TP OCI n°2

L'objectif de ce TP est d'avancer vers la construction de l'application de dessin à la suite du TP précédent.

1 Drag'n drop de couleurs

En vous inspirant des exemples et du support de cours, implémentez les possibilités de *drag'n drop* entre :

- les widgets de classe `ColorViewer`
- les `ColorButton`

Il n'y a rien à faire pour les `ColorButton`. Ceux-ci implémentent déjà des possibilités de *drag'n drop* de couleur (du type « `application/x-color` »). Une fois les *drag'n drop* implémentés entre ces deux widgets, vérifier qu'il fonctionne entre votre application et une application extérieure (gimp par exemple).

Notez qu'il n'est pas nécessaire de surcharger les fonctions du DnD dont vous n'avez pas l'utilité (`on_drag_delete()` ...).

2 Création d'une zone de dessin

À la suite du TP précédent, on se rend compte qu'il est nécessaire de travailler spécifiquement sur la zone de dessin pour avoir un affichage pérenne. La solution que je vous propose repose sur deux principes :

- un dessin est une liste de formes, il peut y avoir plusieurs types de forme et chacune des formes ayant ses propriétés (couleurs et position),
- le dessin et son affichage du dessin est géré par un objet (hérité de la classe `DrawingArea`). Sur cet aspect, il est évident que la conception pourrait être améliorée en séparant le modèle du dessin de sa vue !

Je vous propose donc d'implémenter une classe abstraite `Shape` qui représente n'importe quelle forme qui peut être dessinée à l'aide d'une fonction `draw()` ainsi qu'un classe `DrawingZone` dont l'affichage réalisera quelque chose de la forme :

```
DessinerDrawingZone() {
  Pour tout Shape S faire {
        S.draw(this)
  }
}
```

A Une classe abstraite pour les formes : Shape

Créer une classe `Shape` abstraite qui sera utilisée pour implémenter des formes à dessiner dans la zone de dessin. Dans le cas le plus général, une forme aura une position et une taille (hauteur, largeur) pour son emprise (c'est-à-dire le rectangle dans lequel sera inscrit la forme) et deux couleurs (*foreground*, *background*).

Contrairement à l'application Java qui implémentait les formes comme des `JPanel`, ici les formes seront des objets abstraits mais pas des widget. Ces objets décrivent eux-même comment ils doivent être dessinés dans le `Drawable`, comme le par exemple `DrawingArea`.

Cette classe proposera les méthodes abstraites suivantes :

- une fonction `draw` qui réalise le dessin de la forme (commencer par implémenter une fonction sans paramètre, puis vous modifierez ce profil lors de la création d'une classe `ShapeRectangle` ou après la lecture de la partie 3.B),

- une fonction `bool over(Gdk::Point p)` qui indique si `p` est au dessus de la forme ou non. Cette fonction sera certainement utile plus tard ...

Créer ensuite une classe `ShapeRectangle` qui implémente une forme de rectangle.

B Classe DrawingZone

Créer une classe `DrawingZone` qui hérite d'un `DrawingArea` et qui devra prendre la place du `DrawingArea` actuel pour y dessiner des formes.

Vous ajouterez à cette classe une liste de forme : `list<Shape *> _shapes;`

Implémenter une fonction pour dessiner toutes les formes de la liste `_shapes` dans la zone de dessin en faisant appel à la fonction `draw()` partagée.

Implémenter également une fonction pour déterminer la forme visible à une position donnée de dessin :

`Shape *first_over(Gdk::Point p)`

Pour tester votre travail, vous pourrez créer une fonction dans la classe de la fenêtre principale qui ajoute un rectangle à une position aléatoire avec les couleurs définies dans le `ColorSelector`. Cette fonction pourra ensuite être connectée au bouton créé dans le TP précédent.

Pour la génération de nombres pseudo-aléatoires, voir :
http://www.cplusplus.com/reference/clibrary/cstdlib/rand/

C Bouton Clear

Rajouter un bouton `Clear` pour supprimer l'intégralité des formes précédemment ajoutées.

3 Réflexion sur l'implémentation d'un système d'outils générique

Vous disposez maintenant d'une base et d'un peu de pratique de Gtkmm. Il faut maintenant réfléchir à la conception de votre application pour qu'elle réponde au « cahier des charges ». Il est bien évidement utile de s'inspirer de l'application Java proposée par T. Duval (accessible sur la page de ses cours).

Le plus difficile est de mettre de la généricité dans le système de gestion des outils. Dans le principe, je distingue deux types d'outils : les outils de création de formes (outil rectangle, outil triangle, ...) et les autres outils (suppression, déplacement, …). Néanmoins, il n'est pas souhaitable d'en distinguer la gestion par l'application.

Dans la mesure où on souhaite greffer des plugins pour des nouvelles formes, il faut que de nouvelles classes d'outil puisse facilement s'insérer dans l'application. Il est souhaitable, pour une meilleure décomposition objet et pour une plus grande facilité d'évolution de l'éditeur, de ne pas faire de *switch* ni de successions de tests imbriqués pour déterminer quel type de dessin doit être réalisé en cours d'interaction : **on préfèrera avoir recours à une connexion/déconnexion dynamique des signaux des évènements intéressants sur la zone du `DrawingArea` avec des implémentations de l'outil courant**.

Passer le temps restant de la séance à commencer à définir votre solution … (*ie* la stratégie de connexion des signaux).

Les séances suivantes ne seront pas guidées, elles seront encadrées pour mener à bien vos solutions pour la suite du TP !