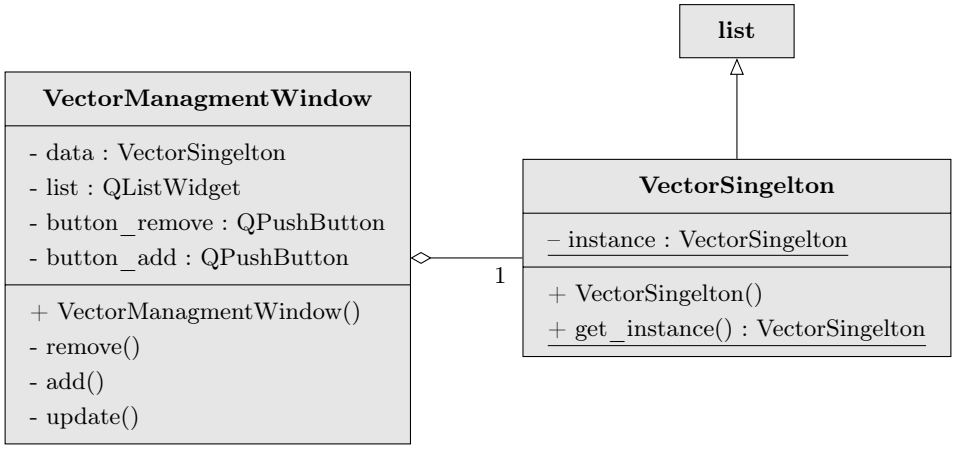
1. But du TP

Implémenter le *design pattern* **Singleton** pour afficher et modifier une liste d'éléments textes. Le Singleton permettra de forcer le partage d'une même instance pour toutes les classes qui accède à la liste.

Nous utiliserons ici la librairie graphique *PyQt5* (GPL/commercial license) (vous pouvez aussi utiliser *PySide6)* (LGPL license).

2. Diagramme UML



3. Classe `VectorSingelton`

En vous inspirant du design pattern Singleton écrivez la classe VectorSingelton. La méthode `get_instance()` doit initialiser l'attribut de classe (qui est l'unique instance de la classe VectorSingelton), si celui-ci vaut `None`, et le renvoyer.

Vous penserez à :

— empêcher l'utilisation du constructeur : si `instance` est `None`, remontez une exception, sinon, instanciez `instance` ;

— créer la méthode statique `get_instance`, qui appelle le constructeur ;

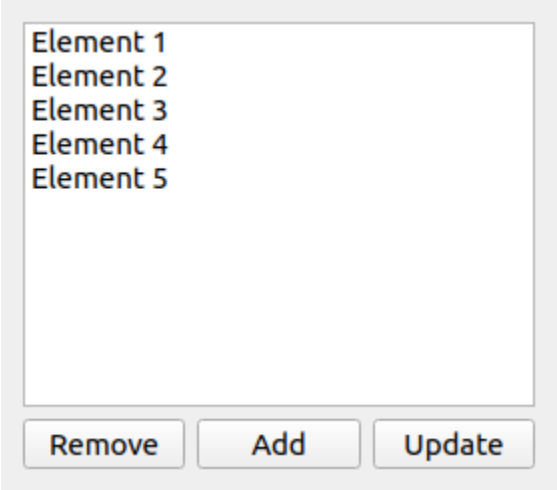
Vous pourrez tester votre programme avec ce `main` :

Fin de vector_singelton.py

```python
if __name__ == "__main__":
    singleton1 = VectorSingelton.get_instance()
    singleton2 = VectorSingelton.get_instance()
    assert singleton1 is singleton2
```

4. Classe `VectorManagmentWindow`

En vous inspirant des captures d'écran suivantes, créez une classe `VectorManagmentWindow` permettant d'afficher les éléments de `VectorSingleton`. Le bouton "Add" doit permettre d'ajouter des élements alors que le bouton "Remove" doit permettre de supprimer un élément de la liste. Pour accéder à la liste de données, vous utiliserez la méthode statique `VectorSingleton.getInstance()`. Votre interface pourra ressembler à la figure ci-dessous. Enfin, les clics sur les deux boutons appeleront la méthode `update`. Le bouton update quant-à lui mettra à jour la fenêtre.



Créez maintenant le `main` qui va permettre d'instancier un `VectorSingelton`, le peupler et instancier deux `VectorManagmentWindow`. Du fait du design pattern, toute modification de la liste aura lieu sur les deux fenêtres.

Rappels

Ici, vous devrez utiliser les classes `QApplication`, `QHBoxLayout`, `QListWidget`, `QVBoxLayout`, `QPushButton`, `QInputDialog`, `QWidget`.

Voici un exemple minimal pour créer une fenêtre :

```python
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Minimal PyQt5 Example")

        # Création d'un bouton
        self.button = QPushButton("Click on me!", self)
        self.button.clicked.connect(self.on_button_click)
```

```
13
14      def on_button_click(self):
15          print("Click!")
16
17  if __name__ == "__main__":
18      app = QApplication(sys.argv)
19      window = MainWindow()
20      window.show()
21      sys.exit(app.exec())
```

5. Pour aller plus loin

Comment pourriez-vous créer un design pattern qui permettrait pas de modéliser un singleton (i.e. un ensemble de taille 1), mais ensemble de d'objet de taille n, au maximum ?

Par exemple, pour créer des connexions à un serveur dont le nombre de connexion simultané est limité à 100.