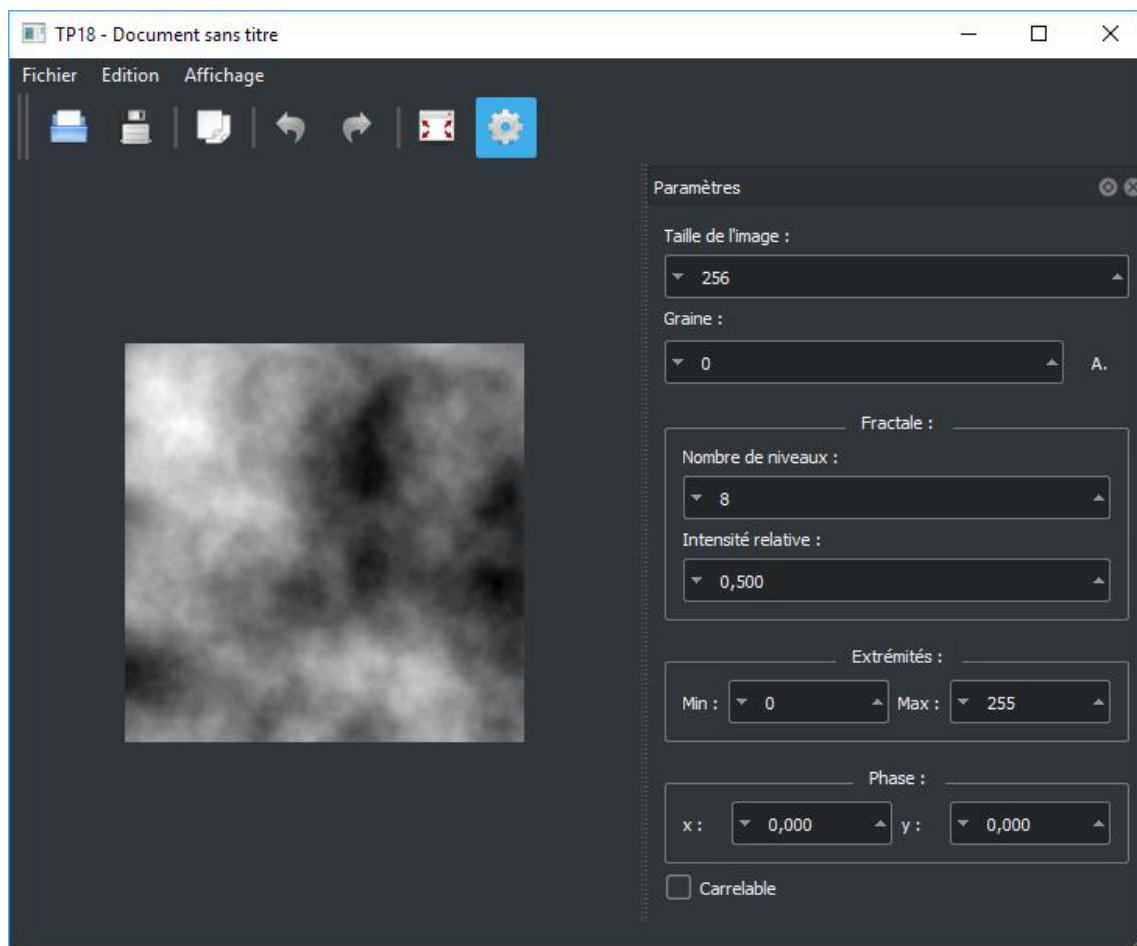


TP 28 – Observateur et Commande

Comme pour les TP précédents, la solution de ce TP est disponible par le gestionnaire de suivi de version Git via le dépôt distant [git@gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/TP28.git](https://gitlab-lepuy.iut.uca.fr/TPs-QualiteDeDev/TP28.git). Des étiquettes vous permettent de naviguer dans le dépôt parmi les différents exercices.

Objectif du TP

À travers une application de synthèse de texture procédurale, vous allez prendre en main deux design patterns classiques en POO : le pattern *Observateur* et le pattern *Commande*. Voici ce que vous obtiendrez :



Capture d'écran de l'application

Dans cette application, le réglage des paramètres dans la partie de droite de la fenêtre permet de générer une texture procédurale de [bruit de Perlin](#). En superposant différente résolution, il est possible de créer une texture de [bruit fractal](#). C'est ce qui est utilisé dans de petits logiciels comme 3DS Max.

Il n'est pas question ici de travailler sur l'algorithme, mais uniquement sur l'application construite autour, permettant de l'exécuter et de recueillir ses résultats.

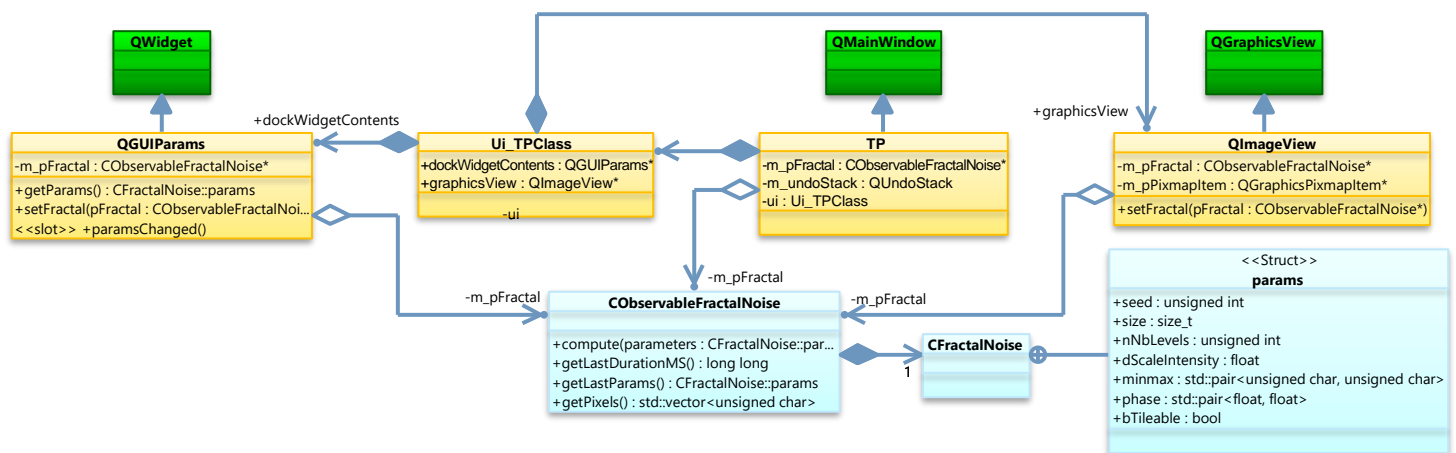
Exercice 0 Récupération des sources initiales

- 📄 Clonez le dépôt [git@gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/TP28.git](https://gitlab-lepuy.iut.uca.fr/TPs-QualiteDeDev/TP28.git).
- 📄 Vous récupérez ainsi un squelette de l'application.
- 📄 Compilez et exécutez.
- 📄 Toute l'interface est présente mais n'est pas fonctionnelle.

Voici ce que nous voulons réussir en termes de fonctionnalités durant ce TP :

1. La modification d'un des paramètres doit lancer automatiquement la génération d'une nouvelle texture.
2. La texture doit s'afficher automatiquement dans la partie de gauche de la fenêtre.
3. Le temps d'exécution de l'algorithme doit s'afficher dans la barre d'état de l'application.
4. Il doit être possible d'annuler et rétablir les modifications de paramètres, et cela doit se répercuter sur la texture et les paramètres affichés.

Voici le diagramme de classes de l'application :



Dans ce diagramme il faut identifier :

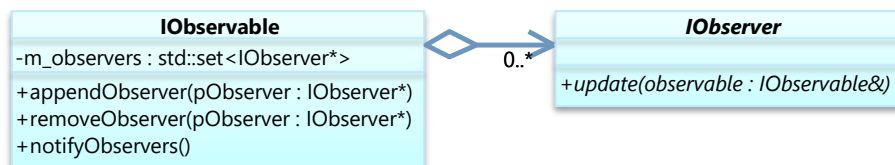
- 📄 La classe **TP** qui gère la fenêtre principale de l'application, contenant notamment un objet **ui.graphicsView** de type **QImageView** pour afficher la texture et un objet **ui.dockWidgetContents** de type **QGUIParams** pour contrôler les paramètres de l'algorithme. Elle permet également de charger / enregistrer les paramètres.
- 📄 La classe **QImageView** qui utilise son membre **m_pPixmapItem** pour afficher la texture sous forme d'un **QPixmap**.
- 📄 La classe **QGUIParams** qui, en plus de gérer les contrôles du formulaire, propose un accesseur unique vers un objet contenant tout le paramétrage saisi dans le formulaire.
- 📄 La classe **CObservableFractalNoise** qui adapte la classe **CFractalNoise** à l'usage de notre application.
- 📄 La structure **CFractalNoise::params** qui permet d'encapsuler tout le paramétrage de l'algorithme de synthèse d'image.

L'application est donc structurée autour d'une part d'un ensemble de classes de données (**CObservableFractalNoise**, **CFractalNoise** et **CFractalNoise::params**) et d'autre part de classes d'affichage/contrôle des données (**QImageView**, **TP** et **QGUIParams**). C'est un classique Modèle / Vue.

Exercice 1 Mise en place de la mise à jour automatique

Afin que les différents affichages de la texture (son image, ses paramètres et le temps de calcul) restent toujours cohérents, il est indispensable que l'ensemble des vues soit prévenu lors de toute modification. Pour cela, nous avons le choix entre utiliser le mécanisme signaux / slots de Qt, ou notre propre mécanisme autour d'un pattern **Observateur**. C'est cette seconde approche qui est retenue pour l'intérêt pédagogique.

Dans le code source, il vous est fourni un squelette de pattern Observateur constitué des classes **IObservable** et **IObserver** :



C'est donc la classe **CObservableFractalNoise** qui doit être rendue observable et les classes **QImageView**, **TP** et **QGUIParams** qui doivent l'observer.

1.1 Rendons la classe CObservableFractalNoise observable

- 📁 Faites hériter la classe **CObservableFractalNoise** de la classe **IObservable**.
- ➡ N'oubliez pas de notifier tous ses observateurs en appelant la fonction **notifyObservers()** en fin de la fonction **CObservableFractalNoise::compute**.

1.2 Rendons les classes d'affichage observatrices de la classe CObservableFractalNoise

- 📁 Sans détruire les héritages existants, faites hériter les classes **QImageView**, **TP** et **QGUIParams** de la classe **IObserver**.
- 📁 Vous devez alors associer ces observateurs à l'objet observé en appelant sa méthode **appendObserver()**.
 - ➡ L'objet **TP** est créé dans la fonction **main()**, c'est l'endroit idéal pour l'associer à l'objet **CObservableFractalNoise f**.
 - ➡ Les objets **QImageView** et **QGUIParams** sont créés dans le constructeur de la classe **TP** (via l'appel de **ui.setupUi()**). C'est donc ici que vous devez les associer à votre observable.
- 📁 Cela vous impose de surcharger la fonction virtuelle pure **update()** de chacune de ces classes. C'est la fonction qui est appelée par **IObservable::notifyObservers()**.
 - ➡ Pour la classe **QImageView**, le rôle de la fonction **update()** est de calculer un **QPixmap** depuis les pixels retournés par **m_pFractal->getPixels()** et de le fournir à l'élément d'affichage **m_pPixmapItem**.
 - ➡ Voici le code source de cette fonction :

```

//Récupère l'objet CObservableFractalNoise par un transtypage descendant
CObservableFractalNoise& fractal =
dynamic_cast<CObservableFractalNoise*>(observable);
//Crée un objet QImage depuis Les données brutes de la texture
QImage img(fractal.getPixels().data(),
fractal.getLastParams().size, fractal.getLastParams().size,

```

```

    fractal.getLastParams().size, QImage::Format_Grayscale8);
//Si La texture est carrelable
if (fractal.getLastParams().bTileable)
{
    //Alors on pose 4 carreaux de texture de façon à bien rendre compte l'effet carrelé
    QImage imgTile(fractal.getLastParams().size * 2, fractal.getLastParams().size * 2,
        QImage::Format_RGB888);
    {
        QPainter painter(&imgTile);
        painter.drawImage(0, 0, img);
        painter.drawImage(fractal.getLastParams().size, 0, img);
        painter.drawImage(0, fractal.getLastParams().size, img);
        painter.drawImage(fractal.getLastParams().size, fractal.getLastParams().size,
            img);
        QPen pen(QBrush(QColor(0,127,127,64)),0.0, Qt::DashLine);
        painter.setPen(pen);
        painter.drawLine(fractal.getLastParams().size, 0,
            fractal.getLastParams().size, fractal.getLastParams().size * 2);
        painter.drawLine(0, fractal.getLastParams().size,
            fractal.getLastParams().size * 2, fractal.getLastParams().size);
    }
    m_pPixmapItem->setPixmap(QPixmap::fromImage(imgTile));
}
else
    //Sinon, La texture est copiée directement
    m_pPixmapItem->setPixmap(QPixmap::fromImage(img));

//Met à jour La taille de scène rendue dans l'objet QGraphicsView
m_scene.setSceneRect(m_pPixmapItem->boundingRect());

```

- ➡ À ce stade, vous devriez pouvoir compiler l'application. À l'exécution, la texture s'affiche lors des changements de paramètres. La mécanique est la suivante : le changement d'un paramètre déclenche un signal connecté au slot **QGUIParams::paramsChanged**. Ce slot lance un nouveau calcul sur l'objet **CObservableFractalNoise**, qui prévient ces observateurs qu'il a été modifié (**notifyObservers()**). La fonction **update()** de l'objet **QImageView** est donc appelée et convertit les pixels bruts en une image affichable par Qt. Cependant, la mécanique n'est pas encore complète, car le temps d'exécution ne s'affiche pas et l'affichage des paramètres n'est pas mis à jour en cas de chargement de fichier.
- ➡ Pour la classe **TP**, la fonction **update()** doit simplement afficher le temps de calcul dans la barre d'état.
- ➡ Voici son code :

```

CObservableFractalNoise& fractal =
dynamic_cast<CObservableFractalNoise&>(observable);
ui.statusBar->showMessage(tr("Temps écoulé : %1 ms")
    .arg(fractal.getLastDurationMS()));

```

- ➡ Dès lors, le temps de calcul d'une nouvelle texture s'affiche dans la barre d'état de l'application, selon le même mécanisme que précédemment.
- ➡ Pour la classe **QGUIParams**, la fonction **update()** doit mettre à jour l'affichage des paramètres dans les contrôles du formulaire. Cependant, il ne faut pas que cette modification des champs du formulaire relance un calcul par le jeu des signaux et slots présents (ceux qui font que dès qu'un paramètre est changé, un nouveau calcul est lancé).

Pour éviter de tomber dans cet écueil, on utilise la méthode **blockSignals()** qui permet de bloquer les signaux d'un **QObject**.

⇒ Voici le code :

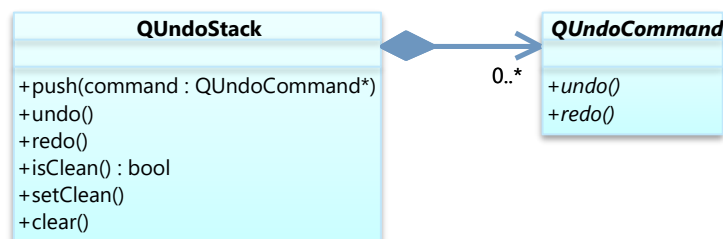
```
CObservableFractalNoise& fractal =
dynamic_cast<CObservableFractalNoise>(observable);
blockSignals(true);
setSize(fractal.getLastParams().size);
setSeed(fractal.getLastParams().seed);
setNbLevels(fractal.getLastParams().nNbLevels);
setRelativeIntensity(fractal.getLastParams().dScaleIntensity);
setMin(fractal.getLastParams().minmax.first);
setMax(fractal.getLastParams().minmax.second);
setPhaseX(fractal.getLastParams().phase.first);
setPhaseY(fractal.getLastParams().phase.second);
setTileable(fractal.getLastParams().bTileable);
blockSignals(false);
```

⇒ Maintenant, la mécanique est complète. Vous pouvez le vérifier en chargeant un fichier de paramètres. La fonction **TP::open()** lance le calcul sur les nouveaux paramètres chargés. Par conséquent, l'objet **CObservableFractalNoise** prévient ses observateurs qu'il a été modifié. Les fonctions **update()** précédemment codées sont donc appelées, est celle de la classe **QGUIParams** affiche les paramètres dans les champs du formulaire.

Exercice 2 Annuler / rétablir

Dans l'état, on a l'impression que le logiciel fonctionne. Cependant, deux petites flèches restent désespérément grises : **Annuler** et **Rétablir**. Pour les faire fonctionner, nous allons mettre en place le pattern **Commande**.

Pour rappel, le pattern commande fonctionne selon le principe suivant :



Les commandes de l'application sont encapsulées dans des objets de type **QUndoCommand** via l'héritage. Ces objets sont alors stockés dans une liste (**QUndoStack**). Au moment du stockage dans la liste, la méthode **QUndoCommand::redo()** de la commande est appelée, exécutant de ce fait les actions de la commande. La pile d'annulation est ensuite manipulée à l'aide des méthodes **QUndoStack::undo()** et **QUndoStack::redo()**.

Des méthodes accessoires sont également disponibles. Parmi elles, j'ai retenu pour vous les méthodes **isClean()** et **setClean()** qui permettent de définir un état « propre » de la pile. Cet état peut correspondre par exemple à l'état des données enregistrées via le menu de l'application. Lorsque l'état de propreté est modifié par l'ajout d'une nouvelle commande ou par une action **undo()** ou **redo()**, un signal est émis. Ce signal a été connecté pour vous dans le constructeur de **TP** à un slot

qui met à jour le titre de la fenêtre, faisant apparaître une « * » lorsque la pile est « sale » (données non enregistrées).

De plus une méthode `clear()` est utile lorsque de nouvelles données sont chargées depuis un fichier. Cela permet de réinitialiser la pile des commandes afin de ne pas annuler / rétablir des actions incohérentes avec les nouvelles données chargées.

La classe `TP` est déjà munie d'un membre `QUndoStack m_undoStack`. Il va donc falloir lui pousser des objets personnalisés depuis la classe `QUndoCommand`.

2.1 Création d'une classe de commande


Le principe de cette classe sera de stocker en elle l'état des paramètres avant et après le calcul. Ainsi, annuler correspondra à réappliquer les paramètres qui étaient présents avant l'appel de l'action alors que rétablir consistera à appliquer les paramètres demandés.


- 📄 Ajoutez au projet une nouvelle classe `QComputeCommand` héritant de la classe `QUndoCommand`.
- ➡ Munissez cette classe des données membres suivantes :
 - ➡ Un pointeur vers l'objet `CObservableFractalNoise` à manipuler pour réaliser les calculs
 - ➡ Un objet `CFractalNoise::params` pour stocker les paramètres présents avant l'action
 - ➡ Un objet `CFractalNoise::params` pour stocker les paramètres de l'action
- ➡ Munissez cette classe des fonctions membres suivantes :
 - ➡ Un constructeur avec deux paramètres permettant d'initialiser le pointeur vers le `CObservableFractalNoise` et les paramètres de l'action. Dans son corps, il devra enregistrer les paramètres présents avant dans l'objet `CObservableFractalNoise`. Ces paramètres sont accessibles via la méthode `CObservableFractalNoise::getParams()`.
 - ➡ La surcharge de la méthode `redo()` qui lancera le calcul avec les paramètres de l'action
 - ➡ La surcharge de la méthode `undo()` qui lancera le calcul avec les paramètres présents avant la réalisation de l'action.

2.2 Utilisation de la classe `QComputeCommand`

Les actions étant lancées au moment des modifications des paramètres, il faut créer les objets de type `QComputeCommand` dans la classe `QGUIParams`.

- 📄 Modifiez la classe `QGUIParams` pour lui ajouter un membre de type pointeur vers une `QUndoStack`.
- ➡ Ajoutez-lui également un accesseur en écriture, de façon à pouvoir initialiser cette donnée depuis la classe `TP`.
- 📄 Dans le slot `QGUIParams::paramsChanged()`, modifiez le corps de la fonction pour ne pas faire l'appel direct de la méthode `compute()`, mais plutôt de créer un objet `QComputeCommand` adéquat que vous pousserez dans l'objet `QUndoStack` que vous venez d'ajouter.

-  Dans le constructeur de la classe **TP**, utilisez l'accessor du pointeur vers **QUndoStack** de la classe **QGUIParams** pour le faire pointer vers le membre **TP::m_undoStack**.

 L'application est maintenant totalement fonctionnelle. Si vous souhaitez l'utiliser (il est possible d'exporter les images en PNG ou de les copier dans le presse-papier) je vous recommande de compiler l'application en mode Release. Comparez les temps de calculs entre le mode Release et Debug, et vous comprendrez où se trouvent les différences entre les deux modes.