TME 9 : groupes d'objets, fonction d'annulation

Dans ce TME, nous reprenons une dernière fois l'application `PinBoard` développée aux deux TME précédents pour y ajouter de nouvelles fonctionnalités. Nous ajoutons d'abord un mécanisme pour grouper des éléments graphiques, puis un système permettant d'annuler des opérations (jusqu'à un niveau arbitraire).

9.1 Groupes d'objets

Objectif pédagogique : motif composite.

Afin de faciliter le déplacement ou la copie simultanée d'un ensemble d'éléments graphiques, nous proposons un support pour grouper des objets.

9.1.1 Enrichissement du modèle de document (`ClipGroup`)

Un groupe est un élément graphique composé d'autres éléments graphiques. Ainsi, un point d'exclamation sera composé d'un rectangle et d'une ellipse. Vu de l'extérieur, un groupe d'éléments peut être manipulé exactement comme un élément graphique : il a un rectangle englobant, il peut être déplacé, dessiné, etc. Un groupe implante donc l'interface `Clip`. Ceci nous permettra de réutiliser, pour les groupes, tout le code existant programmé vis à vis de l'interface `Clip`. L'interface d'un groupe est néanmoins enrichie pour supporter l'ajout et la suppression d'objets au groupe. Nous avons donc ici le *motif composite*.

Un groupe d'objets obéit à l'interface `pobj.pinboard.document.Composite` fournie :

tme/src/tme9/src/pobj/pinboard/document/Composite.java

```
package pobj.pinboard.document;

import java.util.List;

public interface Composite extends Clip {
   public List<Clip> getClips();
   public void addClip(Clip toAdd);
   public void removeClip(Clip toRemove);
}
```

L'implantation concrète, `ClipGroup`, de cette interface aura les caractéristiques suivantes :

- dessiner un groupe se réduit à dessiner tous ses éléments ;
- le rectangle englobant d'un groupe est le plus petit rectangle qui englobe les rectangles englobants de tous ses éléments ; attention à maintenir cette information quand un élément est ajouté ou supprimé du groupe, ou quand le groupe est déplacé ;
- déplacer un groupe déplace tous ses éléments ;
- pour simplifier, nous pouvons supposer que `setGeometry` ne change pas la taille d'un groupe, mais seulement sa position, i.e., il peut se réduire à une opération `move` (ceci évite de perdre du temps à développer des calculs de mise à l'échelle) ;
- copier un groupe nécessite également de copier récursivement ses éléments (copie en profondeur) ;
- un groupe peut bien sûr contenir d'autres groupes.

⇒ **Travail demandé :** Programmez une classe `pobj.pinboard.document.ClipGroup` implantant l'interface `Composite`, et testez-là avec `pobj.pinboard.document.test.ClipGroupTest` fournie. Votre classe peut hériter de `AbstractClip` vu aux TME précédents.

9.1.2 Groupage et dégroupage

Nous ajoutons maintenant deux options dans le menu « Edit » : « Group » et « Ungroup », qui opèrent sur la sélection courante. Notez que grouper des objets se fait en quatre étapes : il faut d'abord ôter les éléments sélectionnés de la planche, puis créer un nouvel objet groupe, ajouter les éléments sélectionnés au groupe et enfin ajouter ce groupe à la planche. L'opération de dégroupage est similaire.

⇒ **Travail demandé :** Ajoutez les options « Group » et « Ungroup ». Vérifier que les fonctions de sélection, de déplacement et de copier / coller du TME précédent fonctionnent avec les groupes d'objets.

9.2 Annuler et refaire

Objectif pédagogique : motif commande.

Nous allons ajouter des options « Undo » et « Redo » permettant respectivement d'annuler la dernière opération et de refaire la dernière opération annulée. Cet effet sera obtenu par *réification* des commandes, qui permet de les stocker pour utilisation future, au lieu de les exécuter immédiatement. En utilisant la fonctionnalité « Undo » plusieurs fois de suite, il est possible d'annuler toute une séquence d'opérations ; de même pour « Redo ». Nous aurons donc besoin de maintenir des *piles* de commandes.

9.2.1 Commande annulable

Une commande est un objet qui encapsule une action à exécuter. Construire une commande n'exécute pas l'action ; l'action est effectivement exécutée en appelant une méthode `execute()`. Pour une commande annulable, nous ajoutons une méthode `undo()` ayant pour effet d'annuler l'effet de `execute()`. Nous définissons donc l'interface `pobj.pinboard.editor.commands.Command` suivante :

tme/src/tme9/src/pobj/pinboard/editor/commands/Command.java

```
package pobj.pinboard.editor.commands;

public interface Command {
   public void execute();
   public void undo();
}
```

où `execute` sert à la fois à exécuter la commande une première fois, et à la refaire après un `undo`, si nécessaire.

Pour chaque type de commande permettant à l'éditeur d'agir sur la planche (ajout, suppression, déplacement d'élément graphique, groupage et dégroupage), nous définissons une classe implantant `Command`. Par exemple, l'action d'ajouter des éléments graphiques sera définie par la classe `CommandAdd`, dont une signature possible est :

```
public class CommandAdd implements Command {
   public CommandAdd(EditorInterface editor, Clip toAdd);
   public CommandAdd(EditorInterface editor, List<Clip> toAdd);
   @Override public void execute();
   @Override public void undo();
}
```

Une remarque importante est que `execute` et `undo` n'ont pas d'argument. Toutes les informations nécessaires à l'exécution sont spécifiées lors de la construction de l'objet et maintenues dans des

attributs de l'objet. Ainsi, dans `CommandAdd`, l'argument `toAdd` précise les éléments à ajouter, et `editor` précise la fenêtre d'édition concernée par l'ajout.

Nous allons isoler ces classes et l'interface dans le sous-package `pobj.pinboard.editor.commands`.

⇒ **Travail demandé :**

1. Programmez dans le package `pobj.pinboard.editor.commands` des classes `CommandAdd`, `CommandMove`, `CommandGroup` et `CommandUngroup` obéissant à l'interface `Command` et correspondant respectivement à : l'ajout d'un élément graphique, le déplacement d'éléments, le groupage et dégroupage d'éléments.
2. Testez vos implantations grâce aux classes de `pobj.pinboard.editor.commands.test` fournies.
3. Modifiez tous les outils (`ToolRect`, etc.) et les actions associées au menu « Edit » afin de modifier le contenu de la planche *via* la commande correspondante, plutôt que de modifier la planche directement.

9.2.2 Pile de commandes annulables (`CommandStack`)

Afin de gérer l'annulation d'une succession arbitrairement longue de commandes, nous devons implanter une pile de commandes sur le modèle *last in / first out*. Par ailleurs, lors de l'annulation d'une commande, il est nécessaire de garder l'objet commande afin de pouvoir l'exécuter à nouveau si l'utilisateur utilise l'option « Redo ». Afin de gérer un nombre arbitraire de « Redo », il nous faut une deuxième pile stockant les commandes annulées. Nous appelons ces deux piles *undo* et *redo*. La gestion d'une paire de piles de commandes est laissée à une classe `CommandStack` ayant les méthodes suivantes :

- `public void addCommand(Command cmd)`
  empile une commande dans la pile *undo* ; elle vide également la pile *redo*, puisque refaire la dernière action annulée peut entrer en conflit avec la commande que nous venons d'ajouter ;
- `public void undo()`
  dépile la dernière commande de la pile *undo*, exécute sa méthode `undo`, et empile la commande dans la pile *redo* ;
- `public void redo()`
  dépile la dernière commande de la pile *redo*, exécute sa méthode `execute`, et empile la commande dans la pile *undo* ;
- `public boolean isUndoEmpty()`
  `public boolean isRedoEmpty()`
  utiles pour permette à l'interface graphique de savoir si les actions « Undo » et « Redo » ont un sens (sinon, les options peuvent être grisées).

⇒ **Travail demandé :** Programmez la classe `CommandStack` dans le package `pobj.pinboard.editor`. Testez-là avec la classe `pobj.pinboard.editor.test.CommandStackTest` fournie.

9.2.3 Intégration à l'éditeur

Nous devons maintenant ajouter aux fenêtres d'édition la gestion des commandes à annuler et à refaire. Il suffit pour cela d'ajouter à `EditorWindow` un attribut `CommandStack` et de s'assurer que les piles de commandes sont correctement remplies. Il est également nécessaire de modifier le *getter* associé dans `EditorWindow` pour qu'il retourne ce nouvel attribut au lieu de `null`.

⇒ **Travail demandé :**

- modifiez les outils et les actions associées aux menus pour qu'ils enregistrent chaque commande auprès de `CommandStack` ;
- ajoutez les options « Undo » et « Redo » au menu « Edit », et programmez les actions associées.

9.3 Bonus : sauvegarde et lecture dans des fichiers

Objectif pédagogique : sérialisation.

Nous ajoutons dans cette extension des options dans le menu « File » pour sauvegarder le dessin dans un fichier et pour ouvrir un dessin depuis un fichier dans une nouvelle fenêtre.

Concernant l'interface pour choisir un nom de fichier, nous avons peu à faire : il nous suffit d'employer les fonctionnalités prêtes à l'emploi dans la classe `FileChooser` fournie par JavaFX.

Concernant le format de fichier, nous pouvons profiter de l'interface de sérialisation intégrée au langage Java. Elle permet de sauvegarder dans un fichier et lire depuis un fichier des objets arbitraires. En première approximation, il suffit donc de sérialiser l'objet `Board`. La sérialisation s'effectue par les classes `ObjectOutputStream` et `ObjectInputStream`. Par ailleurs, il est nécessaire de marquer avec `implements serializable` toutes les classes susceptibles d'apparaître dans un fichier. Il s'agit donc non seulement de `Board`, mais aussi de `Clip`.

Malheureusement, une difficulté particulière surgit : la plupart des classes JavaFX ne sont pas sérialisables ! Ceci inclut en particulier les classes `Color` et `Image`, utilisées dans ce projet. Par conséquent, il n'est pas possible d'utiliser ces classes directement comme attribut dans nos éléments graphiques implantant `Clip`. Nous proposons les solutions suivantes :

- Pour `Color` : définir sa propre classe couleur, contenant des attributs `red`, `blue` et `green` de type `double`. Cette classe peut être rendue sérialisable et peut être utilisée dans les attributs des éléments graphiques. Vous ajouterez des méthodes pour convertir à la volée entre cette classe et la classe `javafx.scene.paint.Color`, utilisée pour dessiner avec JavaFX.
- Pour `Image` : ne pas essayer de stocker l'image, mais seulement le nom du fichier contenant l'image. La classe `ClipImage` garde un attribut de type `Image`, afin d'accéder rapidement à l'image pour l'afficher, mais celui-ci est noté comme `transient`, c'est-à-dire que le système Java ne stockera pas cet attribut lors de la sérialisation. Après un chargement de fichier, l'attribut `transient Image` sera remis à `null` par Java. Il faudra donc recharger l'image à partir du nom de fichier.

⇒ **Travail demandé :** Implantez des options « Open » et « Save » dans le menu « File ».

9.4 Rendu du TME (OBLIGATOIRE)

Comme d'habitude, vous implanterez les classes demandées, ferez un *push* dans Gitab et vérifierez que les tests d'intégration continue des TME 7 à 9 passent sans erreur. Vous créerez ensuite un *tag* dans GitLab.

Dans le champ « Release notes » de votre *tag* vous :

1. fournirez une capture d'écran montrant le rectangle englobant d'un groupe sélectionné, contenant au moins deux objets ;
2. préciserez si vous avez développé l'extension proposée en bonus, ou une autre extension de votre invention.