Références croisées
Spécification et Conception

1 Spécification complète

1.1 Définitions

- **Mot** : Suite de caractères se terminant par un délimiteur
- **Délimiteur** : Un caractère représentant une séparation entre deux mots (Ex : une virgule, un espace, un point, ...)
- **Ligne** : Suite de mots terminée par un retour chariot
- **Identificateur** : Mot sensible à la casse composé uniquement de caractères alphanumériques et du caractère '_'. Un identifacteur commence par une lettre ou par le caractère '_'. Les commentaires ou chaines litérales ne peuvent contenir d'identifcateurs
- **Mot Clef** : Identificateurs que l'utilisateur souhaite rechercher dans un ou plusieur fichier. Par défaut il s'agit de ceux de ceux du C++
- **Référence croisée** : Fait de rechercher un identificateur dans un ou plusieurs fichiers sources pour déterminer sa localisation

1.2 Description du programme

Le but du programme est de permettre de retrouver rapidement l'emplacement d'identifcateurs dans une collection de fichiers. L'utilisateur peut choisir de rechercher les identifacateurs étant des mots cléfs, ou ceux qui n'en sont pas. On cherche à connaitre dans quel(s) fichier(s) et à quelle(s) ligne(s) ces identifcateurs apparaissent.

Dans le cas où un identifacteur apparaitrait plusieurs fois sur une même ligne, nous avons pris la décision d'afficher la ligne concernée autant de fois qu'il y a d'occurences. En effet, un identificateur est un mot clé significatif pour l'utilisateur. Nous pouvons donc nous attendre à ce qu'il n'apparaisse que peu sur la même ligne.

Ex : Pour le code ci-dessous présent dans le fichier "test.cpp" et avec comme identifacteur la lettre i

```
for( int i = 0; i < 42; i++ );
```

Le programme produira la sortie suivante :
i→test.cpp•1•1•1↓

Par defaut les mots clefs sont ceux utilisés par le langage C++. Il est cependant possible de spécifier un fichier en argument du programme pour définir précisement quelles seront les identifcateurs recherchés par la référence

croisée. Le fichier d'identifacteurs ne devra contenir qu'un seul identifacteur valide par ligne, le premier mot de la ligne sera choisi comme tel. Le programme présuposera que le fichier fourni en argument respecte ce formalisme.

1.3 Spécifications des options

$$tp\_stl\ [-e]\ [-kfichier\_mot\_clef]\ [nomfichier]+$$

-e : Permet d'inverser le comportement par défaut du programme. Au lieu de rechercher les identificateurs faisant parti du fichier de mot clé, le programme recheche les identificateurs n'étant pas des mots clefs.
-k fichier_mot_clef : Permet de spécifier au programme une liste de mot clé à rechercher par la référence croisée. Par défaut il s'agirat des mots-clefs du langage C++
nomfichier : Chemin vers un ou plusieurs fichiers où rechercher les identifacateurs

2 Tests fonctionnels

2.1 Méthodologie

Nous sommes parti du principe que le programme doit produire la même sortie écran (et donc par extension les mêmes données stockées en interne) si il est lancé deux fois sur la même collection de fichiers et avec les mêmes arguments.

En partant de ce principe nous avons réaliser les tests de la façon suivante :

1. Lancer le programme sur une collection de fichier
2. Verifier que la sortie est correcte et respecte les spécifications
3. Relancer le programme dans le même contexte qu'en 1, trier la sortie et rediriger le flux de sortie dans un fichier témoin

Pour automatiser les tests nous avons écrit un script bash qui suit les étapes suivantes pour chaque test :

1. Lancer le programme dans le même contexte que lors de la réalisation du fichier témoin, trier la sortie et rediriger la sortie vers un fichier résultat
2. Faire le hash md5 du fichier résultat et du fichier témoin
3. Comparer les deux hash md5
4. Si les hashs diffèrent c'est que le programme ne possède pas les mêmes données en interne et donc le test échoue
5. Si les hashs sont égaux c'est que le programme possède les mêmes données et donc le test réussi

2.2 Critique de la méthode

Avantages : Avec cette méthode de réalisation des tests on s'abstrait de la représentation interne des données. On vérifie uniquement que le programme délivre correctement à l'utilisateur les informations voulues. Ainsi tout changement dans la structure de données interne n'affectera pas les tests tant que le programme délivre les memes informations à l'utilisateur.

De plus la réalisation d'un test supplémentaire se fait facilement et ne prend pas beaucoup de temps par rapport à une analyse compléte de la structure de données interne.

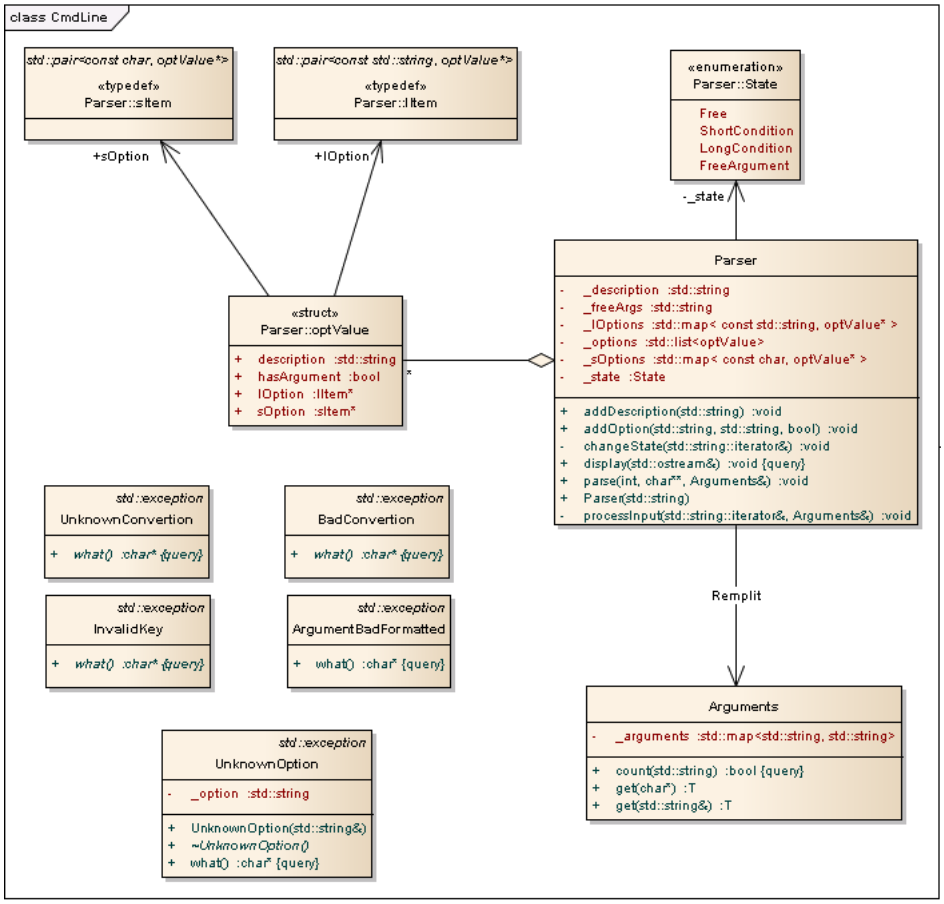
Nous gardons une sauvegarde du flux de sortie lors d'un état fonctionnel de l'application, de ce fait, le développeur peut comparer visuellement les deux sorties pour trouver plus facilement les éléments divergeants. D'où un gain de temps lors du débuggage

Inconvénients : Avec cette méthode, le formatage des informations à l'utilisateur devient un élément critique. Tout changement dans l'affichage des résultats causera inevitablement l'échec de tous les tests. Toutefois, il peut être facile de créer un script bash permettant de regénérer les fichiers témoins si le développeur est sur des changements qu'il a effectués.

Si un fichier témoin venait à être corrompu cela entrainerait l'échec du test le mettant en jeu.

3 Architecture générale

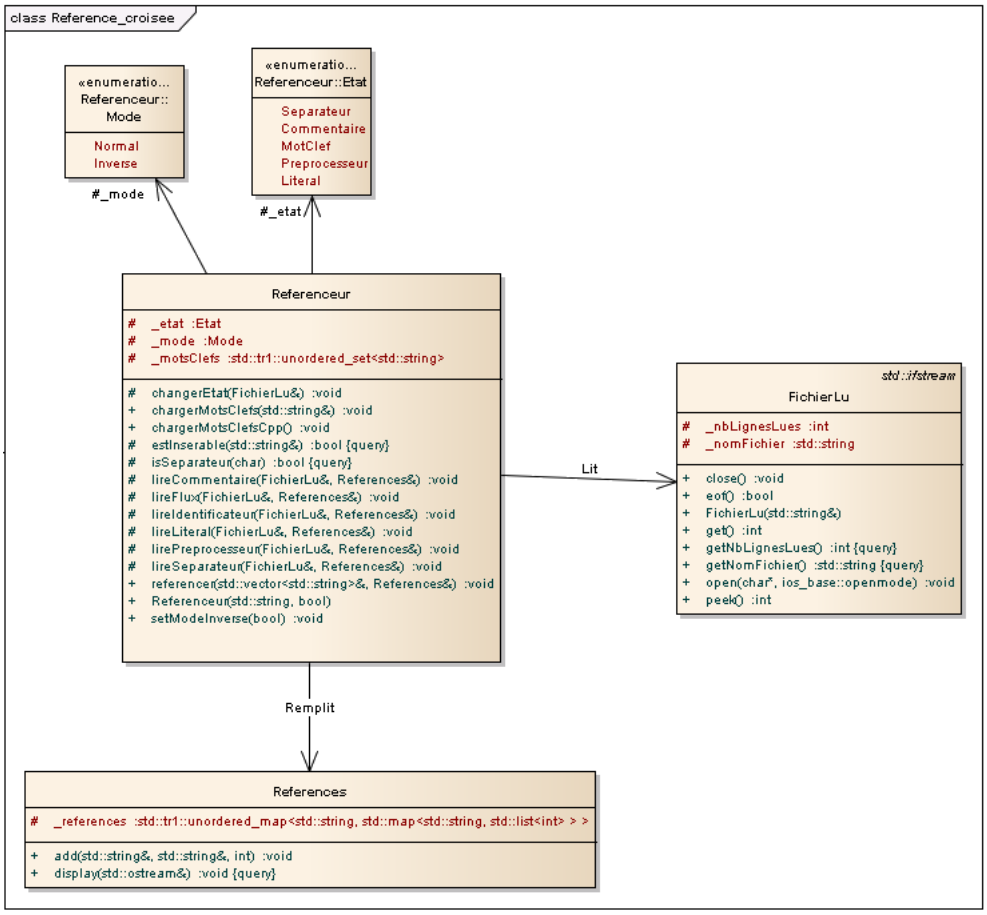
3.1 Diagramme de classe du module CmdLine



Ce diagramme présente les différentes classes présentes pour extraire les informations de la ligne de commande de façon générique.

La classe “Parser” s’occupe d’extraire les informations de la ligne de commande en vérifiant que les options entrées par l’utilisateur respectent celles définies par le développeur du programme. Au fur et à mesure de l’extraction des données, la classe “Parser” remplit un objet de la classe “Arguments”. La classe “Arguments” sert de conteneur et permet de convertir les options vers des types déterminés à la compilation. Ne sachant pas comment représenter des méthodes génériques en UML, j’ai défini le type de retour des fonctions membres “get” comme étant T. Le module possède ses propres exceptions pour remonter les cas d’erreurs possibles.

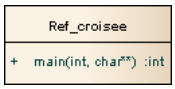
3.2 Diagramme de classe du module Reference_croisée



Ce diagramme présente les différentes classes utilisées pour extraire les identifacateurs d'un fichier source C++. Un objet de la classe "FichierLu" permet de lire un fichier stocké sur le disque, tout en fournissant le nombre de lignes déjà lues ainsi que le nom du fichier ouvert.

Un "Referenceur" se sert d'un "FichierLu" pour lire les fichiers sources et en extraire les identifacateurs. Une collection de "References" permet de stocker les identifacteurs qui sont des mots clefs.

3.3 Diagramme de classe du module principal



Le module principal permet d'orchestrer les deux modules précédents pour que le programme ait le comportement attendu.

4 Algorithmes principaux

4.1 Parseur pour la ligne de commandes

Pour extraire les informations de la ligne de commande nous utilisons un automate avec un nombre d'états fini.

Description des états :

- **Free** : Lorsque l'automate rencontre quelque chose qui n'est pas un argument ou une option
  Exemple : un caractère séparateur comme un espace
- **ShortCondition** : Lorsque l'automate rencontre une option courte
  Exemple : -e ou -k
- **LongCondition** : Lorsque l'automate rencontre une option longue
  Exemple : –exclude ou –keyword
- **FreeArgument** : Lorsque l'automate rencontre un argument rataché à aucune option
  Exemple : le nom d'un fichier à analyser

Une action est déclenché en fonction de l'état de l'automate. L'action extrait, analyse et stocke l'argument de la ligne de commande s'il est valide, sinon une exception est levé.

4.2 Parseur pour les fichiers C++

Pour extraire les informations de la ligne de commande nous utilisons ici aussi un automate avec un nombre fini d'états.

Description des états :

- **Separateur** : Lorsque que l'automate rencontre un caractère séparant deux identifacteurs
  Exemple : Tout caractère non alphanumérique, le tiret du bas non inclus
- **Commentaire** : Lorsque l'automate rencontre un commentaire sur une seule ligne ou multiligne
  Exemple : /* Ceci est un commentaire */

- **MotClef** : Lorsque l'automate rencontre un identifacteur qui peut être un mot clef potentiel
  Exemple : cout
- **Preprocesseur** : Lorsque l'automate rencontre une instruction preprocesseur
  Exemple : #include <iostream>
- **Literal** : Lorsque l'automate rencontre une chaine de caractères ou un caractère
  Exemple : "Bonjour"

Chaque état déclenche une action propre qui a pour tâche d'avancer dans le fichier tout en extrayant les identifacteurs recherchés (ceux qui sont des mots clefs, ou ceux qui n'en sont pas selon l'option choisie).

5 Analyse critique des structures de données

5.1 Structure des mots-clefs

Analyse des besoins : Les mots-clefs sont extraits d'un ou plusieurs fichiers passés en paramètres. À chaque Identificateur rencontré durant l'analyse, il faut vérifier s'il a déjà été référencé auparavant et si non créer son entrée dans la structure de données. Le programme devant tester de nombreuses fois l'égalité avec un mot clef, nous souhaitons optimiser les accès. De plus chaque mot clef référencé possède une liste de fichiers où il apparait. Il faudra donc pouvoir associer des valeurs aux mots-clefs.

Étude d'un arbre binaire : Nous cherchons à optimiser les accès dans la structure de données pour les identificateurs. Dans le pire des cas, si l'arbre n'est pas isométrique il faut parcourir tous les noeuds pour savoir si la clef est présente (O(n)). En revanche, dans le cas d'un un arbre équilibré (Ex : un arbre rouge et noir) la recherche est de complexité O(log(n)). Un autre avantage d'un arbre binaire est que les mots-clefs seront triés.

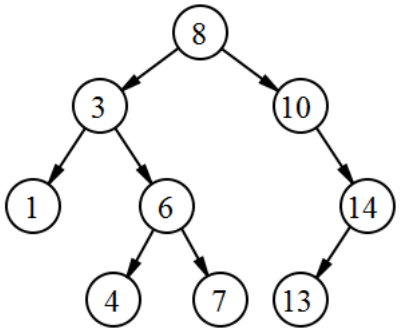


FIGURE 1 – Arbre binaire

Étude d'une table de hashage : Dans le cas d'une table de hash l'accès aux données est divisé par une constante C qui est le temps de calcul de la fonction de hash. Nous pouvons dire que l'accès est de complexité O(1) dans le pire des cas. En contrepartie de cette vitesse d'accès, la table de hash prend plus de place en mémoire que les autres structures de données et les clefs ne sont pas triés. De plus il faut que la fonction de hashage soit bien choisi pour qu'il y ait peu de collisions.

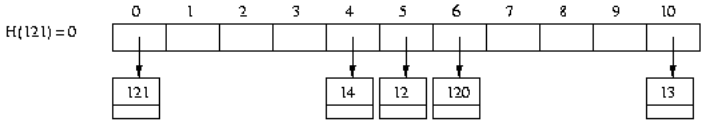


FIGURE 2 – Table de hashage

Decision : Le programme devra fréquement vérifier si un identificateur est un mot clé, donc la structure de donnée doit en priorité favoriser l'acces. Notre cahier des charges n'indiquant rien sur une eventuelle limitation de mémoire, on preferera utiliser une table de hashage car elle possede la meilleure rapidité d'accès.

5.2 Structure localisant les Identificateurs

Analyse des besoins : A chaque occurence est associée une paire contenant le nom du fichier et le numéro de ligne où elle apparait, il nous faut donc une structure de données pouvant représenter cette multiplicité. Si un mot clef apparait plusieurs fois sur une même ligne nous choisissons de référencer cette ligne autant de fois que le mot clef est présent. La complexité en lecture n'a que peu d'importance dans notre programme car nous devons faire un parcours complet pour afficher toutes les occurences.

Dans un soucis de réutilisabilité, on considère que l'utilisateur pourra en plus de voir apparaître les occurences sur la console, vouloir récupérer une structure de donnée représentant ces occurences.

Étude d'un arbre de liste : Nous pouvons utiliser un arbre de liste pour stocker les occurences des Identificateurs. Chaque noeud de l'arbre contiendrait en clef le nom d'un fichier source et en valeur la liste des lignes dans lesquelles le mot clef est présent. L'avantage de cette méthode est que nous stockons juste ce qu'il faut, il n'y a pas de redondances d'informations. En revanche chaque insertion de référence demandera une complexité moyenne en O(log(n)).

Étude d'une liste de pair : Chaque occurence étant une paire d'un nom de fichier et d'un numéro de ligne, nous pourrions utiliser une liste pour stocker chacunes de ces pairs. L'avantage de cette méthode est qu'étant donné que nous

lisons les fichiers séquentiellements, l'insertion des pairs se fera de manière ordonnée et donc sera de complexité O(1). Le désavantage c'est que nous stockons des doublons, le nom des fichiers où apparaissent les mots clefs.

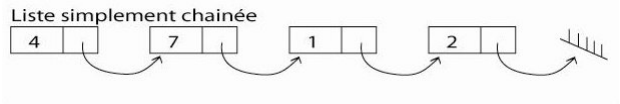


FIGURE 3 – Liste chainée

Décision : Dans un but de réutilisabilité nous préférons utliser un arbre de liste pour stocker les références des occurences. Nous evitons la redondance d'informations et ainsi il est sera plus facile de maintenir la cohérence des données si nous souhaitons appliquer des traitements dessus.