Projet Java - Collector

# Introduction

Le but de ce projet est de gérer une connexion client-serveur par communication TCP et de créer une interface graphique permettant à un utilisateur de visualiser les données récoltées par le serveur.

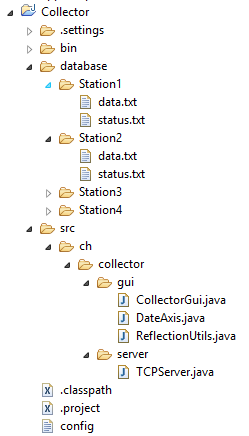
Le projet est donc divisé en deux programmes : d’un côté nous avons un serveur qui récolte et répertorie les données envoyé par le client et de l’autre côté, une interface graphique (GUI) qui traite et affiche les données en fonction des demande de l’utilisateur.

Ce projet traite donc différentes notions de la programmation en java :

* Echange entre «clients-serveur»
* Notion d’«Output Stream» et d’«Input Stream»
* La notion de multithreading
* Gestion d’une «database»
* Traitement et affichage de données dans une interface graphique
* Programmation « évènementielle »

# Organisation du répertoire du projet

Le répertoire de travail est organisé comme suit :



Ce « projet » contient 2 classes principales :

* Le serveur reçoit les données envoyé par les clients et le stock dans la hiérarchie de fichier ci-dessus (en fonction du nom de la station identifié).
* Le GUI scan (selon les désirs de l’utilisateur) les différents fichiers et affiche les données souhaités.

Les Classes « DateAxis » et « ReflectionUtils » sont des classes que j’ai utilisées pour créer un Objet de type « DateAxis ». Ce n’est pas moi qui les ai codées.

Ces deux classes sont disponibles à cette adresse : <https://github.com/dukke/FXCharts>

# 

**Note :** Le fichier « config » contient le nom des différentes stations qui se connectent au serveur.

Ce fichier est utilisé par le GUI qui va le scanner et ajouter le nom de toutes les stations à l’interface. Afin que l’utilisateur puisse sélectionner celle de son choix.

Cependant, si une station est « supprimée », il faut mettre à jour le fichier « config » manuellement.

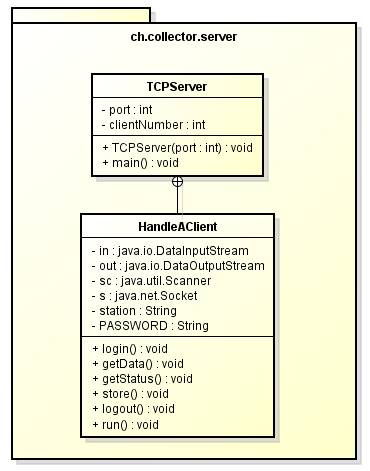
# 

Ce « projet » contient 4 classes qui simulent des «Stations» (clients) qui peuvent se connecter au serveur (TCPServer.java) pour y envoyer des données.

Chaque station génère des données « aléatoire »

# Serveur

## Diagramme de class UML du serveur

****

## Digramme de class UML d’un client « test »

## 

## Echange client-serveur

L’envoi des données au serveur se fait de la façon suivante :

* Le serveur est lancé et attend qu’un client se connecte (un nouveau thread est créé à chaque fois qu’un client se connecte. J’jusqu’à 4 clients peuvent être connecté simultanément).
* Un client se connecte au serveur
* Le client envoi la « commande » « LOGIN » ainsi que son identifiant et le mot de passe au serveur.
* Le serveur reçoit la « commande » « LOGIN » et procède à l’identification de la station.
* Une fois le client identifié, le serveur envoi la « commande » « DATAQUERRY » au client
* Le client reçoit la « commande » « DATAQUERRY » et envoi les données au serveur
* Une fois que le serveur a fini de récolter les données du client (« DATAEND »), le serveur envoi la « commande » « STATUSQUERRY » au client.
* Le client reçoit la « commande » « STATUQUERRY » et envoi ses données au client.
* Une fois que le serveur a fini de récolter les «status» du client (« STATUSEND), le serveur attend la commande « LOGOUT » du client, puis met fin à la connexion.

Ci-dessous, voici un exemple du déroulement d’un échange entre un client et le serveur :

# **Serveur :**

# 

**Client :**

# 

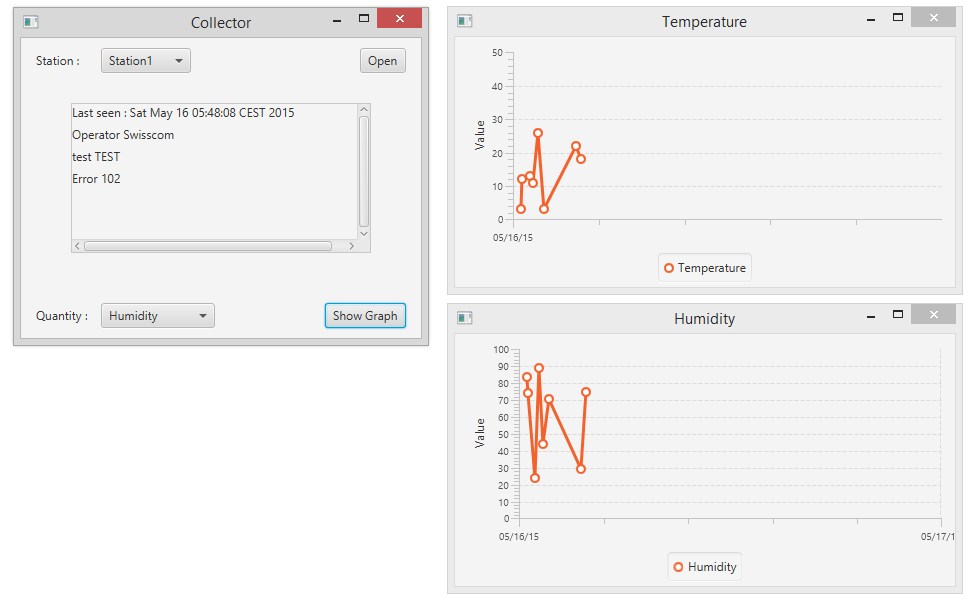
# Graphical User Interface (GUI)

Le GUI a été réalisé selon les exigences présentées dans le cahier des charges. Il y a toutefois des changements apporté entre l’exemple présenté dans le cahier des charges et l’interface réalisé.

La fenêtre ci-dessous est constituée d’un BorderPane dans lequel ont été placés trois GridPane (top, center et bot).

Les données affichées au centre sont les « status » de la station sélectionnée. Les deux « scollbar » permettent de visualiser l’ensemble des données au cas où ces dernières venaient à dépasser du « cadre ».

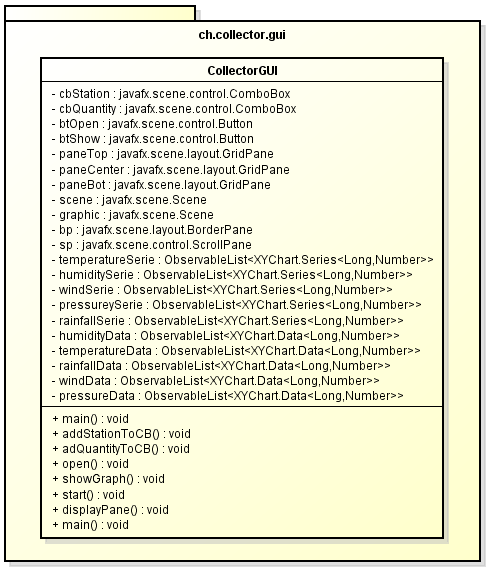
Lorsque le bouton « showGraph » est « toggle », une nouvelle fenêtre qui contient le graphique approprié à la sélection de l’utilisateur s’ouvre. Ce dernier peut alors changer la grandeur à évaluer et ouvrir une nouvelle fenêtre.



Les points sur les graphiques sont affichés en fonction de leur valeur (axe « y », vertical) et de la date à laquelle la mesure a été effectuée (axe « x », horizontal).

**Note :** L’axe « X » s’ajuste automatiquement en fonction des points à afficher.

## Diagramme UML du GUI



# Problèmes rencontrés

* **La méthode writeUTF de DataOutputStream**: la commande writeUTF permet d’envoyer un String dans un OutputStream. Le fait est que deux caractères représentant la taille de la trame sont envoyés au début de la chaine de caractère. C’est pourquoi lorsque des commandes sont échangées entre le client et les serveurs, je commence par envoyer un espace.

Exemple : si le serveur envoi "DATAQUERRY", le client va recevoir

"xxDATAQUERRY" et va donc mal interpréter la commande. Alors que si le serveur envoi

" DATAQUERRY", peu importe les caractères qui précédent l’envoi de la commande, le client va « attraper » la commande DATAQUERRY.