Cahier Des Spécifications
-
Simulation de réseaux de files d'attente

ADOUANE Cylia
ALI AHMEDI Mycipssa
BONNET Ludivine
DOUBI Dylan
HAMMAD Amir
HELLAL Ouiza
LAMOUR Lauriane
MOLINER Emma

21 Avril 2020

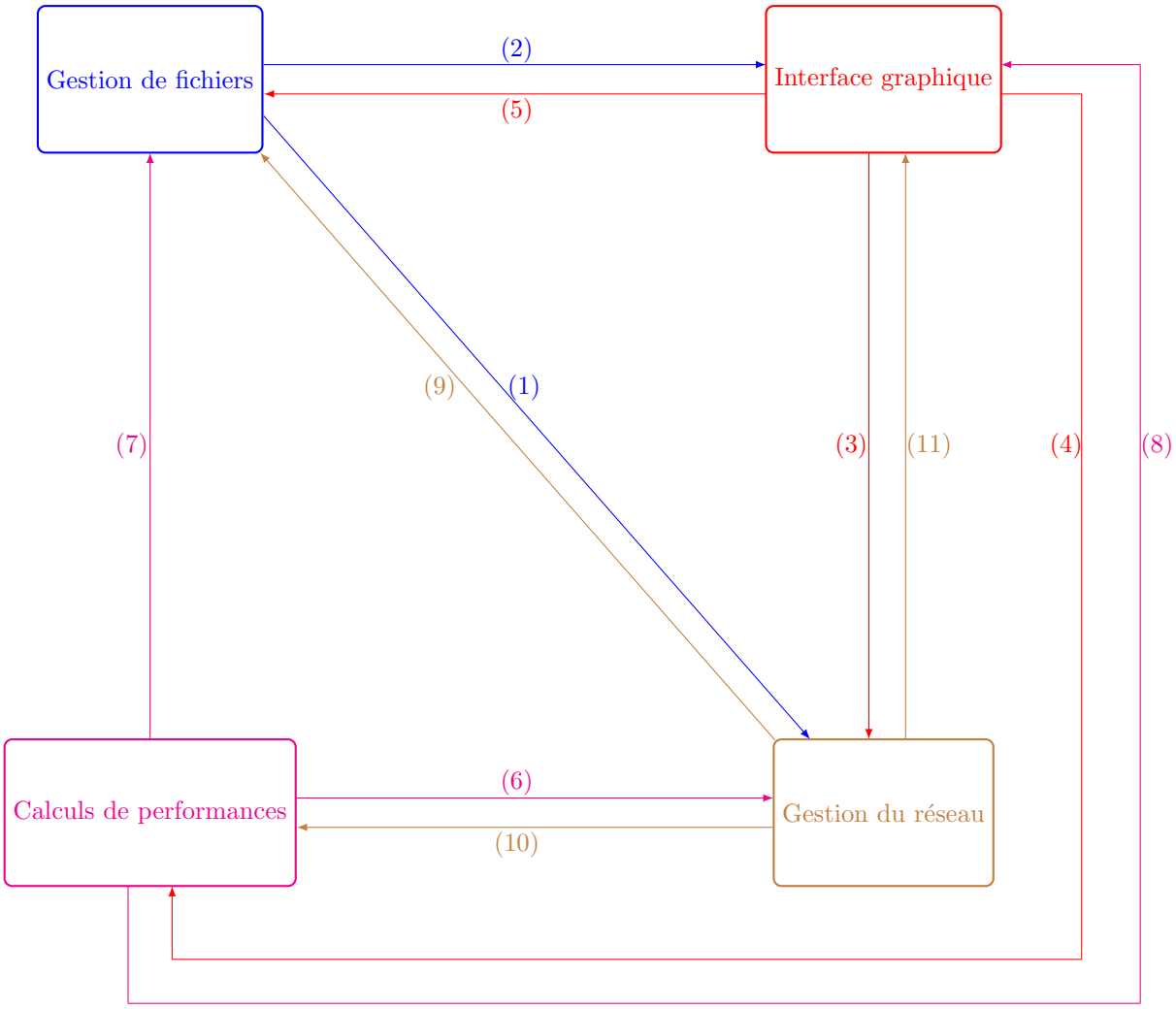
Contents

1 Introduction

Ce projet vise à réaliser une simulation à événement discrets d'un réseau de files d'attente réalisée grâce à un échéancier, en vue d'en étudier les performances. Ces données obtenues seront analysées par un particulier ou des spécialistes du domaine ciblé en guise d'améliorer le réseau simulé.

Pour se faire, le projet est réalisé en "C++", entre autres pour le fort typage que propose ce langage assurant l'intégrité des données traitées ,pour la vitesse d'exécution et pour son aspect orientée-objet. La partie orientée-objet apporte les notions de polymorphisme et d'héritage, notions indispensables au bon fonctionnement de l'application. Le langage "C++" est compatible avec l'API (*Application Programming Interface*) Qt proposant des composants pour créer une interface graphique et qui est celle choisie pour la réalisation de l'interface graphique de l'application.

2 Organigramme et fonctionnalités



2.1 Fonctionnalités - Gestion du réseau

- **Distribuer les clients dans les files au début de la simulation.**

Cette fonctionnalité est associée à la méthode *void distribution_initiale(const int nb_clients_initial)* de la classe *Simulation*

- **Gérer l'échéancier.**

Cette fonctionnalité est associée à la classe *Echeancier* ainsi qu'aux méthodes *void executer_evenement()*, *void EntreeFile(Evenement evenement_sorti)*, *void EntreeServeur(Evenement evenement_sorti)* et *void SortieServeur(Evenement evenement_sorti)* se trouvant dans la classe *Simulation*

- **Distribuer les clients dans les files/sortie pendant la simulation**

Cette fonctionnalité est associée à la méthode *void routage_clients()* de la classe *Simulation*. Selon l'objet qui appelle cette méthode, l'implémentation de la méthode appelée ne sera pas la même.

- **Ordonnancer les clients dans les files selon l'ordonnancement de la file**

Cette fonctionnalité est associée à la méthode *void ordonnancer_clients(Client& client)* de la classe *File*.

- **Calculer le temps d'attente de chaque client**

Cette fonctionnalité correspond à la méthode *void modifier_tps_attente(const int date_actuelle)* se trouvant dans la classe *Client*.

- **Calculer le temps de service de chaque client**

Cette fonctionnalité correspond à la méthode *void calculer_tps_service()* se trouvant dans la classe *Client*.

- **Calculer le taux d'arrivée pour chaque file**

Cette fonctionnalité dépend de la distribution d'arrivée de la file. Si il est constant, il est donné par l'utilisateur et est représenté par l'attribut *distribution_arrivee* dans la classe *File*, sinon il suit une loi de Poisson et est calculé en suivant cette loi dans la méthode *int calculer_taux_arrivee_poisson(const int lamba)*.

2.2 Fonctionnalités - Gestion du fichier

- **Sauvegarder un réseau dans un fichier**

Cette fonctionnalité correspond à la fonction *void lecture_fichier_reseau(Reseau * reseau, const string nomfic)* se trouvant dans le header *lecture_ecriture.hh* ainsi qu'aux sous-fonctions de cette fonction qui sont *void lecture_file(File * file, const string ligne)* et *void lecture_type(ClasseClient * cc, const string ligne)*.

- **Sauvegarder les mesures de performances**

Cette fonctionnalité correspond aux fonctions *void ecriture_fichier_performancesRO(const string nomfic, HistoriqueRO& h)* et *void ecriture_fichier_performancesRF(const string nomfic, HistoriqueRF& h)*. Ces deux fonctions se trouvent dans le header *lecture_ecriture.hh* et leurs appels dépendent du type du réseau.

- **Charger un réseau à partir d'un fichier**

Cette fonctionnalité est associée à la fonction *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)* et aux sous-fonctions *void ecriture_fichier_file(ofstream * fic, Reseau& reseau)* et *void ecriture_fichier_classe(ofstream * fic, Reseau& reseau)* se trouvant dans le header *lecture_ecriture.hh*.

2.3 Fonctionnalités - Calculs de performances

- **Calculer le nombre moyen de clients dans le réseau**

Cette fonctionnalité est associée à la méthode *void calcul_nb_moyen_clients(list<mesure>& listeNbClients)* de la classe *PerfsRF*.

- **Calculer le nombre moyen de clients pour une file donnée**

Cette fonctionnalité est associée à la méthode *void calcul_nb_moyen_clients_file()* de la classe *PerfsFile*.

- **Calculer le temps moyen de séjour**

Cette fonctionnalité est associée à la méthode *void calcul_nb_moyen_clients_file()* de la classe *PerfsFile*.

- **Calculer le temps d'attente moyen pour une file donnée**

Cette fonctionnalité est associée à la méthode *void calcul_tps_attente_moyen()* de la classe *PerfsFile*.

- **Calculer le temps de service moyen**

Cette fonctionnalité est associée à la méthode *void calcul_tps_service_moyen()* de la classe *PerfsFile*.

- **Calculer le temps de réponse moyen**

Cette fonctionnalité correspond à la méthode *void calcul_tps_reponse_moyen()* se trouvant dans la classe *PerfsFile*.

- **Calculer le taux d'utilisation du serveur d'une file donnée**

Cette fonctionnalité correspond à la méthode *void calcul_taux_utilisation_serveurs(const int duree_simulation)* se trouvant dans la classe *PerfsFile*.

- **Calculer le temps d'inter-arrivée moyen d'une file donnée**

Cette fonctionnalité correspond à la méthode *void calcul_tps_interarrivee_moyen()* se trouvant dans la classe *PerfsFile*.

- **Calculer le taux de perte pour une file**

Cette fonctionnalité correspond à la méthode *void calcul_taux_perte()* se trouvant dans la classe *PerfsFile*.

- **Calculer le débit d'entrée (Pour un réseau ouvert)**

Cette fonctionnalité correspond à la méthode *void calcul_debit_entree(list<mesure> & ListeNbClientEntrant)* se trouvant dans la classe *PerfsRO*.

- **Calculer le débit de sortie (Pour un réseau ouvert)**

Cette fonctionnalité correspond à la méthode *void calcul_debit_sortie(list<mesure> & ListeNbClientSortant)* se trouvant dans la classe *PerfsRO*.

2.4 Fonctionnalités - Interface

- **Ajouter une station**

Cette fonctionnalité est reliée au bouton *AddQueue* qui est lui-même relié au slot *void Add_Queue()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Retirer une station**

Cette fonctionnalité est reliée au bouton *RemoveQueue* qui est lui-même relié au slot *void Remove_Queue()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Déplacer une station**

Cette fonctionnalité est reliée au bouton *MoveQueue* qui est lui-même relié au slot *void Move_Queue()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Configurer une station**

Cette fonctionnalité est reliée au bouton *ConfigureQueue* qui est lui-même relié au slot *void Configure_Queue()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal. Le slot ouvre une boîte de dialogue de type *DialogConfigFile*.

- **Ajouter une classe**

Cette fonctionnalité est reliée au bouton *AddClass* qui est lui-même relié au slot *void Add_Class()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal. Le slot ouvre une boîte de dialogue de type *DialogAddClass*.

- **Retirer une classe**

Cette fonctionnalité est reliée au bouton *RemoveClass* qui est lui-même relié au slot *void Supp_Class()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Configurer une classe**

Cette fonctionnalité est reliée au bouton *ConfigureClass* qui est lui-même relié au slot *void configurer_classes()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal. Le slot ouvre une boîte de dialogue de type *DialogConfigClass*.

- **Visualiser les mesures de performances**

Cette fonctionnalité est reliée au bouton *Results* qui est lui-même relié au slot *void Results()* se trouvant dans la classe *FenetreSimulationRF* et dans la classe *FenetreSimulationRO*. Un slot est une méthode s'activant à la réception d'un signal.

- **Lancer la simulation**

Cette fonctionnalité est reliée au bouton *Start* qui est lui-même relié au slot *void Start()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Mettre en pause la simulation**

Cette fonctionnalité est reliée au bouton *Pause* qui est lui-même relié au slot *void Pause()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Reprendre la simulation**

Cette fonctionnalité est reliée au bouton *Resume* qui est lui-même relié au slot *void Resume()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Arrêter la simulation**

Cette fonctionnalité est reliée au bouton *Stop* qui est lui-même relié au slot *void Stop()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Sauvegarder le réseau**

Cette fonctionnalité est reliée au bouton *Save* qui est lui-même relié slot *void Save()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Ouvrir un fichier**

Cette fonctionnalité est reliée au bouton *Open* qui est lui-même relié au slot *void Open()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Proposer un réseau déjà existant**

Cette fonctionnalité est reliée au menu *menuPropreReseau* qui contient les boutons *reseau1*, *reseau2*, *reseau3* et *reseau4* qui sont eux-même relié respectivement aux slots *void charger_reseau1()*, *void charger_reseau2()*, *void charger_reseau3()* et *void charger_reseau4()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Zoomer**

Cette fonctionnalité est reliée au bouton *Zoom_In* qui est lui-même relié au slot *void Zoom_In()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Dézoomer**

Cette fonctionnalité est reliée au bouton *Zoom_Out* qui est lui-même relié au slot *void Zoom_Out()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Supprimer le réseau**

Cette fonctionnalité est reliée au bouton *Reset* qui est lui-même relié au slot *void Reset()* se trouvant dans la classe *FenetreCreation*. Un slot est une méthode s'activant à la réception d'un signal.

- **Afficher l'historique des performances**

Cette fonctionnalité est reliée au bouton *History* qui est lui-même relié au slot *void History()* se trouvant dans la classe *FenetreSimulationRF* et dans la classe *FenetreSimulationRO*. Un slot est une méthode s'activant à la réception d'un signal. Ce slot ouvre une boîte de dialogue de type *DialogHistoriqueRO* ou de type *DialogHistoriqueRF* en fonction du type du réseau.

2.5 Liens entre les modules

Le lien **(1)** correspond à la demande des informations sur le réseau de *Gestion de Fichiers* vers *Gestion du Réseau*. Cette demande est représentée par la présence d'un objet de la classe *Reseau* en paramètre de la fonction *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)*.

Le lien **(2)** correspond aux demandes des noms de fichiers utile pour le chargement ou la sauvegarde d'un réseau, ainsi que pour la sauvegarde des résultats de performances. Ce lien se fait depuis le module *Gestion de Fichiers* vers le module *Interface Graphique*. Il est représenté par la nécessité d'avoir le nom d'un fichier dans les fonctions du module *Gestion de Fichiers* suivantes : *void lecture_fichier_reseau(Reseau * reseau, const string nomfic)*, *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)*, *void ecriture_fichier_performancesRF(const string nomfic,HistoriqueRF& h)* et *void ecriture_fichier_performancesRO(const string nomfic, HistoriqueRO& h)*.

Le lien **(2)** correspond également à l'envoi des données du réseau chargé à partir d'un fichier. Cet envoi est représenté par l'appel de la fonction *void lecture_fichier_reseau (Reseau * reseau, const string nomfic)* dans le slot *void Open()* se trouvant dans la classe *FenetreCreation*, elle-même appartenant au module *Interface Graphique*.

Le lien **(3)** correspond aux signaux contrôlant la simulation, signaux envoyés de *Interface Graphique* vers *Gestion du Réseau*. Ces signaux correspondent aux boutons *Start*, *Pause*, *Stop*, *Resume* associés respectivement aux slots *void Start()*, *void Pause()*, *void Stop()* et *void Resume()* et se trouvant dans la classe *FenetreSimulation*, classe appartenant au module *Interface Graphique*. Ces signaux agissent sur la simulation représentée par un objet de la classe *SimulationRF*, si le réseau est fermé ou par un objet de la classe *SimulationRO* si le réseau est ouvert. Ces classes appartiennent au module *Gestion du Réseau*.

Ce lien correspond également à l'envoi de la durée de simulation, récupérée lors d'une demande à l'utilisateur par une boîte de dialogue dans le slot *void Start()* se trouvant dans la classe *FenetreSimulation*, classe appartenant au module *Interface Graphique*. Sa récupération permet d'initialiser l'attribut *duree_simulation* se trouvant dans la classe *SimualtionRF*, si le réseau est fermé (*SimulationRO*, si le réseau est ouvert), classes appartenant au module *Gestion du Réseau*.

Enfin, ce lien permet d'envoyer les données du réseau dessiné par l'utilisateur ou charger depuis un fichier. Cet envoi est représenté par la présence d'un objet de la classe *Reseau*, classe du module *Gestion du réseau*, dans la classe *FenetreCreation*, classe du module *Interface Graphique*.

Le lien **(4)** correspond à la demande de calculs des performances de *Interface Graphique* vers *Calculs de Performances*. Cette demande est représentée par l'activation du bouton *Results* se trouvant dans la classe *FenetreSimulationRO* et *FenetreSimulationRF*, appelant le slot *void Results()* qui enclenche les calculs de performances se faisant à l'aide des classes *PerfRF* si le réseau est fermé (*PerfRO* si le réseau est ouvert) et *PerfsFile*, classes du module *Calculs de Performances*.

Le lien **(5)** représente les signaux de demande de chargement ou de sauvegarde du réseau et de sauvegarde des performances. Ce lient se fait du module *Interface Graphique* vers *Gestion de Fichiers*. Le signal de chargement d'un réseau est activé par le bouton *Open* provenant du module *Interface Graphique*, enclenchant le slot *void Open()*, lui-même appelant la fonction *void lecture_fichier_reseau(Reseau * reseau, const string nomfic)* provenant du module *Gestion du Fichiers*. Le signal de sauvegarde d'un réseau est activé par le bouton *Save* provenant du module *Interface Graphique*, enclenchant le slot *void Save()*, lui-même appelant la fonction *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)* provenant du module *Gestion de Fichiers*. Le signal de sauvegarde des performances est activé par le bouton *Stop* provenant du module *Interface Graphique*, enclenchant le slot *void Stop()*, lui-même appelant la fonction *void ecriture_fichier_performancesRF(const string nomfic,HistoriqueRF& h)* si le réseau est fermé et *void ecriture_fichier_performancesRO(const string nomfic, HistoriqueRO& h)* si le réseau est ouvert. Ces fonctions proviennent du module *Gestion de Fichiers*.

Ce lien représente également le nom des fichiers utilisés pour sauvegarder un réseau, charger un réseau ou sauvegarder les performances du réseau. Le nom d'un fichier est demandé lors de l'activation d'un des boutons suivants : *Save*, *Stop* ou *Open*. Le nom récupéré est passé en paramètre des fonctions *void lecture_fichier_reseau(Reseau * reseau, const string nomfic)*, *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)*, *void ecriture_fichier_performancesRF(const string nomfic,HistoriqueRF& h)* et *void ecriture_fichier_performancesRO(const string nomfic, HistoriqueRO& h)* provenant du module *Gestion de Fichier* et appelées en fonction du bouton activé.

Le lien **(6)** représente la demande des données utilisées lors des calculs des performances. Ce lien se fait de *Calculs de Performances* vers *Gestion du Réseau*. Cette demande est représentée par l'utilisation des données d'un réseau, représentées par *DonneesCalculsRF* si le réseau est fermé (*DonneesCalculsRO* si le réseau ouvert), classes du module *Gestion du Réseau*, dans la classe *PerfRF* (respectivement *PerfRO*), classes du module *Calculs de Performances*. Les données pour les calculs de performances des files sont contenues dans la classe *File* appartenant au module *Gestion du Réseau*. Elles sont utilisées par la classe *PerfFile*, classe du module *Calculs de Performances*.

Ce lien correspond également à l'envoi de la mise à jour des données des files. Cette mise à jour est récupérée de la

classe *PerfFile*, classe du module *Calculs de Performances* et est appliquée sur les attributs correspondants de la classe *File*, classe du module *Gestion du Réseau*.

Le lien **(7)** correspond à l'envoi des résultats des calculs de performances et à sa sauvegarde dans *Gestion de Fichiers*. Il se fait de *Calculs de Performances* à *Gestion de fichier*. Il correspond à l'appel de la fonction *void ecriture_fichier_performancesRO (const string nomfic,HistoriqueRO& h)* si c'est un réseau ouvert et la fonction *void ecriture_fichier_performancesRF(const string nomfic, HistoriqueRF& h)* si c'est un réseau fermé. Ces fonctions appartiennent au module *Gestion de Fichier*.

Le lien **(8)** correspond à l'envoi des calculs de performances de *Calculs de Performance* vers *Interface Graphique* afin de les afficher. Les résultats des calculs sont contenus dans *PerformancesRO*, si réseau ouvert, ou dans *PerformancesRF*, si le réseau fermé, structures du module *Calculs de Performances* et sont utilisés dans la classe *DialogResultatsRO*, classe du module *Interface Graphique*.

Le lien **(9)** correspond à la sauvegarde du réseau dans un fichier. Le lien va de *Gestion du Réseau* à *Gestion de Fichiers*. Cela est représenté par l'appel de la fonction *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)* dans le module *Gestion de Fichiers*. Le paramètre *reseau* correspond au réseau à sauvegarder.

Le lien **(10)** correspond à l'envoi des données du réseau pour le calcul des performances. Le lien va de *Gestion du Réseau* vers *Calculs de Performances*. Les données du réseau sont représentées par *DonneesCalculsRO*, situé dans *SimulationRO*, et *DonneesCalculsRF* situé dans *SimulationRF*. Elles sont utilisées dans la classe *PerfsRO* ou *PerfsRF* en fonction du type du réseau. Les données des files sont situées dans les files. Elles sont utilisées dans la classe *PerfsFile*. Les performances du réseau et des files sont regroupées dans un objet *PerformancesRF* ou *PerformancesRO* en fonction du type de réseau.

Le lien **(11)** correspond à l'affichage du réseau en fonction des changements qui auront été apportés. Le lien se fait des modules *Gestion du Réseau* vers *Interface Graphique*. La fenêtre de création a accès au réseau directement par attribut. La fenêtre de simulation d'un réseau ouvert à accès au réseau à travers l'attribut de type *SimulationRO* et celle d'un réseau fermé, à travers l'attribut de type *SimulationRF*.

3 Module Gestion du réseau

Le module Gestion du réseau s'occupe de la simulation à événements discrets sous forme d'un échéancier du réseau créé/choisi par l'utilisateur. Il se compose de 8 classes organisées de la façon suivante :

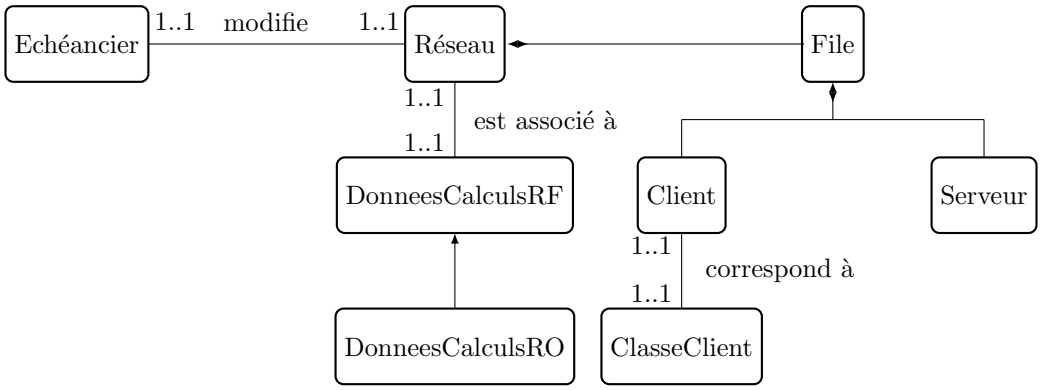


Diagramme de classes

La classe centrale de ce module est la classe *Reseau*. Un objet de cette classe contiendra la sauvegarde logique du réseau créé ou choisi par l'utilisateur. Un réseau se compose d'un ensemble de files représentées par la classe *File*. Elles sont elles-mêmes composés d'un ensemble de clients représentés par la classe *Client* et d'un ensemble de serveurs représentés par la classe *Serveur*. Les clients sont catégorisés en classes qui sont modélisées par la classe *ClasseClient*. Une classe de base est créé automatiquement pour éviter la simulation d'un réseau sans clients.

Lors de la simulation, les changements dans le réseau sont orchestrés par un échéancier, modélisé par un objet de la classe *Echéancier*. Un événement est sorti de l'échéancier pour changer l'état du réseau. Selon le type de réseau (ouvert ou fermé), les changements d'états du réseau sont sauvegardés par un objet de classe différente car des événements modifiants l'état du réseau peuvent se produire dans un réseau ouvert et pas dans un réseau fermé. Si le réseau est fermé, l'objet associé est de la classe *DonneesCalculsRF*, sinon il est de la classe *DonneesCalculsRO*. La classe *DonneesCalculsRO* hérite

de la classe *DonnesCalculsRF* car elle sauvegarde également les mêmes changements qui se produisent dans un réseau fermé.

3.1 Classe ClasseClient

```
#include <vector> [5]
```

Cette bibliothèque permet de créer et gérer les vecteurs. Elle est utile pour garder le routage des clients de la classe, choisi par l'utilisateur.

```
#include <iostream>
```

Cette bibliothèque gére l'entrée et la sortie standard.

```
#include "File.hh"
```

Cette bibliothèque contient la classe *File* nécessaire pour la structure *elemRoutage*.

```
#include <string>
```

Cette bibliothèque gère les chaînes de caractères. Elle est utile pour passer le routage en paramètre de la méthode *void definir_routage(const string newRoutage)*.

```
#include <stdexcept> [1][2]
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

L'appel de l'environnement std permet d'alléger le code en évitant d'écrire "std::" pour les classes faisant partie de l'environnement standard.

```
class ClasseClient{

private:

int IDClasseClient;
```

Cet attribut correspond à l'identifiant de la classe. Il permet de différencier les classes présentes dans la simulation.

```
vector <File *> routage;
```

Cet attribut correspond à la liste de routage de la classe. Elle est composée d'un ensemble de files que les clients de la classe doivent parcourir. La classe *vector* est utile car une extension du vecteur est possible et l'accès aux éléments du vecteur se fait par index ce qui permet de connaître la position du client dans le routage.

```
int priorite;
```

Cet attribut correspond à la priorité de la classe. Plus l'entier est petit et plus un client de ce type est prioritaire dans la file dans le cas de l'ordonnancement avec priorité.

```
int nbClients;
```

Cet attribut correspond au nombre de clients dans la classe.

```
public:

ClasseClient(const int ID, const int priorite);
```

Ce constructeur initialise l'attribut *IDClasseClient* avec la valeur *ID*, l'attribut *priorite* avec la valeur *priorite* et appelle le constructeur par défaut de la classe *vector* pour intialiser l'attribut *routage*. Ce constructeur est nécessaire car l'identifiant d'une classe diffère des autres classes et la priorité est choisie par l'utilisateur donc ils doivent être passés en argument pour initialiser les attributs correspondants. Les deux entiers passés en paramètre sont constants pour éviter leur modification. Une exception de type *invalid_argument* est levée si des entiers invalides sont passés.

~ClasseClient();

Le destructeur va désallouer le vecteur *routage*.

```
int getIDClasseClient();
int getNbClients();
vector<File *>& getRoutage();
int getPriorite();
```

Ces accesseurs permettent d'accéder aux attributs privés de la classe.

```
void setPriorite(const int numero);
```

Cet accesseur permet d'attribuer la valeur *numero* à l'attribut *priorite*. Il est utile si l'utilisateur assigne une valeur de priorité à la classe et qu'il souhaite la modifier. Une exception de type *invalid_argument* est levée si un entier invalide est passé .

```
void setNbClients(const int nombre);
```

Cet accesseur permet d'attribuer la valeur *nombre* à l'attribut *nbClients*. Ce nombre augmente lors de la création des clients routants dans le réseau. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void supprimer_file_routage(const int IDfile);
```

Cette méthode supprime la file ayant pour identifiant *IDfile* du vecteur *routage*. L'identifiant de la file est passé en paramètre sous forme d'un entier constant pour éviter de la modifier. Cette méthode est utilisée lorsque l'utilisateur supprime une file du réseau. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void definir_routage(const string newRoutage);
```

Cette méthode va définir le routage avec un objet de type *string* correspondant aux identifiants des files du routage dans l'ordre. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
File * trouver_file(const int pos_actuelle);
```

Cette méthode prend en paramètre un entier constant pour éviter sa modification correspondant à la position de la file que l'on cherche et renvoie un pointeur sur la file correspondante afin de récupérer l'objet se trouvant à l'adresse donnée par le pointeur et pouvoir le modifier. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
File* prochaine_file_routage(const int pos_precedente);
```

Cette méthode prend en paramètre un entier constant ,pour éviter sa modification, correspondant à la position de la file à laquelle se trouve le client et renvoie un pointeur sur la file suivante afin de récupérer l'objet se trouvant à l'adresse donnée et pouvoir le modifier. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
friend ostream& operator << (ostream &flux, ClasseClient const& cc)
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques de l'objet comme le vecteur *routage* et la priorité. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *ClasseClient*, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur un objet de la classe *ClasseClient* pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.2 Classe Client

```
#include <iostream>
```

Cette bibliothèque gére l'entrée et la sortie standard.

```
#include <cmath> [4]
```

Cette bibliothèque contient des fonctions permettant l'utilisation d'opérations mathématiques. Elle est utile pour le calcul du temps du service du client nécessitant des fonctions mathématiques.

```
#include "ClasseClient.hh"
```

Cette bibliothèque doit être incluse car un client possède un attribut de type *ClasseClient*.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

L'appel de l'environnement std permet d'alléger le code en évitant d'écrire "std::" pour les classes faisant partie de l'environnement standard.

```
class Client{

private :

int ID_client ;
```

Cet attribut correspond à l'identifiant du client et permet de différencier les clients les uns des autres.

```
ClasseClient * classe;
```

Cet attribut correspond à la classe du client. Il permet notamment de savoir la priorité du client par rapport aux autres clients et le routage qu'il suivra. L'attribut correspond à un pointeur sur un objet *ClasseClient* car il est nécessaire d'avoir le bon état du routage. Or, il peut être modifié au cours de la simulation. Il faut donc avoir accés à l'objet qui aura ces modifications.

```
int tps_service ;
```

Cet attribut correspond au temps de service du client. Cette valeur est attribuée au client au moment où il entre dans la file.

```
int tps_attente ;
```

Cet attribut correspond au temps d'attente du client. Cette valeur va être incrémentée dès qu'un client rentre dans un serveur.

```
int pos_actuel_routage;
```

Cet attribut correspond à la position du client dans le routage. Il est utile pour déterminer la prochaine file vers laquelle il devra se diriger.

```
int date_arrivee_file;
```

Cet attribut correspond à la date d'arrivée du client dans une file d'attente. Il est initialisé à -1 quand le client ne se trouve pas dans une file. Cette date permet de calculer le temps d'attente du client dans la file.

```
public :

Client(const int ID, ClasseClient * c);
```

Ce constructeur initialise l'attribut *ID_client* avec la valeur *ID* passée en paramètre, l'attribut *classe* avec le pointeur *c* passé en paramètre et les entiers autres que l'attribut *ID_client* sont initialisés à 0. L'adresse d'un objet de type *ClasseClient* est donnée afin de toujours avoir la bonne version de l'objet. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

~Client();

Les attributs de la classe sont supprimés automatiquement lors de la destruction de l'objet. Donc le destructeur est vide.

```
int getID_client();
int getTps_attente();
int getTps_service();
ClasseClient* getClasse();
File* getFileActuelle();
int getDateArriveeFile();
```

Ce sont des accesseurs qui permettent d'accéder aux attributs privés de la classe.

```
void setDateArrivee(const int newDate);
void setPositionActuelle(const int numero);
```

Ces méthodes vont permettre de modifier les attributs privés *date_arrivee_file* et *pos_actuel_routage*. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
friend ostream& operator << (ostream &flux, Client const& c );
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques du client. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs du *Client c*, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur le client pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
void modifier_tps_attente(const int date_actuelle);
```

Cette méthode modifie la valeur de l'attribut *tps_attente* avec l'entier *date_actuelle* passé en paramètre. L'entier est constant car il ne doit pas être modifié. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void calculer_tps_service();
```

Cette méthode calcule le temps de service du client au moment où il rentre dans la file et l'affecte à l'attribut *tps_service*. Le calcul appliqué dépend de la loi de service de la file dans lequel le client se trouve.

```
};
```

3.3 Classe Serveur

```
#include "Client.hh"
```

Cette bibliothèque doit être incluse car un serveur contient un client.

```
#include <iostream>
```

Cette bibliothèque est nécessaire pour l'entrée et la sortie standard, utile pour afficher dans la sortie standard l'état du serveur.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

L'appel de l'environnement std permet d'alléger le code en évitant d'écrire "std::" pour les classes faisant partie de l'environnement standard.

```
class Serveur{

private :

int ID_SERVEUR;
```

Cet attribut correspond à l'identifiant du serveur. Chaque serveur a un identifiant qui lui est propre ce qui permet de les distinguer les uns des autres.

```
Client* client_actuel;
```

Cet attribut correspond au pointeur qui pointe sur le client présent actuellement dans le serveur. Un pointeur est utilisé car un client n'est pas présent dans le même serveur tout au long de la simulation, par conséquent il est nécessaire de modifier ce pointeur à chaque passage d'un nouveau client dans le serveur.

```
int tps_utilisation;
```

Cet attribut correspond au temps total d'utilisation du serveur. Il est utile pour le calcul du taux d'utilisation du serveur.

```
public :

Serveur(const int ID_SERVEUR);
```

Ce constructeur correspond au constructeur par passage de paramètres. L'attribut *ID_SERVEUR* est initialisé avec l'entier *ID_SERVEUR* passé en paramètre. L'entier est constant car il ne doit pas être modifié. L'attribut *tps_utilisation* est initialisé à 0 et le pointeur *client_actuel* à NULL.Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
 Serveur();
```

Les attributs ne sont pas alloués de manière dynamique dans le constructeur. Par conséquent, il n'y a pas de mémoire à libérer.

```
int getIDServeur();
Client& getClient();
int getTpsUtilisation();
```

Ce sont les accesseurs qui permettent d'accéder aux attributs privés de la classe.

```
friend ostream& operator <<(ostream& flux, Serveur const& s);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques du serveur. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *Serveur*, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur le serveur pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
void Entrer_Client(Client* C);
```

Cette méthode met à jour l'attribut *client_actuel* avec l'adresse d'un objet *Client* passée en paramètre et actualise l'attribut *tps_utilisation* avec le temps de service du client. L'adresse d'un objet *Client* est passé en paramètre pour avoir accès directement à l'objet.

```
Client * Sortir_client();
```

Cette méthode permet de sortir le client du serveur en mettant le pointeur *client_actuel* à NULL. Elle renvoie un pointeur sur le client que l'on vient de sortir afin de pouvoir le récupérer pour le router vers une autre file.

```
};
```

3.4 Classe File

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
#include "Client.hh"
```

La liste *liste_clients* contient des pointeurs sur des objets de la classe *Client*, cet en-tête est donc nécessaire.

```
#include "Serveur.hh"
```

Cet en-tête est nécessaire car la structure *elemServeur* contient un objet de la classe *Serveur*.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct coordonnees{
    int x
    int y
}coordonnees;
```

Cette structure représente l'emplacement de la file sur l'interface graphique. Elle contient deux entiers correspondant aux coordonnées du point central de la file.

```
struct mesure {
    int valeur
    int date
}mesure;
```

Cette structure de données est composé des attributs *valeur* qui correspond à la valeur de la mesure et *date* qui correspond à la date de la mesure. Elle permet de stocker les valeurs pour la liste *liste_nbClients* et *liste_nbClientsEntres*.

```
class File{

private :
 int ID_file;
```

Cet attribut est un entier correspondant au numéro de la file. Il permet de différencier les files les unes des autres.

```
int nb_clients;
```

Cet attribut est un entier correspondant au nombre de clients dans la file. Il est comparé avec la taille maximale de la file pour savoir si de nouveaux clients peuvent y entrer.

```
int taille_max;
```

Cet attribut est un entier correspondant à la capacité maximale de la file, important pour déterminer l'entrée d'un client dans cette file. On choisit 10 000 pour symboliser l'infini.

```
int nb_clients_servis;
```

Cet attribut est un entier correspondant au nombre de clients ayant terminé leur service dans la file. Ce nombre est nécessaire au calcul du temps d'attente moyen et du temps de service moyen.

```
int nb_clients_presentes;
```

Cet attribut est un entier correspondant au nombre de clients envoyés à la file mais pas forcément entrés dans la file. Ce nombre est utilisé pour le calcul du taux de perte.

```
int tps_service_moyen;
```

Cet attribut est un entier correspondant au temps de service moyen qui est une mesure de performances. Cette mesure de performances est mise à jour dans la file car c'est une valeur propre à chaque file.

```
int tps_attente_moyen;
```

Cet attribut est un entier correspondant au temps de attente moyen qui est une mesure de performances. Cette mesure de performances est mise à jour dans la file car c'est une valeur propre à chaque file.

```
int nb_clients_entres;
```

Cet attribut est un entier représentant le nombre de clients ayant réussi à entrer dans la file durant toute la simulation. Ce nombre sert au calcul du taux de perte.

```
int nb_clients_entres_actuel;
```

Cet attribut est un entier représentant le nombre de clients entrés à l'instant. Il permet de savoir si le nombre de clients devant entrer dans la file a été satisfait. Ce nombre est sauvegardé dans la liste de mesures du nombre de clients entrés à chaque instant *liste_nbClientsEntres*.

```
coordonnees centre;
```

Cet attribut correspond aux coordonnées du centre de l'icône représentant la file. Ces coordonnées permettent d'afficher la file sur la fenêtre graphique.

```
int ordonnancement;
```

Cet attribut est un entier représentant l'ordonnancement de la file. Le chiffre 1 représente l'ordonnancement FIFO (First In First Out), 2 représente LIFO (Last In First Out), 3 représente Priorité et 4 représente Aléatoire.

```
int loi_service;
```

Cet attribut est un entier correspondant à la loi de service de la file. Le chiffre -1 représente la loi exponentielle, -2 représente la loi uniforme et pour la loi constante, l'utilisateur choisit un nombre positif correspondant au temps de service de chaque client dans la file.

```
int distribution_arrivee;
```

Cet attribut est un entier correspondant au nombre de clients à faire rentrer dans la file. Le chiffre -1 représente la loi de Poisson et pour la loi constante, cet entier correspond au nombre de clients ainsi le nombre choisi par l'utilisateur est gardé tout en sous-entendant que la loi choisie pour la file est la loi constante.

```
list<Client *> liste_clients;
```

Cet attribut correspond à la liste des clients dans la file. Il utilise la classe *list* et contient des objets *Client ** pour ne pas copier l'objet et y avoir accès.

```
list<Serveur *>liste_serveurs;
```

Cet attribut correspond à la liste des serveurs dans la file. Il utilise la classe *list* et contient des objets *Serveur ** pour ne pas copier l'objet et y avoir accès.

```
list <mesure> liste_nbClients;
```

Cet attribut est la liste des mesures du nombre de clients en fonction du temps. Cette liste permet de calculer le nombre de clients moyen dans la file. Il utilise la classe *list* et contient des objets *mesure*.

```
list <mesure>  liste_nbClientsEntres ;
```

Cet attribut est la liste des mesures du nombre de clients entrés en fonction du temps. Cette liste permet de calculer le temps d'inter-arrivée moyen dans la file. Il utilise la classe *list* et contient des objets *mesure*.

```
public :

File(const int ID, const int abs, const int ord);
```

Ce constructeur initialise l'identifiant à *ID*, l'ordonnancement à 1 (FIFO), la loi de service à -1 (exponentielle), la distribution d'arrivée à -1 (Poisson), la taille maximale à 10000, les autres entiers à 0 et alloue les listes. Il crée un nouvel élément *coordonnees* et initialise son champ x à *abs* et son champ y à *ord*. Lorsque l'utilisateur ajoute une file sur l'interface, il ne l'a pas encore configuré. Seuls les coordonnées du centre de l'image représentant une file sont connues ce qui permet de les donner lors de la construction d'un objet File. Le numéro de la file étant connu grâce à un compteur du nombre d'objets *File* créées, il est également possible de le donner. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
File(File& f);
```

Ce constructeur initialise les attributs de l'objet avec ceux de l'objet *f*. Cela sert pour la classe *PerfsFile*. L'objet est passé en référence pour éviter une copie locale de l'objet.

~File();

Le destructeur doit désallouer la mémoire utilisée pour les listes en attribut de cette classe afin de ne pas perdre de la mémoire.

```
int getIDFile();
int getNbClients();
int getTailleMax();
int getNbClientsServis();
int getNbClientsPresentes();
int getTpsServiceMoy();
int getTpsAttenteMoy();
int getNbClientsEntres();
int getNbClientsEntresActuel();
coordonnees getcoord();
int getOrdonnancement();
int getLoi_service();
int getDistribution_arrivee();
list<Client *>& getListeClients();
list<Serveur *>& getListeServeur();
list<mesure >&  getListeNbClients();
list<mesure>& getListeNbClientsEntres();
```

Ces méthodes correspondant aux accesseurs permettant de récuperer les attributs privés de la classe afin de les exploiter.

```
void setTailleMax(const int nombre);
void setOrdonnancement(const int numero);
void setLoiService(const int numero);
void setDistribution(const int numero);
void setCoordonnees(coordonnees c);
void setNbClientsEntresActuel(const int nombre);
void setNbClientsEntresl(const int nombre);
void setNbClientsPresentes(const int nombre);
void setNbClients(const int nombre);
```

```
void setTempsAttenteMoyen(int temps);
void setTempsServiceMoyen(int temps);
```

Ces méthodes correspondant aux accesseurs qui permettent de modifier les attributs privés suivants : *taille_max*, *ordonnancement*, *loi_service*, *distribution_arrivee*, *centre*, *nb_clients_entres_actuel*, *nb_clients_entres*, *nb_clients_presentes*, *nb_clients*, *tps_attente_moyen* et *tps_service_moyen*. L'attribut *nb_clients_entres_actuel* est modifié durant la phase de routage de la simulation. Les autres attributs cités ci-dessus peuvent être modifiés lors de la création du réseau par l'utilisateur. Une exception de type *invalid_argument* est levée si un entier invalide est passé pour les méthodes nécessitant un entier en paramètre.

```
void ordonnancer_clients(Client& client);
```

Cette méthode ajoute le client *client* à la liste des clients dans la file *liste_clients* et applique la méthode correspondant à l'ordonnancement de la file parmi les quatres ordonnancements proposés sur cette liste. Cette méthode permet de garder l'ordonnancement choisi par l'utilisateur.

```
void FIFO(Client& client);
```

Cette méthode ajoute le client *client* à la fin de la liste *liste_clients*.

```
void LIFO(ClientRF& client);
```

Cette méthode ajoute le client *client* au début de la liste *liste_clients*.

```
void PRIO(ClientRF& client);
```

Cette méthode teste la priorité de *client* avec les autres clients dans la liste*liste_clients* et l'ajoute à l'endroit adéquat.

```
void RANDOM(ClientRF& client);
```

Cette méthode tire un nombre aléatoire correspondant à la position du client *client* dans la liste *liste_clients* et l'ajoute à cette position.

```
void sortir_client();
```

Cette méthode retire le client en tête de la file et met à jour le nombre de clients dans la file *nb_clients*. Elle doit exister car les clients sortent d'une file d'attente.

```
void ajouter_serveur(Serveur& s);
```

Cette méthode permet d'ajouter le serveur *s* passé en argument à la liste *liste_serveurs*. Elle est utile si l'utilisateur souhaite ajouter de nouveaux serveurs.

```
void supprimer_serveur(const int ID_serveur);
```

Cette méthode permet de supprimer le serveur ayant pour id *ID_serveur* à la liste *liste_serveurs*. Elle est utile si l'utilisateur souhaite diminuer le nombre de serveurs de la file. La suppression de tous les serveurs est impossible.Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void ajouter_client(Client& client);
```

Cette méthode ajoute l'objet *client* à la liste *liste_clients* et met à jour le nombre de clients dans la file *nb_clients*, le nombre de clients entrés dans la file durant toute la simulation *nb_clients_entres* et le nombre de clients entrés à l'instant *nb_clients_entres_actuel*.

```
void ajouter_mesure_nb_clients_entres(const int nb_clients_entres, const int tps);
```

Cette méthode ajoute un nouvel élément *mesure* à la liste *liste_NbClientsEntres* initialisé avec les valeurs passées en paramètres. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void ajouter_mesure_nb_clients (const int nb_clients, const int tps);
```

Cette méthode ajoute un nouvel élément *mesure* à la liste *liste_NbClients* initialisé avec les valeurs passées en paramètres. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void actualiser_tps_attente_clients (const int date_actuelle);
```

Cette méthode calcule le laps de temps entre l'entrée d'un client dans la file et la date actuelle de l'horloge et l'ajoute au temps d'attente de chaque client de la file. Le paramètre *date_actuelle* correspond à la date actuelle de l'horloge. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void calcul_temps_attente_moy(const int tps_attente_client_entrant_serveur);
```

Cette méthode permet de mettre à jour la valeur *tps_attente_moyen* à l'aide de la valeur *tps_attente_client_entrant_serveur* passée en paramètre. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
void calcul_temps_service_moy(const int tps_service_client_entrant_serveur);
```

Cette méthode permet de mettre à jour la valeur *tps_service_moyen* à l'aide de la valeur *tps_service_client_entrant_serveur* passée en paramètre. Une exception de type *invalid_argument* est levée si un entier invalide est passé.

```
File& operator = (File const& f);
```

Cette méthode surcharge l'opérateur = afin de copier les objets sans appel du constructeur de recopie. Il sert à l'implémentation du constructeur par recopie.

```
friend ostream& operator << (ostream& flux, File const& f);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques de la file. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de la File f, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur la file pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.5 Classe Reseau

```
#include "File.hh"
```

La classe *Reseau* possède une liste d'objets de la classe *File*, déclarée dans *File.hh*.

```
#include <iostream>
```

Cette bibliothèque gère l'entrée et la sortie standard.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
class Reseau{

private:

int type_reseau;
```

Cet attribut est un entier permettant de spécifier le réseau pour pouvoir différencier un réseau ouvert d'un réseau fermé. Le chiffre 0 représente un réseau fermé et 1, un réseau ouvert.

```
int nb_clients;
```

Cet attribut est un entier correspondant au nombre de clients dans le réseau.

```
int nb_clients_passes;
```

Cet attribut est un entier correspondant au nombre de clients qui sont passés dans le réseau depuis le début de la simulation.

```
int nb_files;
```

Cet attribut est un entier correspondant au nombre de files dans le réseau. Il permet de savoir s'il y a au moins une file afin de ne pas simuler un réseau vide.

```
int nb_classes;
```

Cet attribut est un entier correspondant au nombre de classes. Il permet de savoir s'il y a au moins une classe afin de ne pas simuler un réseau vide.

```
list<File*> liste_File;
```

Cet attribut correspond à la liste des files d'attente du réseau. Elle est utile afin d'effectuer le routage des clients vers les files. La classe *list* été choisie car l'utilisateur ajoute une par une les files sur l'interface et peut décider d'en retirer. Une structure extensible est donc recommandée. Les éléments de la liste sont des pointeurs sur les files présentes dans le réseau pour avoir accès aux modifications qui auront été faites sur les objets.

```
list <Client *> liste_ClientARouter;
```

Cet attribut correspond à la liste des clients à router dans le réseau. Elle est utilisée dans la phase de routage. Une liste a été choisie comme structure car le nombre de clients à router n'est pas connu à l'avance. Une structure extensible est donc recommandée. Les éléments de la liste sont des pointeurs sur les clients à router dans le réseau pour avoir accès aux modifications qui auront été faites sur les objets.

```
list <ClasseClient *> liste_ClasseClient;
```

Cet attribut correspond à la liste des différentes classes présentes dans le réseau. Elle permet de savoir quelles classes peut-on donner aux clients. Cet objet de la classe *list* a été choisie car l'utilisateur peut ajouter et retirer des classes lors de la création du réseau. Une structure extensible est donc utile. Les éléments de la liste sont des pointeurs sur les classes dans le réseau pour avoir accès aux modifications qui auront été faites sur les objets.

```
public :

 Reseau();
```

Ce constructeur initialise tous les attributs de type entier à 0 et tous les objets *list* avec le constructeur par défaut de leur classe.

```
 ~Reseau();
```

Le destructeur désalloue la mémoire allouée pour les listes en attribut.

```
int getTypeReseau();
int getNbFiles();
int getNbClasses();
int getNbClients();
int getNbClientsPasses()
list<File*>& getListeFile();
list <Client *>&  getListeClientARouter();
list <ClasseClient *>& getListeClasseClient();
```

Ces méthodes permettent d'accéder aux attributs privés de la classe *Reseau*.

```
void setNbClients(const int nombre);
```

Cet accesseur met à jour l'attribut *nb_clients* avec la valeur de l'entier passée en paramètre. Cet entier est constant pour éviter sa modification. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void setNbClientsPasses(const int nombre);
```

Cet accesseur met à jour l'attribut *nb_clients_passes* avec la valeur de l'entier passée en paramètre. Cet entier est constant pour éviter sa modification. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void setTypeReseau(const int type);
```

Cet accesseur met à jour l'attribut *type_reseau* avec la valeur de l'entier passée en paramètre. Cet entier est constant pour éviter sa modification. Cet accesseur est nécessaire car l'utilisateur peut modifier le type du réseau autant de fois qu'il le souhaite. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void ajouter_file(File* f);
```

Cette méthode crée un élément dans la liste et ajoute la file passée en paramètre à la fin de la liste *liste_File*. L'adresse de la file est passée en paramètre car la liste attend un pointeur sur un objet *File*.

```
void ajouter_client_routage(Client * client);
```

Cette méthode crée un élément dans la liste *liste_ClientARouter* et ajoute le client passé en paramètre à la fin de la liste . L'adresse du client est passée en paramètre car la liste attend un pointeur sur un objet *Client*.

```
void supprimer_file(const int ID_file);
```

Cette méthode supprime la file avec pour identifiant *ID_file* de la liste *liste_File*. Lors de la conception du réseau, l'utilisateur peut supprimer une file présente dans le réseau. L'entier passé en paramètre est constant pour qu'il ne soit pas modifié. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void vider_liste_routage();
```

Cette méthode désalloue la mémoire de tous les éléments de la liste *liste_ClientARouter*. Cette liste doit être vidée à chaque fin de phase de routage.

```
void ajouter_classe(ClasseClient * classe);
```

Cette méthode crée un élément dans la liste et ajoute la classe passée en paramètre à la fin de la liste *liste_ClasseClient*. L'adresse de la classe est passée en paramètre car la liste attend un pointeur sur un objet *ClasseClient*.

```
void supprimer_classe(const int ID_classe);
```

Cette méthode supprime la classe avec pour identifiant *ID_classe* de la liste *liste_ClasseClient*. Lors de la conception du réseau, l'utilisateur peut supprimer une classe présente dans le réseau. L'entier passé en paramètre est constant pour qu'il ne soit pas modifié. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
friend ostream& operator << (ostream& flux, Reseau const& r);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques du réseau. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs du réseau *r*, cette méthode doit être "amie" de la classe, spécifié avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur le réseau pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.6 Classe Echeancier

La classe *Echeancier* correspond à la fonctionnalité *Gérer l'échéancier*.

```
#include <iostream>
```

Cette bibliothèque gère l'entrée et la sortie standard.

```
#include "File.hh"
```

Une structure *Evenement* contient un objet de la classe *File* déclaré dans *File.hh*.

```
#include <list> [6][7]
```

Cette bibliothèque est nécessaire car la sauvegarde des événements se fera sous forme d'une liste.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct evenement{
        int type_evenement;
        int date_traitement;
        File * file_destinee;
        Serveur * serveur_destine;
    Client * client;
}Evenement;
```

Cette structure permet de représenter un événement qui servira pendant le déroulement de la simulation. On utilise une liste d'événements qui chacun leur tour s'exécutent afin d'avancer la simulation. Un événement est représenté par sa date de traitement, le type d'événement, la file, le serveur et le client concerné. Les objets sont récupérés en pointeur car ils seront modifiés lors de l'exécution de l'événement.

```
class Echeancier{

private :

list <Evenement> liste_evenement;
```

Cet attribut correspond à un objet de type *list* contenant des éléments de type *Evenement*. Il sauvegarde les événements devant se produire dans le réseau.

```
int date_actuelle;
```

Cet attribut correspond à la date à laquelle le réseau se trouve. Il est mis à jour à chaque évènement sorti de l'échéancier.

```
public :

Echeancier();
```

Ce constructeur initialise l'attribut *liste_evenements* en appelant le constructeur par défaut de la classe *list* et met à 0 la date actuelle.

~Echeancier();

Le destructeur va désallouer la liste des événements.

```
list <Evenement>& getListeEvenement();
```

Cet accesseur permet d'accéder à l'attribut privé *liste_evenements* de la classe.

```
int getDateActuelle();
```

Cet accesseur permet d'accéder à l'attribut privé *date_actuelle* de la classe.

```
void ajouter_evenement(Evenement& event);
```

Cette méthode permet d'ajouter un événement *event* passé en paramètre à la liste. La liste est modifiée lors de changements dans le réseau. Cette méthode est donc nécessaire pour la modifier.

```
Evenement sortir_prochain_evenement();
```

Cette méthode va sortir le prochain événement se produisant. Le premier évènement doit être exécuté pour faire avancer la simulation. Cette méthode doit donc exister pour pouvoir récupérer l'événement à exécuter.

```
void supprimer_evenement_file(const int num_file);
```

Cette méthode supprime les événements correspondant au numéro de la file passé en paramètre. Lors de la simulation, l'utilisateur peut faire pause et modifier le réseau. Il peut notamment supprimer une file. Il faut donc supprimer les événements correspondants à la file. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void supprimer_evenement_serveur(const int num_serveur);
```

Cette méthode supprime les événements correspondant au numéro du serveur passé en paramètre. Lors de la simulation, l'utilisateur peut faire pause et modifier le réseau. Il peut notamment supprimer un serveur. Il faut donc supprimer les événements correspondants au serveur. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void trier_croissant();
```

Cette méthode permet de trier les événements par ordre croissant. Les événements possèdent une date de traitement et doivent donc être exécutés dans le bonne ordre. Il est donc nécessaire de trier la liste.

```
friend ostream& operator << (ostream& flux, Echeancier const& e);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques de l'échéancier. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *Echeancier*, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur l'échéancier pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.7 Classe DonneesCalculsRF

Cette classe nous permet de récupérer les données nécessaires pour les calculs des performances d'un réseau fermé.

```
#include "File.hh"
```

L'attribut *PremierNbCLients* est un pointeur sur la structure *mesure* déclarée dans *File.hh*.

```
#include <ostream>
```

Cette bibliothèque gère la sortie standard.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
class DonneesCalculsRF{

protected:
```

La classe *DonneesCalculsRF* est la classe mère de la classe *DonneesCalculsRO*. Afin que les attributs de cette classe soient uniquement accessibles par la classe fille, ils doivent être protégés.

```
list<mesure> listeNbClients;
```

Cet attribut est la liste des mesures du nombre de clients dans le réseau en fonction du temps. Cette liste est utilisée pour le calcul du nombre moyen de clients dans le réseau. Il utilise la classe *list* et contient des objets *mesure*.

```
public :

DonneesCalculsRF() ;
```

Ce constructeur de la classe va allouer la liste *listeNbCLients*.

~DonneesCalculsRF() ;

Le destructeur va désallouer la liste *listeNbCLients*.

```
list<mesure> getListeNbClients();
```

Cette méthode est un accesseur retournant la liste *listeNbClients*.

```
void ajouterMesuresNbclients(const int nb_clients, const int temps) ;
```

Cette méthode va créer un nouvel élément *mesure* ,initialisé aux valeurs *nb_clients* et à la date *temps*, et va l'ajouter à la liste du nombre de clients *listeNbClients*. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
friend ostream& operator << (ostream& flux, DonneesCalculsRF const& d);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques de la classe. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *DonneesCalculsRF*, cette méthode doit être "amie" de la classe, spécifiée avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur les données pour les calculs de performances du réseau pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.8 Classe DonneesCalculsRO

Cette classe hérite de la classe DonneesCalculsRF. Elle nous permet d'avoir les données nécessaires pour le calcul des performances d'un réseau ouvert.

```
#include DonneesCalculsRF.hh
```

Cette bibliothèque doit être incluse car cette classe hérite de la classe *DonneesCalculsRf.hh*.

```
#include <iostream>
```

Cette bibliothèque doit être incluse pour la surcharge de l'opérateur <<.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct TSClient{
        int numero_client;
        int date_sortie;
        int date_entree;
        int tps_sejour;
    struct TSClient * suivant;
}TSClient;
```

Cette structure permet de garder les données nécessaires pour le calcul du temps de séjour d'un client. Elle prend le numéro du client, sa date d'entrée, sa date de sortie et son temps de séjour sous forme d'entier. Elle est utilisée pour former une liste qui contiendra les données pour le calcul du temps de séjour de tous les clients passés dans le réseau.

```
class DonneesCalculsRO : public DonneesCalculsRF {
```

L'héritage doit être public pour que l'encapsulation des données de la classe *DonneesCalculsRF* reste le même.

```
private :

list<mesure> listeNbClientsEntrant ;
```

Cet attribut est la liste des mesures du nombre de clients entrant dans le réseau en fonction du temps. Cette liste est utilisée pour le calcul du débit d'entrée moyen. Il utilise la classe *list* et contient des objets de la classe *mesure*.

```
list <mesure> listeNbClientSortant ;
```

Cet attribut est la liste des mesures du nombre de clients entrant dans le réseau en fonction du temps. Cette liste est utilisée pour le calcul du débit de sortie moyen. Il utilise la classe *list* et contient des objets de la classe *mesure*.

```
list <TSClient> listeTSClient;
```

Cet attribut est la liste qui va contenir des éléments stockant des informations sur les dates d'arrivée, de sortie et le temps de séjour d'un client. Un élément correspond aux données d'un client. Cette liste est utilisée pour le calcul du débit de sortie moyen. Il utilise la classe *list* et contient des objets de la classe *TSClient*.

```
DonneesCalculsRO() ;
```

Le constructeur va allouer les listes.

~DonneesCalculsRO();

Le destructeur va désallouer les listes en attribut.

```
list <mesure>& getListeNbClientEntrant();
list <mesure>& getListeNbClientSortant();
list <TSClient>& getListeClientEntres();
```

Ces méthodes permettent d'accéder aux attribut privés de la classe.

```
void ajouterMesuresNbclientsEntrants (const int nb_clients_entres, const int temps) ;
```

Cette méthode va créer un nouvel élément *mesure* ,initialisé aux valeurs *nb_clients_entres* et à la date *temps*, et va l'ajouter à la liste du nombre de clients entrants *listeNbClientsEntrant*. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void ajouterMesureNbClientsSortants (const int nb_clients_sortis, const int temps);
```

.

Cette méthode va créer un nouvel élément *mesure* ,initialisé aux valeurs *nb_clients_sortis* et à la date *temps*,et va l'ajouter à la liste du nombre de clients sortants *listeNbClientsSortant*. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void ajouter_element_TSclient(const int IDClient, const int date_entree);
```

Cette méthode ajoute un nouvel élément *TSclient* initialisé le numéro du client au nombre *IDClient* et la date d'entrée à la valeur *date_entree*. Elle initialise les autres champs entiers à 0. Elle est appelée lorsqu'un nouveau client est entré dans le réseau. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
 void calcul_tps_sejour_client(const int IDClient, const date_sortie);
```

Cette méthode va mettre à jour un élément *TSClient* ayant pour *numero_client* la valeur *IDClient* en paramètres en lui affectant la valeur *date_sortie* passée en paramètres à l'attribut *date_sortie* du *TSClient*. Ensuite la méthode va calculer le temps de séjour en fonction de la date d'arrivée et la date de sortie. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
friend ostream& operator << (ostream& flux, DonneesCalculsRO const& d);
```

La surcharge de l'opérateur << permet d'afficher les caractéristiques de la classe. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *DonneesCalculsRO*, cette méthode doit être "amie" de la classe, spécifié avec le mot-clé *friend*. Elle prend en paramètre un objet *ostream* pour gérer la sortie standard, et une référence constante sur les données pour les calculs de performances pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante donne l'accès à l'objet en lecture uniquement.

```
};
```

3.9 Classe Simulation

Cette classe est mère des deux classes *simulationRO* et *simulationRF*. Ces deux classes partagent certains attributs et certaines méthodes. C'est une classe abstraite car il existe une méthode virtuelle pure qui sera implémentée dans les classes dérivées. Aucune instance de cette classe ne pourra être créer.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
#include "Echeancier.hh"
```

L'échéancier est un outil indispensable de la simulation. Cet outil est représenté par un objet *Echeancier* dont la classe se trouve dans le header *Echeancier.hh*.

```
#include "Reseau.hh"
```

Le réseau est indispensable à la simulation. Il est représenté par un objet *Reseau* dont la classe se trouve dans le header *Reseau.hh*.

```
protected :
```

Les attributs sont protégés car cette classe est utilisée dans un héritage et donc l'accès des attributs de la classe mère à partir des classes dérivées peut se faire facilement (sans les appels d'accesseurs).

```
Reseau * reseau
```

Cet attribut est un pointeur sur un objet de la classe *Reseau*. La simulation se fait sur un réseau. Il est donc nécessaire d'avoir accès à ce réseau. Le réseau est passé en pointeur pour accéder aux modifications qui seront faites sur le réseau par l'utilisateur pendant la simulation.

```
Echeancier echeancier
```

Cet attribut est un objet de type *Echeancier*. Il correspond à l'échéancier, outil orchestrant la simulation à événements discrets du réseau.

```
int duree_simulation
```

Cet attribut est un entier correspondant à la durée de simulation du réseau donné par l'utilisateur. Il est nécessaire de la connaître pour arrêter la simulation quand l'utilisateur le souhaite.

```
int nbClientsTot;
```

Cet attribut correspond au nombre de clients créés lors de la simulation. Il permet de les compter et de leur donner un identifiant.

```
int nbFilesTot;
```

Cet attribut correspond au nombre de files créées lors de la simulation. Il permet de donner des identifiants aux files.

```
int nbClassesTot;
```

Cet attribut correspond au nombre de classes créées lors de la simulation. Il permet de donner un identifiant à chaque classe.

```
int nbServeursTot;
```

Cet attribut correspond au nombre de serveurs créés lors de la simulation. Il permet de donner un identifiant à chaque serveur.

```
public :

Reseau * getReseau();
Echeancier& getEcheancier();
int getDureeSimulation();
int getNbClientsTot();
int getNbFileTot();
int getNbClassesTot();
int getNbServeursTot();
```

Ces accesseurs permettent de récupérer les attributs privés.

```
void executer_evenement();
```

Cette méthode permet d'exécuter le premier événement de l'échéancier en appelant la méthode correspondante à l'événement en cours. Cette fonction va modifier le réseau *reseau* et l'échéancier *echeancier*.

```
void EntreeFile(evenement evenement_sorti);
```

Cette méthode va exécuter un événement *Entrée dans une file* correspondant à la nature de *evenement_sorti*, sorti fraîchement de l'échéancier. Elle va également modifier l'échéancier en y ajoutant des événements. La structure *evenement* est directement passé en paramètre pour pouvoir récupérer les informations qu'elle contient.

```
void EntreeServeur(evenement evenement_sorti);
```

Cette méthode va exécuter un événement *Entrée dans un serveur* correspondant à la nature de l'événement *evenement_sorti*, sorti fraîchement de l'échéancier. Elle va également modifier l'échéancier en y ajoutant des événements. La structure *evenement* est directement passé en paramètre pour pouvoir récupérer les informations qu'elle contient.

```
void SortieServeur(evenement evenement_sorti);
```

Cette méthode va exécuter un événement *Sortie d'un serveur* correspondant à la nature de *evenement_sorti*, sorti fraîchement de l'échéancier. Elle va également modifier l'échéancier en y ajoutant des événements. La structure *evenement* est directement passée en paramètre pour pouvoir récupérer les informations qu'elle contient.

```
virtual void routage_clients() = 0;
```

Cette méthode correspond au routage des clients dans le réseau. Selon le type du réseau, le routage diffère car dans un réseau ouvert, un client peut router vers la sortie du réseau. Ainsi dans une simulation sur un réseau sans type prédéfini, router un client ne peut être défini. La méthode est donc passée en virtuelle pure pour qu'elle ne soit pas implémentée dans cette classe. Elle devra être obligatoirement implémentée dans les classes dérivant de cette classe.

```
void distribution_initiale(const int nb_clients_initial);
```

Cette méthode s'occupe de la création des clients au début de la simulation en fonction du nombre *nb_clients_initial* et de leur routage dans le reseau *reseau* grâce à la fonction *routage_clients*. La valeur passée en paramètre est constante pour que cette méthode ne la modifie pas. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
int calculer_taux_arrivee_poisson(const int lambda);
```

Cette fonction calcule le nombre de clients devant entrer dans une file en fonction d'un lambda calculé lors du routage des clients. Elle renvoie le nombre de clients devant entrer afin de router le bon nombre de clients vers la file en question. La valeur passée en paramètre est constante pour que cette méthode ne la modifie pas. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

3.10 Classe SimulationRF

```
#include "simulation.hh"
```

La classe *simulationRF* hérite de *simulation*. La bibliothèque contenant la classe *simulation* doit donc être incluse.

```
class SimulationRF : public Simulation{
```

L'héritage est public pour garder l'encapsulation des membres de la classe mère.

```
private:
DonneesCalculsRF donnees_calculs
```

Cet attribut est un objet de la classe *DonneesCalculsRF*. Elle contient des listes stockant des données du réseau nécessaire aux calculs des performances pour un réseau fermé.

```
public :
simulationRF(Reseau *reseau, const int duree);
```

Le constructeur va appeler le constructeur de sa superclasse *Simulation* en initialisant son attribut *reseau* avec l'objet *reseau* en paramètre et son attribut *duree_simulation* avec la valeur *duree* en paramètre. Il va aussi créer un objet *Echeancier* et *DonneesCalculsRF* en appelant leur constructeur par défaut. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

~ simulationRF();

Le destructeur va désallouer l'attribut *reseau*.

```
DonneesCalculsRF& getDonnees_calculs();
```

Cet accesseur renvoie l'attribut *donnees_calculs*.

```
void routage_clients();
```

Cette méthode va créer des événements "entrée d'un client dans le réseau" pour les clients dans la liste de routage du réseau. Elle est redéfinie dans cette classe car le routage se fait différemment dans un réseau fermé.

```
};
```

3.11 Classe SimulationRO

```
#include simulation.hh
```

La classe *simulationRO* hérite de *simulation*. La bibliothèque contenant la classe *simulation* doit donc être incluse.

```
#include Reseau.hh
```

La bibliothèque de la classe *Réseau* doit être incluse car les fonctions ci-dessous prennent en paramètre un objet de cette classe.

```
class SimulationRO : public Simulation{
```

L'héritage est public pour garder l'encapsulation des membres de la classe mère.

```
private:
DonneesCalculsRO donnees_calculs;
```

Cette attribut est un objet de la classe *DonneesCalculsRO*. Elle contient des listes stockant des données du réseau nécessaire aux performances.

```
public :
simulationRO(Reseau *reseau, const int duree);
```

Le constructeur va appeler le constructeur de sa superclasse *Simulation* en initialisant son attribut *reseau* avec l'objet *reseau* en paramètre et son attribut *duree_simulation* avec la valeur *duree* en paramètre. Il va aussi créer un objet *Echeancier* et *DonneesCalculsRO* en appelant leur constructeurs par défaut. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

~simulationRO();

Le déstructeur va désallouer son attribut *Reseau* alloué dynamiquement.

```
DonneesCalculsRO& getDonnees_calculs();
```

Cet accesseur renvoie l'attribut *donnees_calculs*.

```
void routage_clients();
```

Cette méthode va créer des événements "entrée d'un client dans le réseau" pour les clients dans la liste de routage du réseau. Elle est redéfinie dans cette classe car le routage se fait différemment dans un réseau ouvert.

```
void entrer_clients_reseau(Reseau * reseau);
```

Cette méthode s'occupe de l'entrée des clients dans le réseau *reseau*. Un nombre aléatoire est tiré pour connaître le nombre de clients à créer qui sont ajoutés à la liste de routage contenue dans *reseau*. L'objet est passé en pointeur car il est modifié et on veut accéder à ces modifications.

```
void sortir_clients_reseau(Reseau * reseau);
```

Cette méthode s'occupe de la sortie des clients du réseau *reseau*. Un nombre aléatoire est tiré pour connaître le nombre de clients à sortir. Leur temps de séjour est mis à jour puis ils sont supprimés de la liste de routage de *réseau*. L'objet est passé en pointeur car il est modifié et on veut accéder à ces modifications.

```
};
```

4 Module Gestion de fichiers

lecture_ecriture.hh

```
#include "Reseau.hh"
```

Cette bibliothèque est nécessaire car certaines fonctions permettent de charger un réseau.

```
#include <fstream>  [8]
```

Cette bibliothèque nous permet d'ouvrir un fichier en lecture ou en écriture.

```
#include "HistoriqueRO.hh"
```

Cette bibliothèque est utile car deux fonctions de ce .hh prennent un objet *HistoriqueRO* et un objet *HistoriqueRH*.

```
#include <string>
```

Cette bibliothèque nous permet de récupérer le nom du fichier sous forme d'un string.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
void lecture_fichier_reseau(Reseau * reseau, const string nomfic);
```

Cette fonction prend en paramètre le nom du fichier à ouvrir sous forme d'un string. Le paramètre de type string est constant pour éviter sa modification. Une exception de type *invalid_argument* est levée si le nom du fichier correspond à un fichier qui ne contient pas un réseau. La fonction prend également un pointeur sur un objet de type *Reseau*. Ce pointeur permet d'accéder à l'objet *Reseau* créé en dehors de la fonction et de le modifier avec les données récupérées dans le fichier. Elle utilise les deux sous-fonctions *lecture_file* et *lecture_type*.

```
void lecture_file(File * file, const string ligne);
```

Cette fonction nous permet de lire les données nécessaires pour la construction d'une file, donc ses caractéristiques. Elle prend en paramètre une ligne du fichier récupérée sous forme de chaîne de caractères et renvoie la file correspondante. Le paramètre de type string est constant pour éviter sa modification. Une exception de type *invalid_argument* est levée si la chaîne de caractère ne correspond pas aux données d'une file. La fonction prend également un pointeur sur un objet de type *File*. Ce pointeur permet d'accéder à l'objet *File* créé en dehors de la fonction et de le modifier avec les données récupérées dans le fichier.

```
void lecture_type(ClasseClient * cc, const string ligne);
```

Cette fonction nous permet de lire les caractéristiques d'une classe de clients. Elle prend en paramètre une ligne du fichier récupérée sous forme de chaîne de caractères et renvoie l'objet *ClasseClient* correspondant à la classe. Le paramètre de type string est constant pour éviter sa modification. Une exception de type *invalid_argument* est levée si la chaîne de caractère ne correspond pas aux données d'une classe. La fonction prend également un pointeur sur un objet de type *ClasseClient*. Ce pointeur permet d'accéder à l'objet *ClasseClient* créé en dehors de la fonction et de le modifier avec les données récupérées dans le fichier.

```
void ecriture_fichier_reseau(const string nomfic, Reseau& reseau);
```

Cette fonction prend en paramètre, le nom du fichier à ouvrir sous forme de string et une référence sur l'objet *Reseau* pour éviter une copie inutile de l'objet. Le paramètre de type string est constant pour éviter sa modification. Si le fichier n'existe pas, le réseau est sauvegardé dans un nouveau fichier ayant le nom donné en paramètre, sinon le contenu du fichier est écrasé. Elle utilise les méthodes *ecriture_fichier_file* et *ecriture_fichier_classe* ci-dessous.

```
void ecriture_fichier_file(ofstream* fic, Reseau& reseau);
```

Cette fonction permet d'écrire dans le fichier récupéré en paramètre grâce au pointeur *fic*, les caractéristiques des files du réseau *reseau* . Les caractéristiques d'une file à sauvegarder sont les attributs suivants :

- l'identifiant *ID_FILE*
- la taille de la file *taille_max*
- les coordonnées *coord* de la file sur l'interface
- l'ordonnancement *ordonnancement*
- la loi de service *loi_service*
- la distribution d'arrivée *distribution_arrivee*

ainsi que le nombre de serveurs liés à la file.

Le fichier est récupéré sous forme de pointeur car il va être modifié par la fonction et le réseau sous forme de référence pour éviter une copie inutile de l'objet.

```
void ecriture_fichier_classe(ofstream* fic, Reseau& reseau);
```

Cette fonction permet d'écrire dans le fichier *fic*, les différentes classes composant le réseau. Le fichier est récupéré sous forme de pointeur car il va être modifié par la fonction et le réseau sous forme de référence pour éviter une copie inutile de l'objet. Les caractéristiques d'une classe à sauvegarder sont les attributs suivants :

- l'identifiant ID_classe
- le routage *routage*

- la priorité de la classe *priorite*

```
void ecriture_fichier_performancesRF(const string nomfic,HistoriqueRF& h);
```

Cette fonction prend en paramètre le nom du fichier à ouvrir sous forme d'un string constant pour éviter sa modification et une référence sur un objet *HistoriqueRF* qui contient l'historique des performances d'un réseau fermé. L'objet est passé par référence pour éviter une copie d'objet inutile. La fonction va ouvrir le fichier correspondant au nom donné par l'utilisateur si il existe, sinon le fichier est créé avec le nom donné. L'historique des performances du réseau et de ses files est écrit dans le fichier ouvert.

```
void ecriture_fichier_performancesRO(const string nomfic,HistoriqueRO& h);
```

Cette fonction prend en paramètre le nom du fichier à ouvrir sous forme d'un string constant pour éviter sa modification et une référence sur un objet *HistoriqueRO* qui contient l'historique des performances d'un réseau ouvert. L'objet est passé par référence pour éviter une copie d'objet inutile. La fonction va ouvrir le fichier correspondant au nom donné par l'utilisateur si il existe, sinon le fichier est créé avec le nom donné. L'historique des performances du réseau et de ses files est écrit dans le fichier ouvert.

5 Module Calculs de Performances

Le module Calculs de Performances est chargé de calculer les performances du réseau après avoir reçu les données nécessaires du module Gestion Réseau. Il se compose de 5 classes organisées de la manière suivante :

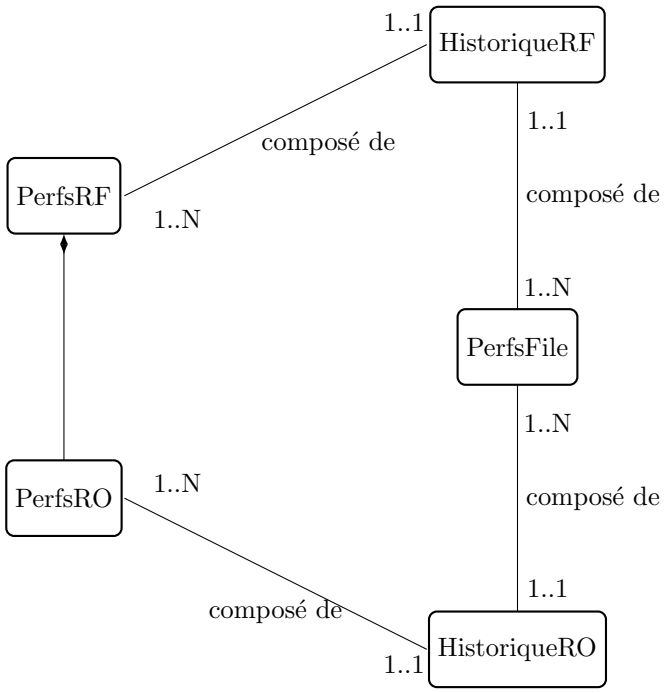


Diagramme de classes

Les performances du réseau sont calculées dans *PerfsRF* pour un réseau fermé ou dans *PerfsRO* pour un réseau ouvert. La classe *PerfsRO* hérite de *PerfsRF* car les performances d'un réseau ouvert sont les mêmes qu'un réseau fermé auxquelles nous ajoutons le débit d'entrée et de sortie du réseau avec le temps de séjour moyen d'un client. Les performances d'une file sont calculées dans *PerfsFile*. Pour sauvegarder l'historique de toutes les performances (c'est-à-dire les performances du réseau et de ses files), nous avons créé deux classes distinctes : *HistoriqueRO* pour l'historique d'un réseau ouvert et *HistoriqueRF* pour l'historique d'un réseau fermé. Ces deux classes sont composées de la classe *PerfsFile* pour enregistrer les performances des files du réseau et d'un objet de type *PerfsRO* ou *PerfsRF* selon si nous sommes dans la classe *HistoriqueRO* ou *HistoriqueRF*.

5.1 Classe PerfsRF

Cette classe nous permet d'effectuer le calcul des performances d'un réseau fermé.

```
#include "DonneesRF.hh"
```

Cet en-tête est nécessaire pour la liste des mesures effectuées sur le réseau fermé.

```
#include <ostream>
```

Cette bibliothèque doit être incluse pour la surcharge de l'opérateur <<.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
class PerfsRF{
```

C'est dans cette classe que les performances d'un réseau fermé seront calculées. Cette classe permet de récupérer le nombre moyen de clients dans le réseau et la date à laquelle le calcul a été effectué. Elle possède une méthode qui nous permet de calculer le nombre moyen de clients dans le réseau.

```
protected :
```

La classe *PerfsRF* est la classe mère de la classe *PerfsRO*. Afin que les attributs de cette classe soient uniquement accessibles par la classe fille, ils doivent être protégés.

```
int nb_moyen_clients;
```

Cet attribut correspond à la performance "nombre moyen de clients dans le réseau" du cahier des charges.

```
int date;
```

Cet attribut correspond à la date à laquelle les performances ont été calculées.

```
public :
```

```
PerfsRF(const int date);
```

Le constructeur prend un entier *date* en paramètre pour initialiser l'attribut *date* et initialise l'attribut *nb_moyen_clients* à 0. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
PerfsRF(PerfsRF& P);
```

Le constructeur de recopie va initialiser ses attributs avec ceux de l'objet P en paramètre.

~PerfsRF();

Le destructeur ne va pas désallouer d'attributs alloués dynamiquement car il n'y en a aucun.

```
int getNbMoyenClients();
int getDate();
```

Ces méthodes correspondent aux accesseurs pour accéder aux attributs privés de la classe.

```
void calcul_nb_moyen_clients(list<mesure>& listNbClients)
```

Cette méthode calcule le nombre moyen de clients dans le réseau à partir d'une liste de mesures données en paramètre où le nombre de clients dans le réseau a été mesuré à certaines dates. Cette méthode ne renvoie rien. Elle s'occupe de mettre à jour l'attribut *nb_moyen_clients*.

```
friend ostream& operator <<(ostream& flux, PerfsRF const& perfs);
```

La surcharge de l'opérateur << permet d'afficher les performances d'un réseau fermé ainsi que la date à laquelle le calcul a été effectué. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet *PerfsRF*, cette méthode doit être "amie" de la classe, spécifiée par le mot-clé friend. Elle prend en paramètres un objet de type *ostream* pour gérer la sortie standard et une référence constante sur les performances d'un réseau fermé pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante permet de donner l'accès à l'objet en lecture uniquement.

```
PerfsRF& operator = (PerfsRF const& p);
```

La surcharge de l'opérateur = permet de recopier les attributs de l'objet p. L'objet est passé en référence constante pour éviter une copie locale et sa modification.

```
};
```

5.2 Classe PerfsRO

Cette classe hérite de la classe PerfsRF, elle nous permet de calculer les performances d'un réseau ouvert.

```
#include "PerfsRF.hh"
```

Etant donné que *PerfsRO* hérite de *PerfsRF*, il est obligatoire d'inclure cet en-tête.

```
#include "DonneesRO.hh"
```

Cet en-tête est nécessaire pour la liste des mesures effectuées sur le réseau ouvert.

```
#include <ostream>
```

Cette bibliothèque doit être incluse pour la surcharge de l'opérateur <<.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
class PerfsRO : public PerfsRF{
```

C'est dans cette classe que les performances d'un réseau ouvert seront calculées. Cette classe permet de récupérer le temps de séjour moyen, le débit d'entrée, le débit de sortie du réseau. Elle possède une méthode qui nous permet de calculer le temps de séjour moyen du réseau. Une autre méthode qui calcule le débit d'entrée dans le réseau en prenant la liste des clients entrés dans le réseau . La troisième méthode calcule le débit de sortie dans le réseau en prenant la liste des clients sortis du réseau .

```
private :

int debit_entree;
```

Cet attribut correspond à la performance "débit d'entrée dans le réseau".

```
int debit_sortie;
```

Cet attribut correspond à la performance "débit de sortie dans le réseau ".

```
float tps_sejour_moyen;
```

Cet attribut correspond à la performance "temps de séjour moyen dans le réseau". C'est un *float* car n résultat plus précis est souhaitée.

```
public :

PerfsRO(const int date) ;
```

Le constructeur initialise l'attribut *date* avec la valeur *date* en paramètres, les entiers à 0 et les flottants à 0.0. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
PerfsRO(PerfsRO const& P);
```

Le constructeur de recopie initialise tous ses attributs avec les attributs de P en paramètres.

~PerfsRO();

Le destructeur ne va pas désallouer de la mémoire allouée dynamiquement car les attributs sont tous statiques.

```
int getDebitEntree();
int getDebitSortie();
float getTpsSejourMoyen();
```

Ces méthodes correspondent aux accesseurs pour accéder aux attributs privés de la classe.

```
void calcul_tps_sejour_moyen(list<TSClient>& ListeTSClient);
```

Cette méthode calcule le temps de séjour moyen dans le réseau. Elle prend en paramètre la liste des clients déjà passés dans le réseau avec leur date d'entrée, de sortie ainsi que leur temps de séjour. Elle s'occupe de mettre à jour l'attribut *tps_sejour_moyen*.

```
void calcul_debit_entree(list<mesure>& ListeNbClientEntrant) ;
```

Cette méthode calcule le débit d'entrée dans le réseau à partir du paramètre qui est la liste de mesures effectuées à certaines dates sur le nombre de clients entrant dans le réseau. Elle s'occupe de mettre à jour l'attribut *debit_entree*.

```
void calcul_debit_sortie(list<mesure>& ListeNbClientSortant) ;
```

Cette méthode calcule le débit de sortie dans le réseau à partir du paramètre qui est la liste de mesures effectuées à certaines dates sur le nombre de clients sortant du réseau. Elle s'occupe de mettre à jour l'attribut *debit_sortie*.

```
friend ostream& operator << (ostream& flux, PerfsRO const& perfs);
```

La surcharge de l'opérateur << permet d'afficher les performances d'un réseau ouvert ainsi que la date à laquelle le calcul a été effectué. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet PerfsRO, cette méthode doit être "amie" de la classe, spécifiée par le mot-clé friend. Elle prend en paramètre un objet de type ostream pour gérer la sortie standard et une référence constante sur les performances d'un réseau ouvert pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante permet de donner l'accès à l'objet en lecture uniquement.

```
PerfRO& operator = (PerfsRO const& p);
```

La surcharge de l'opérateur = permet de recopier les attributs de l'objet *p*. L'objet est passé en référence constante pour éviter une copie locale et sa modification.

```
};
```

5.3 Classe PerfsFile

La classe *PerfsFile* gère les calculs des performances des files.

```
#include "File.hh"
```

Cet en-tête est nécessaire car un objet de la classe *PerfsFile* contient un attribut de type *File*.

```
#include <ostream>
```

Cette bibliothèque doit être incluse pour la surcharge de l'opérateur << .

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct PerfServeur{
        int numero_serveur;
        float taux_utilisation;
}PerfServeur;
```

Cette structure permet de garder le taux d'utilisation d'un serveur. Elle est utilisée pour former une liste qui contiendra le taux d'utilisation de chaque serveur relié à la file. Le taux d'utilisation est gardé sous forme de float pour avoir une meilleure précision.

```
class PerfsFile {

private :

File file;
```

Cet attribut de type *File* permet de récupérer les données nécessaires d'une file pour en calculer ses performances. Il est nécessaire d'avoir la file en attribut pour garder l'intégrité des données traitées.

```
int date;
```

Cet attribut correspond à la date à laquelle les performances de la file ont été calculées.

```
int nb_moyen_clients_file;
```

Cet attribut correspond au résultat du nombre moyen de clients dans la file. Il s'agit d'un entier car des clients sont comptés.

```
float tps_attente_moyen;
```

Cet attribut correspond au résultat du temps d'attente moyen dans la file. Il s'agit d'un float car la précision est nécessaire.

```
float tps_service_moyen;
```

Cet attribut correspond au résultat du temps de service moyen dans la file. Il s'agit d'un float car la précision est nécessaire.

```
float tps_reponse_moyen;
```

Cet attribut correspond au résultat du temps de réponse moyen dans la file. Il s'agit d'un float car la précision est nécessaire.

```
list <PerfServeur> ListeResPerfServeur;
```

Cet attribut correspond à la liste des serveurs reliés à la file avec leur taux d'utilisation. Une liste a été choisie car le nombre de serveurs change d'une file à une autre. Une structure extensible est donc nécessaire.

```
float tps_interarrivee_moyen;
```

Cet attribut correspond au résultat du temps d'inter-arrivée moyen dans la file. Il s'agit d'un float car la précision est nécessaire.

```
int taux_perte;
```

Cet attribut correspond au résultat du taux de perte d'une file. Il s'agit d'un entier car des clients sont comptés.

```
public :
```

```
PerfsFile(File& f, const int date);
```

Ce constructeur prend un entier constant *date* et une réference sur un objet *File* en paramètre. L'entier est constant pour éviter sa modification et l'objet de type *File* est passé en référence pour éviter une copie locale inutile de l'objet. Le constructeur initialise l'attribut *date* et *file* avec les champs donnés en paramètre. Il initialise les autres attributs entiers à 0 et les attributs flottants à 0.0. Une exception de type *invalid_argument* si l'entier passé en paramètre n'est pas valide.

```
PerfsFile(PerfsFile const& perfs);
```

Ce constructeur correspond au constructeur de recopie qui initialise tous les attributs avec les champs de l'objet de type *PerfsFile* donné en paramètre. L'objet est passé par référence constante pour éviter une copie locale de l'objet et sa modification.

~PerfsFile();

Le destructeur supprime la mémoire allouée par la liste.

```
int getDate();
File& getFile();
int getNbMoyenClientsFile();
float getTpsAttenteMoy();
float getTpsServiceMoy();
float getTpsReponseMoy();
list<PerfServeur>& getListeResPerfServeur();
float getTpsinterarriveeMoy();
int getTauxPerte();
```

Ces accesseurs permet d'accéder aux attributs privés de la classe.

```
void calcul_nb_moyen_clients_file();
```

Cette méthode calcule le nombre moyen de clients dans la file. Ce calcul sera possible grâce aux données que contient l'attribut *file*. Elle s'occupe de mettre à jour l'attribut *nb_moyen_clients_file*.

```
void calcul_tps_attente_moyen();
```

Cette méthode calcule le temps d'attente moyen dans la file. Ce calcul sera possible grâce aux données que contient l'attribut *file* de la classe. Elle s'occupe de mettre à jour l'attribut *tps_attente_moyen*.

```
void calcul_tps_service_moyen();
```

Cette méthode calcule le temps de service moyen d'une file. Ce calcul sera possible grâce aux données que contient l'attribut *file* de la classe. Elle s'occupe de mettre à jour l'attribut *tps_service_moyen*.

```
void calcul_tps_reponse_moyen();
```

Cette méthode calcule le temps de réponse moyen de la file. Ce calcul sera possible grâce aux résultats des deux méthodes précédentes. Elle s'occupe de mettre à jour l'attribut *tps_reponse_moyen*.

```
void calcul_taux_utilisation_serveurs(const int duree_simulation);
```

Cette méthode calcule le taux d'utilisation des serveurs reliés à la file. Ce calcul sera possible grâce à la durée de la simulation donnée en paramètre et le temps d'utilisation du serveur que l'on peut récupérer grâce à la liste des serveurs présents dans l'objet de type *File*. Elle s'occupe de mettre à jour l'attribut *ListeResPerfServeur*.

```
void calcul_tps_interarrivee_moyen();
```

Cette méthode calcule le temps d'inter-arrivée moyen entre chaque client pour une file. Ce calcul est possible grâce à la liste de mesures du temps d'inter-arrivée de la file en fonction du temps contenue dans l'attribut *file*. Elle s'occupe de mettre à jour l'attribut *tps_interarrivee_moyen*.

```
void calcul_taux_perte();
```

Cette méthode calcule le taux de perte d'une file. Ce calcul sera possible grâce à la connaissance du nombre de clients entrés dans la file et le nombre de clients présentés à la file. Elle s'occupe de mettre à jour l'attribut *taux_perte*.

```
friend ostream& operator << (ostream& flux, PerfsFile const& perfs);
```

La surcharge de l'opérateur << permet d'afficher les performances d'une file ainsi que la date à laquelle le calcul a été effectué. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet PerfsFile, cette méthode doit être "amie" de la classe, spécifiée par le mot-clé friend. Elle prend en paramètre un objet de type ostream pour gérer la sortie standard et une référence constante sur les performances d'une file pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante permet de donner l'accès à l'objet en lecture uniquement.

```
PerfsFile& operator = (PerfsFile const& p);
```

Cette surcharge correspond à la surcharge de l'opérateur = pour pouvoir recopier les attributs de l'objet *p*. L'objet est passé en référence constante pour éviter une copie locale de l'objet et sa modification.

5.4 Classe HistoriqueRF

La classe *HistoriqueRF* nous permet d'avoir un historique des performances pour un réseau fermé, ce qui nous permet de les afficher et de les sauvegarder.

```
#include <iostream>
```

Cette bibliothèque gère l'entrée et la sortie standard. Elle est utile vu que l'on appelle une fonction lui appartenant.

```
#include <list>
```

Cette bibliothèque s'occupe de la gestion d'une liste. Elle est nécessaire vu que l'unique attribut est un objet *list*.

```
#include "PerfRF.hh"
```

Cette bibliothèque gère le calcul des performances d'un réseau fermé. Elle doit être incluse car les performances du réseau sont sauvegardées dans l'historique.

```
#include "PerfFile.hh"
```

Cette bibliothèque gère le calcul des performances des files d'un réseau. Elle doit être incluse car les performances des files sont sauvegardées dans l'historique.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct PerformancesRF{
        PerfsRF perfReseau;
        list <PerfFile> ListePerfsFiles;
}PerformancesRF;
```

Cette structure contient les performances du réseau et de ses files à une date donnée. Les performances du réseau sont représentées par un objet *PerfsRF* et ceux de ses files sont représentées par des objets *PerfFile* regroupées dans un objet *list*. Une liste a été choisie car le nombre de files n'est pas le même selon le réseau créé par l'utilisateur.

```
class HistoriqueRF{

private :

list <PerformancesRF> historique
```

Cet attribut correspond à l'ensemble des performances du réseau et de ses files, calculé pendant la simulation. Cette liste contient des structures de type *PerformancesRF*. La structure choisie est une liste car le nombre de fois où ses performances vont être calculées est variable.

```
public :

HistoriqueRF();
```

Il s'agit du constructeur par défaut. Il initialise l'attribut *historique* avec le constructeur par défaut de la classe *list*.

~HistoriqueRF();

Le destructeur désalloue la mémoire allouée pour la liste.

```
list<PerformancesRF>& getHistorique();
```

Cet accesseur permet d'avoir accès à l'attribut *historique*.

```
void ajouterPerfsReseau(PerfsRF& pR);
```

Cette méthode alloue la mémoire pour un nouvel élément *PerformancesRF* et initialise le champ *perfsReseau* avec l'objet *pR* passé en paramètre et la liste *ListePerfsFiles* avec le constructeur par défaut de la classe *list*. *pR* est passé en reférence pour éviter une copie temporaire de l'objet.

```
void ajouterPerfsFile(File& f);
```

Cette méthode alloue la mémoire pour un nouvel élément dans la liste *ListePerfsFiles* et initialise l'élément avec l'objet *f* passé en paramètre. *f* est passé en référence pour éviter une copie temporaire de l'objet.

```
friend ostream& operator <<(ostream& flux, HistoriqueRF const& H);
```

La surcharge de l'opérateur << permet d'afficher l'historique des performances du réseau. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet PerfsFile, cette méthode doit être "amie" de la classe, spécifiée par le mot-clé friend. Elle prend en paramètre un objet de type ostream pour gérer la sortie standard et une référence constante sur l'historique pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante permet de donner l'accès à l'objet en lecture uniquement.

5.5 Classe HistoriqueRO

La classe *HistoriqueRO* hérite de *HistoriqueRF*. Elle nous permet d'avoir un historique des performances calculées pour un réseau ouvert, ce qui nous permet de les afficher et de les sauvegarder.

```
#include <iostream>
```

Cette bibliothèque gère l'entrée et la sortie standard.

```
#include <list>
```

Cette bibliothèque gère les listes. Elle est nécessaire vu que l'unique attribut est un objet *list*.

```
#include "PerfRO.hh"
```

Cette bibliothèque gère le calcul des performances d'un réseau ouvert. Elle doit être incluse car les performances du réseau sont sauvegardées dans l'historique.

```
#include "PerfFile.hh"
```

Cette bibliothèque gère le calcul des performances des files d'un réseau. Elle doit être incluse car les performances des files sont sauvegardées dans l'historique.

```
using namespace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
struct PerformancesRO{
        PerfsRO perfReseau;
        list <PerfsFile> ListePerfsFiles;
}PerformancesRO;
```

Cette structure contient les performances du réseau et de ses files à une date donnée. Les performances du réseau sont représentées par un objet *PerfsRO* et ceux de ses files sont représentées par des objets *PerfFile* regroupées dans un objet *list*. Une liste a été choisie car le nombre de files n'est pas le même selon le réseau créé par l'utilisateur.

```
class HistoriqueRO

private :

list <PerformancesRO> historique
```

Cet attribut correspond à une liste des performances du réseau à chaque fois que l'utilisateur le demande. Chaque élément de la liste est de type *PerformancesRO*. Une liste est utilisée pour l'historique car le nombre de demandes de calculs des performances est variable. Une structure extensible est donc nécessaire.

```
public :

HistoriqueRO();
```

Ce constructeur appelle le constructeur par défaut de l'objet *historique*.

~HistoriqueRO()

Le destructeur libère la mémoire allouée pour la liste *historique*.

```
list<PerformancesRO>& getHistorique();
```

Cet accesseur permet d'avoir accès à l'attribut *historique*.

```
void ajouterPerfsReseau(PerfsRO& reseau);
```

Cette méthode va créer un nouvel élément *PerformancesRO* dans la liste *historique*. Le champ *perfReseau* de ce nouvel élément sera initialisé grâce à la surcharge de l'opérateur = de la classe *PerfsRO*. Le champ *ListePerfsFiles* est initialisé avec le contructeur par défaut de la classe *list*. L'objet en paramètre est passé par référence pour éviter une copie locale inutile de l'objet.

```
void ajouterPerfsFile(File& f);
```

Cette méthode va ajouter au dernier élément *PerformancesRO* de la liste *historique*, l'objet *f* à la liste *ListePerfsFile*. L'objet en paramètre est passé par référence pour éviter une copie locale inutile de l'objet.

```
friend ostream& operator <<(ostream& flux, HistoriqueRO const& H);
```

La surcharge de l'opérateur << permet d'afficher les éléments de la liste *historique*. Cette méthode n'appartient pas à la classe, ainsi pour éviter d'utiliser des méthodes accesseurs pour accéder aux attributs de l'objet PerfsFile, cette méthode doit être "amie" de la classe, spécifiée par le mot-clé friend. Elle prend en paramètre un objet de type ostream pour gérer la sortie standard et une référence constante sur *historique* pour pouvoir récupérer ses caractéristiques étant donné que la méthode ne fait pas partie de la classe. La référence constante permet de donner l'accès à l'objet en lecture uniquement.

6 Module Interface Graphique

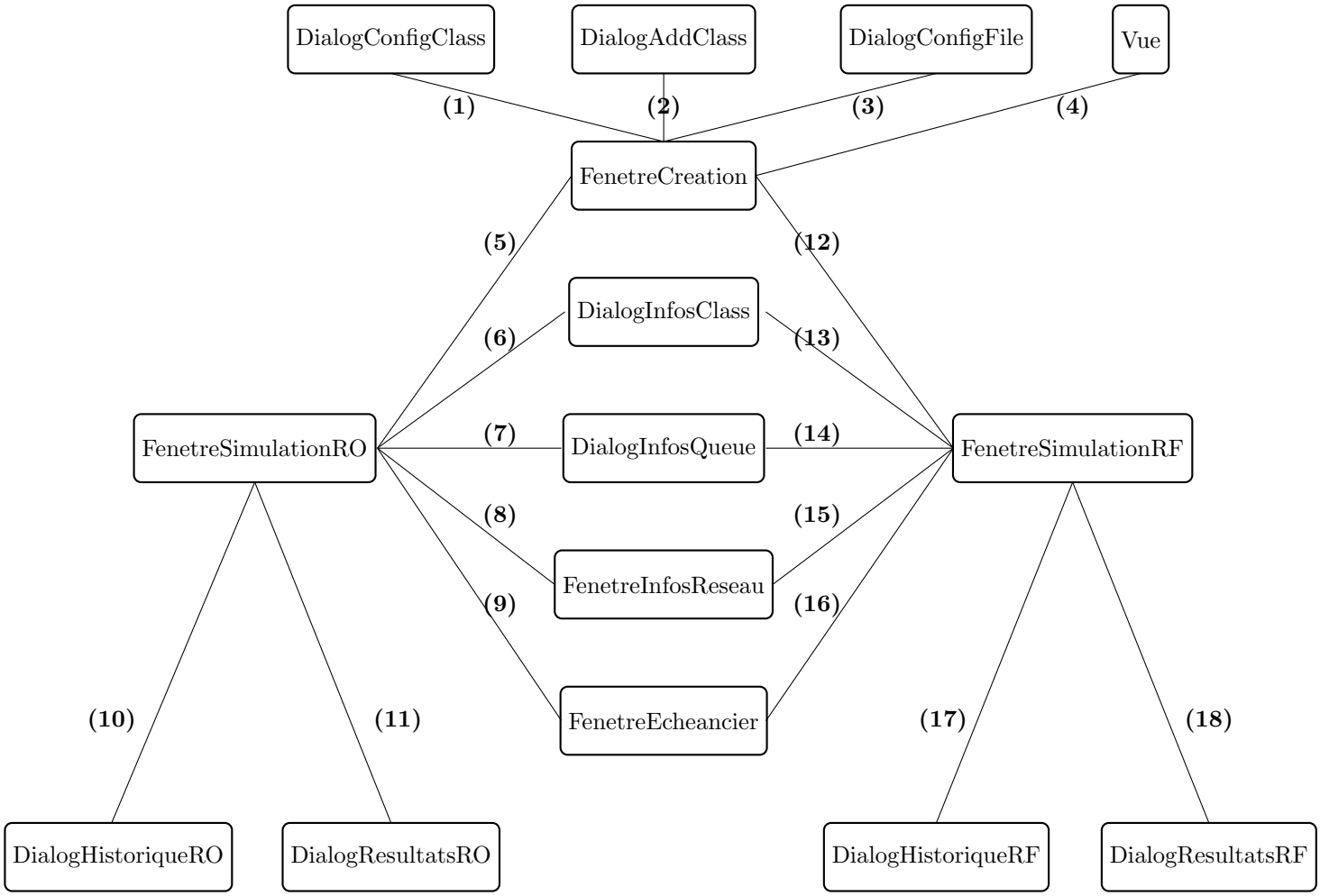


Diagramme de classes

(1) DialogConfigClass - 1..* — est ouvert — *FenetreCreation* - 1..1

(2) DialogAddClass - 1..* — est ouvert — *FenetreCreation* - 1..1

(3) DialogConfigClass - 1..* — est ouvert — *FenetreCreation* - 1..1

(3) DialogConfigFile - 1..* — est ouvert — *FenetreCreation* - 1..1

(4) FenetreCreation - 1..1 — se compose — *Vue* - 1..1

(5) FenetreCreation - 1..1 — est liée — *FenetreSimulationRO* - 0..*

(6) FenetreSimulationRO - 1..1 — ouvre — *DialogInfosClass* - 0..*

(7) FenetreSimulationRO - 1..1 — ouvre — *DialogInfosQueue* - 0..*

(8) FenetreSimulationRO - 1..1 — se compose — *FenetreInfosReseau* - 1..1

(9) FenetreSimulationRO - 1..1 — se compose — *FenetreEcheancier* - 1..1

(10) FenetreSimulationRO - 1..1 — ouvre — *DialogHistoriqueRO* - 0..*

(11) FenetreSimulationRO - 1..1 — ouvre — *DialogResultatsRO* - 0..*

(12) FenetreCreation - 1..1 — est liée — *FenetreSimulationRF* - 0..*

(13) FenetreSimulationRF - 1..1 — ouvre — *DialogInfosClass* - 0..*

(14) FenetreSimulationRF - 1..1 — ouvre — *DialogInfosQueue* - 0..*

(15) FenetreSimulationRF - 1..1 — se compose — *FenetreInfosReseau* - 1..1

(16) *FenetreSimulationRF* - 1..1 — se compose — *FenetreEcheancier* - 1..1

(17) *FenetreSimulationRF* - 1..1 — ouvre — *DialogHistoriqueRF* - 0..*

(18) *FenetreSimulationRF* - 1..1 — ouvre — *DialogResultatsRF* - 0..*

Le module *Interface Graphique* s'occupe de l'interface graphique de l'application. L'interface graphique est le lien entre l'utilisateur et le reste de l'application. Le module se compose de 15 classes.

L'une des classes les plus importantes du module est la classe *FenetreCreation*. Elle permet de créer la fenêtre permettant à l'utilisateur de créer son réseau.

Cette classe est associée à deux autres classes importantes, la classe *FenetreSimulationRO* et la classe *FenetreSimulationRF*, qui permettent d'afficher le déroulement de la simulation du réseau en fonction de son type à l'utilisateur et lui permettre d'interagir avec.

Ces trois fenêtres se composent d'une vue représentée par un objet de la classe Vue. Cet objet permet d'afficher le réseau dessiné par l'utilisateur.

Les classes *DialogConfigFile*, *DialogConfigClass* et *DialogAddClass* permettent la création de boîtes de dialogue permettant la configuration des files et des classes et sont ouvertes à partir de la fenêtre de création.

La classe *FenetreInfosReseau* permet la création d'une fenêtre affichant les informations du réseau simulé et est ouvert dans les fenêtres de simulation.

Les classes *DialogInfosClass* et *DialogInfosQueue* permettent la création de boîtes de dialogues pour donner les informations sur les classes et les files. Elles sont ouvertes à partir des fenêtres de simulation.

Les classes *DialogHistoriqueRF* et *DialogHistoriqueRO* permettent la création de boîtes de dialogue affichant l'historique du réseau en fonction de son type et sont ouvertes à partir des fenêtres de simulation.

Les classes *DialogResultatsRF* et *DialogResultatsRO* permettent la création de boîtes de dialogue affichant les résultats des performances du réseau et de ses files. Ces boîtes sont ouvertes à partir des fenêtres de simulation.

La classe *FenetreEcheancier* permet la création de fenêtres affichant l'état de l'échéancier et sont ouvertes dans les fenêtres de simulation.

6.1 Classe Vue

```
#include <QGraphicsView>[15]
```

Cette bibliothèque gère la vue de la scène graphique. La classe *Vue* hérite de la classe *QGraphicsView*.

```
#include "File.hh"
```

Cette bibliothèque gère les files d'attente. Elle possède une structure *coordonnees* utile pour garder la position de la souris.

```
#include <QMouseEvent>[13]
```

Cette bibliothèque gère les événements se produisant avec la souris. Une méthode de la classe *Vue* nécessite un objet de cette bibliothèque.

```
Class Vue : public QGraphicsView{
```

La classe *Vue* hérite de la classe *QGraphicsView* de façon publique pour garder l'encapsulation des membres de la classe héritée.

```
private :
```

```
coordonnees pos_clic;
```

Cet attribut correspond à la position de la souris après un clic. Il correspond à la structure *coordonnees* car cette structure a comme champs, deux entiers, utile pour garder l'abscisse et l'ordonnée du clic.

```
public:
```

```
void mousePressEvent(QMouseEvent *event); [33]
```

Cette méthode est une méthode de la classe *QWidget* permettant de récupérer un clic. Le clic est représenté par l'objet *QMouseEvent* passé en paramètre sous forme de pointeur. Cette méthode est redéfinie dans cette classe pour récupérer la position du clic.

```
Vue(QWidget * parent = NULL)
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget*, initialisé à NULL de base. Ce pointeur permet d'indiquer le widget parent. Le constructeur va appeler un constructeur de la classe mère et initialisé les champs de *pos_clic* à 0.

```
~Vue()
```

Aucune mémoire ne sera allouée dynamiquement. Le destructeur sera donc vide.

```
int getPosXClic();
```

Cet accesseur permet de récupérer l'abscisse de la position du clic sous forme d'entier.

```
int getPosYClic();
```

Cet accesseur permet de récupérer l'ordonnée de la position du clic sous forme d'entier.

```
};
```

6.2 Classe FenetreCreation

La classe *FenetreCreation* est la classe principale de l'interface. L'application se déroule grâce à la fenêtre graphique réalisée à partir de cette classe. Elle laisse la possibilité à l'utilisateur d'utiliser différentes fonctionnalités. En conséquence, des fenêtres graphiques et boîtes de dialogue se lanceront lorsque l'utilisateur souhaite les exécuter.

```
#include <QWidget> [10]
```

Cette bibliothèque contient la classe *QWidget*. Elle est nécessaire car la classe *FenetreCreation* hérite de la classe *QWidget*.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
#include <QAction>[11]
```

Cette bibliothèque contient la classe *QAction* qui fournit une action d'interface utile à l'utilisation des *QMenu*.

```
#include <QMenu>[12]
```

Cette bibliothèque contient la classe *QMenu* qui fournit des menus. La fenêtre de création utilise des menus pour la gestion des fichiers et la création du réseau.

```
#include <QPushButton>[14]
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. La fenêtre de création se compose de multiples boutons afin de gérer la création du réseau.

```
#include "Vue.hh"
```

Cette bibliothèque contient la classe Vue permettant d'afficher une scène graphique. La fenêtre de création est composé d'une zone de dessin permettant de dessiner le réseau et devant être afficher.

```
#include <QInputDialog>[16]
```

Cette bibliothèque contient la classe *QInputDialog* qui permet de générer des boîtes de dialogue pour demander une entrée à l'utilisateur. Ces boîtes sont utiles pour gérer la configuration des classes et des files.

```
#include <QFileDialog>[17]
```

Cette bibliothèque contient la classe *QFileDialog* qui permet de créer des boîtes de dialogue demandant à l'utilisateur de sélectionner un fichier. Ces boîtes sont utiles pour gérer les ouvertures et les fermetures de fichiers.

```
#include <QMessageBox>[18]
```

Cette bibliothèque contient la classe *QMessageBox* qui fournit une boîte de dialogue modale informant ou questionnant l'utilisateur. Ces boîtes sont utiles pour avertir l'utilisateur sur des potentiels changements.

```
#include <QHBoxLayout>[24]
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Elle va permettre de ranger les boutons de la fenêtre horizontalement.

```
#include "FenetreSimulationRF.hh"
```

Cette bibliothèque contient la classe *FenetreSimulationRF* qui permet d'utiliser une fenêtre de simulation d'un réseau fermé. Cette fenêtre est liée à la fenêtre de création.

```
#include "FenetreSimulationRO.hh"
```

Cette bibliothèque contient la classe *FenetreSimulationRO* qui fournit une fenêtre de simulation s'occupant de la simulation d'un réseau ouvert. Cette fenêtre est liée à la fenêtre de création.

```
#include "DialogConfigFile.hh"
```

Cette bibliothèque contient la classe*DialogConfigFile* qui fournit une boîte de configuration des files. Ces boîtes permettent à l'utilisateur de configurer les files du réseau.

```
#include "DialogConfigClass.hh"
```

Cette bibliothèque contient la classe *DialogConfigClass* permettant d'appeler une fenêtre afin de configurer les classes.

```
#include "DialogAddClass.hh"
```

Cette bibliothèque contient la classe *DialogAddClass* permettant d'appeler une fenêtre afin de configurer les classes au moment de leur ajout dans le réseau.

```
#include "Reseau.hh"
```

Cette bibliothèque contient la classe *Reseau*. Un objet de cette classe est nécessaire car la fenêtre de création permet de créer et configurer le réseau. La configuration est gardée dans un objet de cette classe.

```
#include "lecture_ecriture.hh"
```

Cette bibliothèque contient les fonctions utiles pour la lecture et l'écriture d'un réseau dans un fichier ainsi que l'écriture des performances dans un fichier. La fenêtre de création possède des boutons permettant ces actions.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
class FenetreCreation : public QWidget {
```

La classe *FenetreCreation* hérite de la classe *QWidget* afin de pouvoir afficher la classe sous forme de fenêtre. L'héritage est public pour utiliser les méthodes en public.

```
Q_OBJECT
```

Cette macro est nécessaire pour définir des signaux et des slots.

```
private :

Reseau * reseau;
```

Cet attribut est un pointeur sur un objet de type *Reseau*. L'objet est crée et modifié grâce à la fenêtre de création qui permet à l'utilisateur de créer et configurer son réseau. Un pointeur sur cet objet est utilisé car il ne fait pas partie intégrante de la fenêtre.

```
QGraphicsScene * scene;
```

Cet attribut est un pointeur sur un objet de type *QGraphicsScene* permettant de contenir tous les objets qui vont servir à dessiner le réseau afin de pouvoir l'afficher. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
Vue * vue;
```

Cet attribut est un pointeur sur un objet de type *Vue* permettant l'affichage de la scène graphique *scene*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
FenetreSimulationRF  * fenetre_RF;
```

Cet attribut est un pointeur sur un objet de type *FenetreSimulationRF* qui affiche la simulation et son déroulement si le réseau choisi est de type fermé. La fenêtre de création agit sur la fenêtre de simulation. Ainsi un pointeur est utilisé pour montrer la liaison entre la fenêtre de création et la fenêtre de simulation.

```
FenetreSimulationRO * fenetre_RO;
```

Cet attribut est un pointeur sur un objet de type *FenetreSimulationRO* qui va afficher la simulation et son déroulement si le réseau choisi est de type ouvert. Il s'agit d'un pointeur pour montrer le lien entre la fenêtre de création et de simulation.

QMenu *menuFichier;

Cet attribut est un pointeur sur un objet de type *QMenu* correspondant à un menu. Il permet d'accéder aux actions *Open* et *Save*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction *Open;

Cet attribut est un pointeur *QAction* correspondant au bouton *Open* du menu *menuFichier*. Ce bouton est relié au slot *void Open()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * Save;

Cet attribut est un pointeur *QAction* correspondant au bouton *Save* du menu *menuFichier*. Ce bouton est relié au slot *void Save()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QMenu * menuQueue;

Cet attribut est un pointeur sur un objet de type *QMenu* correspondant à un menu. Il permet d'accéder aux options *AddQueue, MoveQueue, RemoveQueue et ConfigureQueue*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * AddQueue ;

Cet attribut est un pointeur *QAction* correspondant au bouton *AddQueue* du menu *menuQueue*. Ce bouton est relié au slot *void Add_Queue()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * MoveQueue ;

Cet attribut est un pointeur *QAction* correspondant au bouton *MoveQueue* du menu *menuQueue*. Ce bouton est relié au slot *void Move_Queue()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * RemoveQueue;

Cet attribut est un pointeur *QAction* correspondant au bouton *RemoveQueue* du menu *menuQueue*. Ce bouton est relié au slot *void Remove_Queue()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * ConfigureQueue;

Cet attribut est un pointeur *QAction* correspondant au bouton *ConfigureQueue* du menu *menuQueue*. Ce bouton est relié au slot *void Configure_Queue()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QMenu * menuClass;

Cet attribut est un pointeur sur un objet de type *QMenu* correspondant à un menu. Il permet d'accéder aux options *ConfigureClass, AddClass et RemoveClass*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * ConfigureClass;

Cet attribut est un pointeur *QAction* correspondant au bouton *ConfigureClass* du menu *menuClass*. Ce bouton est relié au slot *void configurer_classes()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * AddClass;

Cet attribut est un pointeur *QAction* correspondant au bouton *AddClass* du menu *menuClass*. Ce bouton est relié au slot *void Add_Class()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * RemoveClass;

Cet attribut est un pointeur *QAction* correspondant au bouton *RemoveClass* du menu *menuClass*. Ce bouton est relié au slot *void Supp_Class()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QMenu * menuPropreReseau;

Cet attribut est un pointeur sur un objet de type *QMenu* correspondant à un menu. Il permet d'accéder aux options *reseau1, reseau2, reseau3 et reseau4*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * reseau1;

Cet attribut est un pointeur *QAction* correspondant au bouton *reseau1* du menu *menuPropreReseau*. Ce bouton est relié au slot *void charger_reseau1()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

QAction * reseau2;

Cet attribut est un pointeur *QAction* correspondant au bouton *reseau2* du menu *menuPropreReseau*. Ce bouton est relié au slot *void charger_reseau2()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QAction * reseau3;
```

Cet attribut est un pointeur *QAction* correspondant au bouton *reseau3* du menu *menuPropreReseau*. Ce bouton est relié au slot *void charger_reseau3()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QAction * reseau4;
```

Cet attribut est un pointeur *QAction* correspondant au bouton *reseau4* du menu *menuPropreReseau*. Ce bouton est relié au slot *void charger_reseau4()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Pause;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Pause*. Ce bouton est relié au slot *void Pause()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Resume;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Resume*. Ce bouton est relié au slot *void Resume()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton* Start;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Start*. Ce bouton est relié au slot *void Start()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Stop;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Stop*. Ce bouton est relié au slot *void Stop()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Reset;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Reset*. Ce bouton est relié au slot *void Reset()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Zoom_in;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Zoom_in*. Ce bouton est relié au slot *void Zoom_in()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Zoom_out;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Zoom_out*. Ce bouton est relié au slot *Zoom_out()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Quit;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Quit*. Ce bouton est relié au slot *void Quitter()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * changeType;
```

Cet attribut est un pointeur *QPushButton* correspondant au bouton *Change Type*. Ce bouton est relié au slot *void changer_type_reseau()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
int taille_zoom;
```

Cet attribut est un entier correspondant au zoom appliqué sur la scène graphique.

```
QHBoxLayout * menu_haut;
```

Cet attribut est un pointeur sur un objet *QHBoxLayout*.Il permet de placer horizontalement les *QWidget menuFichier, Start, Pause, Resume, Stop, Reset* et *Quit*.

```
QVBoxLayout * menu_gauche;
```

Cet attribut est un pointeur sur un objet *QVBoxLayout* permettant de ranger des widgets de façon vertical. Cette boîte de rangement contient les menus *menuQueue*, *menuClasse* et le bouton *changeType*.

```
QVBoxLayout * menu_droite;
```

Cet attribut est un pointeur sur un objet *QVBoxLayout* permettant de ranger des widgets de façon vertical. Cette boîte de rangement contient les boutons *Zoom_in* et *Zoom_out*.

```
public slots : [34]
```

Les slots sont passés en public pour pouvoir y accéder en dehors de la classe.

```
void Pause();
```

Cette méthode met en pause la simulation. Elle émet un signal *stop()* au *QTimer* présent dans une des fenêtre gérant la simulation. La fenêtre de création sera à nouveau accessible.

```
void Start();
```

Cette méthode permet le lancement de la simulation. Elle affiche une fenêtre de dialogue qui demande la durée de simulation et le nombre de clients initial. Elle lance le chronomètre et la fenêtre de simulation correspondant au type du réseau. Pour éviter toute erreur de manipulation, certains boutons de la fenêtre de création sont déconnectés.

```
void Stop();
```

Cette méthode arrête la simulation. Elle affiche les résultats de calculs de performances. Elle demande également, par une boîte de dialogue, la sauvegarde des performances dans un fichier. Les boutons désactivés de la fenêtre de création sont réactivés.

```
void Resume();
```

Cette méthode permet de reprendre la simulation. Elle relance le *QTimer* présent dans une des fenêtres gérant la simulation. Certains boutons de la fenêtre de création seront déconnectés pour éviter des erreur de manipulations.

```
void Reset();
```

Cette méthode supprime le réseau de la scène graphique et supprime l'objet de type *Réseau*. Puis le constructeur de la classe *Reseau* est rappelé pour recréer un objet *Reseau*. Les boutons désactivés de la fenêtre de création sont réactivés.

```
void Zoom_In();
```

Cette méthode agrandit la taille des objets se trouvant sur la scène graphique. Elle modifie l'attribut *taille_zoom* afin de conserver le zoom effectué.

```
void Zoom_Out();
```

Cette méthode rétrécit la taille des objets se trouvant sur la scène graphique. Elle modifie l'attribut *taille_zoom* afin de conserver le zoom effectué.

```
void Add_Queue();
```

Cette méthode va construire un nouvel objet *File* quand l'utilisateur va cliquer sur la fenêtre, l'ajouter au réseau puis ajouter une image représentant la file sur la fenêtre aux coordonnées du clic de la souris. Un nouvel objet *Serveur* sera également instancié, celui-ci sera associé à la nouvelle *File* créée.

```
void Remove_Queue();
```

Cette méthode permet la suppression d'une file d'attente en cliquant sur l'image la représentant. Toutes les données associées seront à leur tour supprimées.

```
void Move_Queue();
```

Cette méthode permet à l'utilisateur de déplacer une file d'attente présent sur la fenêtre graphique en cliquant dessus ainsi dans un prmier temps puis aux nouvelles coordonnées dans un second temps.

```
void Configure_Queue();
```

Cette méthode permet la configuration d'une file d'attente. Elle effectue cette modification grâce à une boîte de dialogue DialogConfigFile.

```
void Add_Class();
```

Cette méthode permet l'ajout d'une classe de clients dans le réseau. Un objet de la classe *ClasseClient* est créé et ajouté à l'objet sur lequel pointe l'attribut *reseau*. Puis une boîte de dialogue *DialogAddClass* s'affiche et propose à l'utilisateur de configurer la classe. Ses configurations sont ajoutées à l'objet *ClasseClient* qui vient d'être créé.

```
void Supp_Class();
```

Cette méthode permet la suppression d'une classe de clients dans le réseau et demande à l'utilisateur de confirmer ce choix grâce à une boîte de dialogue.

```
void configurer_classes();
```

Cette méthode permet la modification des données d'une classe de clients dans le réseau grâce à une boîte de dialogue *DialogConfigClass*.

```
void Open();
```

Cette méthode permet de récupérer un réseau à partir d'un fichier. Une boîte de dialogue s'affiche et demande à l'utilisateur le nom du fichier. Ce nom est passé en paramètre de la méthode *void lecture_fichier_reseau(Reseau * reseau, const string nomfic)* permettant de récupérer les informations du fichier.

```
void Save();
```

Cette méthode permet la sauvegarde d'un réseau grâce à une boîte de dialogue. Une boîte de dialogue s'affiche et demande à l'utilisateur le nom du fichier. Ce nom est passé en paramètre de la méthode *void ecriture_fichier_reseau(const string nomfic, Reseau& reseau)*

```
void Quitter();
```

Cette méthode permet de quitter l'application. Une boîte de dialogue s'affiche et propose à l'utilisateur de sauvegarder son réseau construit.

```
void changer_type_reseau();
```

Cette méthode permet la modification du type de réseau. Elle appelle la méthode *setTypeReseau(const int type)* qui modifie le type du réseau de l'objet *Reseau* et affiche une entrée et une sortie sur la scène graphique si le réseau est ouvert.

```
void charger_reseau1();
void charger_reseau2();
void charger_reseau3();
void charger_reseau4();
```

Cette méthode charge un réseau préétabli à partir d'un fichier spécifique.

```
public :

FenetreCreation();
```

Le constructeur va allouer dynamiquement les pointeurs en attribut, initialiser l'attribut *taille_zoom* à 1 et appeler le constructeur de la classe mère.

~FenetreCreation();

La mémoire est allouée dynamiquement pour tous les pointeurs en attribut de la classe. Le destructeur va donc libérer cette mémoire.

```
int setTaille_Zoom(const int newTaille);
```

Cet accesseur modifie la valeur de l'attribut *taille_zoom* avec la valeur de *newTaille*. L'entier passé en paramètre est une constante pour éviter de la modifier. Une exception de type *invalid_argument* est levée si un entier invalide est passé en paramètre.

```
void afficher_reseau();
```

Cette méthode permet d'afficher ou de réafficher le réseau dans le cas d'un ajout ou d'une suppression d'éléments du réseau.

```
void afficher_routage_classe(ClasseClient& cc);
```

Cette méthode permet l'affichage du routage sur la scène graphique de la classe de clients passée en paramètre. L'objet de type *ClasseClient* est passé en référence pour éviter une copie locale inutile.

```
void afficher_file(coordonnees centre_dessin);
```

Cette méthode permet l'affichage d'une station sur la scène graphique aux coordonnées récupérées du clic de la souris.

```
void afficher_entree_sortie();
```

Cette méthode permet l'affichage d'une entrée et d'une sortie sur la scène graphique dans le cas où le réseau est de type ouvert.

```
};
```

6.3 Classe FenetreSimulationRF

La classe *FenetreSimulationRF* permet de créer la fenêtre dans laquelle se déroule la simulation d'un réseau fermé. Elle contient le réseau créé par l'utilisateur ainsi que 2 fenêtres : celle indiquant les caractéristiques du réseau et celle indiquant les événements de l'échéancier.

`#include <QWidget>`

Cette bibliothèque contient la classe permettant de créer une fenêtre en Qt. La classe *FenetreSimulationRF* hérite de la classe *QWidget*. La bibliothèque doit donc être incluse.

`#include "FenetreInfosReseau.hh"`

Cette bibliothèque est nécessaire pour l'attribut de type *FenetreInfosReseau*.

`#include "FenetreEcheancier.hh"`

Cette bibliothèque est nécessaire pour l'attribut de type *FenetreEcheancier*.

`#include "Vue.hh"`

Cette bibliothèque doit être incluse pour l'attribut *vue* de type *Vue ** et permet d'afficher une scène graphique représentée par un objet de type *QGraphicsScene*.

`#include <QPushButton>`

Cette bibliothèque contient la classe permettant de créer un bouton en Qt. Elle est nécessaire pour les différents boutons contenus dans la fenêtre.

`#include <QTimer>`[23]

Cette bibliothèque est utile pour l'attribut de la classe de type *QTimer*. Elle permet de créer un minuteur (relancé toutes les secondes dans notre cas).

`#include <QTime>`[20]

Cette bibliothèque est nécessaire pour l'attribut de la classe de type *QTime*. Elle permet de créer un chronomètre pour garder en mémoire le temps qui s'écoule au cours de la simulation.

`#include "DialogResultatsRF.hh"`

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogResultatsRF* dans la méthode *Results()*.

`#include "DialogHistoriqueRF.hh"`

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogHistoriqueRF* dans la méthode *History()*.

`#include "DialogInfosClass.hh"`

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogInfosClass* dans la méthode *Infos_Classes()*.

`#include "DialogInfosFile.hh"`

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogInfosFile* dans la méthode *Infos_Queue()*.

`#include <QInputDialog>`

Cette bibliothèque permet d'ouvrir des boîtes de dialogue demandant une entrée à l'utilisateur. Elle est utile pour ouvrir une boîte demandant à l'utilisateur d'entrer le nom d'un fichier pour la sauvgearde des performances.

`#include <stdexcept>`

Cette bibliothèque gère les exceptions.

`#include <QMessageBox>`

Cette bibliothèque sert à créer des objets de la classe *QMessageBox* qui permet d'interagir avec l'utilisateur à l'aide de boîtes de message. A la fin de la simulation, une fenêtre s'affichera pour demander à l'utilisateur s'il souhaite sauvegarder les performances du réseau.

`#include "lecture_ecriture.hh"`

Cette bibliothèque est nécessaire pour accéder aux fonctions de lecture et d'écriture dans un fichier. Ces fonctions sont utilisées pour la sauvegarde des performances dans un fichier.

`#include <QVBoxLayout>`[25]

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Elle va permettre de ranger les boutons de la fenêtre verticalement.

```
Class FenetreSimulationRF : public QWidget
```

La classe *FenetreSimulationRF* hérite de la classe *QWidget* pour pouvoir créer la fenêtre qui est un widget. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe *QWidget*.

Q_OBJECT

Cette macro est nécessaire pour définir de nouveaux slots et signaux.

```
private :

SimulationRF simulation ;
```

Cet attribut est objet de type *SimulationRF*. Cet objet permet d'effectuer une simulation sur un réseau fermé. Il est nécessaire d'avoir cet objet en attribut pour afficher la simulation à l'utilisateur.

```
FenetreInfosReseau fenetre_infos_reseau ;
```

Cet attribut correspond à la fenêtre affichant les caractéristiques du réseau simulé. Cette fenêtre sera affichée dans la fenêtre de simulation, d'où le fait que cet attribut est contenu dans la classe.

```
FenetreEcheancier fenetre_echancier ;
```

Cet attribut correspond à la fenêtre affichant les événements de l'échéancier. Cette fenêtre sera affichée dans la fenêtre de simulation, d'où le fait que cet attribut est contenu dans la classe.

```
Vue * vue ;
```

Cet attribut permet d'afficher la scène graphique récupérée de la fenêtre de création et afficher le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Results ;
```

Cet attribut correspond au bouton pour afficher les résultats de performances du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * History ;
```

Cet attribut correspond au bouton pour afficher l'historique des performances du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Infos_Queue ;
```

Cet attribut correspond au bouton pour afficher les informations sur les files composant le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Infos_Classes ;
```

Cet attribut correspond au bouton pour afficher les différentes classes composant le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * layout ;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage des boutons composant la fenêtre de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QTime * chrono ;
```

Cet attribut correspond au chronomètre qui garde en mémoire le temps qui s'écoule au cours de la simulation. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QTimer * timer_chrono ;
```

Cet attribut est un minuteur qui se renouvelle sans cesse selon un intervalle de temps défini . Ici, c'est toutes les secondes. À chaque fois qu'une seconde s'écoule, l'attribut *chrono* ci-dessus s'incrémente d'une seconde. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
int secondes ;
```

Cet attribut permet de garder en mémoire le nombre de secondes écoulées lors de la simulation, utile pour décrémenter l'attribut *chrono* lorsque l'on veut le remettre à 0 dans la méthode *start_chrono*.

```
public slots :
```

Les slots sont des méthodes pouvant être appelées par des signaux comme le clic sur un bouton.

```
void Results() ;
```

Cette méthode va répondre au signal *clicked()* de *Results*. Un objet de type *DialogResultatsRF* sera instancié pour permettre d'afficher les résultats de performances d'un réseau fermé.

```
void History() ;
```

Cette méthode va répondre au signal *clicked()* de *History*. Un objet de type *DialogHistoriqueRF* sera instancié pour permettre d'afficher l'historique des performances d'un réseau fermé.

```
void Infos_Queue() ;
```

Cette méthode va répondre au signal *clicked()* de *Infos_Queue*. Elle attend de nouveau un signal *clicked()* pour récupérer les coordonnées de la file cliquée. Un objet de type *DialogInfosFile* sera instancié permettant d'afficher les caractéristiques correspondantes à la file dont l'utilisateur souhaite connaître les informations.

```
void Infos_Classes() ;
```

Cette méthode va répondre au signal *clicked()* de *Infos_Classes*. Un objet de type *DialogInfosClasss* sera instancié pour permettre d'afficher les informations des différentes classes composant le réseau simulé.

```
void chrono_refresh() ;
```

Cette méthode va répondre au signal *timeout()* de *timer_chrono*. Elle incrémente d'une seconde *chrono* et *secondes*.

```
public :

FenetreSimulationRF (QWidget * parent = NULL, Reseau * r, Vue * v) ;
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base NULL, un pointeur sur une instance de *Reseau* et un pointeur sur une instance de *Vue*. Les pointeurs *r* et *v* sont des pointeurs afin d'éviter une copie d'objet et l'objet *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la fenêtre de simulation ce qui permet de les lier. Ce constructeur appelle le constructeur de *QWidget* car la classe *FenetreSimulationRF* hérite de *QWidget*. L'attribut *simulation* est instanciée en ayant comme attribut l'adresse du réseau passé en paramètre du constructeur. L'attribut *vue* est instanciée avec l'adresse de la vue passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait.

~FenetreSimulationRF();

Le destructeur désalloue la mémoire allouée dynamiquement par les attributs de la classe.

```
void start_chrono() ;
```

Cette méthode permet d'actionner le chronomètre *chrono* à partir de t = 0 s.

```
void resume() ;
```

Cette méthode permet de reprendre le chronomètre *chrono* à partir du moment où le chronomètre a été mis en pause.

```
void pause() ;
```

Cette méthode permet de mettre en pause le chronomètre *chrono*. La mise en pause du chonomètre arrête la simulation.

```
void afficher_compteur_clients() ;
```

Cette méthode permet d'afficher au-dessus de chaque file le nombre de clients les composant.

```
void executer_simulation(const int nbClientsInitial) ;
```

Cette méthode permet d'exécuter la simulation selon un nombre initial de clients donné en paramètre. Ce paramètre est constant car il ne change pas, il ne doit donc pas être modifié dans la méthode. Cette méthode appellera la méthode *start_chrono* ci-dessus. Une exception de type *invalid_argument* est lancée si l'entier passé en paramètre est invalide.

```
};
```

6.4 Classe FenetreSimulationRO

La classe *FenetreSimulationRO* permet de créer la fenêtre dans laquelle se déroule la simulation d'un réseau ouvert. Elle contient le réseau créé par l'utilisateur ainsi que 2 fenêtres : celle indiquant les caractéristiques du réseau et celle indiquant les événements de l'échéancier.

```
#include <QWidget>
```

Cette bibliothèque contient la classe permettant de créer une fenêtre en Qt. La classe *FenetreSimulationRO* hérite de la classe *QWidget*. La bibliothèque doit donc être incluse.

```
#include "FenetreInfosReseau.hh"
```

Cette bibliothèque est nécessaire pour l'attribut de type *FenetreInfosReseau*.

```
#include "FenetreEcheancier.hh"
```

Cette bibliothèque est nécessaire pour l'attribut de type *FenetreEcheancier*.

```
#include "Vue.hh"
```

Cette bibliothèque doit être incluse pour l'attribut *vue* de type *Vue ** et permet d'afficher une scène graphique représentée par un objet de type *QGraphicsScene*.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant de créer un bouton en Qt. Elle est nécessaire pour les différents boutons contenus dans la fenêtre.

```
#include <QTimer>
```

Cette bibliothèque est utile pour l'attribut de la classe de type *QTimer*. Elle permet de créer un minuteur (relancé toutes les secondes dans notre cas).

```
#include <QTime>
```

Cette bibliothèque est nécessaire pour l'attribut de la classe de type *QTime*. Elle permet de créer un chronomètre pour garder en mémoire le temps qui s'écoule au cours de la simulation.

```
#include "DialogResultatsRO.hh"
```

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogResultatsRO* dans la méthode *Results()*.

```
#include "DialogHistoriqueRO.hh"
```

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogHistoriqueRO* dans la méthode *History()*.

```
#include "DialogInfosClass.hh"
```

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogInfosClass* dans la méthode *Infos_Classes()*.

```
#include "DialogInfosFile.hh"
```

Cette bibliothèque est nécessaire pour instancier un objet de type *DialogInfosFile* dans la méthode *Infos_Queue()*.

```
#include <QInputDialog>
```

Cette bibliothèque permet d'ouvrir des boîtes de dialogue demandant une entrée à l'utilisateur. Elle est utile pour ouvrir une boîte demandant à l'utilisateur d'entrer le nom d'un fichier pour la sauvgearde des performances.

```
#include <stdexcept>
```

Cette bibliothèque gère les exceptions.

```
#include <QMessageBox>
```

Cette bibliothèque sert à créer des objets de la classe *QMessageBox* qui permet d'interagir avec l'utilisateur à l'aide de boîtes de message. A la fin de la simulation, une fenêtre s'affichera pour demander à l'utilisateur s'il souhaite sauvegarder les performances du réseau.

```
#include "lecture_ecriture.hh"
```

Cette bibliothèque est nécessaire pour accéder aux fonctions de lecture et d'écriture dans un fichier. Ces fonctions sont utilisées pour la sauvegarde des performances dans un fichier.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Elle va permettre de ranger les boutons de la fenêtre verticalement.

```
Class FenetreSimulationRO : public QWidget
```

La classe *FenetreSimulationRO* hérite de la classe *QWidget* pour pouvoir créer la fenêtre qui est un widget. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe *QWidget*.

Q_OBJECT

Cette macro est nécessaire pour définir de nouveaux slots et signaux.

```
private :

SimulationRO simulation ;
```

Cet attribut est objet de type *SimulationRO*. Cet objet permet d'effectuer une simulation sur un réseau ouvert. Il est nécessaire d'avoir cet objet en attribut pour afficher la simulation à l'utilisateur.

```
QVBoxLayout * layout ;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage des boutons composant la fenêtre de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
FenetreInfosReseau fenetre_infos_reseau ;
```

Cet attribut correspond à la fenêtre affichant les caractéristiques du réseau simulé. Cette fenêtre sera affichée dans la fenêtre de simulation, d'où le fait que cet attribut est contenu dans la classe.

```
FenetreEcheancier fenetre_echancier ;
```

Cet attribut correspond à la fenêtre affichant les événements de l'échéancier. Cette fenêtre sera affichée dans la fenêtre de simulation, d'où le fait que cet attribut est contenu dans la classe.

```
Vue * vue ;
```

Cet attribut permet d'afficher la scène graphique récupérée de la fenêtre de création et afficher le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Results ;
```

Cet attribut correspond au bouton pour afficher les résultats de performances du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * History ;
```

Cet attribut correspond au bouton pour afficher l'historique des performances du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Infos_Queue ;
```

Cet attribut correspond au bouton pour afficher les informations sur les files composant le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * Infos_Classes ;
```

Cet attribut correspond au bouton pour afficher les différentes classes composant le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QTime * chrono ;
```

Cet attribut correspond au chronomètre qui garde en mémoire le temps qui s'écoule au cours de la simulation. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QTimer * timer_chrono ;
```

Cet attribut est un minuteur qui se renouvelle sans cesse selon un intervalle de temps défini par l'utilisateur. Ici, c'est toutes les secondes. À chaque fois qu'une seconde s'écoule, l'attribut *chrono* ci-dessus s'incrémente d'une seconde. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
int secondes ;
```

Cet attribut permet de garder en mémoire le nombre de secondes écoulées lors de la simulation, utile pour décrémenter l'attribut *chrono* lorsque l'on veut le remettre à 0 dans la méthode *start_chrono*.

```
public slots :
```

Les slots sont des méthodes pouvant être appelées par des signaux comme le clic sur un bouton.

```
void Results() ;
```

Cette méthode va répondre au signal *clicked()* de *Results*. Un objet de type *DialogResultatsRO* sera instancié pour permettre d'afficher les résultats de performances d'un réseau ouvert.

```
void History() ;
```

Cette méthode va répondre au signal *clicked()* de *History*. Un objet de type *DialogHistoriqueRO* sera instancié pour permettre d'afficher l'historique des performances d'un réseau ouvert.

```
void Infos_Queue() ;
```

Cette méthode va répondre au signal *clicked()* de *Infos_Queue*. Elle attend de nouveau un signal *clicked()* pour récupérer les coordonnées de la file cliquée. Un objet de type *DialogInfosFile* sera instancié permettant d'afficher les caractéristiques correspondantes à la file dont l'utilisateur souhaite connaître les informations.

```
void Infos_Classes() ;
```

Cette méthode va répondre au signal *clicked()* de *Infos_Classes*. Un objet de type *DialogInfosClasss* sera instancié pour permettre d'afficher les informations des différentes classes composant le réseau simulé.

```
void chrono_refresh() ;
```

Cette méthode va répondre au signal *timeout()* de *timer_chrono*. Elle incrémente d'une seconde *chrono* et *secondes*.

```
public :
```

```
FenetreSimulationRO (QWidget * parent = NULL, Reseau * r, Vue * v) ;
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base NULL, un pointeur sur une instance de *Reseau* et un pointeur sur une instance de *Vue*. Les pointeurs *r* et *v* sont des pointeurs afin d'éviter une copie d'objet et l'objet *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la fenêtre de simulation ce qui permet de les lier. Ce constructeur appelle le constructeur de *QWidget* car la classe *FenetreSimulationRO* hérite de *QWidget*. L'attribut *simulation* est instanciée en ayant comme attribut l'adresse du réseau passé en paramètre du constructeur. L'attribut *vue* est instanciée avec l'adresse de la vue passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait.

~FenetreSimulationRO();

Le destructeur désalloue la mémoire allouée dynamiquement par les attributs de la classe.

```
void start_chrono() ;
```

Cette méthode permet d'actionner le chronomètre *chrono* à partir de t = 0 s.

```
void resume() ;
```

Cette méthode permet de reprendre le chronomètre *chrono* à partir du moment où le chronomètre a été mis en pause. La reprise du chonomètre relance la simulation.

```
void pause() ;
```

Cette méthode permet de mettre en pause le chronomètre *chrono*. La mise en pause du chronomètre stoppe la simulation.

```
void afficher_compteur_clients() ;
```

Cette méthode permet d'afficher au-dessus de chaque file le nombre de clients les composant.

```
void executer_simulation(const int nbClientsInitial) ;
```

Cette méthode permet d'exécuter la simulation selon un nombre initial de clients donné en paramètre. Ce paramètre est constant car il ne change pas, il ne doit donc pas être modifié dans la méthode. Cette méthode appellera la méthode *start_chrono* ci-dessus. Une exception de type *invalid_argument* est lancée si l'entier passé en paramètre est invalide.

6.5 Classe DialogConfigFile

La classe *DialogConfigFile* permet de créer des boîtes de dialogue permettant la configuration d'une file. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Configure Queue*.

```
#include <QDialog>[28]
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. Elle est utile car la classe *DialogConfigFile* hérite de *QDialog*.

```
#include <QSpinBox>[26]
```

Cette bibliothèque contient la classe permettant de présenter un ensemble de valeurs entières à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différentes valeurs qui peuvent être sélectionnées.

```
#include <QComboBox>[27]
```

Cette bibliothèque contient la classe permettant de présenter une liste d'options à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différents éléments qui peuvent être sélectionnés.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. Une boîte de dialogue de la classe *DialogConfigFile* a un bouton OK pour fermer la boîte de dialogue.

```
#include <QLabel>[19]
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt. Un objet de cette classe va permettre d'afficher l'ancienne configuration d'une caractéristique de la file.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de *QLabel* crées pour afficher les caractéristiques de la file.

```
#include "File.hh"
```

Cette bibliothèque contient la classe permettant de donner accès aux caractéristiques de la file.

```
#include <QString>[30]
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de *QLabel* utilise une instance de *QString*.

```
using namepsace std;
```

Cet ajout de l'environnement std permet d'alléger le code car à chaque fois qu'il y a *std::* dans le code, il n'est pas nécessaire de le spécifier.

```
Class DialogConfigFile : public QDialog [32]{
```

La classe DialogConfigFile hérite de la classe QDialog pour pouvoir créer de nouvelles boîtes de dialogue avec les caractéristiques souhaitées. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe QDialog.

```
private :
```

Tous les attributs sont des pointeurs car une maîtrise sur la gestion de la mémoire est souhaitée.

```
File * file_a_config;
```

Cet attribut est un pointeur sur un objet de la classe *File*. Il est utile d'avoir accès à cette objet pour afficher les caractéristiques de la file. Il s'agit d'un pointeur pour éviter une copie de l'objet.

```
QVBoxLayout * layout;
```

Cet attribut correspond à la boîte permettant d'ordonner verticalement l'affichage des boutons composant la fenêtre.

```
QLabel *LoiArrivee;
```

Cet attribut correspond au texte représentant la distribution d'arrivée de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel *LoiService;
```

Cet attribut correspond au texte représentant la loi de service de la file du réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel *Ordonnancement;
```

Cet attribut correspond au texte représentant la loi d'ordonnancement des clients dans chaque file du réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte.

```
QLabel *TailleMax;
```

Cet attribut correspond au texte représentant la taille maximale de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel *NbServeurs;
```

Cet attribut correspond au texte représentant le nombre de serveurs de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QComboBox *newLoiArrivee;
```

Cet attribut correspond à la boîte proposant les différents choix de distribution d'arrivée. Il est de type *QComboBox* pour avoir un bel affichage des différents choix.

```
QComboBox *newLoiService;
```

Cet attribut correspond à la boîte proposant les différents choix de loi de service. Il est de type *QComboBox* pour avoir un bel affichage des différents choix.

```
QComboBox *newOrdonnancement;
```

Cet attribut correspond à la boîte proposant les différents choix d'ordonnancement. Il est de type *QComboBox* pour avoir un bel affichage des différents choix.

```
QSpinBox *newTailleMax;
```

Cet attribut correspond à la boîte proposant un ensemble de valeurs entières à l'utilisateur, ensemble correspondant aux tailles possibles d'une file. Il est de type *QSpinBox* pour avoir un bel affichage des différents choix.

```
QSpinBox *newNbServeurs;
```

Cet attribut correspond à la boîte proposant un ensemble de valeurs entières à l'utilisateur, ensemble correspondant aux nombres possbiles de serveurs pour une file. Il est de type *QSpinBox* pour avoir un bel affichage des différents choix.

```
QPushButton *ok;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue.

```
QPushButton *annuler;
```

Cet attribut correspond au bouton d'annulation qui empêche la prise en compte des changements donnés par l'utilisateur.

```
public :
```

```
DialogConfigFile(QWidget * parent=NULL, File * f);
```

Ce constructeur prend en paramètre un pointeur sur un objet QWidget avec comme valeur de base NULL et un pointeur sur l'objet de type *File*. L'objet de type *File* est passé en pointeur pour intialiser le pointeur *file_a_config* en attribut de la classe. Le pointeur sur l'objet de type *QWidget* permet de récupérer l'adresse du parent d'une fenêtre de cette classe. Il est initialisé de base à NULL si la fenêtre ne possède pas de parent. Ce constructeur appelle le constructeur de la classe *QDialog* vu que la classe *DialogConfigFile* en hérite. Pour les autres pointeurs en attribut, leur constructeur est appelé.

```
~DialogConfigFile()
```

De la mémoire est allouée dynamiquement pour les pointeurs en attribut de la classe (excepté pour l'attribut *file_a_config*). Elle doit donc être libérée.

```
string recuperer_valeur_newLoiArrivee();
```

Cette méthode récupère la nouvelle loi d'arrivée insérée par l'utilisateur et la renvoie sous forme d'un string.

```
string recuperer_valeur_newLoiService();
```

Cette méthode récupère la nouvelle loi de service insérée par l'utilisateur et la renvoie sous forme d'un string.

```
string recuperer_valeur_newOrdonnancement();
```

Cette méthode récupère le nouvel ordonnancement inséré par l'utilisateur et le renvoie sous forme d'un string

```
int recuperer_valeur_newTailleMax();
```

Cette méthode récupére la nouvelle taille maximale de la file insérée par l'utilisateur et la renvoie sous forme d'un entier.

```
int recuperer_valeur_newNbServeurs();
```

Cette méthode récupére le nouveau nombre de serveurs inséré par l'utilisateur et le renvoie sous forme d'un entier.

6.6 Classe DialogConfigClass

La classe *DialogConfigClass* permet de créer des boîtes de dialogue permettant la configuration d'une classe après qu'elle ait été créée. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Configure class* dans la fenêtre de la simulation.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe *DialogConfigClass* hérite de la classe *QDialog*. La bibliothèque doit donc être incluse.

```
#include <QSpinBox>
```

Cette bibliothèque contient la classe permettant de présenter un ensemble de valeurs entières à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différentes valeurs qui peuvent être sélectionnées.

```
#include <QComboBox>
```

Cette bibliothèque contient la classe permettant de présenter une liste d'options à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différents éléments qui peuvent être sélectionnés.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant de créer un bouton en Qt. Elle est nécessaire pour les différents boutons contenus dans la fenêtre.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'ajouter du texte sur une fenêtre et de le mettre en forme Qt. Un objet de cette classe va permettre d'afficher une caractéristique de la classe.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt.

```
#include <QLineEdit>[29]
```

Cette bibliothèque contient la classe permettant de modifier une ligne en Qt. L'utilisateur va pouvoir entrer et modifier une seule ligne de texte brut dans la boîte de dialogue.

```
class DialogConfigClass: public QDialog{

private :

ClasseClient * class;
```

Cet attribut correspond à une pointeur sur un objet de type *ClasseClient*. Cet objet correspond à une classe de clients. L'attribut est nécessaire pour pouvoir accéder facilement aux caractéristiques d'une classe de clients pour les afficher. Elle est passée sous forme de pointeur pour éviter de copier l'objet.

```
QVBoxLayout * layout;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * configRoutageLabel;
```

Cet attribut correspond au texte représentant le routage actuel de la classe contenue dans le pointeur *class*. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * prioriteLabel;
```

Cet attribut correspond au texte représentant la priorité actuelle de la classe contenue dans le pointeur *class*. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLineEdit * configRoutage;
```

Cet attribut correspond à la ligne où l'utilisateur pourra rentrer le routage qu'il souhaite. Il est de type *QLineEdit* afin de pouvoir récupérer une entrée de l'utilisateur. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QSpinBox* priorite;
```

Cet attribut est la boîte permettant de gérer les entiers qui correspondent aux priorités. Il est de type *QSpinBox* pour proposer à l'utilisateur une boîte permettant d'augmenter ou de réduire la valeur entière s'y trouvant . Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * BoutonOk;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue une fois les modifications terminées. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * BoutonAnnuler;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue sans prendre en compte les possibles modifications qui ont été faites. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public:

DialogConfigClass(QWidget * parent = NULL, ClasseClient * cc);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur un objet de type *ClasseClient*. L'objet de type *ClasseClient* est passé en pointeur afin d'éviter une copie de l'objet et l'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogConfigClass* hérite de *QDialog*. L'attribut *class* est initialisé à l'adresse de la structure *ClasseClient* passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait.

```
int getPriorite();
```

Cette méthode renvoie la valeur de la priorité récupérée dans l'objet *priorite* sous forme de *int*.

```
string getRoutage();
```

Cette méthode renvoie sous forme de *string* le routage récupérée dans l'objet *configRoutage*.

```
};
```

6.7 Classe DialogAddClass

La classe *DialogAddClass* permet de créer des boîtes de dialogue permettant la configuration d'une classe au moment où elle est ajoutée. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Add Class* dans la fenêtre de la simulation.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe *DialogAddClass* hérite de la classe *QDialog*. La bibliothèque doit donc être incluse.

```
#include <QSpinBox>
```

Cette bibliothèque contient la classe permettant de présenter un ensemble de valeurs entières à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différentes valeurs qui peuvent être sélectionnées.

```
#include <QComboBox>
```

Cette bibliothèque contient la classe permettant de présenter une liste d'options à l'utilisateur en Qt. Elle permet d'avoir un bel affichage des différents éléments qui peuvent être sélectionnés

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant de créer un bouton en Qt. Elle est nécessaire pour les différents boutons contenus dans la fenêtre.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'ajouter du texte sur une fenêtre et de le mettre en forme Qt. Un objet de cette classe va être utilisé pour afficher le nom d'un champ à remplir.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt.

```
#include <QLineEdit>
```

Cette bibliothèque contient la classe permettant de modifier une ligne en Qt. L'utilisateur va pouvoir entrer et modifier une seule ligne de texte brut dans la boîte de dialogue.

```
class DialogAddClass: public QDialog{

private :

QVBoxLayout * layout;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur afin de pouvoir gérer la mémoire.

```
QLabel * titreRoutage;
```

Cet attribut correspond au texte représentant le nom du champ à remplir. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * titrePriorite;
```

Cet attribut correspond au texte représentant le nom du champ à remplir. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLineEdit * configRoutage;
```

Cet attribut correspond à la ligne où l'utilisateur pourra rentrer le routage qu'il souhaite. Il est de type *QLineEdit* afin de pouvoir récupérer une entrée de l'utilisateur. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QSpinBox* priorite;
```

Cet attribut est la boîte permettant de gérer les entiers qui correspondent aux priorités. Il est de type *QSpinBox* afin de proposer à l'utilisateur une boîte permettant d'augmenter ou de réduire la valeur entière s'y trouvant. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * BoutonOk;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue, elle vérifie si les champs sont bien remplis. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public :

DialogAddClass(QWidget * parent = NULL);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL*. L'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogAddClass* hérite de *QDialog*.Pour les pointeurs, un appel de leur constructeur sera fait.

~DialogAddClass(QWidget * parent = NULL);
Le destructeur désalloue la mémoire allouée dynamiquement pour les attributs de la classe.

```
int getPriorite();
```

Cette méthode renvoie la valeur de la valeur récupérée dans l'objet *priorite* sous forme de int.

```
string getRoutage();
```

Cette méthode renvoie sous forme de *string* le routage récupéré de l'objet *configRoutage*.

```
};
```

6.8 Classe FenetreInfosReseau

```
#include <QWidget>
```

Cette bibliothèque contient la classe QWidget. La classe *FenetreInfosReseau* hérite de la classe *QWidget*.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe *QVboxLayout* et permet de placer des widgets verticalement.

```
#include <QLabel>
```

Cette bibliothèque permet l'utilisation d'objet de la classe *QLabel* afin d'afficher des éléments textuels.

```
#include <QString>
```

Cette bibliothèque contient la classe *QString* et permet de les utiliser.

```
#include "Reseau.hh"
```

Cette bibliothèque est nécessaire car il possède un attribut de la classe *Reseau*.

```
class FenetreInfosReseau : public QWidget {
```

La classe FenetreInfosReseau hérite de la classe *QWidget* de façon publique pour garder l'encapsulation des membres de la classe mère.

Q_OBJECT

Cette macro est nécessaire pour que la classe déclare ses propres signaux et slots.

```
private :

Reseau * reseau;
```

Cet attribut est un pointeur sur un objet de type *Reseau*. Il s'agit d'un pointeur pour accéder directement à l'objet et à ses modifications tout au long de la simulation.

```
QVBoxLayout * rangement;
```

Cet attribut est un pointeur sur un objet *QVBoxLayout*. Un objet de ce type permet de ranger des widgets à la verticale. Il va permettre de ranger l'affichage des informations du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nbClientsActuel;
```

Cet attribut est un pointeur sur un objet *QLabel*. Il permet un affichage textuel du nombre de clients actuel dans le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nbClientsPasses;
```

Cet attribut est un pointeur sur un objet *QLabel*. Il permet l'affichage textuel du nombre de clients sortis du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nbClasses;
```

Cet attribut est un pointeur sur un objet *QLabel*. Il permet l'affichage textuel du nombre de classes de clients dans le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nbClientsClasse;
```

Cet attribut est un pointeur sur un objet *QLabel*. Il permet l'affichage textuel du nombres de clients par classe dans le réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * typeReseau;
```

Cet attribut est un pointeur sur un objet *QLabel*. Il permet l'affichage textuel du type du réseau. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public :

FenetreInfosReseau(Reseau * r, QWidget * parent=NULL);
```

Ce constructeur prend en paramètre un pointeur sur un objet QWidget avec comme valeur de base NULL et un pointeur sur l'objet Reseau. L'objet de type *Reseau* est passé en pointeur pour intialiser le pointeur *reseau* en attribut de la classe. Le pointeur sur l'objet de type *QWidget* permet de récupérer l'adresse du parent d'une fenêtre de cette classe. Il est initialisé de base à NULL si la fenêtre ne possède pas de parent. Ce constructeur appelle le constructeur de la classe *QWidget* vu que la classe *FenetreInfosReseau* en hérite. Pour tous les autres pointeurs en attribut, leur constructeur est appelé.

~FenetreInfosReseau();

La mémoire est allouée de manière dynamique pour tous les pointeurs excepté le pointeur *reseau*. Le destructeur va libérer cette mémoire.

```
void setInfoNbClientsActuel();
```

Cette méthode récupère le nombre de clients actuellement dans le réseau, grâce à la méthode *getNbClients()* appliquée sur l'objet pointée par le pointeur *reseau*. La valeur est transformée en string pour pouvoir l'afficher.

```
void setInfoNbClientsPasses();
```

Cette méthode récupère le nombre de clients passés dans le réseau grâce à la méthode *getNbClientsPasses()* appliquée sur l'objet pointée par le pointeur *reseau*. La valeur est transformée en string pour pouvoir l'afficher.

```
void setInfoNbClasses();
```

Cette méthode récupère le nombre de classes de clients du réseau grâce à la méthode *getNbClasses()* appliquée sur l'objet pointée par le pointeur *reseau*. La valeur est transformée en string pour pouvoir l'afficher.

```
void setInfoNbClientsClasse();
```

Cette méthode récupère le nombre de client par classe grâce à l'attribut *reseau* de la classe. Les valeurs sont transformées en string pour pouvoir les afficher.

```
void setInfoTypeReseau();
```

Cette méthode récupère le résultat correspondant au type du réseau grâce à la méthode *getTypeReseau()* et le transforme en string.

6.9 Classe DialogInfosQueue

La classe *DialogInfosQueue* permet de créer des boîtes de dialogue affichant les informations d'une file. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Infos Queue* dans la fenêtre de simulation.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe hérite de *QDialog*.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt. Un objet de cette classe va afficher une caractéristique de la file passée en attribut.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de *QLabel* crées pour afficher les caractéristiques.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. Une boîte de dialogue de la classe *DialogInfosQueue* a un bouton OK pour fermer la boîte de dialogue.

```
#include <QString>
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de QLabel utilise une instance de *QString*.

```
#include "File.hh"
```

Cette bibliothèque contient la classe permettant de donner accès aux caractéristiques de la file.

```
Class DialogInfosQueue : public QDialog {
```

La classe *DialogInfosQueue* hérite de la classe *QDialog* pour pouvoir créer de nouvelles boîtes de dialogue avec les informations souhaitées. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe QDialog.

Q_OBJECT

Cette macro est nécessaire pour définir de nouveaux slots et signaux.

```
private:
```

Les attributs sont des pointeurs afin d'avoir une maîtrise sur la mémoire.

```
File * file;
```

Cet attribut est un pointeur sur un objet de la classe *File*. Il est nécessaire d'avoir un objet de ce type en attribut pour accéder aux informations qu'il contient. L'objet est passé sous forme de pointeur pour éviter une copie de l'objet.

```
QLabel * ordonnancement;
```

Cet attribut correspond au texte représentant l'ordonnancement de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * loiService;
```

Cet attribut correspond au texte représentant la loi de service de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nbClients;
```

Cet attribut correspond au texte représentant le nombre de clients dans la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nbClientsServis;
```

Cet attribut correspond au texte représentant le nombre de clients servis de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nbClientsPresentes;
```

Cet attribut correspond au texte représentant le nombre de clients présentes à la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nbClientsPerdus;
```

Cet attribut correspond au texte représentant le nombre de clients perdus de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nbServeurs;
```

Cet attribut correspond au texte représentant le nombre de serveurs de la file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QVBoxLayout * layout;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage des boutons composant la fenêtre de façon verticalement. Elle contiendra les *QLabel*.

```
QPushButton * ok;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue.

```
QPushButton * Infos_Clients;
```

Cet attribut correspond au bouton *Infos Clients* qui ouvre une boîte de dialogue donnant des informations sur les clients de la file.

```
public slots :
```

```
void informations_clients();
```

Cette méthode est un slot s'activant à l'enclenchement du bouton *Infos Clients*. Elle ouvre une boîte de dialogue affichant des informations sur les clients présents dans la file.

```
public :

DialogInfosQueue(Widget * parent = NULL, File * f);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur un objet de type *File* qui permet d'initialiser l'attribut *file*. L'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogInfosQueue* hérite de *QDialog*. Pour les autres pointeurs, un appel de leur constructeur sera fait.

```
~DialogInfosQueue();
```

De la mémoire ets allouée dynamiquement pour tous les pointeurs de la classe sauf *file*. Le destructeur doit libérer cette mémoire.

```
void setInfoOrdonnancement();
```

Cette méthode récupère la loi d'ordonnancement de la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *ordonnancement*.

```
void setInfoLoiService();
```

Cette méthode récupère la loi de service de la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *loiService*.

```
void setInfoNbClients();
```

Cette méthode récupère le nombre de clients dans la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *nbClients*.

```
void setInfoNbClientsServis();
```

Cette méthode récupère le nombre de clients servis de la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *nbClientsServis*.

```
void setInfoNbClientsPresentes();
```

Cette méthode récupère le nombre de clients présentés à la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *nbClientsPresentes*.

```
void setInfoNbClientsPerdus();
```

Cette méthode récupère le nombre de clients perdus de la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *nbClientsPerdus*.

```
void setNbServeurs();
```

Cette méthode récupère du nombre de serveurs de la file dans l'objet pointé par *file* et le transforme en string avant de le mettre dans l'attribut *nbServeurs*.

6.10 Classe DialogInfosClass

La classe *DialogInfosClass* permet de créer des boîtes de dialogue affichant les caractéristiques d'une classe. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Infos Class* dans la fenêtre de simulation.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe *DialogInfosClass* hérite de la classe QDialog. La bibliothèque doit donc être incluse.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt. Une boîte de dialogue de la classe *DialogInfosClass* devant afficher les caractéristiques d'une classe, cette bibliothèque sera utile.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de QLabel créées pour afficher les résultats.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. Une boîte de dialogue de la classe *DialogInfosClass* a un bouton OK pour fermer la boîte de dialogue.

```
#include <QString>
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de *QLabel* utilise une instance de *QString*.

```
#include "ClasseClient.hh"
```

Cette bibliothèque contient la classe permettant de créer une instance de type *ClasseClient*. Cet élément est utile pour afficher ses caractéristiques.

```
Class DialogInfosClass : public QDialog
```

La classe DialogInfosClass hérite de la classe *QDialog* pour pouvoir créer de nouvelles boîtes de dialogue avec les caractéristiques souhaitées. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe *QDialog*.

Q_OBJECT

Cette macro est nécessaire pour déclarer des nouveaux signaux et slots.

```
private :
ClasseClient * classe_client;
```

Cet attribut correspond à un pointeur sur un objet *ClasseClient*. Cet objet représente une classe du réseau. Elle est passée sous forme de pointeur pour éviter une copie inutile d'objet.

```
QLabel * priorite;
```

Cet attribut correspond au texte représentant la priorité de la classe . Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * routage;
```

Cet attribut correspond au texte représentant le routage. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * layout
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage des widgets dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée

```
QPushButton * ok;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée

```
public :
DialogInfosClass(QWidget * parent = NULL, ClasseClient * cc);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur un objet de type *ClasseClient* qui permet d'initialiser l'attribut *classe_client*. L'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogInfosClass* hérite de *QDialog*. Pour tous les autres pointeurs, un appel de leur constructeur sera fait.

~ DialogInfosClass();

Le destructeur désalloue la mémoire allouée dynamiquement pour les attributs de la classe.

```
void setInfoPriorite()
```

Cette méthode récupère le résultat correspondant à la priorité de la classe dans la structure de type *ClasseClient* et le transforme en string avant de l'ajouter à l'attribut *priorite*.

```
void seInfoRoutage()
```

Cette méthode récupère le résultat correspondant au routage de la classe dans la structure de type *ClasseClient* et le transforme en string avant de l'ajouter à l'attribut *routage*.

6.11 Classe DialogHistoriqueRF

```
#include "HistoriqueRF.hh";
```

Ce header est nécessaire pour l'attribut *histo*.

```
#include <QDialog>;
```

Cette classe hérite de *QDialog*, il faut donc inclure la bibliothèque associée.

```
#include <QLabel>;
```

Ce header permet l'utilisation de *QLabel* permettant d'ajouter des éléménts textuels sur une fenêtre.

```
#include <QVBoxLayout>;
```

Ce header sert à pouvoir utiliser des objets de la classe *QVBoxLayout* permettant de placer verticalement des objets de la classe *QWidget*.

```
#include <QPushButton>;
```

Ce header sert à pouvoir créer des boutons dans la fenêtre.

```
#include <QScrollArea>;[31]
```

Cette bibliothèque contient la classe permettant de définir un périmètre qui aura une barre de défilement si les éléments affichés dans la fenêtre venaient à dépasser la taille de la fenêtre.

```
#include <QString>;
```

Ce header sert à pouvoir utiliser des *QString* qui vont être affectés à des *QLabel*.

```
Class DialogHistoriqueRF : public QDialog
```

La classe *DialogHistoriqueRF* hérite de *QDialog* afin de pouvoir faire une boîte de dialogue affichant les différentes mesures de performances à une date. La classe *QDialog* possède des méthodes telles que *open()* ou bien *exec()* qui permettent d'ouvrir la boîte de dialogue. Un héritage *public* est effectué pour utiliser les méthodes en *public*.

Q_OBJECT

Cette macro est nécessaire pour définir de nouveaux slots et signaux.

```
private :
```

Les attributs sont des pointeurs car on souhaite gérer la mémoire.

```
HistoriqueRF * histo;
```

Cet attribut est un objet de la classe *HistoriqueRF** qui correspond à la liste des mesures de performances. Elle est utilisée pour afficher les performances pendant la simulation.

```
int position_historique;
```

Cet attribut permet de connaître la position dans la liste des perfomances de l'attribut *histo*.

```
QLabel * tps_rep_moy_reseau;
```

Cet attribut correspond au texte représentant le résultat du temps de réponse moyen dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nb_moy_clients_reseau;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nb_moy_clients_files;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans chaque file du réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_attente_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'attente moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_service_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de service moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_interarrivee_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'inter-arrivée moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * taux_perte_files;
```

Cet attribut correspond au texte représentant le résultat du taux de perte de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * taux_utilisation_serveurs;
```

Cet attribut correspond au texte représentant le résultat du taux d'utilisation de chaque serveur de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * date;
```

Cet attribut sert à stocker un *QString* correspondant aux performances à la position *position_historique*.

```
QVBoxLayout * layout_affichage;
```

Cet attribut sert à ordonner verticalement des objets héritant de *QWidget*. Il va servir pour les attributs qui sont des *QLabel* et des *QPushButton*.

```
QVBoxLayout * rangement_scroll;
```

Cet attribut correspond à la boîte verticale qui contiendra l'objet de type *ScrollArea*. Il est nécessaire que cet objet soit stocké dans une boîte afin de pouvoir être affiché sur la boîte de dialogue.

```
QScrollArea * scrollArea;
```

Cet attribut permet de délimiter l'environnement qui possèdera une barre de défilement. L'objet *scrollArea* contiendra la fenêtre *fenetre_affichage*.

```
QWidget * fenetre_affichage;
```

Cet attribut correspond à la fenêtre qui contiendra la boîte de rangement *layout_affichage*.

```
QPushButton * ok;
```

Ce bouton sert à envoyer un signal *accepted* pour fermer la fenêtre.

```
QPushButton * suivant;
```

Cet attribut est un bouton permettant d'appeler la méthode *perf_suivante()*.

```
QPushButton * precedent;
```

Cet attribut est un bouton permettant d'appeler la méthode *perf_precedente()*.

```
public slots:
```

Les slots sont des méthodes pouvant être appelées par des signaux comme le clic sur un bouton.

```
void perf_suivante();
```

Cette méthode va répondre au signal *clicked()* de *suivant*. Les méthodes *set* vont être appelées pour mettre à jour les *QLabel* avec les performances de la date suivante et ensuite elles seront affichées.

```
void perf_precedent();
```

Cette méthode va répondre au signal *clicked()* de *precedent*. Les méthodes *set* vont être appelées pour mettre à jour les *QLabel* avec les performances de la date précédente et ensuite elles seront affichées.

```
public :

DialogHistoriqueRF(QWidget * parent = NULL, HistoriqueRF * h);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur un objet de type *HistoriqueRF*. L'objet est passé en pointeur afin d'éviter une copie et l'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogResultatsRF* hérite de *QDialog*. L'attribut *histo* est intialisé avec l'adresse de l'objet de type *HistoriqueRF* passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait et pour l'entier *position_historique*, il est initialisé à 0.

~DialogHistoriqueRF();

Le destructeur va désallouer les attributs qui ont été alloués dynamiquement dans le constructeur.

```
void setInfoTpsRepMoyReseau();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps de réponse moyen dans le réseau et va l'attribuer, sous forme de *string*, à *tps_rep_moy_reseau*.

```
void setInfoNbMoyClientsReseau();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au nombre moyen de clients dans le réseau et va l'attribuer, sous forme de *string*, à *nb_moy_clients_reseau*.

```
void seInfotNbMoyClientsFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au nombre moyen de clients dans chaque file et va l'attribuer, sous forme de *string*, à *nb_moy_clients_files*.

```
void setInfoTpsAttenteMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'attente moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_attente_moy_files*.

```
void setInfoTpsServiceMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps de service moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_service_moy_files*.

```
void setInfoTpsinterarriveeMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'interarrivée moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_interarrivee_moy_files*.

```
void setInfoTauxPerteFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au taux de perte de chaque file et va l'attribuer, sous forme de *string*, à *taux_perte_files*.

```
void setInfoTauxUtilisationServeurs();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'utilisation moyen pour chaque serveur de chaque file et va l'attribuer, sous forme de *string*, à *taux_utilisation_serveurs*.

```
};
```

6.12 Classe DialogHistoriqueRO

```
#include "HistoriqueRO.hh";
```

Ce header est nécessaire pour l'attribut *histo*.

```
#include <QDialog>;
```

Cette classe hérite de *QDialog*, il faut donc inclure la bibliothèque associée.

```
#include <QLabel>;
```

Ce header permet l'utilisation de *QLabel* permettant d'ajouter des éléménts textuels sur une fenêtre.

```
#include <QVBoxLayout>;
```

Ce header sert à pouvoir utiliser des objets de la classe *QVBoxLayout* permettant de placer verticalement des objets de la classe *QWidget*.

```
#include <QPushButton>;
```

Ce header sert à pouvoir créer des boutons dans la fenêtre.

```
#include <QScrollArea>;
```

Cette bibliothèque contient la classe permettant de définir un périmètre qui aura une barre de défilement si les éléments affichés dans la fenêtre venaient à dépasser la taille de la fenêtre.

```
#include <QString>;
```

Ce header sert à pouvoir utiliser des *QString* qui vont être affectés à des *QLabel*.

```
Class DialogHistoriqueRO : public QDialog
```

DialogHistoriqueRO hérite de *QDialog* afin de pouvoir faire une boîte de dialogue affichant les différentes mesures. La classe *QDialog* possède des méthodes telles que *open()* ou bien *exec()* qui permet d'ouvrir la boîte de dialogue en bloquant les autres fenêtres. On fait un héritage *public* pour utiliser les méthodes en *public*.

Q_OBJECT

Cette macro est nécessaire pour définir de nouveaux slots.

```
private :
```

Les attributs sont des pointeurs car on souhaite gérer la mémoire.

```
HistoriqueRO * histo;
```

Cet attribut est un objet de la classe *HistoriqueRO** qui correspond à la liste des mesures de performances. Elle est utilisée pour afficher les performances pendant la simulation.

```
int position_historique;
```

Cet attribut permet de connaître la position dans la liste des perfomances de l'attribut *histo*.

```
QLabel * tps_rep_moy_reseau;
```

Cet attribut correspond au texte représentant le résultat du temps de réponse moyen dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nb_moy_clients_reseau;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * nb_moy_clients_files;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans chaque file du réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_attente_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'attente moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_service_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de service moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_interarrivee_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'inter-arrivee moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * taux_perte_files;
```

Cet attribut correspond au texte représentant le résultat du taux de perte de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * taux_utilisation_serveurs;
```

Cet attribut correspond au texte représentant le résultat du taux d'utilisation de chaque serveur de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * tps_sejour_moy_reseau;
```

Cet attribut correspond au texte représentant le résultat du temps de séjour moyen des clients dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * debit_entree_reseau;
```

Cet attribut correspond au texte représentant le résultat du débit d'entrée des clients dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * debit_sortie_reseau;
```

Cet attribut correspond au texte représentant le résultat du débit de sortie des clients dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte.

```
QLabel * date;
```

Cet attribut sert à stocker un *QString* correspondant aux performances à la position *position_historique*.

```
QVBoxLayout * layout_affichage;
```

Cet attribut sert à ordonner verticalement des objets héritant de *QWidget*. Il va servir pour les attributs qui sont des *QLabel* et des *QPushButton*.

```
QVBoxLayout * rangement_scroll;
```

Cet attribut correspond à la boîte verticale qui contiendra l'objet de type *ScrollArea*. Il est nécessaire que cet objet soit stocké dans une boîte afin de pouvoir être affiché sur la boîte de dialogue.

```
QScrollArea * scrollArea;
```

Cet attribut permet de délimiter l'environnement qui possèdera une barre de défilement. L'objet *scrollArea* contiendra la fenêtre *fenetre_affichage*.

```
QWidget * fenetre_affichage;
```

Cet attribut correspond à la fenêtre qui contiendra la boîte de rangement *layout_affichage*.

```
QPushButton * ok;
```

Ce bouton sert à envoyer un signal *accepted* pour fermer la fenêtre.

```
QPushButton * suivant;
```

Cet attribut est un bouton permettant d'appeler la méthode *perf_suivante()*.

```
QPushButton * precedent;
```

Cet attribut est un bouton permettant d'appeler la méthode *perf_precedente()*.

```
public slots:
```

Les slots sont des méthodes pouvant être appelées par des signaux comme le clic sur un bouton.

```
void perf_suivante();
```

Cette méthode va répondre au signal *clicked()* de *suivant*. Les méthodes *set* vont être appelés pour mettre à jour les *QLabel* avec les performances de la date suivante et ensuite elles seront affichées.

```
void perf_precedent();
```

Cette méthode va répondre au signal *clicked()* de *precedent*. Les méthodes *set* vont être appelés pour mettre à jour les *QLabel* avec les performances de la date précédente et ensuite elles seront affichées.

```
public :
DialogHistoriqueRO(QWidget * parent = NULL, HistoriqueRO * h);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur un objet de type *HistoriqueRO*. L'objet est passé en pointeur afin d'éviter une copie et l'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogResultatsRO* hérite de *QDialog*. L'attribut *histo* est intialisé avec l'adresse de l'objet de type *HistoriqueRO* passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait et pour l'entier *position_historique*, il est initialisé à 0.

~DialogHistoriqueRO();

Le destructeur va désallouer les attributs.

```
void setInfoTpsRepMoyReseau();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps de réponse moyen dans le réseau et va l'attribuer, sous forme de *string*, à *tps_rep_moy_reseau*.

```
void setInfoNbMoyClientsReseau();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au nombre moyen de clients dans le réseau et va l'attribuer, sous forme de *string*, à *nb_moy_clients_reseau*.

```
void seInfotNbMoyClientsFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au nombre moyen de clients dans chaque file et va l'attribuer, sous forme de *string*, à *nb_moy_clients_files*.

```
void setInfoTpsAttenteMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'attente moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_attente_moy_files*.

```
void setInfoTpsServiceMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps de service moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_service_moy_files*.

```
void setInfoTpsinterarriveeMoyFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'interarrivée moyen de chaque file et va l'attribuer, sous forme de *string*, à *tps_interarrivee_moy_files*.

```
void setInfoTauxPerteFiles();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au taux de pertes de chaque file et va l'attribuer, sous forme de *string*, à *taux_perte_files*.

```
void setInfoTauxUtilisationServeurs();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps d'utilisation moyen pour chaque serveur de chaque file et va l'attribuer, sous forme de *string*, à *tps_utilisation_serveurs*.

```
void setInfoTpsSejourMoyReseau();
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au temps de sejour moyen dans le réseau et va l'attribuer, sous forme de *string*, à *tps_sejour_moy_reseau*.

```
void setInfoDebitEntreeReseau()
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au débit moyen d'entrée dans le réseau et va l'attribuer, sous forme de *string*, à *debit_entree_reseau*.

```
void setInfoDebitSortieReseau()
```

Cette méthode va récupérer la valeur,dans *histo* qui correspond au débit moyen de sortie dans le réseau et va l'attribuer, sous forme de *string*, à *debit_sortie_reseau*.

```
};
```

6.13 Classe DialogResultatsRF

La classe *DialogResultatsRF* permet de créer des boîtes de dialogue affichant les résultats des performances d'un réseau fermé. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Results* dans la fenêtre de simulation.

```
#include "HistoriqueRF.hh"
```

Cette bibliothèque contient la classe permettant de créer un historique des résultats de performances demandés tout au long de la simulation pour un réseau fermé. L'historique est représenté par une liste dont les éléments représentent une demande de calcul des performances. Cet élément est utile pour afficher les résultats de performances.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe *DialogResultatsRF* hérite de la classe *QDialog*. La bibliothèque doit donc être incluse.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt. Une boîte de dialogue de la classe *DialogResultatsRF* doit afficher les résultats de performances, cette bibliothèque sera donc utile.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner verticalement les widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de QLabel créées pour afficher les résultats.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. Une boîte de dialogue de la classe *DialogResultatsRF* a un bouton *OK* pour fermer la boîte de dialogue.

```
#include <QScrollArea>
```

Cette bibliothèque contient la classe permettant de définir un périmètre qui aura une barre de défilement si les éléments affichés dans la fenêtre venaient à dépasser la taille de la fenêtre. Une boîte de dialogue de la classe *DialogResultatsRF* devant afficher les résultats du réseau et ceux de ses files, il est possible que leur affichage amène à avoir une barre de défilement.

```
#include <QString>
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de *QLabel* utilise une instance de *QString*.

```
class DialogResultatsRF : public QDialog{
```

La classe *DialogResultatsRF* hérite de la classe *QDialog* pour pouvoir créer de nouvelles boîtes de dialogue avec les caractéristiques souhaitées. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe *QDialog*.

```
private :

PerformancesRF * perfsRF;
```

Cet attribut correspond à la structure contenant les résultats des performances d'un réseau fermé et de ses files. Elle est utile pour pouvoir accéder facilement aux résultats. Elle est passée sous forme de pointeur pour éviter de copier la structure.

```
QLabel * nb_moy_clients_reseau;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans le réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_rep_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de réponse moyen pour chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nb_moy_clients_files;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans chaque file du réseau. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_attente_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'attente moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_service_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de service moyen dans chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_interarrivee_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'inter-arrivee moyen dans chaque file. Il est de type*QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * taux_perte_files;
```

Cet attribut correspond au texte représentant le résultat du taux de perte de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * taux_utilisation_serveurs;
```

Cet attribut correspond au texte représentant le résultat du taux d'utilisation de chaque serveur de chaque file. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * layout_affichage;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * rangement_scroll;
```

Cet attribut correspond à la boîte verticale qui contiendra l'objet de type *ScrollArea*. Il est nécessaire que cet objet soit stocké dans une boîte afin de pouvoir être afficher sur la boîte de dialogue. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QScrollArea * scrollArea;
```

Cet attribut permet de délimiter l'environnement qui possèdera une barre de défilement. L'objet *scrollArea* contiendra la fenêtre *fenetre_affichage*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QWidget * fenetre_affichage;
```

Cet attribut correspond à la fenêtre qui contiendra la boîte de rangement *layout_affichage*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * ok;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public :
```

```
DialogResultatsRF(QWidget * parent = NULL, PerformancesRF * perfs);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur la structure de type *PerformancesRF*. La structure est passée en pointeur afin d'éviter une copie de la structure et l'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogResultatsRF* hérite de *QDialog*. L'attribut *perfsRF* est intialisé avec l'adresse de la structure *PerformancesRF* passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait.

```
~DialogResultatsRF();
```

Le destructeur désalloue la mémoire allouée dynamiquement par les attributs de la classe.

```
void setInfoTpsRepMoyFiles();
```

Cette méthode récupère le résultat correspondant au temps de réponse moyen des files du réseau dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *tps_rep_moy_reseau*.

```
void setInfoNbMoyClientsReseau();
```

Cette méthode récupère le résultat correspondant au nombre moyen de clients dans le réseau dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *nb_moy_clients_reseau*.

```
void seInfotNbMoyClientsFiles();
```

Cette méthode récupère les résultats correspondant au nombre moyen de clients dans chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *nb_moy_clients_files*.

```
void setInfoTpsAttenteMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps d'attente moyen dans chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *tps_attente_moy_files*.

```
void setInfoTpsServiceMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps de service moyen dans chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *tps_service_moy_files*.

```
void setInfoTpsinterarriveeMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps d'inter-arrivee moyen dans chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *tps_interarrivee_moy_files*.

```
void setInfoTauxPerteFiles();
```

Cette méthode récupère les résultats correspondant au taux de perte de chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *tps_perte_files*.

```
void setInfoTauxUtilisationServeurs();
```

Cette méthode récupère les résultats correspondant au taux d'utilisation de chaque serveur de chaque file dans la structure de type *PerformancesRF* et le transforme en string avant de l'ajouter à l'attribut *taux_utilisation_serveurs*.

```
};
```

6.14 Classe DialogResultatsRO

La classe *DialogResultatsRO* permet de créer des boîtes de dialogue affichant les résultats des performances d'un réseau ouvert. Une boîte de dialogue de cette classe se lance lorsque l'utilisateur clique sur le bouton *Results* dans la fenêtre de simulation.

```
#include "HistoriqueRO.hh"
```

Cette bibliothèque contient la classe permettant de créer un historique des résultats de performances demandées tout au long de la simulation pour un réseau ouvert. L'historique est représentée par une liste dont les éléments représentent une demande de calcul des performances. Cet élément est utile pour afficher les résultats de performances.

```
#include <QDialog>
```

Cette bibliothèque contient la classe permettant de créer une boîte de dialogue en Qt. La classe *DialogResultatsRO* hérite de la classe *QDialog*. La bibliothèque doit donc être incluse.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt. Une boîte de dialogue de la classe *DialogResultatsRO* devant afficher les résultats de performances, cette bibliothèque sera utile.

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner les widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de QLabel créés pour afficher les résultats.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre. Une boîte de dialogue de la classe *DialogResultatsRO* a un bouton *OK* pour fermer la boîte de dialogue.

```
#include <QScrollArea>
```

Cette bibliothèque contient la classe permettant de définir un périmètre qui aura une barre de défilement si les éléments affichés dans la fenêtre venaient à dépasser la taille de la fenêtre. Une boîte de dialogue de la classe *DialogResultatsRO* devant afficher les résultats du réseau et ceux de ses files, il est possible que leur affichage amène à avoir une bar de défilement.

```
#include <QString>
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de QLabel utilise une instance de QString.

```
class DialogResultatsRO : public QDialog{
```

La classe *DialogResultatsRO* hérite de la classe *QDialog* pour pouvoir créer de nouvelles boîtes de dialogue avec les caractéristiques souhaitées. L'héritage est public pour pouvoir garder l'encapsulation des membres de la classe *QDialog*.

```
private :
```

```
PerformancesRO * perfsRO;
```

Cet attribut correspond à la structure contenant les résultats des performances d'un réseau ouvert et de ses files. Elle est utile pour pouvoir accéder facilement aux résultats. Elle est passé sous forme de pointeur pour éviter de copier la structure.

```
QLabel * nb_moy_clients_reseau;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans le réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_rep_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de réponse moyen pour chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_sejour_moy_reseau;
```

Cet attribut correspond au texte représentant le résultat du temps de séjour moyen des clients dans le réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * debit_entree_reseau;
```

Cet attribut correspond au texte représentant le résultat du débit d'entrée des clients dans le réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * debit_sortie_reseau;
```

Cet attribut correspond au texte représentant le résultat du débit de sortie des clients dans le réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * nb_moy_clients_files;
```

Cet attribut correspond au texte représentant le résultat du nombre de clients moyen dans chaque file du réseau. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_attente_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'attente moyen dans chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_service_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps de service moyen dans chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * tps_interarrivee_moy_files;
```

Cet attribut correspond au texte représentant le résultat du temps d'inter-arrivee moyen dans chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * taux_perte_files;
```

Cet attribut correspond au texte représentant le résultat du taux de perte de chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * taux_utilisation_serveurs;
```

Cet attribut correspond au texte représentant le résultat du taux d'utilisation de chaque serveur de chaque file. Il est de type QLabel afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * layout_affichage;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * rangement_scroll;
```

Cet attribut correspond à la boîte verticale qui contiendra l'objet de type *ScrollArea*. Il est nécessaire que cet objet soit stocké dans une boîte afin de pouvoir être afficher sur la boîte de dialogue. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QScrollArea * scrollArea;
```

Cet attribut permet de délimiter l'environnement qui possèdera une barre de défilement. L'objet *scrollArea* contiendra la fenêtre *fenetre_affichage*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QWidget * fenetre_affichage;
```

Cet attribut correspond à la fenêtre qui contiendra la boîte de rangement *layout_affichage*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * ok;
```

Cet attribut correspond au bouton de fermeture de la boîte de dialogue. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public :

DialogResultatsRO(QWidget * parent = NULL, PerformancesRO * perfs);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base *NULL* et un pointeur sur la structure de type *PerformancesRO*. La structure est passée en pointeur afin d'éviter une copie de la structure et l'objet de type *QWidget* est passé également en pointeur pour récupérer l'adresse de la fenêtre parent de la boîte de dialogue ce qui permet de les lier. Ce constructeur appelle le constructeur de *QDialog* car la classe *DialogResultatsRO* hérite de *QDialog*. L'attribut *perfsRO* est intialisé avec l'adresse de la structure *PerformancesRO* passée en paramètre du constructeur. Pour les autres pointeurs, un appel de leur constructeur sera fait.

```
~DialogResultatsRO();
```

Le destructeur désalloue la mémoire allouée dynamiquement par les attributs de la classe.

```
void setInfoTpsRepMoyFiles();
```

Cette méthode récupère le résultat correspondant au temps de réponse moyen des files du réseau dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_rep_moy_reseau*.

```
void setInfoNbMoyClientsReseau();
```

Cette méthode récupère le résultat correspondant au nombre moyen de clients dans le réseau dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *nb_moy_clients_reseau*.

```
void setInfoTpsSejourMoyReseau();
```

Cette méthode récupère le résultat correspondant au temps de séjour moyen des clients dans le réseau dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_sejour_moy_reseau*.

```
void setInfoDebitEntreeReseau();
```

Cette méthode récupère le résultat correspondant au débit d'entrée des clients dans le réseau dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *debit_entree_reseau*.

```
void setInfoDebitSortieReseau();
```

Cette méthode récupère le résultat correspondant au débit de sortie des clients du réseau dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *debit_sortie_reseau*.

```
void seInfotNbMoyClientsFiles();
```

Cette méthode récupère les résultats correspondant au nombre moyen de clients dans chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *nb_moy_clients_files*.

```
void setInfoTpsAttenteMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps d'attente moyen dans chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_attente_moy_files*.

```
void setInfoTpsServiceMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps de service moyen dans chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_service_moy_files*.

```
void setInfoTpsinterarriveeMoyFiles();
```

Cette méthode récupère les résultats correspondant au temps d'inter-arrivee moyen dans chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_interarrivee_moy_files*.

```
void setInfoTauxPerteFiles();
```

Cette méthode récupère les résultats correspondant au taux de perte de chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *tps_perte_files*.

```
void setInfoTauxUtilisationServeurs();
```

Cette méthode récupère les résultats correspondant au taux d'utilisation de chaque serveur de chaque file dans la structure de type *PerformancesRO* et le transforme en string avant de l'ajouter à l'attribut *taux_utilisation_serveurs*.

```
};
```

6.15 Classe FenetreEcheancier

```
#include <QVBoxLayout>
```

Cette bibliothèque contient la classe permettant d'ordonner des widgets d'une fenêtre en Qt. Pour avoir un bel affichage, il est donc nécessaire d'ordonner les instances de *QLabel* créées pour afficher les résultats.

```
#include <QPushButton>
```

Cette bibliothèque contient la classe permettant d'ajouter des boutons à une fenêtre.

```
#include <QLabel>
```

Cette bibliothèque contient la classe permettant d'écrire du texte et de le mettre en forme en Qt.

```
#include <QWidget>
```

Cette bibliothèque contient la classe *QWidget* qui est une classe permettant d'utiliser des *QWidget* pour l'affichage d'une fenêtre.

```
#include <QString>
```

Cette bibliothèque contient la classe permettant de traiter un string en Qt. Elle est utile car une instance de *QLabel* utilise une instance de *QString*.

```
#include "Echeancier.hh"
```

Cette bibliothèque contient la classe permettant de créer une instance de la classe *Echeancier*. Cet élément est utile pour l'afficher.

```
Class FenetreEcheancier : public QWidget
```

La classe *FenetreEcheancier* hérite de la classe *QWidget* pour permettre d'utiliser ses méthodes en public.

```
Q_OBJECT
```

Cette macro est nécessaire pour créer des signaux et des slots.

```
private :
Echeancier * echeancier;
```

Cet attribut correspond à l'échéancier. Il est en attribut pour permettre de l'afficher. Il s'agit d'un pointeur car il n'est pas contenu directement dans la fenêtre.

```
int position_echeancier;
```

Cet attribut correspond à la position d'un évènement dans l'échéancier.

```
QLabel * date_actuelle;
```

Cet attribut correspond au texte représentant la date actuelle de la simulation. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * date_evenement;
```

Cet attribut correspond au texte représentant la date d'un événement. Il est de type *QLabe*l afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * file_concernee;
```

Cet attribut correspond au texte représentant la file concernée par l'événement à la position *position_echeancier* dans l'échéancier. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * serveur_concerne;
```

Cet attribut correspond au texte représentant le serveur concerné par l'événement à la position *position_echeancier* dans l'échéancier. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QLabel * client_concerne;
```

Cet attribut correspond au texte représentant le client concerné par l'événement à la position *position_echeancier* dans l'échéancier. Il est de type *QLabel* afin de pouvoir mettre en forme le texte. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QVBoxLayout * rangement_echeancier;
```

Cet attribut correspond à la boîte permettant d'ordonner l'affichage dans la boîte de dialogue de façon verticale. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * suivant;
```

Cet attribut correspond au bouton de passage au prochain événement dans l'échéancier. Le signal *clicked* de ce bouton est relié au slot *evenement_suivant()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
QPushButton * precedent;
```

Cet attribut correspond au bouton de retour à l'événement précédent dans l'échéancier. Le signal *clicked* de ce bouton est relié au slot *evenement_precedent()*. Il s'agit d'un pointeur car une maîtrise sur la gestion de la mémoire est souhaitée.

```
public slots :
```

Les slots sont des méthodes qui peuvent être appelés par un signal sur Qt.

```
void evenement_precedent();
```

Cette méthode permet de passer à l'événement précedent.

```
void evenement_suivant();
```

Cette méthode permet de passer à l'événement suivant.

```
public :
FenetreEcheancier(Echeancier * ech, QWidget * parent=NULL);
```

Ce constructeur prend en paramètre un pointeur sur un objet *QWidget* avec comme valeur de base NULL et un pointeur sur un objet de la classe *Echeancier*. *ech* est passé en pointeur afin d'éviter une copie de l'objet et va initialiser l'attribut *echeancier*. Le pointeur sur l'objet de type *QWidget* permet de récupérer l'adresse du parent d'une fenêtre de cette classe. Il est initialisé de base à NULL si la fenêtre ne possède pas de parent. Ce constructeur appelle le constructeur de la classe *QWidget* vu que la classe *FenetreEcheancier* en hérite. Pour tous les pointeurs en attribut, leur constructeur est appelé.

~FenetreEcheancier();

Le destructeur désalloue la mémoire allouée dynamiquement par les attributs de la classe.

```
void setInfoDate();
```

Cette méthode récupère le résultat correspondant à la date de l'événement à la position *position_echeancier* dans l'échéancier et le transforme en string avant de l'ajouter à l'attribut *date_evenement*.

```
void setInfoClient();
```

Cette méthode récupère le résultat correspondant au client concerné par l'événement à la position *position_echeancier* dans l'échéancier et le transforme en string avant de l'ajouter à l'attribut *client_concerne*.

```
void setInfoFile();
```

Cette méthode récupère le résultat correspondant à la file concernée par l'événement à la position *position_echeancier* dans l'échéancier et le transforme en string avant de l'ajouter à l'attribut *file_concernee*.

```
void setInfoServeur();
```

Cette méthode récupère le résultat correspondant à un serveur concerné par l'événement à la position *position_echeancier* dans l'échéancier et le transforme en string avant de l'ajouter à l'attribut *serveur_concerne*.

```
void setDateActuelle();
```

Cette méthode récupère le résultat correspondant à la date actuelle de la simulation et le transforme en string avant de l'ajouter à l'attribut *date_actuelle*.

7 Conclusion

Le *Cahier des Spécifications* a pour but de modéliser l'architecture de notre projet *"Simulation de réseaux de files d'attentes"*. Il nous permet de nous pencher sur une vision plus approfondie et technique de chacun des modules du *Cahier des Charges* : *Gestion de Fichier*, *Gestion du Réseau*, *Calculs de Performances* et *Interface Graphique*. Le travail fourni durant ces dernières semaines pour rédiger ce *Cahier des Spécifications* sera un support pour l'implémentation de notre projet. Il nous permet d'avoir une vue d'ensemble sur le codage et de quelle manière l'implémenter.

8 Références

[1] <stdexcept> - C++ Reference

[2] Gérez des erreurs avec les exceptions - Programmez avec le langage C++

[3] Programmez avec le langage C++

[4] <cmath> (math.h) - C++ Reference

[5] Vector in C++ STL

[6] C++ : La classe list de la librairie standard (STL)

[7] List in C++ Standard Template Library (STL)

[8] Lisez et modifiez des fichiers - Programmez avec le langage C++

[9] Initiez-vous à Qt - Programmez avec le langage C++

[10] QWidget Class — Qt Widgets 5.14.2

[11] QAction Class — Qt Widgets 5.14.2

[12] QMenu Class — Qt Widgets 5.14.2

[13] QMouseEvent Class — Qt GUI 5.14.2

[14] QPushButton Class — Qt Widgets 5.14.2

[15] QGraphicsView Class — Qt Widgets 5.14.2

[16] QInputDialog Class — Qt Widgets 5.14.2

[17] QFileDialog Class — Qt Widgets 5.14.2

[18] QMessageBox Class — Qt Widgets 5.14.2

[19] QLabel Class — Qt Widgets 5.14.2

[20] QTime Class — Qt Core 5.14.2

[21] Apprenez à utiliser les boîtes de dialogue usuelles

[22] [Résolu] [Qt] Chrono avec fonction pause

[23] QTimer Class — Qt Core 5.14.2

[24] QHBoxLayout Class — Qt Widgets 5.14.2

[25] QVBoxLayout Class — Qt Widgets 5.14.2

[26] QSpinBox Class — Qt Widgets 5.14.2

[27] QComboBox Class — Qt Widgets 5.14.2

[28] QDialog Class — Qt Widgets 5.14.2

[29] QLineEdit Class — Qt Widgets 5.14.2

[30] QString Class — Qt Core 5.14.2

[31] QScrollArea Class — Qt Widgets 5.14.2

[32] QInputDialog with multiple fields

[33] Qt : Position Souris QMainWindow - Récupérer la position XY de la souris dans la zone centrale par betsprite

[34] Utilisez les signaux et les slots - Programmez avec le langage C++