Institut Supérieur d'Informatique de
Modélisation et de leurs Applications

Campus des Cézeaux
24 avenue des Landais
BP 10125
63173 AUBIÈRE Cedex

Compte rendu de TP
Filière Génie Logiciel et Systèmes Informatiques

# Auto-complétion

TP réalisé par Nicolas PRUGNE et Antoine COLMARD

TP demandé dans le cadre du cours de compilation de M. DELEPLANQUE

# Table des matières

## Table des matières

# Table des figures et illustrations

# 1 Introduction

## 1.1 L'auto-complétion

L'auto-complétion ou complément automatique est une fonctionnalité informatique qui permet à son utilisateur de saisir plus rapidement une chaine de caractères. Au fur et à mesure que l'utilisateur saisit une chaine, un algorithme va analyser les premiers caractères tapés, et proposer à l'utilisateur un ensemble de mots qui débutent de la même manière que la chaine en cours de saisie. L'utilisateur n'a plus qu'à sélectionner la chaine correspondante au mot qu'il souhaite taper. Ainsi, il limite la quantité d'informations à saisir directement au clavier, ce qui lui procure un gain de temps important. Pour fonctionner, l'auto-complétion a besoin d'établir un dictionnaire de mots proposables à l'utilisateur. Pour ce faire, elle peut se baser sur ce que l'utilisateur a déjà saisi, ou utiliser une liste de mots préconçus qu'elle peut charger depuis un fichier texte ou une base de données par exemple.

## 1.2 Objectifs du TP

L'objectif de ce TP est donc de coder un programme capable de gérer une auto-complétion simple en console. Ainsi, il doit pouvoir lire et charger en mémoire une liste de mots à partir d'un fichier texte passé en paramètre. Dans ce fichier, chaque mot est associé à un nombre d'occurrences. Dans un cas concret, ce nombre pourrait être calculé en fonction du nombre d'apparitions d'un mot dans un texte (analysé au préalable). Cependant dans ce TP, la liste a été construite de manière empirique et non à partir de l'analyse réelle d'un texte. Après le chargement de la liste de mots, le logiciel doit proposer à l'utilisateur de saisir une sous chaine de caractères correspondant au début d'un mot. Le programme doit ensuite analyser cette sous chaine et fournir une liste de mots correspondante aux caractères qu'elle contient. Cette liste doit être affichée en fonction du nombre d'occurrences de chaque mot. Plus un mot a un nombre d'occurrences important, plus il doit apparaitre au début de la liste proposée.

Divers classes et structures de données sont nécessaires à l'implémentation d'un tel programme. La partie suivante a pour but de présenter l'analyse et la conception des classes créées afin de répondre aux spécifications du TP.

# 2 Génie logiciel

Avant d'être implémenté, le logiciel présenté ci-dessus a nécessité une phase de réflexion et conception afin de concevoir un modèle répondant à ses spécifications. Cette conception s'appuie le paradigme de la programmation orientée objet, c'est pourquoi les composants logiciels produits au cours de ce TP s'apparentent à des classes et non à de simples structures de données. La partie suivante a pour rôle de présenter ces divers composants logiciels.

## 2.1 Présentation du package UML développé

L'auto-complétion nécessite donc la gestion d'un dictionnaire de mots en mémoire. Pour ce faire, deux classes ont été développées et regroupées dans un package UML nommé `Auto-Completion` (Figure 1).

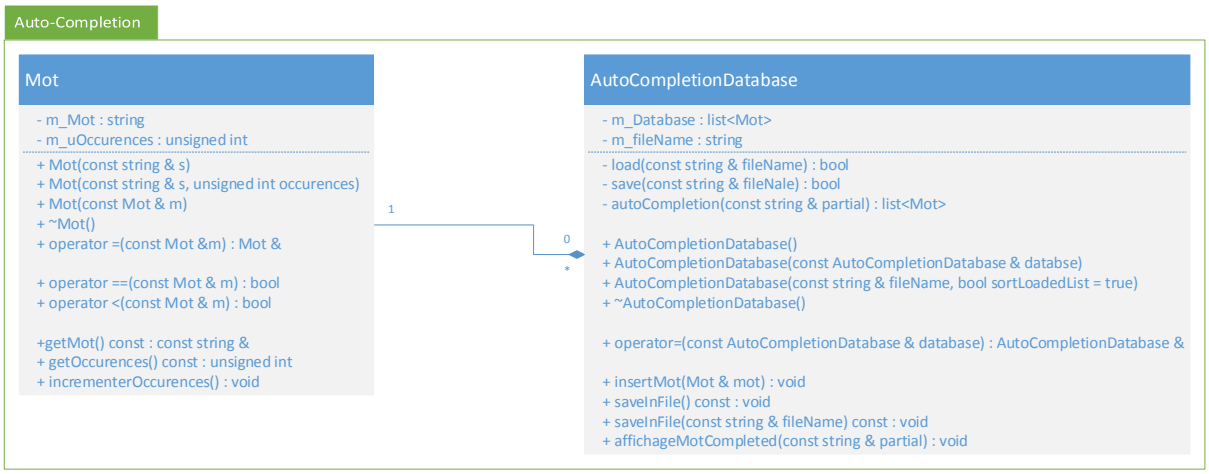


**Figure 1 - Package Auto-Completion**

Les structures de chacune des classes de ce package seront détaillées à la suite du compte rendu, cependant il est possible d'établir une analyse générale de l'architecture conçue. Tout d'abord, le package montre que le programme se base sur deux classes pour gérer son dictionnaire de mots. La première appelée `AutoCompletionDatabase` permet de référencer tous les mots connus par le programme. Elle fournit également une API basique afin de rechercher un des mots du dictionnaire qui débute de la même manière qu'une séquence de caractères passée en paramètre. Cette classe constitue le cœur du programme. Cependant, le package indique également qu'elle utilise une classe annexe afin de stocker les mots qu'elle répertorie. Cette classe baptisée `Mot` a pour rôle de stocker une chaine de caractères représentant ainsi un mot. Dans le même temps, elle associe à cette chaine un entier non signé afin de compter son nombre d'occurrences.

La classe `AutoCompletionDatabase` est donc constituée d'une liste d'objets de type `Mot`. La classe `Mot` présente également d'autres fonctions et la partie suivante a pour rôle d'expliciter clairement son rôle.

## 2.2 La classe Mot

La classe `Mot` a donc pour rôle de stocker les chaînes de caractères du dictionnaire que gère le programme tout en leur associant un entier afin de compter leur nombre d'utilisations (Figure 2).

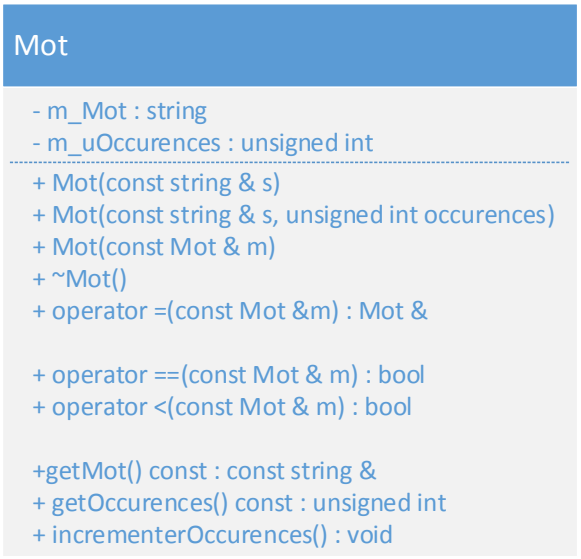


**Figure 2 - La classe Mot**

Parmi ses méthodes, la classe `Mot` compte plusieurs constructeurs dont l'un d'entre eux est un constructeur par recopie. En effet, la classe respecte la forme de Copelien et possède donc également un opérateur d'affectation et un destructeur.

Deux opérateurs de comparaison lui ont été ajoutés. Ils permettent d'effectuer des traitements algorithmiques sur la classe, comme des tris par exemple. De plus, deux accesseurs sont disponibles afin de pouvoir consulter le contenu de la chaine stockée, ainsi que son nombre d'occurrences. Enfin, une dernière méthode permet d'incrémenter le nombre d'occurrences du mot stocké.

D'une manière synthétique, on peut résumer le rôle de la classe `Mot` à celui d'un conteneur associant une chaine de caractères à un compteur. Elle sert principalement comme structure de stockage à la classe `AutoCompletionDatabase` dont le modèle est expliqué à la suite de cette partie.

## 2.3 La classe AutoCompletionDatabase

La classe `AutoCompletionDatabase` est donc au cœur du mécanisme d'auto-complétion implémenté au cours de ce TP. Pour remplir son rôle, elle stocke une liste d'objets de type `Mot` ainsi, qu'une chaine de caractères correspondant au d'un nom d'un fichier. Cet attribut lui permet de lire et d'écrire dans un fichier texte les mots qu'elle contient (Figure 3).

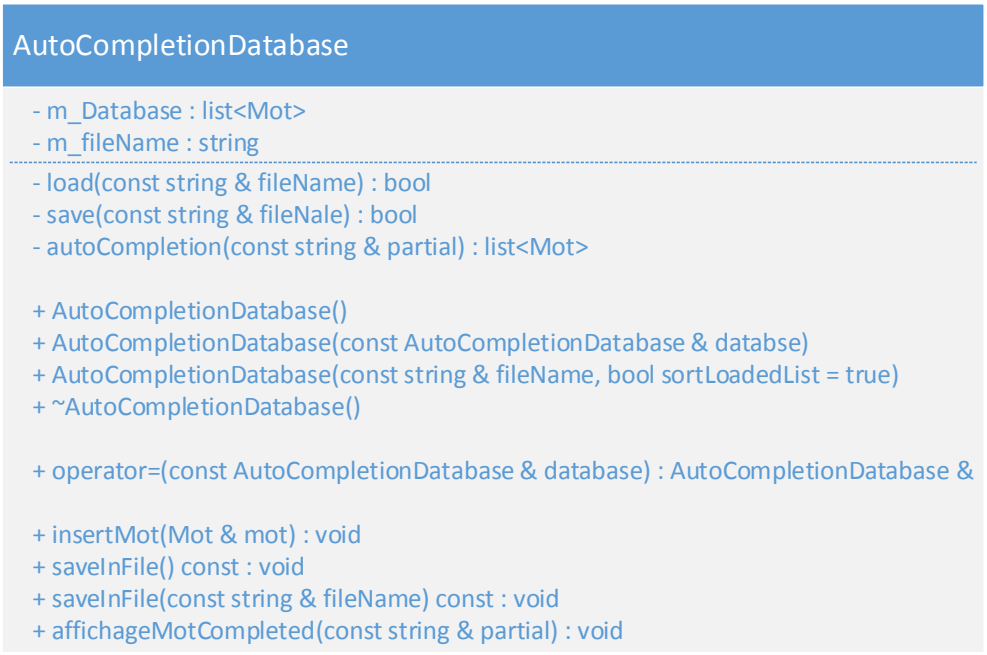


**Figure 3 - La classe AutoCompletionDatabase**

La classe `AutoCompletionDatabase` fournit trois constructeurs. Le premier permet de construire un objet à partir d'un fichier texte dont le nom est passé en paramètre. Cela permet d'initialiser la base de mots à partir d'un fichier existant. Le second constructeur est un constructeur par défaut (sans paramètre). En effet, il est également possible de partir d'une base vide et d'ajouter au fur et à mesure de l'exécution du programme, des mots dans cette base. Cet ajout peut être fait via la méthode `insertMot`. Enfin le troisième et dernier constructeur permet de recopier une base existante lors de l'instanciation (constructeur par recopie). La classe contient également un opérateur d'affectation et un destructeur dans le but de respecter une nouvelle fois la forme de Copelien, facilitant ainsi son utilisation lors de l'implémentation du programme.

La classe `AutoCompletionDatabase` possède également deux méthodes destinées à lire et sauvegarder son contenu dans un fichier texte. Enfin, la méthode `affichageMotCompleted` permet de réaliser, à proprement parler, le traitement nécessaire à l'auto-complétion. Cette méthode se destine à afficher dans la console une liste de mots débutants de la même manière que le paramètre qu'on lui passe. Pour ce faire, la classe utilise la liste de mots qu'elle contient ainsi qu'un algorithme de recherche détaillé dans la partie suivante.

# 3 Algorithmique

Afin de proposer une fonctionnalité d'auto-complétion performante, le programme se base sur un algorithme de recherche simple mais efficace. Tout d'abord, une précision doit être apportée par rapport à la manière dont sont stockés les mots dans la liste de la classe `AutoCompletionDatabase`. En effet, celle-ci gère une liste de mots triés par ordre alphabétique. Cet ordre permet de garantir une recherche plus efficace lors de l'auto-complétion, car il est possible d'utiliser un algorithme de recherche dichotomique.

Le tri est effectué d'une part lors du chargement de la base de mots depuis un fichier texte. Une fois la totalité du fichier parsé, le programme lance un algorithme de tri fusion sur la liste de

mots chargée en mémoire. C'est à ce moment que les opérateurs de comparaisons de la classe `Mot`, présentés plus haut, sont utilisés. D'autre part, lors de l'insertion d'un nouveau mot dans la base, la classe `AutoCompletionDatabase` utilise un algorithme d'insertion triée de manière à ne pas altérer l'ordre de stockage des mots. La partie suivante présente succinctement l'algorithme mis en œuvre pour effectuer cette insertion.

## 3.1 Insertion triée

L'ajout de nouveaux mots dans la base est géré via la méthode `insertMot` de la classe `AutoCompletionDatabase`. Cette méthode utilise un algorithme d'insertion triée afin de permettre l'ajout de nouveaux mots dans la base en respectant l'ordre alphabétique de stockage. Pour ce faire, elle va rechercher la position du nouveau mot dans la liste que contient la classe `AutoCompletionDatabase` grâce à un algorithme de recherche dichotomique. Puis une fois la position localisée, la méthode va soit insérer le nouveau s'il n'est pas déjà présent dans la base, soit incrémenter son nombre d'occurrences s'il existe déjà (Algorithme 1).

Algorithme 1 : insertMot

Entrée:

`mot` : un nouveau mot à ajouter dans la base

Pseudocode :

```
Procédure insertMot
      database := liste de mots chargée, puis triée, à partir d'un fichier
      texte;
      positionMot := recherche dichotomique dans la liste database du mot à
      insérer;

      [Test si le mot existe déjà]
      Si contenu(database, positionMot) = mot alors:
            Incrémenter le nombre d'occurence du mot recherché;
      Sinon
            insérer(database, posistionMot, mot);
      fsi;
fin insertMot;
```

Les fonctions `contenu()` et `insérer()` permettent respectivement, d'accéder au contenu d'une liste à une position donnée, et d'insérer un nouveau mot dans une liste à une position donnée. L'implémentation réelle de la fonction est disponible en annexe.

La question de l'insertion étant résolu, il est temps d'aborder le cœur de ce TP : l'auto-complétion. La partie suivante explicite l'algorithme qui se cache derrière cette fonctionnalité.

## 3.2 Mécanisme d'auto-complétion

L'auto-complétion est gérée via la méthode privée `autoCompletion` de la classe `AutoCompletionDatabase`. Celle-ci renvoie une liste d'objets de type Mot triée par rapport à leur nombre d'occurrences. Pour ce faire, elle va tout d'abord repérer et mémoriser le premier mot

de la liste qui contient la séquence de caractères passée en paramètre. Puis à partir de ce premier élément de la liste, la fonction va rechercher parmi les mots suivants, le premier qui ne contient plus la séquence recherchée et le mémoriser. De cette manière, elle sait qu'entre le premier et le second mot qu'elle vient de mémoriser, tous les mots comportent la séquence de caractères recherchée. Il ne lui reste plus qu'à faire un tri de ces mots par rapport à leur nombre d'occurrences (Algorithme 2).

Algorithme 2 : autoCompletion

Entrée :

`partial` : une chaine de caractères correspondant au début d'un mot à rechercher

Pseudocode :

```
Fonction autoCompletion
    database := liste de mots chargée, puis triée, à partir d'un fichier
    texte;
    autoCompletionList := ∅;

    [Recherche de l'ensemble des mots contenant la chaine modèle]
    first := Recherche dichotomique dans database sur le premier élément de
    la liste qui contient la chaine partial;
    last := first;

    Tant que last != fin(database) et que le mot pointé par last contient
    la sous chaine partial faire:
      last := suivant(last);
    fin;

    [Tri des mots sélectionnés par rapport au nombre d'occurrences]
    mot := first;
    Tant que mot != last faire :
      positionMot := Recherche dichotomique dans autoCompletionList par
      rapport au nombre d'occurence du mot;
      inserer(mot, positionMot, autoCompletionList);
    fin;

    Retourner autoCompletionList;
fin autoCompletion;
```

Les fonctions `fin()` et `suivant()` permettent respectivement, de vérifier que l'on n'a pas atteint la fin d'une liste de mots, et de passer à l'élément suivant dans un liste chainée. La fonction `insérer()`, quant à elle permet, d'insérer un nouveau mot dans une liste à partir d'une position donnée. L'implémentation réelle de la fonction est disponible en annexe.

# 4 Résultats

L'implémentation réelle du TP a donc été réalisée en C++ et la compilation effectuée dans un environnement Windows. Ainsi le compilateur utilisé pour ce jeu d'essais est celui de la suite logicielle Visual Studio. Plus précisément, il s'agit du compilateur de Visual C++ 2012. Les tests ont été menés sur une version release du programme développé et sur une machine dotée d'un processeur Intel Core i7-QMCPU @2,30Ghz.

Le test a été mené sur une base de plus de 120000 mots[1]. Pour ce faire, le programme a dans un premier temps, parsé un fichier texte comportant tous ces mots associés à un nombre d'occurrences fictif. Puis dans un second temps, il a recherché une liste de mots correspondante à une séquence de caractères passée en paramètre. Ces deux phases d'exécution ont été chronométrées (Figure 4).

```
Chargement de la base de donnees "../data/dat
Chargement termine (1.045s).
Recherche de mots commencant par "verbal" ...
verbal
verbale
verbales
verbalement
verbalisme
verbaliser
verbalisation
verbalisant
verbalise
Recherche terminee (0.006s).
Dechargement de la base de donnees ...
Appuyez sur une touche pour continuer...
```

Figure 4 - Sortie console du programme

Le chargement met donc 1,045 seconde à être effectué. Puis la recherche des mots commençant par la chaine de caractères « verbal » met 0,006 seconde à se terminer et le programme affiche les neufs mots qu'il a pu trouver dans la base.

# 5 Discussion

En termes de complexité et de vitesse d'exécution, le mécanisme d'auto-complétion implémenté dans ce TP donne de bons résultats. En effet, la fonctionnalité se base essentiellement sur une recherche dichotomique pour assurer la production de sa liste de mots. Or, la complexité d'une telle recherche croit de manière logarithmique avec la quantité de données scrutées. C'est pourquoi, en termes de complexité, l'algorithme d'auto-complétion produit ici est en $O(\ log\ (n))$.

Cependant, l'une des critiques qui peut être émise vient de la quantité de mémoire vive consommée par le programme. En effet, la classe principale du programme utilise une simple liste chainée pour gérer les mots qu'elle stocke. De ce fait, étant donné que de nombreux mots possè-

[1] Le fichier `data.txt`, présent dans le répertoire `auto_completion/src/data` du projet, comporte très exactement 128 918 mots.

dent des racines communes, comme verbal et verbalement par exemple, de nombreux caractères sont stockés de manière redondante. Une structure sous forme d'arbre où chaque nœud est un caractère pourrait pallier ce problème. La recherche ne perdrait pas en efficacité et la quantité de mémoire nécessaire diminuerait. De plus, stockée sous cette forme, la liste de mots ne nécessiterait plus de tri. Néanmoins, l'insertion de nouveaux mots dans la base deviendrait plus complexe.

# 6 Annexes

L'intégralité du projet est disponible à l'adresse suivante : https://github.com/LebonNic/auto_completion. Le reste des annexes se compose essentiellement d'extraits provenant du code source du programme développé. Les algorithmes de recherche dichotomique et de tri fusion sont assurés grâce à des méthodes disponibles dans la STL du C++.

## 6.1 Implémentation C++ de la procédure d'insertion triée

```
/**
 * @brief AutoCompletionDatabase::insertMot Insert un mot dans la database, incrémente
ses occurences s'il existe déjà
 * @param mot Mot à insérer
 */
void AutoCompletionDatabase::insertMot(Mot &mot)
{
    std::list<Mot>::iterator insertIterator = std::lower_bound(this-
    >m_Database.begin(), this->m_Database.end(), mot);
    // La liste n'est pas vide ou mot recherché existe déjà dans la database, on in-
    crémente la valeur associée
    if (insertIterator != this->m_Database.end() && (*insertIterator) == mot)
    {
        insertIterator->incrementerOccurences();
    }
    // Le mot n'existe pas dans la database, insertion
    else
    {
        this->m_Database.insert(insertIterator, mot);
    }
}
```

## 6.2 Implémentation C++ de la fonction autoCompletion

```
/**
 * @brief AutoCompletionDatabase::autoCompletion Récupère la liste des complétion de
la chaîne partielle triés sur leur nombre d'occurences
 * @param partial Chaîne partielle à compléter
 * @return Liste des complétions triées par occurences
 */
std::list<Mot> AutoCompletionDatabase::autoCompletion(const std::string &partial)
{
       std::list<Mot> autoCompletionList;
       std::list<Mot>::iterator first = std::lower_bound(this->m_Database.begin(),
       this->m_Database.end(), Mot(partial));
       std::list<Mot>::iterator last = first;
       unsigned int length = (unsigned int) partial.length();

       while(last != this->m_Database.end() && last->getMot().substr(0,length) == par-
       tial)
              ++last;

       for(std::list<Mot>::iterator it = first; it != last; ++it)
       {
              std::list<Mot>::iterator insertIterator =
              std::lower_bound(autoCompletionList.begin(), autoCompletionList.end(),
              *it, compareOccurences);
              autoCompletionList.insert(insertIterator, *it);
       }

    return autoCompletionList;
}
```