Rapport d’avancement : Bakefile

Marco Orfao
Maxence Raymond

Sommaire

I. Introduction

1. Sujet

Le Makefile est un utilitaire de compilation servant à maintenir à jour des fichiers en fonction de leur sources via la commande “make” en compilant uniquement les fichiers sources ayant besoin d’être recompilés.
Le but principal de ce projet est donc de programmer un utilitaire de compilation appelé “Bakefile” ayant une structure simplifiée mais similaire au Makefile.

II. Bakefile

1. Fonctionnalités

Le Bakefile comportes différentes options qui peuvent être assignées à la commande “bake” qui sont proposées à l’utilisateur. Il y a tout d’abord l’option “--help” qui permet d’afficher dans le terminal certaines informations sur le Bakefile ainsi que la liste des différentes commandes disponibles avec une brève description de leur comportement (voir Figure 1.0). Il y a ensuite l’option “--version” qui permet d’afficher dans le terminal la version actuelle du Bakefile (voir Figure 1.1).

Il y a enfin l’option “-d” qui permet à l’utilisateur d’activer l’option de debug du Bakefile qui va expliciter les actions réalisées par le programme (voir Figure 1.2) permettant de mieux situer un problème si nécessaire.
Les principales similitudes du Bakefile par rapport au Makefile se trouve dans la rédaction du fichier Bakefile. L’utilisateur a la possibilité de créer des variables de la même manière que dans un Makefile (voir Figure 1.3) et de créer des cibles avec les dépendances et la liste de commandes qui va avec (voir Figure 1.4). Il y a ensuite le ”.PHONY”, qui se rédige de la même manière qu’une cible normale mais qui a pour but de permettre à l’utilisateur de renseigner au Bakefile quelle cible n’est pas un fichier et doit donc être traitée sans avoir à rechercher si le fichier existe en mettant en dépendances du .PHONY les cibles concernées.

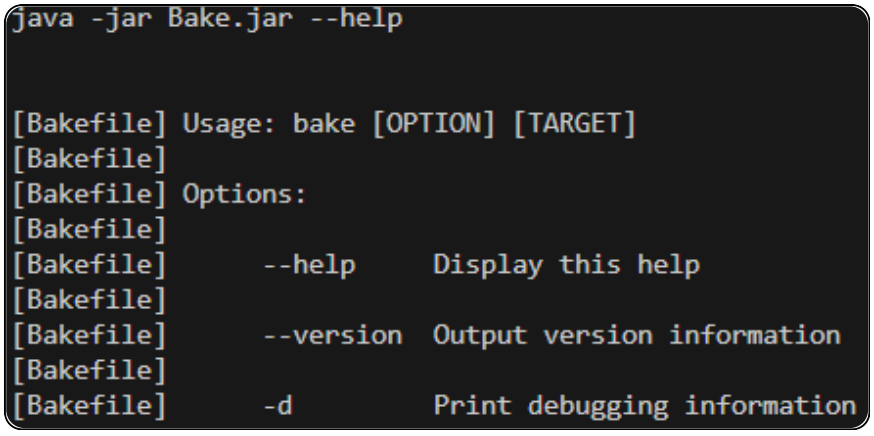

Figure 1.0 : Illustration de l’option “--help”

```
java -jar Bake.jar --version

[Bakefile] Bakefile version 1.0
```

Figure 1.1 : Illustration de l’option “--version”

```
java -jar Bake.jar -d all

[Bakefile] Building target: all
Considering rebuilding : clean
Considering rebuilding : all
[Bakefile]   echo ./build/
./build/
[Bakefile]   javac Banane.java
error: file not found: Banane.java
Usage: javac <options> <source files>
use --help for a list of possible options
```

Figure 1.2 : Illustration de l’option “-d”

```
PACKAGE = fr.variable
ENTRY = Sorty
```

Figure 1.3 : Illustration de l’écriture de variables dans un Bakefile

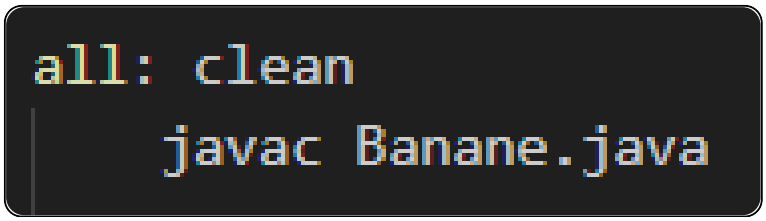

Figure 1.4 : Illustration de l’écriture d’une cible dans un Bakefile

2. Structure du programme

Le programme se compose de 17 classes réparties dans 4 packages différents : le package principal, le sous package “commandline” regroupant les classes relatifs au traitement de la ligne de commande, le sous package “structure” qui regroupe les classes servant à l’utilisation de structures de données abstraites et il y a ensuite le sous package “tree” du sous package “structure” qui regroupe les classes interagissant avec la structure de l’arbre.

fr.kanoulier.bakefile

Le package principal regroupe les sous packages ainsi que les classes qui servent au parcours et au traitement du fichier Bakefile (voir Figure 2.0). Les classes présentes dans ce package sont les suivantes :

- Bake.java
- BlockCommandExecuter.java
- LineReader.java
- Parser.java
- TargetHandler.java

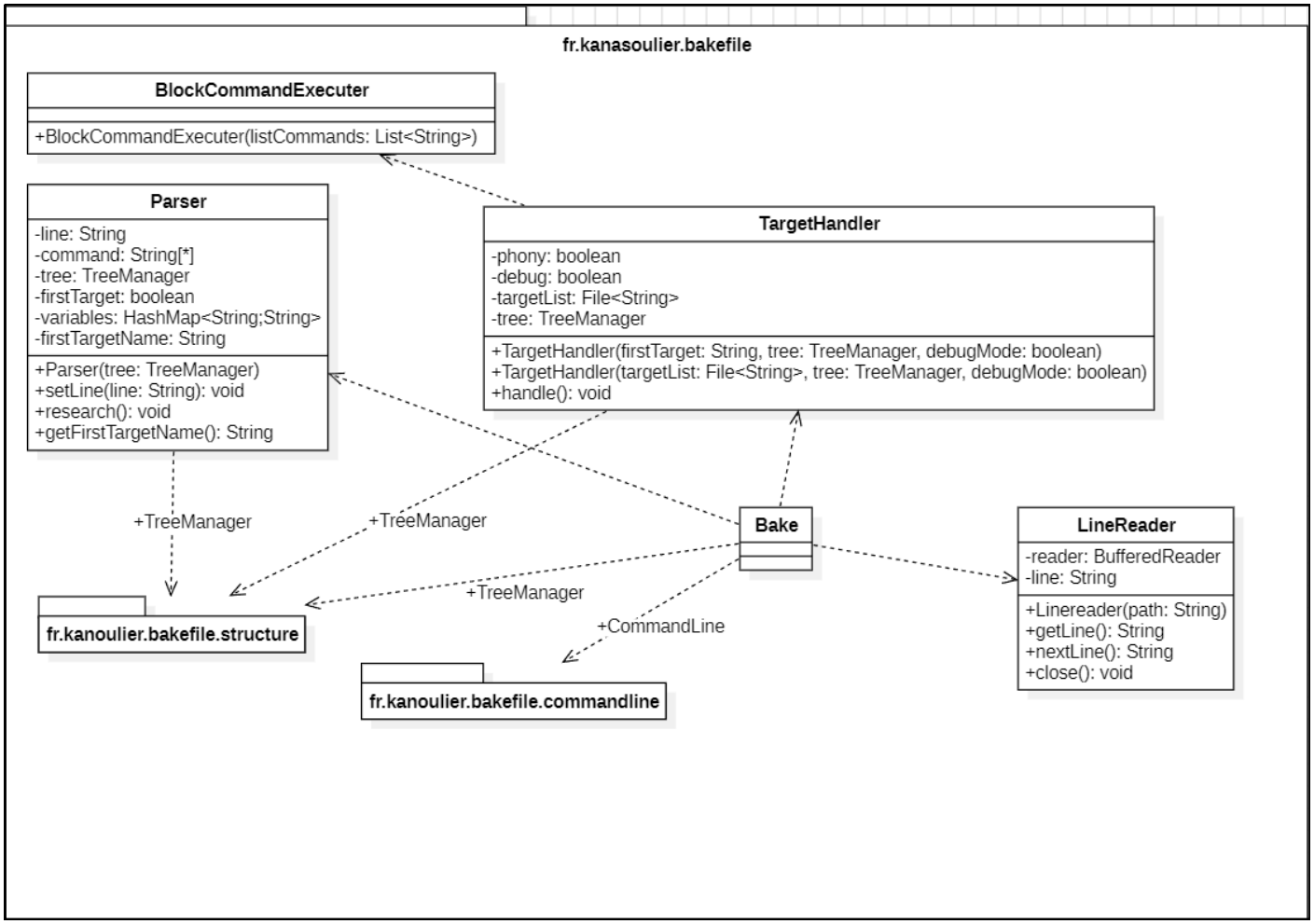

Figure 2.0 : Diagramme de classe du package principal

fr.kanoulier.bakefile.commandline

Le sous package "commandline" regroupe les classes qui servent au traitement de la ligne de commande (voir Figure 2.1). Les classes présentes dans ce package sont les suivantes :

- CommandLine.java
- Options.java

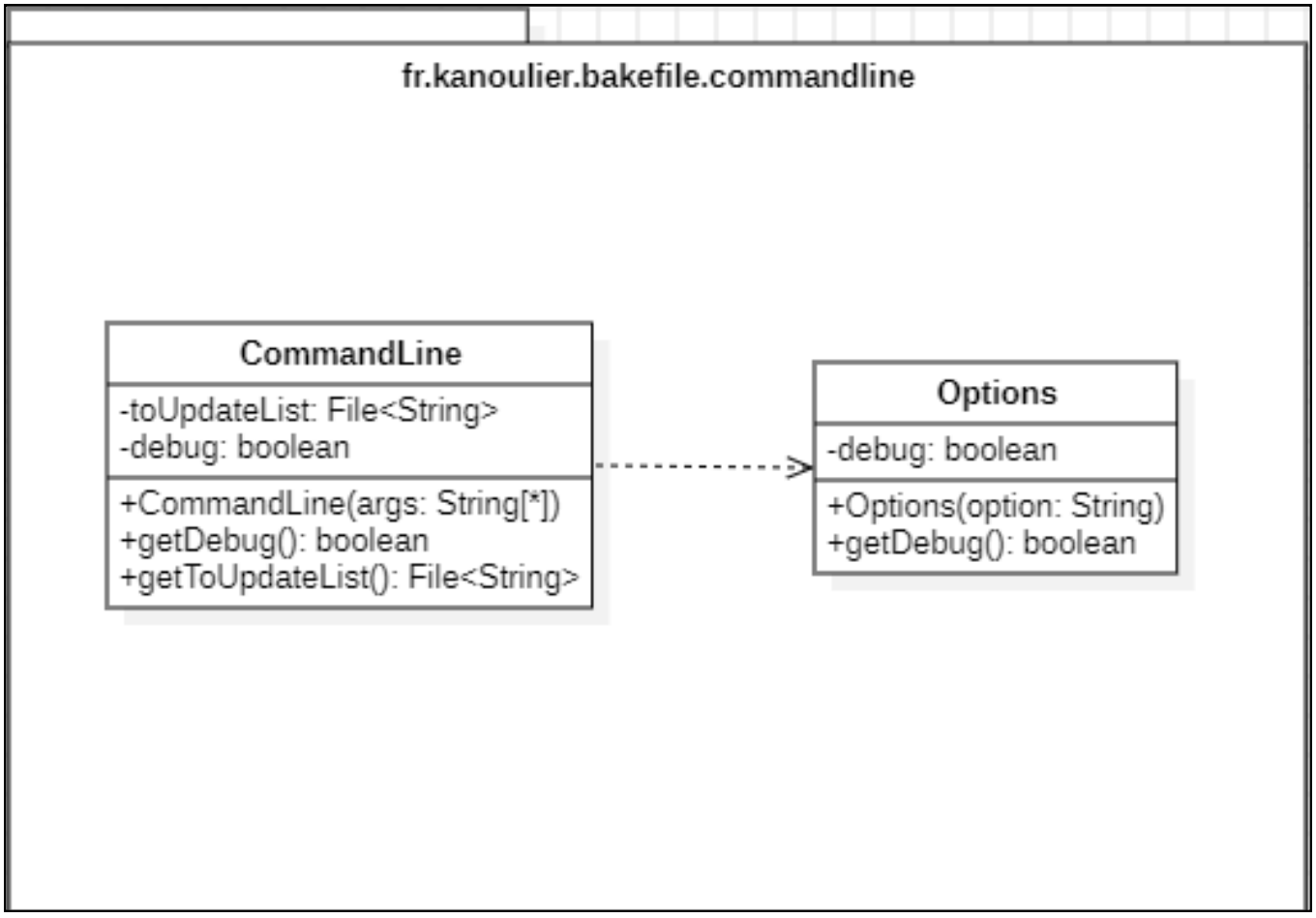

Figure 2.1 : Diagramme de classe du sous package commandline

fr.kanoulier.bakefile.structure

Le sous package “structure” regroupe les classes implémentant des structures de données ou servant d’interface avec le système de fichiers.
Les classes présentes dans ce package sont les suivantes :

- File.java
- FileChecker.java

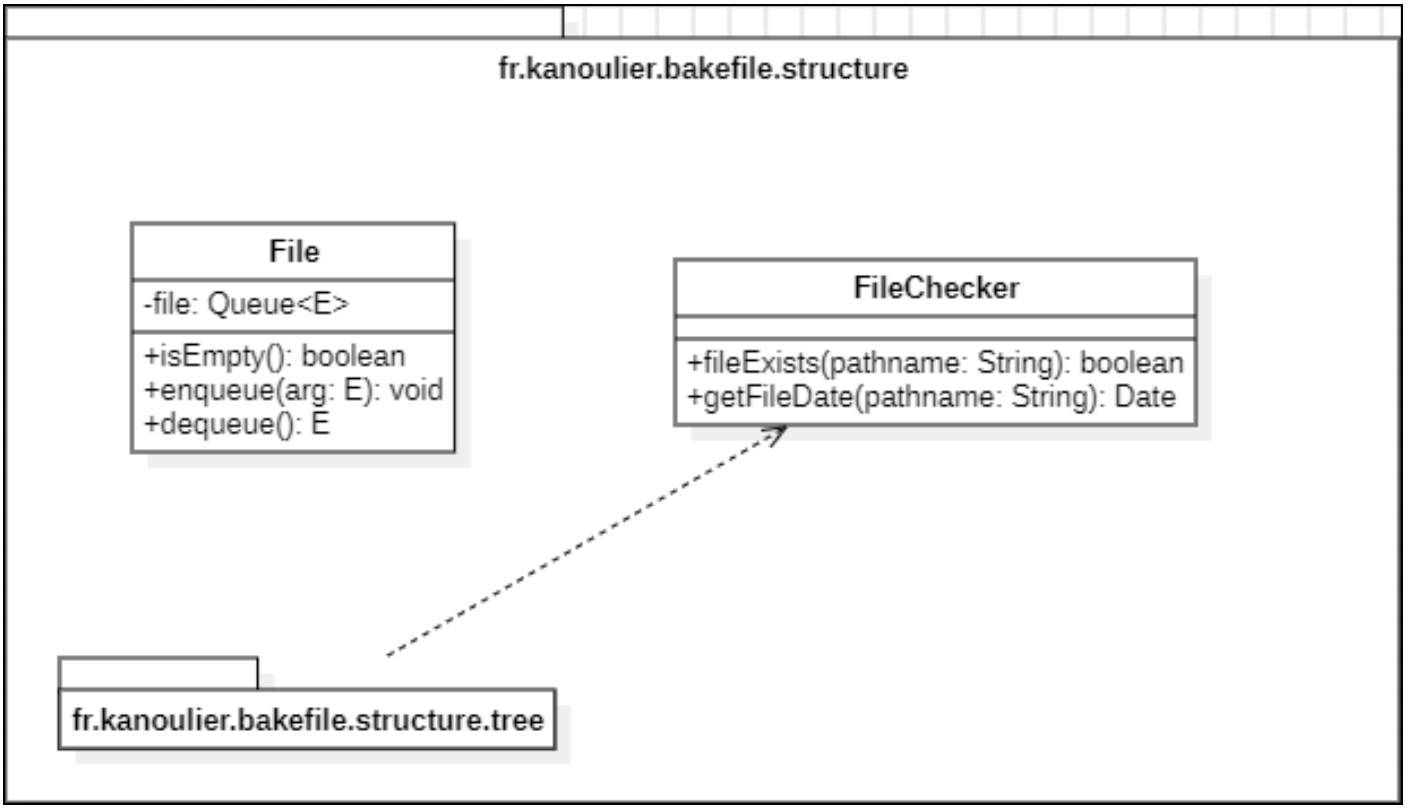

Figure 2.2 : Diagramme de classe du sous package structure

fr.kanoulier.bakefile.structure.tree

Le sous package “tree” regroupe les classes appliquant le concept de structure de données abstraite d’un arbre (voir Figure 2.3). Les classes présentes dans ce package sont les suivantes :

- AbstractNode.java
- DependencyNode.java
- DependencyTree.java
- FileNode.java
- NodeIterator.java
- TimeNodeIterator.java
- TransitNode.java
- TreeManager.java

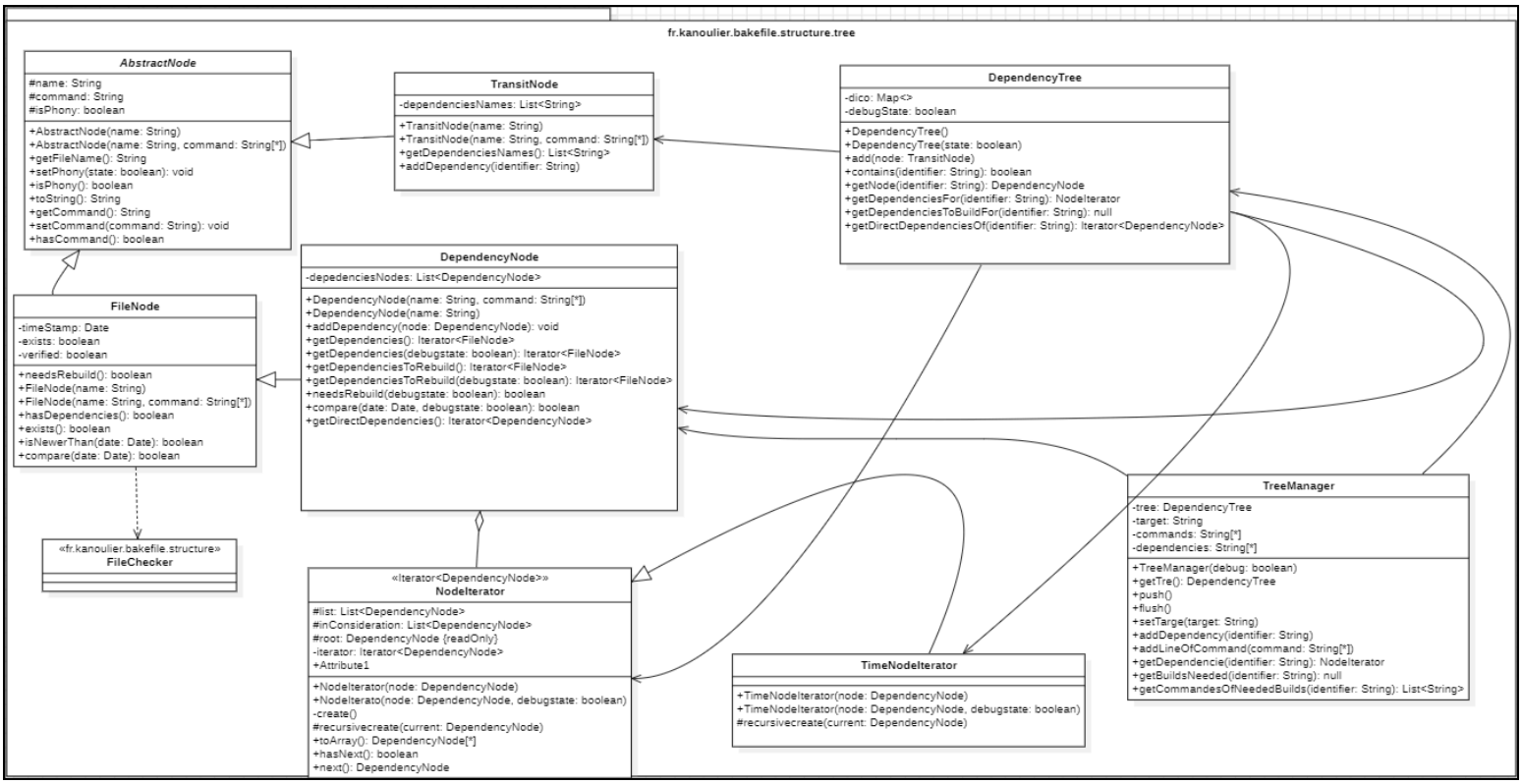

Figure 2.3 : Diagramme de classe du sous package tree

3. Graphe des dépendances

L'ensemble des classes participant au traitement des dépendances sont toutes regroupées dans le sous package fr.kanoulier.bakefile.structure.tree et elles se divisent en trois catégories : les classes des noeuds, de gestion de l'arbre et les classes relatives au parcours de l'arbre.

Les classes représentant les noeuds dans le graphe des dépendances sont les suivantes :

- AbstractNode : Une classe abstraite représentant un noeud possédant une commande, un objectif et un état "Phony".
- FileNode : Implémentation de AbstractNode prenant en compte les méthodes nécessaires au calcul des dates et de vérification avec le système de fichiers.
- DependencyNode : Extension de FileNode implémentant le code nécessaire pour gérer les dépendances.

Les classes servant à gérer l'arbre dans le graphe des dépendances sont les suivantes :

- DependencyTree : L'arbre des dépendances.
- TransitNode : Implémentation de AbstractNode servant à ajouter un noeud dans l'arbre.
- TreeManager : Interface à DependencyTree afin de faciliter la création de nœuds notamment.

Enfin, les classes relatives au parcours de l'arbre sont les suivantes :

- NodeIterator : Classe implémentant Iterator ainsi qu'une méthode toArray afin de récupérer les dépendances d'un nœud, gère les dépendances circulaires et l'unicité des nœuds retournés.

- TimeNodeIterator : Extension de NodeIterator ajoutant l'interaction avec le système de fichiers.

4. Dépendances circulaires

Voici l'algorithme qui s'occupe de la gestion des dépendances circulaires :

Tout d'abord, à l'instanciation d'un NodeIterator, un nœud de dépendances y est passé en argument. Une méthode récursive est ensuite appelé sur ce nœud. Cet méthode va tout d'abord regarder si ce nœud a déjà été traité, si c'est le cas, le nœud sera ignoré, sinon, le nœud sera ajouté dans une liste des nœuds en cours de traitement, attribut de la classe NodeIterator. Ensuite, si ce nœud possède des dépendances, la méthode est appelée sur chacune de ses dépendances directes de manière récursive. Finalement, le nœud est ajouté dans une deuxième liste, aussi attribut de la classe qui sert à contenir les nœuds à retourner de manière ordonnée. Via cet algorithme, si une dépendance circulaire se présente, le premier nœud à traiter a déjà été ajouté à la liste des nœuds en cours de traitement, si un nœud y fait donc référence, cette référence sera ignorée et la méthode pour ce lien s'arrêtera de manière prématurée. Cet algorithme garantit également que les nœuds ne se retrouveront pas en double sur la liste des nœuds retournée.

Il est techniquement possible de détecter les dépendances circulaires lors de l'exécution de l'algorithme en comparant les noeuds dans les deux listes mais cela n'a pas été implémenté.

5. Structure de données abstraites

Lors de cette SAE, les structures de données abstraites utilisées sont les suivantes :

- File : les classes Queue et ArrayDeque sont utilisées.
- Listes : ArrayList est utilisée
- Iterator
- Dictionnaire : HashMap est utilisée
- Arbre

III. Conclusions

1. Maxence Raymond

J'ai trouvé cette SAE intéressante car nous permettant d'explorer de la complexité dans un intervalle de temps relativement court. Cette SAE m'aura également permis de saisir l'importance de la documentation afin de détailler le comportement de certaines méthodes qui n'est pas toujours intuitif.

Finalement, ce travail m’aura permis d’améliorer mes compétences de travail en groupe, où chacun a pu mettre à contribution ses points forts lors de la création du programme et a pu compter sur l’aide de l’autre tout du long.

2. **Marco Orfao**

Ayant déjà commencé la création de mon propre langage de programmation, j’étais déjà assez familier avec les concepts de Lexer et de Parser qui ont pu grandement nous guider pour le parcours du fichier et la compréhension des données qu’il contient. Cette SAE m’a permis d’approfondir mes connaissances du Makefile, outil que nous utilisons depuis un certain temps mais que je ne maîtrisait pas encore correctement. Enfin, cette SAE m’a permis de raffermir mes habitudes et mes méthodes de travail que ce soit seul ou en groupe avec la mise en valeur de comportements essentiels tel que ma communication et d’autres composites importants dans le bon déroulement d’un projet en groupe. Dans l’ensemble, cette SAE a été une réussite et une source d’amusement lors de sa réalisation.