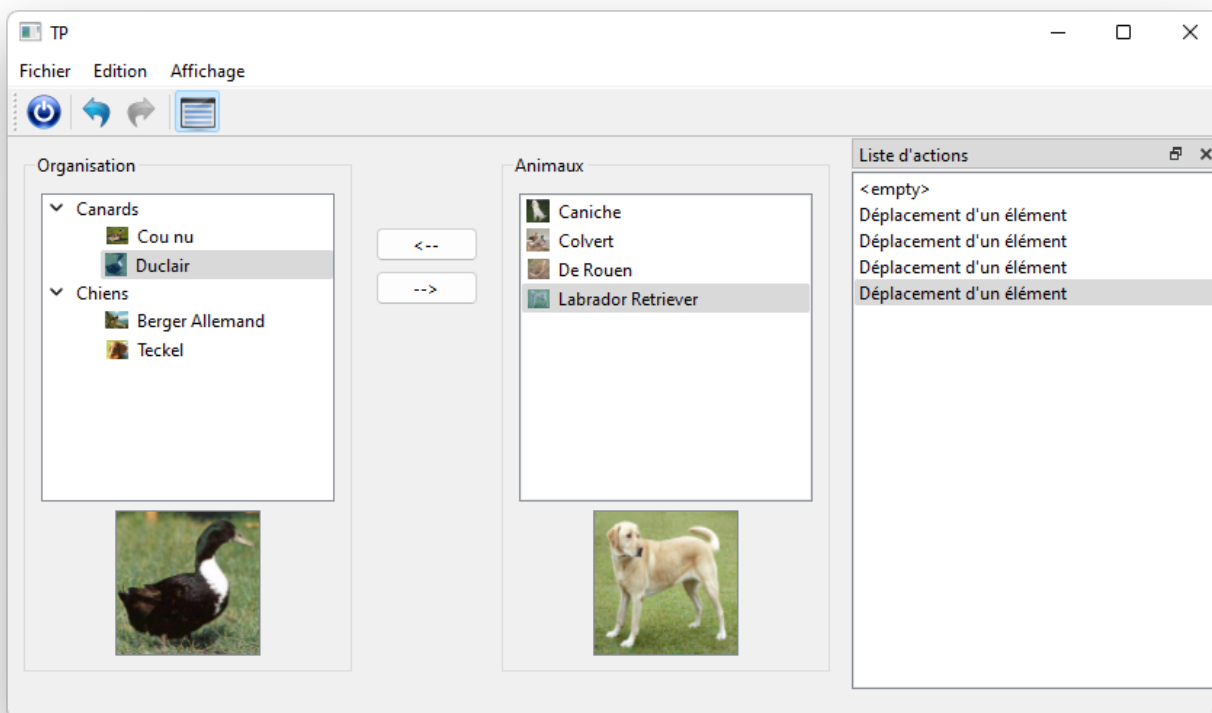


TP 27 – Drag and Drop

✎ Pour rappel, vous pouvez suivre la correction étape par étape de ce sujet en clonant le dépôt [git@gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/TP27.git](https://gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/TP27.git). Des étiquettes vous permettent de vous déplacer dans les différentes versions de chaque exercice.

Objectif du TP

L'objectif de ce TP est de développer une petite application mettant en œuvre le drag & drop et les commandes annuler / rétablir. Le programme consiste principalement en deux zones d'affichage de listes. Les éléments affichés dans la liste de droite doivent être glissés et déposés dans la liste hiérarchique de gauche avec pour objectif de trier manuellement des races d'animaux par familles. La sélection d'un élément permet d'afficher son image agrandie sous la liste.



✎ Si vous avez besoin de savoir à quoi ça sert, outre l'aspect pédagogique de votre travail, vous n'avez qu'à penser qu'il s'agit d'un jeu pour enfant...

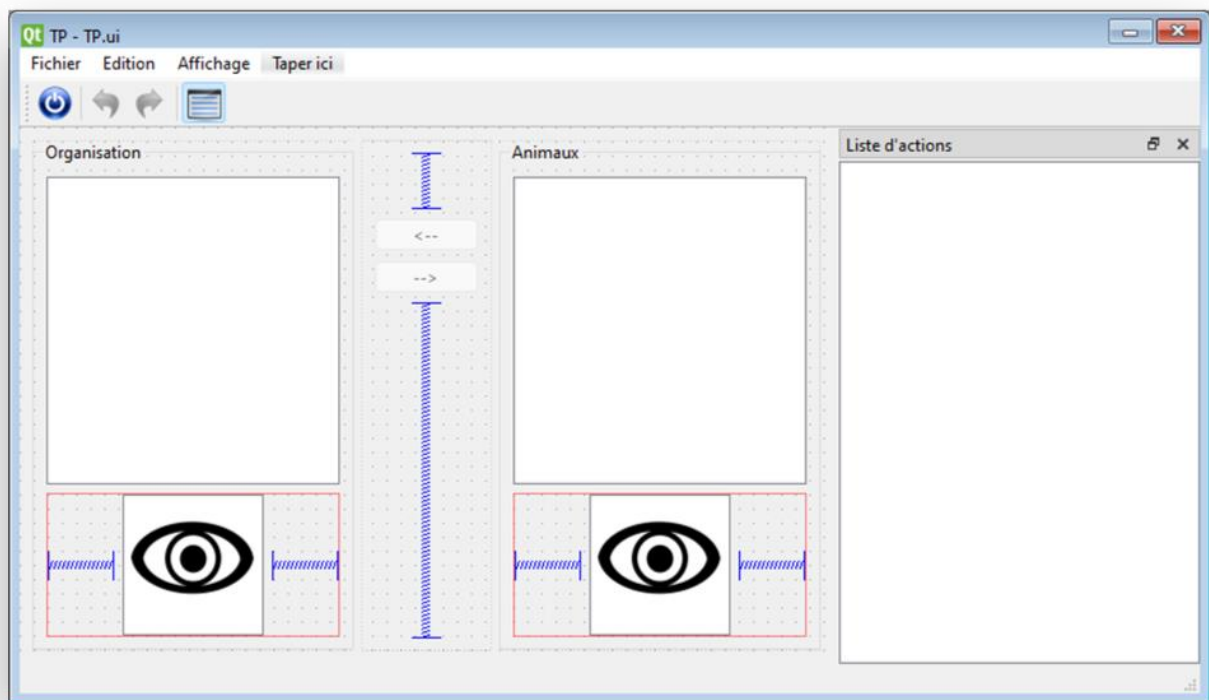
✎ Plus sérieusement, à travers ce TP, vous allez apprendre à vous débrouiller dans la documentation !

Exercice 0 Création d'un nouveau projet

- Créez un nouveau projet **Qt** basé sur des widgets, dont la fenêtre principale hérite d'un **QMainWindow**.

Par la suite, nous supposons que vous avez nommé la classe correspondant à la fenêtre principale « **TP** »


Exercice 1 Création de l'interface avec Qt Designer





1.1 Menus et barre d'outils

- Créez les éléments de menu suivant. Pour configurer les propriétés des actions, utilisez l'éditeur d'actions (situé généralement en bas à droite de la fenêtre de **Qt Designer**)


➤ Fichier

- ➔ **Quitter** : Image de l'action : **quit.png** , raccourcis clavier : **Alt+F4**

➤ Edition



- ➔ **Annuler** (par défaut désactivé) : Image de l'action : **stock_undo.png** , raccourcis clavier : **Ctrl+Z**
- ➔ **Rétablir** (par défaut désactivé) : Image de l'action : **stock_redo.png** , raccourcis clavier : **Ctrl+Y**

➤ Affichage







- ➔ **Historique des commandes** (cochable, par défaut coché) : Images de l'action : **playlist.png** 



- Créez la barre d'outils comme sur la capture d'écran en utilisant les actions précédentes.


1.2 Liste d'actions

-  Ajouter un **Dock Widget** sur la droite de la fenêtre principale
 - ➡ Dans ses propriétés, spécifiez un **windowTitle** judicieux (par exemple « **Liste d'actions** »)
-  Dans ce **Dock Widget**, insérez un **Undo View**.
 - ➡ Vous pouvez définir un **layout** adéquat sur le **Dock Widget** de façon à ce que le **Undo View** enfant épouse toute la place disponible


1.3 Zones de listes

-  Dans la partie centrale de la fenêtre, insérez un **Group Box** que vous titrerez « **Organisation** »
 - ➡ Insérez à l'intérieur de celui-ci un **Tree View** et un **Label** qui affichera l'image aperçu de l'animal sélectionné.
 - ➡ Décochez la propriété « **headerVisible** » du **Tree View**.
 - ➡ En utilisant judicieusement des **layouts** et des **spacers**, faites en sorte que le **Label** soit centré sous le **Tree View**.
 - ➡ Le **Label** doit afficher par défaut l'image stylisé d'un œil : 
 - ➡ Pour cela, fixez la taille du **Label** à **100x100** (**minimumSize** et **maximumSize**). Modifiez la palette de couleur du **Label** pour que son fond soit blanc (propriété **palette / Window**). Activez la propriété **autoFillBackground**, le **frameShape** à **StyledPanel** et définissez le **pixmap** à une ressource construite par le fichier **oeil.png**. Enfin, pour que l'image soit contenue entièrement dans le **Label**, activez la propriété **scaledContent**.
-  Répétez cette opération pour créer le groupe « **Animaux** » en choisissant une **List View** à la place d'un **Tree View**
-  Placez entre les deux **Group Box** un **Widget** standard qui contiendra les deux **Push Button** <-- et --> de la capture d'écran.
-  Sélectionnez les deux **Group Box** et le **Widget** précédent puis cliquez sur le bouton de mise en page horizontale avec un séparateur 
 - ➡ Affectez ensuite un **layout** horizontal à la partie centrale de la fenêtre principale
 - ➡ Dans l'inspecteur d'objet, sélectionnez l'objet **splitter** pour désactiver sa propriété **childrenCollapsible**



 L'utilisation d'un splitter  permet de redimensionner à la souris les éléments de l'interface.

-  Questions : (utilisez l'aide en ligne pour répondre)
 - ➡ À quoi sert la propriété **autoFillBackground** ?
 - ➡ À quoi sert la propriété **childrenCollapsible** ?


1.4 Connexion de signaux et slots

 Par l'éditeur de signaux et slots, établissez les connexions suivantes :

Emetteur	Signal	Receveur	Slot
Bouton <--	clicked()	Fenêtre principale	toTree() (à créer)
Bouton -->	clicked()	Fenêtre principale	toList() (à créer)
Action Rétablir	triggered()	Fenêtre principale	redo() (à créer)
Action Annuler	triggered()	Fenêtre principale	undo() (à créer)
Action Quitter	triggered()	Fenêtre principale	close()
Act. Hist. des cmds	triggered(bool)	Dock Widget	setVisible(bool)
Dock Widget	visibilityChanged(bool)	Act. Hist. des cmds	setChecked(bool)

 Votre interface est enfin prête !
 Compilez et exécutez pour vous faire plaisir.

Exercice 2 Création des modèles d'éléments

 Comme vous le savez puisque vous êtes un·e étudiant·e studieux·se, les widgets de liste que vous avez insérés dans l'interface fonctionnent en appliquant le patron de conception modèle – vue. Nous allons donc définir le modèle et nous personnaliserons les vues.

Les éléments sont contenus dans deux fichiers **XML** fournis dans le sous répertoire **xmls**. Le fichier **liste.xml** contient le modèle de la zone de liste des animaux, le fichier **arbre.xml** contient le modèle de la zone de liste hiérarchique des familles. Vous devez créer une classe qui fait le lien entre un fichier **XML** et un modèle compatible avec l'affichage des vues.


```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <elem display="Cou nu" decoration="cou-nu.jpg"/>
  <elem display="Berger Allemand" decoration="bergerallemand.jpg"/>
  <elem display="De Rouen" decoration="derouen.jpg"/>
  <elem display="Duclair" decoration="duclair.jpg"/>
  <elem display="Teckel" decoration="teckel.jpg"/>
  <elem display="Caniche" decoration="caniche.jpg"/>
  <elem display="Colvert" decoration="colvert.jpg"/>
  <elem display="Labrador Retriever" decoration="labrador.jpg"/>
</root>
```

Contenu du fichier liste.xml pour la liste complète des animaux

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <class display="Canards">
    <elem display="Cou nu" decoration="cou-nu.jpg"/>
    <elem display="De Rouen" decoration="derouen.jpg"/>
    <elem display="Duclair" decoration="duclair.jpg"/>
    <elem display="Colvert" decoration="colvert.jpg"/>
  </class>
  <class display="Chiens">
    <elem display="Berger Allemand" decoration="bergerallemand.jpg"/>
    <elem display="Teckel" decoration="teckel.jpg"/>
    <elem display="Caniche" decoration="caniche.jpg"/>
    <elem display="Labrador Retriever" decoration="labrador.jpg"/>
  </class>
</root>
```


Contenu du fichier `arbre.xml` pour la liste complète des familles d'animaux

Comme vu en cours, le modèle `QStandardItemModel` est parfait pour notre cas, car il permet de définir aussi bien des modèles de liste que des modèles de hiérarchie. Nous allons donc étendre cette classe en lui ajoutant la capacité de construire le modèle depuis un fichier **XML** et de l'enregistrer dans un fichier **XML**.

En utilisant le menu  `Projet / Add Qt Class...` ou **QC** faites un clic droit sur le projet puis choisissez `Add New... / Qt / C++ Class`, créez une classe `QXmlItemModel` héritant de `QStandardItemModel`

Vous lui ajouterez entre autres deux méthodes publiques :

- ➔ `ReadFromFile()` → Construit le modèle depuis un fichier **XML**
- ➔ `WriteToFile()` → Enregistre le modèle dans un fichier **XML**

 Pour implémenter ces deux fonctions, utilisez un `QXmlStreamReader` et un `QXmlStreamWriter`. [RTFM](#) !

Munissez la classe principale de votre application de deux membres de type de votre nouvelle classe de modèle.

- ➔ Un pour la liste d'animaux
- ➔ L'autre pour la hiérarchie de familles

Faites ce qu'il faut dans le code pour qu'en lançant l'application, les éléments s'affichent dans les listes correspondantes.

Faites ce qu'il faut pour qu'à la fermeture de l'application, il soit demandé à l'utilisateur d'enregistrer ses modifications dans les deux fichiers **XML**.

- ➔ Pour savoir où écrire le code qui sera exécuté à la fermeture de la fenêtre principale : [RTFM](#) !


Exercice 3 Affichage des images des animaux

A la sélection d'un animal dans une liste, son image doit s'afficher dans le **Label** correspondant. Pour cela, nous devons être prévenus lorsque l'élément courant change. **Qt** ne propose bizarrement pas de signal dans les vues pour connaître cela. Cependant, les vues disposent d'une méthode protégée **currentChanged** appelée lorsque cet événement se produit. Nous allons donc surcharger ces fonctions pour émettre notre signal.


 Ajoutez une classe **Qt** héritant de **QListView**


 Surchargez la méthode :


```
virtual void currentChanged(const QModelIndex & current, const QModelIndex & previous);
```

 Ajoutez le signal suivant à la classe :

```
void selectionChanged(const QModelIndex & index);
```


 Émettez ce signal dans le corps de la fonction **currentChanged**


 Dans l'interface, l'objet **QListView** doit être promu en le nouveau type que vous venez de créer


 Vous pouvez également utiliser l'éditeur de signaux et slots pour connecter le nouveau signal de votre nouvelle classe à un slot que vous créerez dans la classe principale de l'application.

➔ Dans ce slot, vous récupérerez les informations de l'élément courant pour afficher l'image dans la zone prévue à cet effet.

➔ Toujours dans ce slot, vous activerez le bouton « **toTree** » dans le cas où les indices courants de la liste et de l'arbre sont valides, c'est-à-dire lorsqu'un élément est sélectionné de part et d'autre.


 Procédez de même pour l'objet de type **QTreeView**

 Dans le slot associé, en plus du bouton « **toTree** » qui doit être activé dans les mêmes conditions que précédemment, vous devez activer le bouton « **toList** » lorsque l'élément sélectionné correspond à un animal (et non à un groupe d'animaux)


 Dorénavant, la sélection d'un élément dans l'une ou l'autre vue aura pour effet d'afficher son image dans le **Label** correspondant.

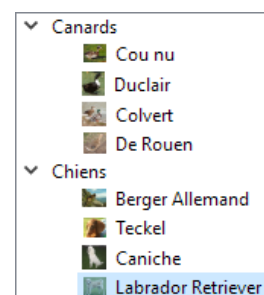
Exercice 4 Passage des éléments de la liste à l'arbre et vice-versa

 Complétez les slots **toList** et **toTree** de la classe **TP**.

 Pour réaliser un déplacement, vous avez besoin de l'objet modèle de départ avec l'indice de l'élément déplacé et de l'objet modèle d'arrivée avec l'indice de l'élément sur lequel le déplacement s'opère. Vous devez également savoir si l'élément doit être inséré avant, après ou dans l'élément sélectionné.


➔ Pour repérer les éléments, il vous est conseillé de stocker dans une liste son « adresse ». Par exemple, en considérant l'élément « **Labrador Retriever** » sélectionné sur la capture d'écran ci-contre, son adresse est **1 - 3** (**1** = indice de « **Chiens** » au premier niveau de la hiérarchie – **3** = (indice de « **Labrador Retriever** » dans le groupe « **Chiens** »)

 Pour éviter de copier/coller du code, il serait pertinent de créer une fonction qui serait appelée dans ces deux slots et qui









réaliserait le déplacement de l'élément sélectionné d'un **QAbstractItemView** vers un autre :

```
void TP::moveElem(QAbstractItemView* source, QAbstractItemView* dest)
```










 Testez cette classe de commande en utilisant les boutons <-- et --> de l'interface


Exercice 5 Activation de l'annulation / rétablissement

-  Tout d'abord, ajouter à la classe principale un membre **QUndoStack**
 -  Faites ce qu'il faut pour que cette donnée soit liée à la **QUndoView** de votre interface.
 -  Faites également ce qu'il faut pour que les actions *Annuler* et *Rétablir* soit désactivées (grisées) lorsqu'aucune action n'est annulable et/ou rétablissable
 -  N'oubliez pas de rendre possible l'annulation et le rétablissement.
-  Créez une classe de commande comme vu en cours permettant d'annuler et de rétablir un déplacement d'élément
 -  Refactorisez la fonction **TP::moveElem** de façon à déplacer le code nécessaire à l'annulation / rétablissement dans la classe de commande.


Exercice 6 Activation du Drag and Drop entre les vues

L'activation du « **drag and drop** » (« glissé déposé » en français) se fait par l'intermédiaire de propriétés définies dans **Qt Designer**.

-  Pour chacune des vues, réglez les propriétés suivantes :
 -  Activez **showDropIndicator**
 -  Affiche en surbrillance le lieu du dépôt lors d'une action drag & drop
 -  Activez **dragEnabled**
 -  Active le glissement des éléments
 -  Définissez la propriété **dragDropMode** à **DragDrop**
 -  La vue peut faire glisser des éléments mais également en accueillir de nouveaux par dépôt.
 -  Définissez la propriété **defaultDropAction** à **MoveAction**
 -  Par défaut, un drag & drop déplacera un élément

 À l'aide de ces quelques propriétés, vous avez activé la fonctionnalité du drag & drop. Compilez et testez.

En maintenant la touche **Ctrl** enfoncée pendant l'action de drag & drop, l'élément n'est plus simplement déplacé mais copié. Ce comportement ne nous convient pas car nous ne voulons en aucun cas que les éléments soient copiés. Il suffit de spécifier que le modèle ne supporte pas la copie d'élément en surchargeant la fonction **QStandardItemModel::supportedDropActions()**.


-  En vous référant à l'aide en ligne, surchargez cette fonction dans votre classe de modèle. La fonction doit retourner la valeur **Qt::MoveAction**.

 Cette fois-ci, les éléments ne sont plus copiables.


Exercice 7 Annuler / Rétablir



Pour le moment, le programme est fonctionnel, mais les actions par drag & drop ne sont pas annulables. Nous devons intercepter les actions de drag & drop pour pouvoir les insérer dans un **QUndoStack**.

L'action de drag and drop peut être interceptée au moment du drop. À ce moment, l'origine du drag & drop est connue (l'élément source) et la destination également. Pour intercepter cet événement, il faut surcharger la méthode **dropEvent** des vues.

-  Surchargez la méthode **dropEvent** des vues
 - ➡ À l'aide des informations véhiculées par le paramètre de cette fonction ([RTFM](#)), reconstruisez l'adresse source et destination de l'élément déposé puis utilisez la classe de commande précédente pour réaliser l'action
 - ➡ N'oubliez pas de désactiver l'action de drag & drop car vous venez de faire l'action à la place du comportement par défaut de Qt.
 - ➡ Il suffit d'appeler la méthode **setDropAction(Qt::IgnoreAction)** sur le paramètre de la fonction **dropEvent**
 - ➡ Enfin, n'oubliez pas d'appeler la méthode **dropEvent** de la classe mère de façon que l'opération de drag & drop gérée par Qt se termine correctement.

Exercice 8 Dernier signolage

-  Faite en sorte qu'à la fermeture de l'application, si aucune modification n'a été faite, il ne soit pas demandé à l'utilisateur d'enregistrer ses données.
 - ➡ Indice : s'il n'est pas possible d'annuler, c'est qu'aucune modification n'a été faite.

 Voilà, c'est tout !
 Félicitation si vous êtes arrivé-e jusqu'ici.