Rapport Tuteur Algèbre

Noms: *Di Gangi Mattia Antonino & Pillitteri Roberta*

Formation: *M2 SSIO (Erasmus)*

Prof. Michael Landschoot

Université Paris-Est Marne-la-Vallée

Table des matières

Chapitre 1

Introdution

Le tuteur algèbre est une application dont but est de fournir un instrument pour échanger des exercices de math et leur corrections entre les élèves et un professeur.
Les exércises seront fait avec un simple langage algèbrique dont l'interpréteur est fournit ensemble au program client, celui donné aux éleves. On peut distinguer deux type d'usage de l'application en base à qui va l'utiliser :

- les élèves peuvent effectuer trois actions differents : envoyer leur exercices, vérifier l'exactitude de la syntaxe des exercices, récupérer les corrections.
- le professeur devrait avoir la possibilité de récupérer les exercices et envoyer les corrections, mais en ayant réalisé une version réduite de l'application, le professeur devra insérer les corrections manuellement, dans le dossier *corrections* qu'on peut trouver dans la partie serveur de l'application.

Pour raisons de temps limité on a choisi d'implementer la plupart des expressions, l'interpréteur du langage et l'architecture client-server, avec le serveur qui peut recevoir plus connexions dans plusieures threads.

On a choisi de composer le projet en trois parties :

(a) l'interpréteur d'expressions algébriques, construit en utilisant les outils flex et bison, qui définit le langage de programmation à utiliser ;

(b) le client ;

(c) le serveur.

En plus, on a aussi une librairie nommée *tcpsocket*, qui fournit une abstraction pour les sockets POSIX. Cette librarie est utilisée soit par le client soit par le serveur.

L'interpréteur a été integré dans le client parce-qu'il est le seul composant qui l'utilise, mais toutes ses classes sont dans le dossier expression qui pourrait devenir une librairie statique de la même façon de *tcpsocket* si ça est utile.

Chapitre 2

Mode d'emploi de l'application

L'application a deux fichiers exécutables différents : le client et le serveur.
Le serveur devrait être toujours en exécution parce-qu'il fait de interface de communication entre le professeur et les éleves et son but est d'accepter les connexions des clients et répondre à leur réquêtes.

Le client a trois utlisations différents (fig.2.1) :
— Envoyer un fichier contenant un program syntactiquement correct et prêt pour être correcté par le professeur.
— Donner un fichier contenant un program à l'interpréteur pour en vérifier la correction syntactique et avoir le résultat.
— Visualiser les programs correctés par le professeur et en récuperer.

Pour le professeur le mode d'employ est encore à ameliorer parce-qu'il aura tous les programs envoyés dans le dossier *received* et il pourra mettre les corrections dans le dossier *corrections*.
De cette façon le professeur doit connaitre les dossiers du serveur pour pouvoir utiliser l'application et envoyer les correction à les éleves. Une version complète de l'application devrait avoir un autre outil pour le professeur qui communique avec le serveur ainsi que le professeur ne doit pas connaitre les dossiers utilisés.

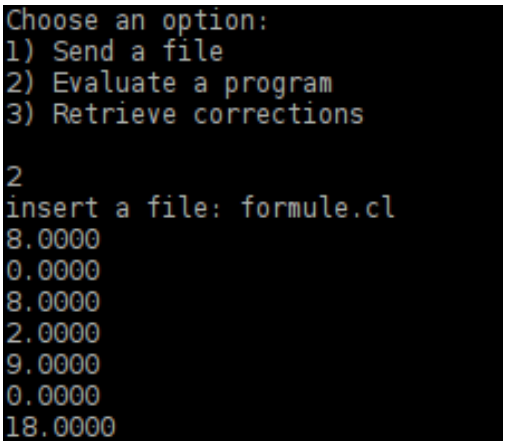

FIGURE 2.1 – Exemple d'execution de l'application

Chapitre 3

Architecture fonctionnelle

Le projet est composé par deux modules principales, le client et le serveur, chacun ayant des dépendances par des librairies externes.
Le but du client est de fournir un logiciel qui permet aux éleves de évaluer leur programmes écrits dans notre petit langage algèbrique, et efféctuer la connexion avec le serveur.
Le but du serveur est de collecter les fichiers envoyés par les clients ainsi que le prof puisse les

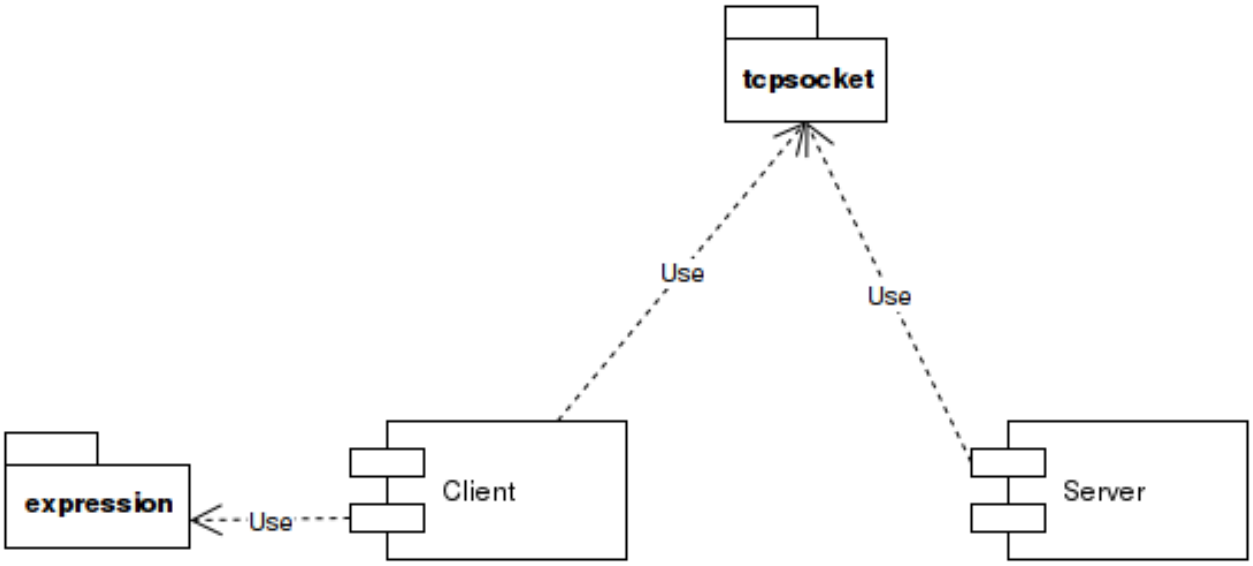

FIGURE 3.1 – Interaction entre les composants logiciel

corriger et leur mettre dans un dossier avec toutes les corrections.
Tant le client que le serveur dépendent de tcpsocket, une librairie qui fournit des APIs haute niveau pour les communications TCP.
Enfin, le client dépend aussi du module expression qui fournit un parseur utilisant la hiérarchie d'Expression implantée.

Le logiciel a été conçu pour être extensible dans toutes les composants. Dans les sections suivantes on montre les design patterns utilisé pour avoir un logiciel extensible et entretienable.

3.1 Expression

La hiérarchie des classes qu'implémentent Expression forme naturellement une structure arborescent qui permet une facile realisation du Pattern Composite où les "composite" sont toutes les operations tandis que les "components" sont les classes Constante et Variable.
Le diagram de la hiérarchie est montré dans la figure 3.2. Les relations entre les classes sont montré

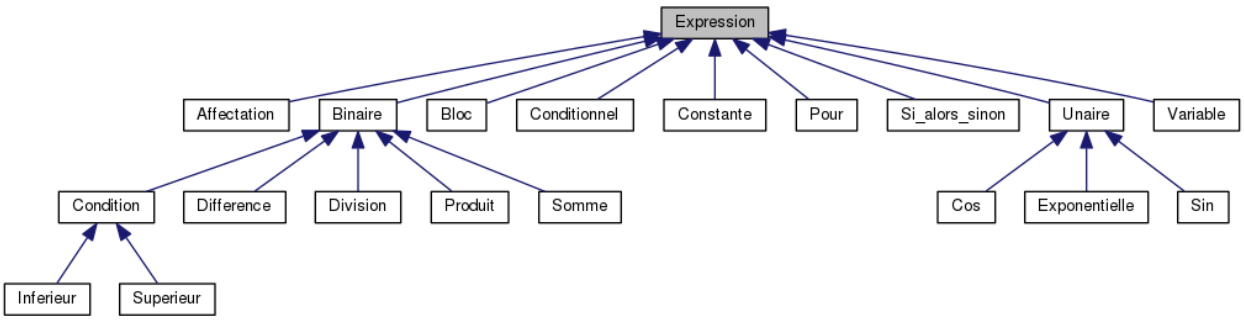

FIGURE 3.2 – Hiérarchie Expression

dans la documentation du projet crée par doxygen.
Le parseur connaît toutes les classes de la hiérarchie mais il les stocke comme des pointeurs à Expression. De cette façon, si on veut augmenter le langage, il faut juste créer les nouvelles classes comme des implémentations d'Expression et les appeler dans le parseur (fichier calc.y). Grâce au Composite Pattern, le nouveau composant sera parfaitement integré.

La seule classe qui utilise une structure des données différent, elle est Variable. Cette classe répresent une variable de type réel dans notre langage, donc sa valeur doit être stockée dans une tableau des symboles.
Ce tableau est implémenté comme une std : :map statique et privée dans la classe Variable, dont clè est le nom de la variable et la valeur est la valeur de type double.
Le constructeur de Variable crée ou met-à-jour la valeur dans le tableau tandis que la fonction membre **eval()** retourne la valuer y stockée.
Pour pouvoir assurer la correction des affectation aux variables, surtout au dedans d'un boucle, on utilise une deuxième classe appelé Affectation, qui stocke le nom d'une variable et une Expression* qui calcul la valeur à affecter. La fonctione membre **eval()** stockera la valeur si calculée dans le tableau des symboles, et si l'Expression contient des variables, chaque fois qu'on appel eval() on aura la valuer correct calculée en utilisant les valeurs des variables mises-à-jour.

3.2 Client

Le client est defini par le fichier client.cpp et ses dépendances (fig. 3.3).
Le client crée un menu en utilisant la classe ClientMenu et la remplit avec des taches qui se trouvent dans les implémentations de l'interface MenuOption. De cette façon on a une realisation du Command Pattern (fig. 3.4) qui permet des faciles extensions au menu, et donc aux fonctionalités du client. En effet, si on veut ajouter une nouvelle fonctionalité au client on doit que créer la classe concrete de MenuOption implementant le tâche souhaité et après, dans client.cpp appeler une instance de la classe et l'enregistrer dans ClientMenu.
Une fois que le ClientMenu est initialisé, le main ne fait qu'un loop qui montre le menu et demande

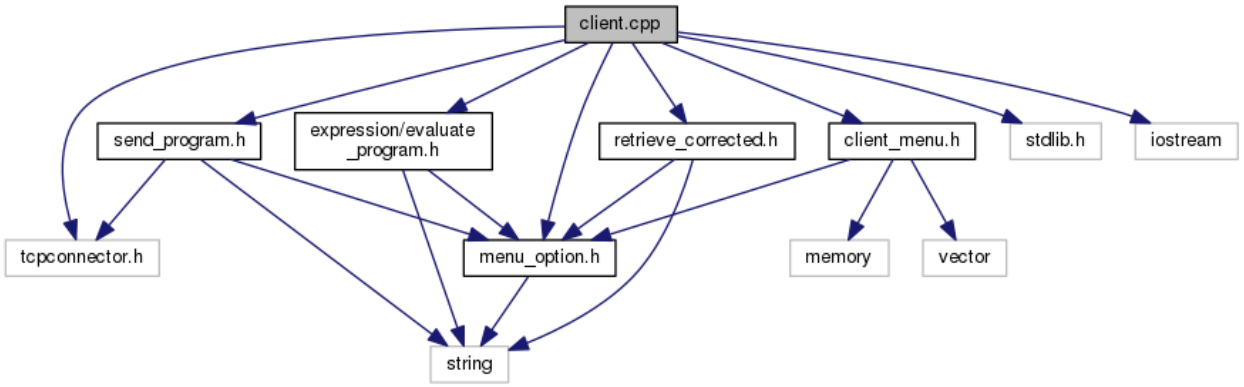

FIGURE 3.3 – Graph des dépendances de client.cpp

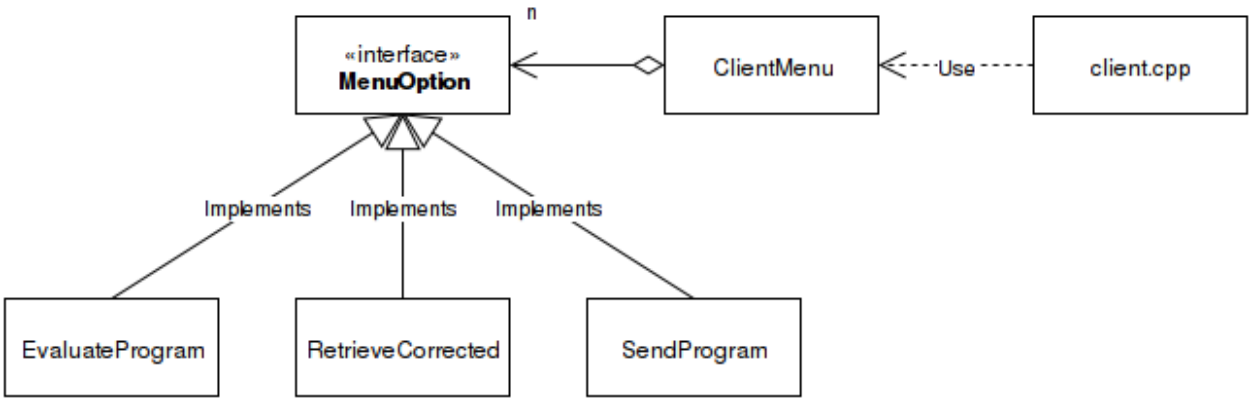

FIGURE 3.4 – Command Pattern dans le client

à l'utilisateur de choisir une option.

3.3 Serveur

Le serveur est assez simple il-même. Le fichier principale est server.cpp qui definit le boucle principale pour accepter connexions, vérifier la correction des rêquetes et les envoyer aux classes implémentant le tâches rêquetes de façon multi-threaded.
Les tâches se trouvent dans des classes implèmentant l'interface "functor" ServerTask.
Pour avoir un serveur multi-threaded on a utilisé les threads fournit par le fichier d'en-tête <thread>.
Cettes threads prennent comme argument une fonction, donc on utilise une fonction appelée *execute_task* dont seul but est d'être un wrapper pour l'appel au tâche.
Au dedans de la fonctione *main* on a le serveur qui accepte les rêquetes faites par les clients, vérifie le type de tâche demandé et crée une nouvelle thread exécutant le tâche.
Au moment il n'y a que deux tâches, donc la selection est fait par moyen d'un simple if-else, mais si l'architecture devient plus compliquée, on a déjà tout ce que c'est nécessaire pour implanter un design pattern qui rend le logiciel plus extensible comme on a fait dans le cas du client.

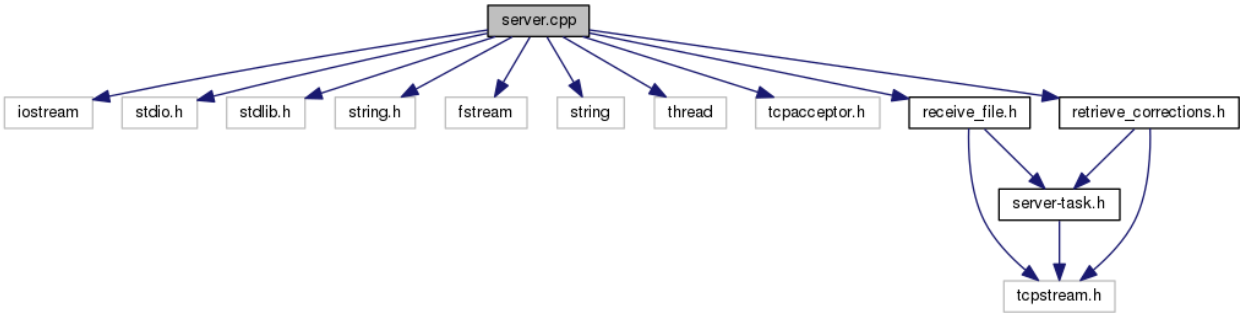

FIGURE 3.5 – Graph des dépendances de server.cpp

Chapitre 4

Conclusion

Ce projet-là nous a fournit un exemple de travaille en C++ qui n'est pas exactement un outil au niveau d'entreprise, mais qui nous a donné la possibilité d'effectuer un projet pas triviale avec ce langage.
On a eu beaucoup de difficultés, en commençant par la choix de l'IDE, en fait on a commencé en utilisant codeblocks, mais à cause de problèmes sur Debian on a utilisé enfin eclipse pour C++.
On a decouvert bientôt que la compilation n'est pas facile pour des novices du langage et on a du configurer beaucoup de choses même si on a utilisé un IDE pour automatiser parties du travaille.

Une autre difficulté qu'on a eu a été l'integration du parser bison dans le projet. En fait, nous avions déjà utilisé bison en langage C et nous avons écrit assez vite une parseur pas mal, mais beaucoup de temps a été nécessaire pour appeler yyparse() par le client. Dans un premier temps nous avions compilé le parseur comment un fichier C++ mais ça marchait pas. La solution a été de traiter les deux fichiers correctement comme des fichiers .c, mais les compiler avec le compilateur g++ et utiliser le linkage "C".

Des fonctionalités intéressants pourraient être, sans considerer la creation d'une GUI comme dit dans le sujet, de créer des idéntifiants pour les clients, de façon que chaque client peut acceder que à ses fichiers dans le serveur. La securité sera un facteur intéressant comme elle est dans toutes les applications réseau.
De plus, en ayant une GUI et une liste des vrais exércises, on pourrait changer la communication entre client et serveur pour faire récuperer toujours la liste des exércises au client, et en cliquant sur un exércise montrer les options disponibles comme récuperer le texte, charger une solution et récuperer sa solution s'elle est disponible.
La définition d'un format de fichiers pour le corrections pourrait être considerée elle-même.

En conclusion, ce projet a été intéressant pour comprendre un petit peu comme travailler en C++, un langage très intéressant et très utile pour les calculs à haute performances, mais dont courbe d'apprentissage est très raide.

Bibliographie

cplusplus.com. `http://www.cplusplus.com/`.

Tcp/ip network programming design patterns in c++. `http://vichargrave.com/network-programming-design-patterns-in-c/`.

Bruce Eckel. *Thinking in C++, Volume 1, 2nd Edition*. Prentice Hall Inc, 2000.

Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

Scott Meyers. *Effective C++, 3rd Edition*. Addison-Wesley, 2005.

Bjarne Stroustrup. *The C++ Programming Language (4th Edition)*. Addison-Wesley, 2013.