TP3 : Manipulation directe

Ce TP est consacré au développement d'interactions de manipulation directe. **Durée : 8h**

Objectifs

- Dessiner des formes géométriques en utilisant les classes *JavaFX* de haut niveau permettant des interactions,
- Utiliser des machines à état dans la programmation d'interactions,
- Programmer les interactions de manipulation directe suivantes : *pan*, *drag*, *zoom* centré souris standard et zoom centré souris avec grossissement variable selon les composants (*zoom* différencié).

Documentation

Toute la doc citée pour les TP1 et 2,

Applying Transformations in *JavaFX*
http://docs.oracle.com/javase/8/javafx/visual-effects-tutorial/transforms.htm#CHDGCBAH

Exercice 1 : Architecture, graphe de scène et formes géométriques

1.1 Créez un nouveau projet *JavaFX*, importez et étudiez le code fourni.

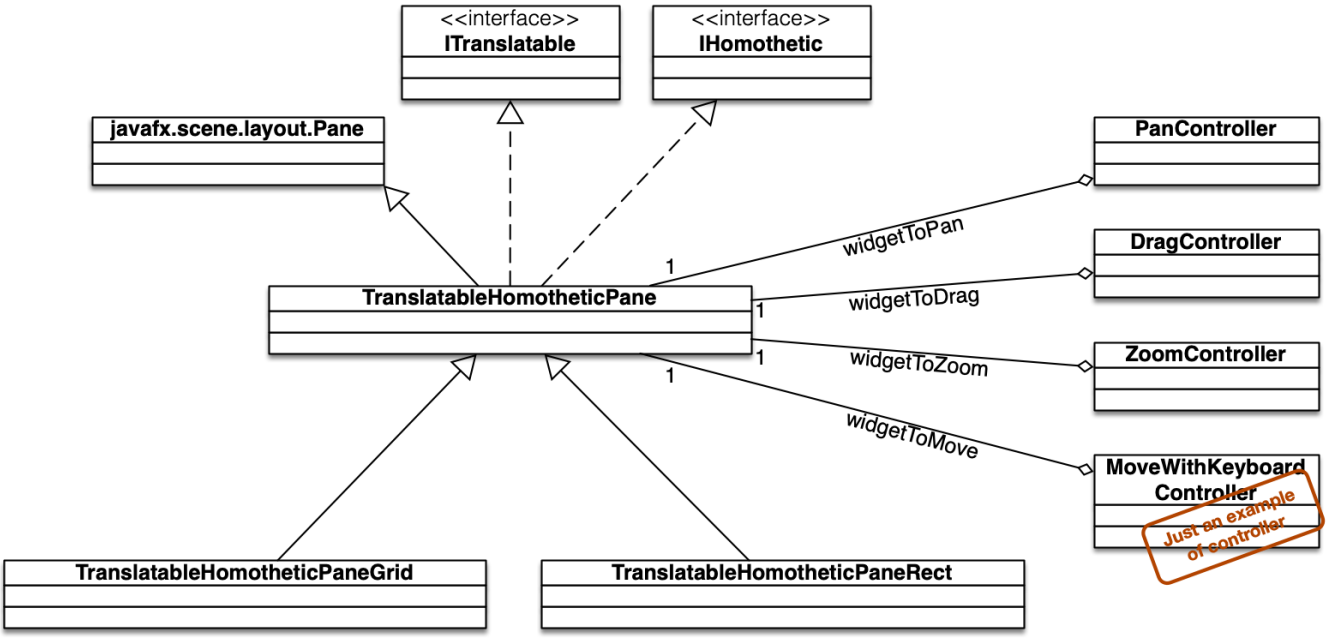
Ce code servira de base pour créer des interactions de manipulation directe d'une ou plusieurs formes graphiques dans un conteneur pouvant lui-même être manipulé. Les manipulations seront les suivantes :

- *pan* du conteneur (translation du conteneur, et donc de tout ce qu'il contient, par manipulation directe avec la souris),
- *drag* des formes (translation indépendamment du conteneur par manipulation directe avec la souris),
- *zoom* (du conteneur) centré sur la position pointée par la souris et différencié selon les widgets.

L'application aura les spécifications générales suivantes :

- les différentes interactions seront gérées dans les classes `PanController`, `DragController` et `ZoomController`,
- le conteneur devra afficher une grille en arrière-plan et sera implémenté sous forme d'une classe nommée `TranslatableHomotheticPaneGrid`,
- la forme graphique à manipuler sera un rectangle et sera implémentée sous forme d'une classe nommée `TranslatableHomotheticPaneRect`.

Analysez l'extrait de diagramme de classes ci-dessous présentant schématiquement les classes fournies (qui seront à compléter au cours du TP) ainsi que quelques autres (qui restent à créer). Expliquez à quoi servent les interfaces introduites ici, et pourquoi il y a 3 niveaux d'héritage pour les widgets.



1.2 La suite de l'exercice est consacrée au dessin de la grille dans la classe `TranslatableHomotheticPaneGrid`. Pour préparer ce travail parcourez la javadoc du package `javafx.scene.shape` et choisissez la ou les classes d'objets graphiques utilisables pour dessiner la grille.

Sachant que vous devrez intégrer la grille dans un `Group` (un autre type de conteneur *JavaFX*), lui-même intégré dans le graphe de scène de la classe, dessinez le graphe de scène complet de l'application et annotez en regard de chaque composant ses coordonnées dans le repère de son parent.

1.3 Le travail préparatoire de la question 1.2 étant mené, codez maintenant la classe `TranslatableHomotheticPaneGrid`.

Dans la suite de ce TP, pour éviter des lourdeurs dans l'énoncé, chaque mention à la grille évoquera l'instance de la classe contenant la grille (`TranslatableHomotheticPaneGrid`) et non plus les lignes qui constituent la grille elle-même.

Exercice 2 : *Pan* de la grille (version classique)

2.1 Positionnement et translation

Chaque nœud du graphe de scène est positionné dans le repère de son parent grâce à deux mécanismes qui font la même chose, à savoir placer le nœud en x,y (diapo 31 du cours) mais sont offerts par commodité : l'un incarne un positionnement absolu (attributs *layoutX* et *layoutY*), et l'autre est offert pour y cumuler les translations que l'on voudrait apporter ultérieurement (attributs *translateX* et translate*Y*).

Dans `DragPanZoomApplication`, le mécanisme associé au layout est bien utilisé pour positionner les différents nœuds lors de leur création. En revanche, pour les translations exigées par les interactions de *Pan* et de *Drag* que vous allez réaliser dans les exercices qui suivent, vous utiliserez plutôt le mécanisme de translation fourni par les matrices de transformation (diapo 40 du cours).

A partir du cours identifiez les méthodes associées à ces mécanismes et la façon dont ils sont utilisés dans le code fourni (et devront continuer à l'être pour les interactions à implémenter).

2.2 A l'aide de la logique du *pan* (décrite ci-après) et de la doc de `MouseEvent`, déterminez quels sont les différents types de cet événement (les constantes `MouseEvent.MOUSE_`...) ainsi que les informations convoyées potentiellement utiles pour gérer le *pan*.

Logique du *pan* :

- lorsque le bouton de la souris est pressé, sa position P_0 doit être mémorisée,
- lorsque la souris est déplacée vers le point P_1, l'objet écouté doit être translaté d'un vecteur P_0P_1,
- si les coordonnées de la souris ont pu être récupérées dans le repère de l'objet translaté il n'y a plus rien à faire. Si en revanche elles ont été récupérées dans un autre repère (scène, écran...), les coordonnées de P_1 doivent alors être mémorisées dans P_0 pour servir de nouvelle base au prochain déplacement de souris. Par ailleurs, avec des coordonnées non locales, il faudra aussi tenir compte par la suite du facteur de *zoom* sinon le *pan* sera cassé. Il vaut donc vraiment mieux privilégier la récupération des coordonnées dans le repère local de l'objet translaté, ce qu'on peut faire en *JavaFX*…

2.3 Dans le package `javafxdragpanzoom.view.controls` créez la classe `PanController` chargée de gérer l'interaction de *pan* de la grille. Pour ce faire vous utiliserez des instances de classes implémentant l'interface `EventHandler<MouseEvent>` et enregistrerez ces instances comme des filtres sur les différents types de `MouseEvent` émis par la grille grâce à la méthode d'abonnement `addEventFilter` (diapo 76 du cours).

2.4 Reprenez la classe `DragPanZoomApplication` et intégrez y une instance de votre nouvelle classe `PanController` associée à votre grille. Testez votre application. Que se passe-t-il ?

Indice : la classe `TranslatableHomotheticPane` qui vous est fournie n'est en fait que partiellement implémentée (suffisamment pour que la première version de l'application fonctionne dans l'exercice 1). Dans cette première version, toutes les méthodes spécifiées par l'interface `ITranslatable` devaient

absolument être implémentées si on voulait que l'application soit utilisable. Ces méthodes auraient pu être implémentées vides mais le choix qui a été fait ici est plutôt de leur faire propager une exception. Cette exception est elle sous contrôle du compilateur ou pas ? Expliquez la raison de cette façon de faire.

2.5 Reprenez le code de `TranslatableHomotheticPane` et implémentez y la fonction nécessaire au fonctionnement du *pan*.

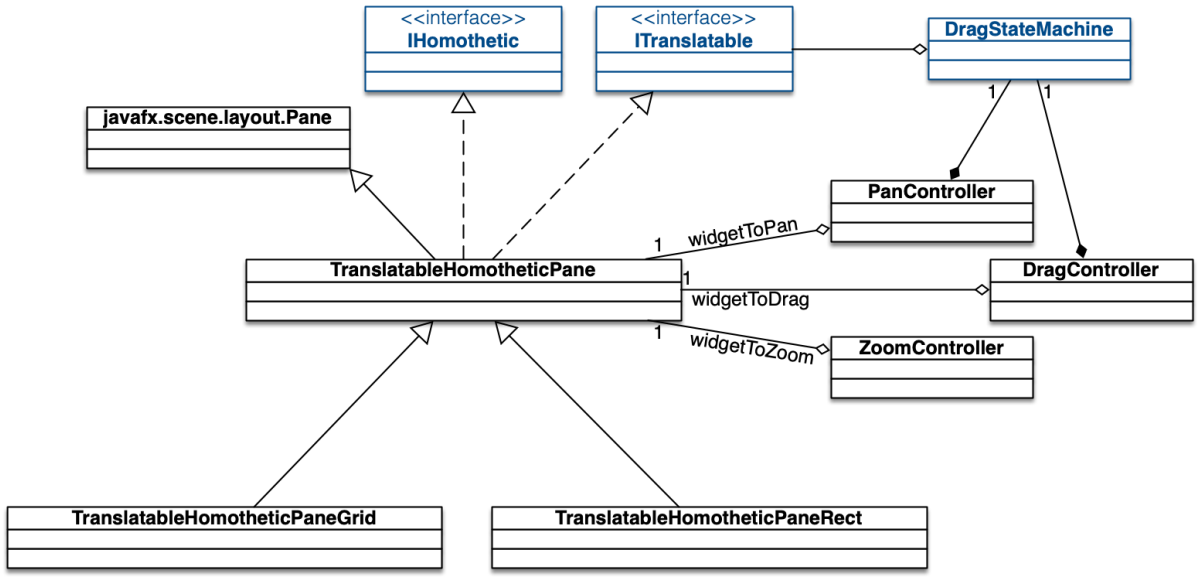
Exercice 3 : *Pan* de la grille (version avec machine à états)

3.1 Félicitations vous venez juste de réaliser votre interaction de *pan*. Malheureusement votre code est intégralement lié à *JavaFX*, ce qui, en termes d'architecture logicielle est peu satisfaisant puisque la logique de cette interaction est toujours la même quelle que soit la bibliothèque graphique utilisée. Pour des raisons de réutilisabilité il serait donc plus satisfaisant de créer un mécanisme logique agnostique en termes de bibliothèque graphique, et de l'utiliser conjointement avec la bibliothèque de votre choix.

Par ailleurs, et comme vous l'avez vu en cours, toutes les logiques d'interaction sont modélisables grâce à des *états* (graphiques ou pas) et des *transitions* entre ces états (actions à réaliser), activées sur réception d'un événement particulier. C'est ce qu'on appelle une *machine à états*. Dans n'importe quel langage informatique les transitions peuvent très bien être décrites à l'aide de structures de contrôle standards (des `if` ou des `switch` imbriqués) mais cela a pour inconvénient de mêler de façon peu lisible les tests (conditions pour effectuer une transition), les actions à effectuer lors d'une transition et les changements d'état. Vous utiliserez donc plutôt une bibliothèque fournie avec le code de l'exercice (package `fr.liienac.statemachine`).

Pour pouvoir utiliser cette bibliothèque facilement, lisez tout d'abord attentivement la documentation « StateMachine API » disponible avec le cours sur e-campus, puis dessinez sur papier la machine à états associée à l'interaction de *pan*.

3.2 Passez maintenant au code : créez dans le package `javafxdragpanzoom.statemachines` une nouvelle classe `DragStateMachine` (vous verrez dans le prochain exercice pourquoi elle s'appelle ainsi) héritant de la classe `fr.liienac.statemachine.StateMachine`. Comme précisé dans la question 1, pour être agnostique, cette classe ne devra pas contenir de code *JavaFX*. Elle aura donc la charge de manipuler un `ITranslatable` grâce à sa méthode `translate`, conformément au nouveau diagramme de classe ci-dessous, et se verra envoyer (depuis `PanController`) uniquement des événements abstraits, comme ceux définis dans le package `event` de la bibliothèque `statemachine`. De cette manière, toute la partie bleue du diagramme sera directement réutilisable avec une autre bibliothèque graphique.



3.3 Améliorez votre machine à états en y introduisant une hystérèse (la translation de l'objet manipulé ne commence que si le déplacement de la souris depuis le *press* est supérieur à un certain seuil).

Exercice 4 : *Drag* du rectangle

4.1 On souhaite maintenant réaliser l'interaction de *drag* (déplacer le rectangle au sein de la grille dans laquelle il se trouve).

En considérant que *pan* et *drag* sont similaires (translation d'un nœud du graphe par manipulation directe), mettez en place ce dernier dans une nouvelle classe `DragController` qui utilisera la machine à états déjà réalisée pour le *pan* de la grille (d'où son nom `DragStateMachine`).

4.2 Testez votre *drag*. Il semble y avoir un conflit entre plusieurs interactions. D'où provient il et comment le gérer ? Décrivez deux solutions distinctes (pensez aux différentes propriétés d'un événement et au cycle de propagation décrits en cours).

4.3 Implémentez l'une des deux solutions à votre convenance.

Exercice 5 : *Zoom* basique

5.1 Dans la même logique que ce qui a été fait pour la méthode `translate(double dx, double dy)` dans les questions 2.4 et 2.5 implémentez la méthode `appendScale(double deltaScale)` (lisez bien la *javadoc* de l'interface `IHomothetic` et les commentaires dans la classe `TranslatableHomotheticPane` pour le faire correctement, avec les matrices de transformation).

5.2 Testez le résultat en modifiant l'écouteur des événements clavier dans la classe `MoveWithKeyboardController` pour zoomer lorsqu'on appuie sur une touche de votre choix.

Quel est le centre de la transformation ainsi réalisée ?

5.3 A l'aide de la documentation de `ScrollEvent` déterminez quel est le type de cet événement ainsi que les informations convoyées potentiellement utiles pour gérer un *zoom* de la grille par action sur la molette de la souris.

Justifiez l'interêt (ou pas) d'une machine à états pour gérer cette interaction.

5.4 Créez la classe `ZoomController` pour gérer le *zoom* de la grille (comme pour les autres contrôleurs d'interaction, dans le package `javafxdragpanzoom.view.control`).

Avec toujours le même centre de transformation est ce une interaction pratique ? Que pourrait on préférer avoir ?

Exercice 6 : *Zoom* centré souris

6.1 Implémentez maintenant la méthode `appendScale(double scale, double x, double y)` permettant de mettre à l'échelle par rapport à un point quelconque et testez la en choisissant un point arbitraire facile à repérer (un des coins de la grille par exemple).

6.2 A l'aide de la documentation de `ScrollEvent` déterminez les informations supplémentaires à exploiter pour gérer un zoom centré souris.

6.3 Améliorez le *zoom* réalisé à l'exercice 4.

Exercice 7 : *Zoom* différencié

7.1 L'utilisation du *zoom* réalisé précédemment entraîne quel problème de visualisation ?

Quelle solution peut on envisager ?

7.2 Quel type d'application peut il y avoir pour une image radar ? Considérez pour cela que l'application que vous êtes en train de créer constitue un modèle graphique et interactif simplifié de la démo de l'image radar fournie sous forme de *jar* avec l'énoncé. Vous pourrez exécuter et explorer le fonctionnement de cette dernière afin d'identifier les correspondances entre les composants visuels des deux applications, ainsi que les différences de traitements qu'ils subissent lorsque vous zoomez.

7.3 Nous allons mettre en place un premier *zoom* dit différencié, qui aura vocation à être appliqué à la grille. Il fera en sorte que l'épaisseur de ses traits reste visuellement invariante au fur et à mesure des modifications d'échelle.

Attention, toutes les transformations étant cumulées le long du graphe de scène, visuellement invariant ne signifie pas réellement invariant. Cela signifie plutôt qu'au fur et à mesure qu'on applique des transformations à la grille, il faudra appliquer les transformations inverses aux nœuds dont on souhaite qu'ils ne changent pas d'aspect (ou à une propriété particulière des ces nœuds pour qu'elle reste visuellement invariante après le cumul des transformations).

Dans notre cas le traitement ne se fera pas en bloc sur la grille ni même sur les lignes qui la composent (les instances de `Line`) mais sur la propriété *strokeWidth* de chacune de ces lignes, qu'il faudra diviser par la valeur de l'échelle (la propriété *scale* héritée de `TranslatableHomotheticPane`) chaque fois que celle-ci change.

Pour pouvoir réagir aux changements de valeur de la propriété *scale* il faudra vous abonner grâce à la méthode `addListener(…)` spécifiée dans l'interface `ObservableValue` qu'implémentent toutes les `Property`. Comme écouteur pour cet abonnement vous utiliserez un `ChangeListener<Double>`.

7.4 A quel type de composant visuel de l'image radar devrez vous appliquer le mécanisme mis en place ?

7.5 Grâce à la question 7.2 vous avez dû identifier `TranslatableHomotheticPaneRect` à une représentation simplifiée de l'étiquette d'un trafic dans l'image radar. Vous allez maintenant implémenter le mécanisme de *zoom* différencié qui y correspond. L'étiquette nécessite un mécanisme différent de la grille puisqu'elle ne doit pas du tout changer de taille à l'écran lorsqu'on zoome. Cette fois ci la transformation est plus simple puisqu'elle est globale : lorsque la propriété *scale* de la grille change, il suffit d'appliquer le facteur d'échelle inverse à l'ensemble du rectangle. Vous aurez donc besoin dans la classe `TranslatableHomotheticPaneRect` :

- de récupérer une référence à la propriété *scale* de la grille (c'est elle qui est zoomée),
- d'implémenter la méthode `replaceScale(double scale)`,
- tant que vous y êtes implémentez également la méthode `replaceScale(double scale, double pivotX, double pivotY)` pour avoir une *API* complète et cohérente (vous n'en avez pas besoin dans ce TP mais votre code doit être réutilisable).

Félicitations ! Vous en savez maintenant assez pour développer la plupart des interactions d'une application cartographique. Vos collègues de la mineure SITA développent sur cette base une **image radar**. Cette réalisation vous est épargnée car vous avez un **projet Java**… où vous pourriez, selon le sujet, avoir à implémenter des interactions de manipulation directe similaires.

Exercice 8 : Alternative aux transformations inverses

Nous avons réalisé nos *zooms* différenciés en appliquant des transformations inverses. N'y aurait il pas un autre moyen de faire du *zoom* différencié ? Pensez à une organisation différente du graphe de scène et à l'utilisation de bindings.

Quels en seraient les avantages et inconvénients par rapport à la solution retenue ?

Exercice 9 : *Drag & Drop*

L'interaction de *drag* que nous avons réalisée consiste à déplacer des objets au sein d'une scène graphique. Une autre interaction de manipulation directe similaire est très utilisée, le *Drag & Drop*, qui permet de réellement déplacer un objet (ou une information représentée par cet objet) depuis un conteneur vers un autre, voire depuis (ou vers) une autre application (par exemple ouvrir un fichier en le faisant directement glisser vers une zone de votre application, plutôt que de cliquer sur un menu « Ouvrir » et de choisir le fichier dans un « File chooser »).

Il y a dans presque toutes les technos une *API* de *Drag & Drop*. Celle de *JavaFX* est décrite ici : https://docs.oracle.com/javase/8/javafx/events-tutorial/drag_drop_feature.htm

Vous allez dans cet exercice suivre les indications laissées en commentaire dans le code fourni, en commençant par la classe `DragAndDropTextContent`, puis `DragAndDropObjectSimple`.

Le *Drag & Drop* améliore considérablement l'utilisabilité d'une application. Pensez à l'utiliser si besoin **dans votre projet**.