Compte-rendu du TP C++ 3
Spécification et conception

Marc Gagné Selma Nemmaoui

Décembre 2015

Préambule

Le TP C++ 3 a pour objectif la création d'un programme capable de traiter des fichiers de log générés par Apache pour en extraire l'historique de parcours ; le programme peut ensuite afficher les informations extraites sur la sortie standard, les filtrer selon certains critères ou générer un fichier Graphviz représentant les parcours entre documents.

1 Spécifications

Pour répondre au cahier des charges, notre application doit respecter certaines spécifications que nous avons établies et qui sont listées ci-dessous.

1.1 Définitions

Les termes ci-dessous sont utilisés dans le reste du document, et sont donc préalablement définis ci-dessous :

- **Document :** une ressource se trouvant sur le serveur local et qui a soit été atteint par un utilisateur dans le fichier de log, soit été la source d'une visite sur un autre document. Un document est identifié par son URI relative à la racine du serveur local. Toutes les ressources venant d'un autre serveur que le serveur local sont considérées comme appartenant à un document virtuel, identifié par la fausse URI `*`.
- **Serveur local :** le serveur qui a généré le fichier de log Apache. Doit être associé à une URL pour que tout document référant venant de ce serveur soit correctement identifié.
- **Popularité :** la popularité d'un document est liée au nombre de fois qu'il a été atteint à partir d'un autre document ; plus ce nombre est grand, plus le document est populaire.

1.2 Spécifications générales

1.3 Spécifications détaillées

Les spécifications détaillées sont des spécifications qui impliquent des tests (réalisés dans le répertoire `tests/`) pour vérifier qu'elles sont respectées. Elles sont présentées dans la table de l'annexe 1.

2 Conception

2.1 Classes principales

L'application fonctionne principalement grâce à une classe centrale, la classe `HistoryManager`, qui est initialisée par le programme principal (dans la fonction `main`) et qui utilise toutes les autres classes pour traiter les fichiers de log.

Les présentations détaillés des méthodes sont disponibles dans les commentaires des interfaces de chaque classe.

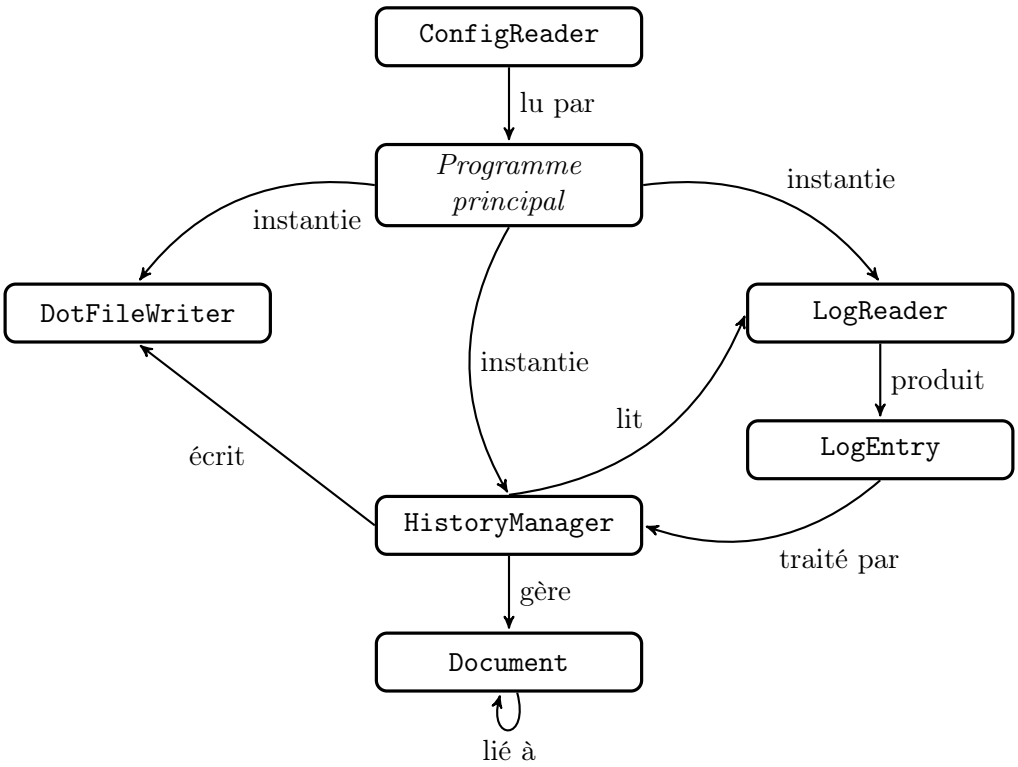

FIGURE 1 – Intéractions entre les classes principales

2.1.1 `ConfigReader`

2.1.2 `Document`

La classe `Document` représente un document du fichier de log fourni en entrée. Il permet de compter le nombre de fois qu'il a été atteint à partir d'un autre document (`AddLocalHit()` et `GetLocalHits()`), ainsi que le nombre de fois qu'il a été utilisé pour atteindre un autre document (`AddRemoteHit(documentId)` et `GetRemoteHits()`. Un document doit être initialisé avec son URI, qui l'identifie de manière unique (`GetUri()`). Un `Document` peut également déterminer s'il est "supérieur" à un autre document en comparant leurs popularités respectives.

Un `Document` est donc essentiellement une structure pour stocker des données sur un document, avec quelques méthodes pour faciliter sa mise à jour.

2.1.3 `DotFileWriter`

La classe `DotFileWriter` permet de créer un graphe destiné à être écrit dans un fichier DOT, utilisable par Graphviz pour générer des images. La classe ne fait pas de suppositions sur la nature du graphe, et ne le considère que comme un ensemble de nœuds, de liens et d'étiquettes, afin de rendre la classe plus réutilisable. Comme elle manipule des flux de fichiers, cette classe ne peut pas être copiée, ce qui permet d'éviter des conflits d'écriture sur un même fichier.

Un des avantages de cette classe est qu'elle peut être utilisée pour générer un graphe, puis l'écrire dans plusieurs fichiers, et même changer de graphe sans avoir à en créer une nouvelle instance. L'écriture d'un graphe se fait donc en deux étapes, pouvant être réalisé dans l'ordre désiré :

- **Le fichier DOT doit être ouvert en mode écriture :** si un fichier avait déjà été ouvert (`Open(filename)`), il est préalablement fermé (`Close(filename)`) avant d'être ouvert pour écriture subséquente.
- **Le graphe doit être initialisé :** un nouveau graphe doit être initialisé en spécifiant son nombre de nœuds (`InitGraph(graphNodes)`). Le programme doit ensuite ajouter tous les nœuds (`AddNode(id, label)`), identifiés par leur ID et étiquetés avec leur nom, puis ajouter tous les liens entre nœuds (`AddLink(sourceId, targetId, linkLabel)`), qui ont eux-mêmes une

étiquette.

Une fois que les deux étapes ont été réalisées et que le graphe est finalisé, le résultat peut être écrit dans le fichier DOT (`Write()`), au format de Graphviz.

Les structures de données utilisées pour stocker les nœuds et liens sont détaillées dans la section suivante.

2.1.4 `HistoryManager`

La classe `HistoryManager` est responsable de la gestion de l'historique de parcours du serveur local, obtenu à partir d'un fichier de log Apache au format attendu (`FromFile(logFile, excludedIntensions, startHour, endHour)`), par le biais d'une instance de `LogReader` déjà ouverte.. Chaque ligne du fichier de log est lue une par une, le programme décidant à la lecture s'il va conserver cette ligne ou non, selon les critères spécifiés par les options `-e` et `-t` : ceci permet d'économiser en mémoire et en temps de recherche par la suite, puisque l'alternative serait de tout conserver et de trier à chaque recherche d'information, une opération coûteuse.

Une fois que toutes les informations ont été extraites (une opération relativement lente, puisqu'il faut lire beaucoup de chaînes de caractères), elles sont stockées sous forme d'un ensemble de `Document`s et peuvent soit être affichés sous la forme d'une liste des documents les plus populaires (`ListDocuments(max)`), soit sous la forme d'un fichier Graphviz au format DOT (`ToDotFile(dotFile)`), par le biais d'une instance de `DotFileWriter` déjà ouverte.

Les structures de données utilisées pour stocker les documents sont détaillées dans la section suivante.

2.1.5 `LogEntry`

La classe `LogEntry` représente une ligne valable du fichier de log, réduite aux informations qui intéressent le reste du programme, à savoir l'heure (`GetHour()`), la méthode de la requête HTTP (`GetRequestMethod()`), l'URI du document demandé, épurée de ses paramètres ou d'un numéro de port éventuels (`GetRequestUriConverted()`), l'extension du document demandé, si possible (`GetRequestUriExtension()`), le code de statut de la requête (`GetStatusCode()`) et l'URL du document demandeur, convertie en URI si le document est local ou en `*` sinon (`GetRefererUrlConverted(local)`).

Cette classe sert d'intermédiaire entre la classe `LogReader` et la classe `HistoryManager`, puisqu'elle permet d'extraire les informations utiles à la création des `Document`s. Ses informations peuvent être extraites directement d'un flux de données ; si la ligne n'est pas valable, une exception est lancée.

2.1.6 `Logger`

La classe `Logger` est une classe utilitaire permettant d'afficher des messages à l'utilisateur. Elle permet d'utiliser des couleurs (définies dans `TerminalColor`) et d'afficher des messages à plusieurs niveaux (`Debug(args)`, `Error(args)`, `Info(args)` et `Warning(args)`) avec un nombre variable d'arguments. Ceci simplifie grandement l'écriture de messages de debug, d'erreur ou d'information ailleurs dans le programme.

2.1.7 `LogReader`

La classe `LogReader` permet de lire des fichiers de log générés par Apache et d'en extraire les informations importantes. Comme elle manipule des flux de fichiers, cette classe ne peut pas être copiée, ce qui permet d'éviter des conflits de lecture sur un même fichier. Pour commencer à lire un fichier, il faut l'ouvrir au préalable (`Open(filename)`), puis récupérer une instance de `LogEntry` par ligne (`ReadLine(entry)`, sachant que si la ligne n'est pas utilisable une exception est lancée), tant que la fin du fichier n'est pas atteinte (`Eof()`). Le fichier doit ensuite être fermé (`Close()`), puis le lecture est prêt à traiter un autre fichier de log. Cette classe permet notamment de compter toutes les lignes qui sont lues, ce qui permet de rendre tout message d'erreur plus utile.

2.2 Structures de données

Puisque l'application a besoin de gérer des collections d'entités liées entre elles (c'est-à-dire les documents et les passages d'un document à un autre), la structure la plus adaptée pour modéliser ceci est le graphe. Notre programme réalise donc deux implémentations de graphe, de deux manières différentes : une dans `HistoryManager`, qui doit favoriser une mise à jour fréquente et qui est de taille inconnue, et une autre dans `DotFileWriter` qui doit permettre un parcours rapide et être de taille connue. Le programme commence par construire la première version, puis l'exporte vers la deuxième si l'utilisateur a demandé de générer un fichier DOT.

Le premier graphe fonctionne grâce à un tableau `documents` de taille variable contenant les `Document`s (implémenté avec `std::vector`) et à un dictionnaire `documentsByName` associant les URI des documents à leur index dans `documents` (implémenté avec `std::unordered_map`). Comme le tri n'est nécessaire que pour une seule opération réalisée à la toute fin du programme (lors de la liste des documents), ces structures ne sont pas triées, ce qui facilite l'insertion, la modification et la recherche par URI, qui est l'identifiant utilisé dans les fichier de log. L'idée est donc d'utiliser `documentsByName` pour trouver l'index du document recherché à partir de l'URI (une opération de temps généralement $\mathcal{O}(1)$, grâce aux tables de hachage, et au pire de temps $\mathcal{O}(n)$), puis de retrouver ensuite le document dans `documents` par son index (une opération de temps $\mathcal{O}(1)$). De plus, pour stocker les visites d'un document à un autre, il est plus efficace d'utiliser les index des documents dans `documents` que leur URI, d'où l'intérêt de séparer `documents` et `documentsByName`.

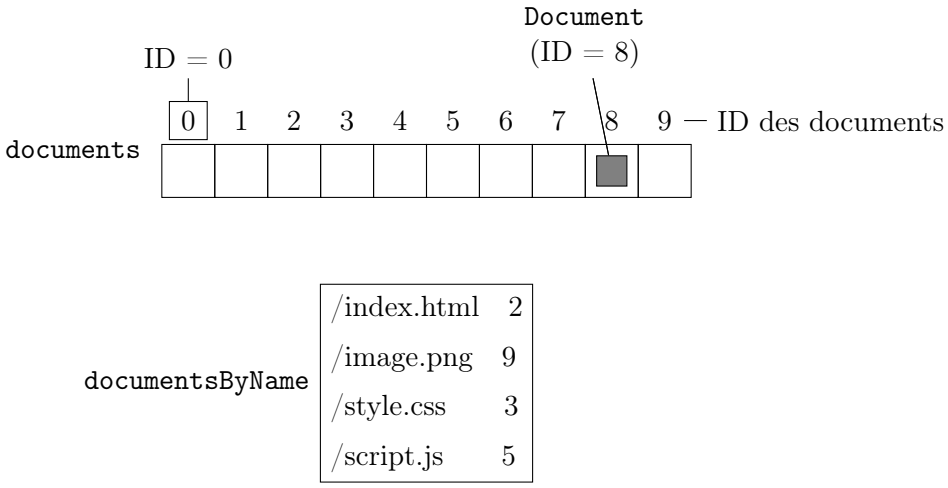

FIGURE 2 – Fonctionnement du graphe de `HistoryManager`

Le deuxième graphe est plus proche du concept de graphe car il reflète le format du fichier Graphviz, séparé en nœuds et en liens. Il consiste en une liste simplement chaînée `links` de liens (implémentée avec `std::forward_list`) et un tableau fixe `nodes` (implémenté par un tableau C alloué dynamiquement) qui liste les étiquettes correspondant aux ID de nœuds. Dans ce modèle, chaque document est associé à un nœud, l'ID du nœud étant la position du document dans `documents` et l'étiquette étant son URI, et les nombres de visites d'autres documents à partir de celui-ci sont associés à des liens dont l'étiquette est le nombre de visites. Ce graphe se construit en temps $\mathcal{O}(n^2)$ à partir du précédent, puisqu'il faut itérer sur chaque document de `documents`, puis sur chaque visite réalisée à partir de ce document vers les autres documents. Cependant, une fois construit, il est très rapide d'écrire le fichier DOT correspondant puisqu'il y a une correspondance directe entre le format du fichier et la représentation interne de ce graphe : il n'a qu'à itérer sur tous les documents, puis à itérer sur tous les liens pour construire le fichier.

Puisque nous avons deux graphes pour représenter les mêmes données, il y a simultanément un gaspillage de la mémoire par redondance d'informations et une petite perte de performance lors de la conversion du premier vers la deuxième. Cependant, le premier est optimisé pour sa situation, qui nécessite beaucoup de recherches par URI, tandis que le deuxième, qui n'est utilisé que dans certains cas, est optimisé pour la sienne, qui nécessite de parcourir rapidement la liste de nœuds et de liens. Les gains

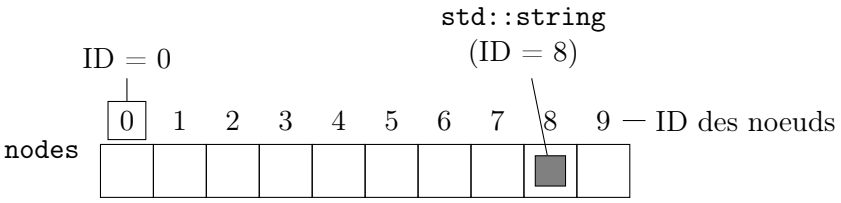

FIGURE 3 – Fonctionnement du graphe de `DotFileWriter`

en performance sur ces aspects compensent donc les désavantages liés au dédoublement des données (sans oublier que ceci permet également à la classe `DotFileWriter` de fonctionner indépendamment du reste du programme).

Annexe 1 - Liste des spécifications détaillées

Spécification	Test
L'application doit s'interrompre et afficher un message d'erreur si le nom de fichier de log n'est pas spécifié.	`test_01`
L'application doit s'interrompre et afficher un message d'erreur si le fichier de log a été spécifié mais n'a pas pu être ouvert.	`test_02`
Un fichier de log vide ne devrait pas poser de problème particulier pour l'application, qui n'affichera rien en sortie.	`text_03`
Un fichier de log "normal", avec plus de 10 documents, doit pouvoir être traité sans problèmes ; les 10 documents les plus populaires seront affichés dans l'ordre décroissant de leur popularité.	`test_04`
Un fichier de log "normal", avec moins de 10 documents, doit pouvoir être traité sans problèmes ; tous les documents atteints plus d'une fois devront être affichés dans l'ordre décroissant de leur popularité.	`test_05`
Les lignes qui ne sont pas conformes à nos attentes dans un fichier de log doivent être ignorées et un message d'avertissement doit être affiché un l'utilisateur.	`text_06`
L'option `-e` doit permettre d'éliminer tous les documents de certains types (par défaut, les images, les scripts et les feuilles de style) ; elle doit fonctionner même s'il y a des paramètres à la suite de l'extension de fichier.	`test_07`
L'option `-t` doit permettre d'éliminer toutes les entrées ayant eu lieu à un certaine heure de la journée ; le décalage horaire n'est pas pris en compte.	`test_08`
Si l'option `-t` est utilisée avec un paramètre qui n'est pas compris entre 0 et 23, l'application doit s'interrompre avec un message d'erreur.	`text_09`
L'option `-g` doit permettre de générer un fichier DOT à partir duquel un graphie Graphviz peut être généré ; tous les documents externes au serveurs doivent être représentées par `*`.	`test_10`
L'application doit fonctionner avec toutes les options précédentes activées en même temps.	`test_11`
L'application doit pouvoir lire le fichier de configuration optionnel `tp-oo_3.cfg` et remplacer les valeurs par défaut par les valeurs dans le fichier ; tous les tests précédents vérifient également que l'application peut fonctionner sans le fichier de configuration.	`text_12`
L'application doit pouvoir lire des fichiers volumineux, de plusieurs dizaines de Mo en taille.	`test_13`

Table 1 – Spécifications et tests associés