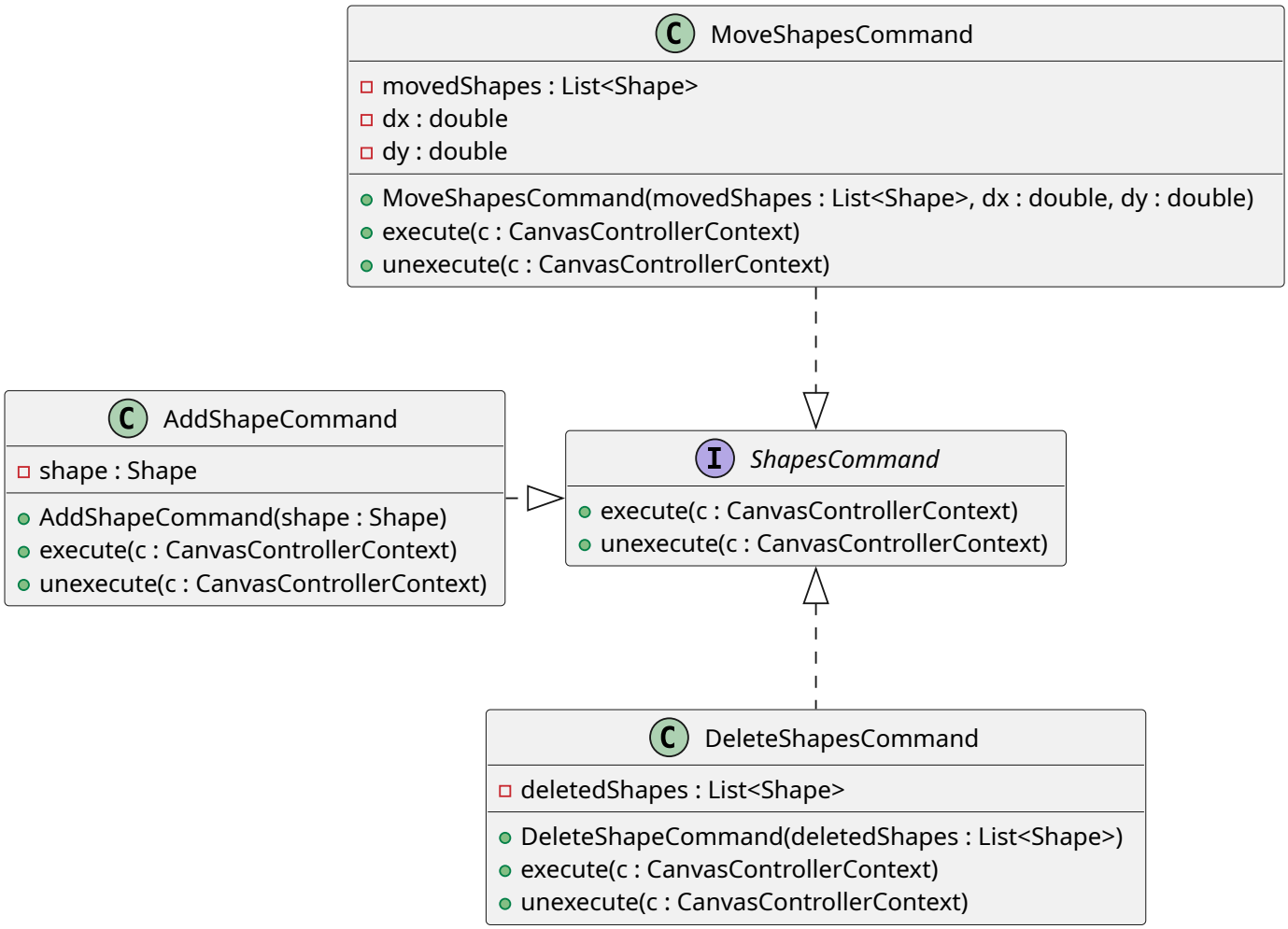
— Pour `ShapeFileWriterVisitor`, il suffit d'utiliser le bon *serializer* pour implémenter chaque méthode `visit` ;
— Pour `NaiveShapeFileReader`, il faut construire un `HashMap` associant chaque *serializer* à son code (le nom de la classe du *serializer*) dans le code du constructeur et utiliser ce `HashMap` dans `readShapes` afin d'obtenir le *serializer* correspondant au premier mot de la ligne lorsqu'on lit une ligne correspondant à une forme. Vous renommerez la classe `NaiveShapeFileReader` en `SerializerShapeFileReader` afin d'obtenir une classe correspondant au diagramme suivant :



Tâche 13 : Modifiez les deux classes `ShapeFileWriterVisitor` et `NaiveShapeFileReader` afin d'utiliser les `Serializer` en suivant les consignes ci-dessus.

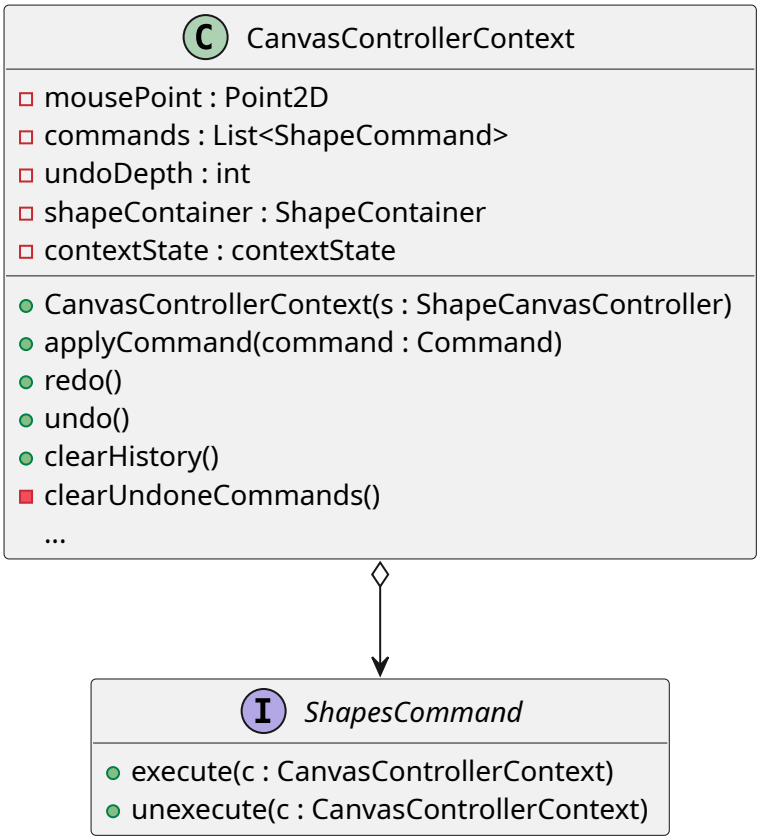
3.7 Fonctionnalités *undo* et *redo*

On cherche à rajouter un *undo* et *redo* permettant d'annuler et de rétablir des opérations sur les formes : créations, suppressions et déplacement. Pour cela, on va utiliser le patron de conception *command*. La première étape sera donc de modéliser les actions de l'utilisateur via des classes implémentant une même interface `ShapesCommand`. On aura donc les classes décrites dans le diagramme suivant :



Tâche 14 : Créez dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/controller/canvas` l'interface `ShapeCommand` et les classes `MoveShapesCommand`, `AddShapeCommand` et `DeleteShapesCommand` correspondant au diagramme ci-dessus.

Maintenant, il ne reste plus qu'à modifier la classe `CanvasControllerContext` et les classes implémentant `ContextState` pour qu'elles utilisent les commandes selon le diagramme ci-dessous :



Tâche 15 : Modifiez la classe `CanvasControllerContext` ainsi que toutes les classes implémentant l'interface `ContextState` (`RectangleEditionClicked`, `CircleEditionClicked`, `PolygonEditionClicked`, `DeletionMode` et `MoveMode`) afin qu'elles utilisent les classes `DeleteShapesCommand`, `AddShapeCommand` et `MoveShapesCommand`.

3.8 Fonctionnalités supplémentaires

Une fois que vous avez terminé toutes les tâches précédentes, vous pouvez travailler sur les nouvelles fonctionnalités suivantes. Pour rajouter un mode, il vous faut ajouter une valeur à l'`enum Mode` dans le répertoire `src/main/java/fr/univ_amu/l3mi/drawing_app/controller/canvas` et modifier la méthode `switchToMode` de la classe `CanvasControllerContext`. Pour rajouter un bouton, il vous faut modifier les classes `DrawingApp` (en rajoutant un bouton dans la liste) et modifier la méthode `void buttonActionOnClick(String buttonId)` de la classe `DrawingAppController`.

Les fonctionnalités possibles sont :

- L'ajout d'un mode *Update* qui mettrait à jour les formes sur lesquelles on clique en changeant leurs attributs de couleurs et d'épaisseur de trait pour leur donner les valeurs courante dans la barre de menu ;
- L'ajout d'un mode *Rotate* qui permettrait d'appliquer des rotations sur les figures, le clic gauche permet de sélectionner les figures, un premier clic droit définit le centre de la rotation et le bouton droit est ensuite utilisé pour définir un angle en le laissant appuyé pour faire tourner les figures ;
- L'ajout d'un mode d'édition de forme qui permettrait de bouger les points des formes déjà dessinées ;
- L'ajout de l'édition de texte : cela demande de modifier `CanvasView`, `JavaFXDrawingAppView` et

`DrawingCanvasView` afin de rajouter une méthode permettant d'écrire du texte.

— L'ajout d'un export en format bitmap comme *PPM*. Cela demande de calculer les couleurs en appliquant les transparences. Pour cela on peut utiliser la formule suivante qui permet de calculer la couleur résultat o issue d'une couleur c de transparence α sur une couleur de fond b : $o = b \times (1 - \alpha) + c \times \alpha$

— Toutes autres fonctionnalités que l'enseignant en charge valide.