Rapport d’avancement :

Bakefile

Par :
Louis Tanchou
Gaston Chenet

Sommaire

I. Rappel du sujet

a) Description brève

Bakefile est un programme inspiré du célèbre outil GNU d'assistance à la compilation nommé « Make ». L'objectif du programme est d'exécuter des commandes de compilation seulement si le fichier cible ou une de ses dépendences a besoin d'être compilé/recompilé.

b) Fonctionnement classique

Le **Bakefile** utilise une syntaxe simplifiée du **Makefile** traditionnel.

Il se compose de blocs de données (appelés « target » dans le langage du Makefile), lui-même composé du fichier cible, de ses dépendences et bien évidemment de ses commandes de compilation.

Syntaxe d'un bloc de données :

```
<fichier> : [...dépendences]
    [commande 1]
    ...
    [commande n]
```

fichier : Le nom du fichier issu de la commande de compilation.
dépendences : Le nom des fichiers dont dépend la compilation.
commandes : Les commandes qui devront être exécutés afin de compiler le fichier.

Ainsi que de déclarations/invocation de variables, permettant de rendre le langage plus dynamique, compact et facile d'utilisation.

Déclaration et invocation d'une variable :

```
# Déclaration
<nom_var>=<valeur>

# Invocation
$(nom_var)
```

nom_var : Nom de la variable à invoquer.
valeur : contenu de la variable, ce par quoi son invocation sera remplacée.

Note : Le caractère du croisillon est utilisé pour écrire des commentaires.

Le reste de la compilation d'un fichier se fait en exécutant le fichier `bakefile.jar` suivi du nom du bloc à exécuter (ne pas mettre de nom de bloc revient à prendre le premier bloc du fichier si celui-ci n'est pas présent (et

non, il ne cherche pas pour un bloc nommé « all » car celui-ci est issu d'un biais du survivant disant que comme tous les blocs « all » sont ceux qui s'exécutent même sans paramètres c'est forcément un nom réservé, alors qu'en réalité c'est une convention de nommer le premier bloc « all » afin de définir une action par défaut)) et le paramètre `--debug` (ou `-d`) permet de passer en mode **débogage**, (toutes les visites de fichiers, comparaisons et recompilations sont affichées), si une commande retourne une erreur, le programme s'arrête immédiatement sans exécuter les commandes suivantes.

Exception : `.PHONY`

Un bloc de données sans commandes nommé `.PHONY` peut être utilisé afin d'éviter que le programme pense que le nom d'un bloc de donnée correspond au nom d'un fichier alors qu'il s'agit d'une action. Il suffit juste de rajouter les noms des blocs de données visés dans les dépendances du `.PHONY` pour que le programme ignore la recherche du fichier dans l'arborescence et passe directement à son exécution. Ce système est utile si jamais imaginons une fonction « clean » est créé et qu'un fichier « clean » soit créé pour X raison, celui-ci ne s'exécutera pas qu'une seule fois mais à chaque invocation.

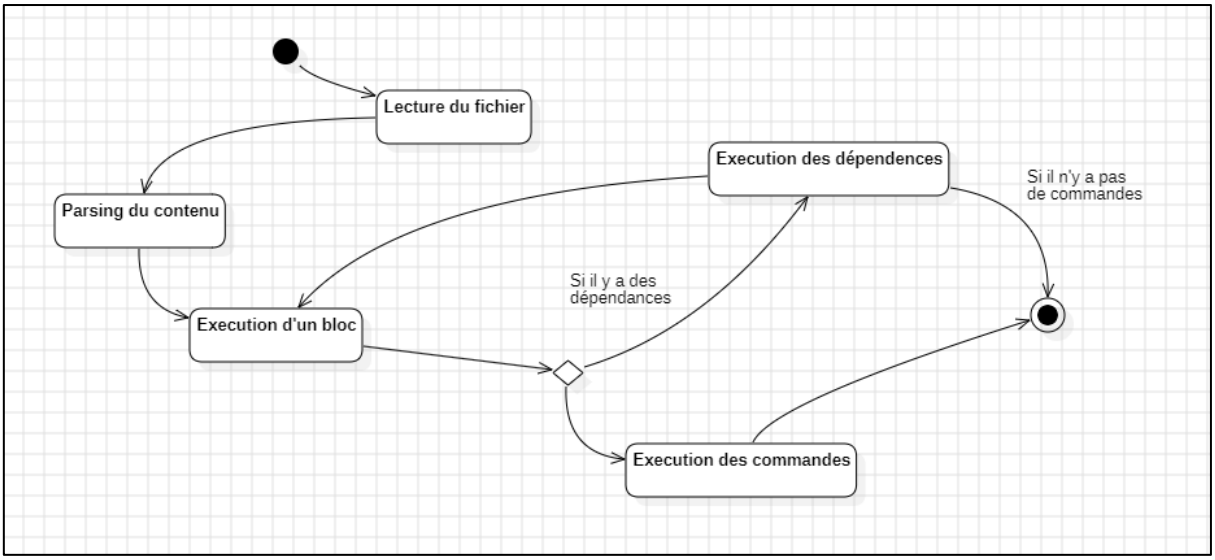

Figure 1.0 : Diagramme d'activité simplifié

Exécution normale du programme :

Lors de l'exécution du fichier source de Bakefile (voir figure 1.1) sur un fichier Bakefile (du test 01), nous pouvons observer (voir figure 1.2) que l'exécution du programme affiche les bonnes commandes, trois fichiers sont créés dans le répertoire (voir figure 1.3 et 1.4).

```
java -jar ../../bakefile.jar
```

Figure 1.1 - commande d'exécution

```
gcc -Wall -Wextra -g -c main.c
gcc -Wall -Wextra -g -c hello_world.c
gcc -Wall -Wextra -g -o main main.o hello_world.o
```

Figure 1.2 - résultats des commandes

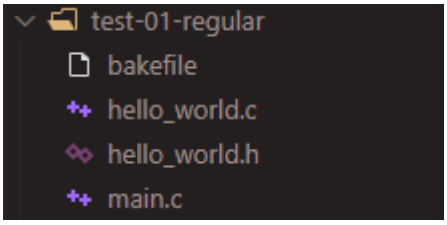

Figure 1.3 - répertoire avant exécution

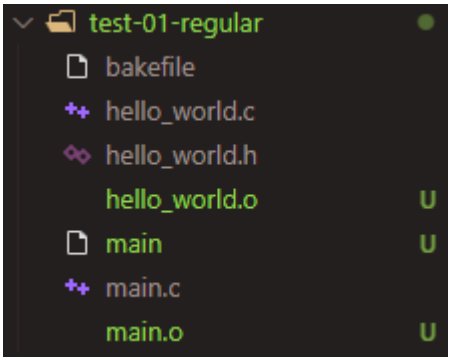

Figure 1.4 – répertoire après exécution

Exécution normale du programme :

Ajout de l'argument `--debug` à la commande d'exécution (voir figure 1.5). Les fichiers visités sont affichés, ceux qui ont besoin d'être recompilés sont eux aussi affichés. S'il y avait eu des différences de dates entre les références et les blocs, cela aurait aussi été affiché.

```
The file 'all' does not exist, recompiling.
 The file 'main' does not exist, recompiling.
  The file 'main.o' does not exist, recompiling.
gcc -Wall -Wextra -g -c main.c
  The file 'hello_world.o' does not exist, recompiling.
gcc -Wall -Wextra -g -c hello_world.c
gcc -Wall -Wextra -g -o main main.o hello_world.o
```

Figure 1.5 – répertoire après exécution

II. Diagramme de classes

Le programme se divise en trois principales classes (voir figure 2.0) :

- `fr.tanchoulet.bakefile.Parser` qui permet de transformer le contenu du fichier **Bakefile** en données compréhensibles par le programme (en blocs de données).

- `fr.tanchoulet.bakefile.Block` qui correspond à un bloc de données de compilation d'un fichier. Cette classe ne comporte que des méthodes statiques et un constructeur car elle sert seulement à regrouper les données d'un bloc.

- `fr.tanchoulet.bakefile.Executor` qui permet d'exécuter les commandes contenues dans un bloc de données, tout en s'assurant que celles de ses dépendances ont également été exécutées.

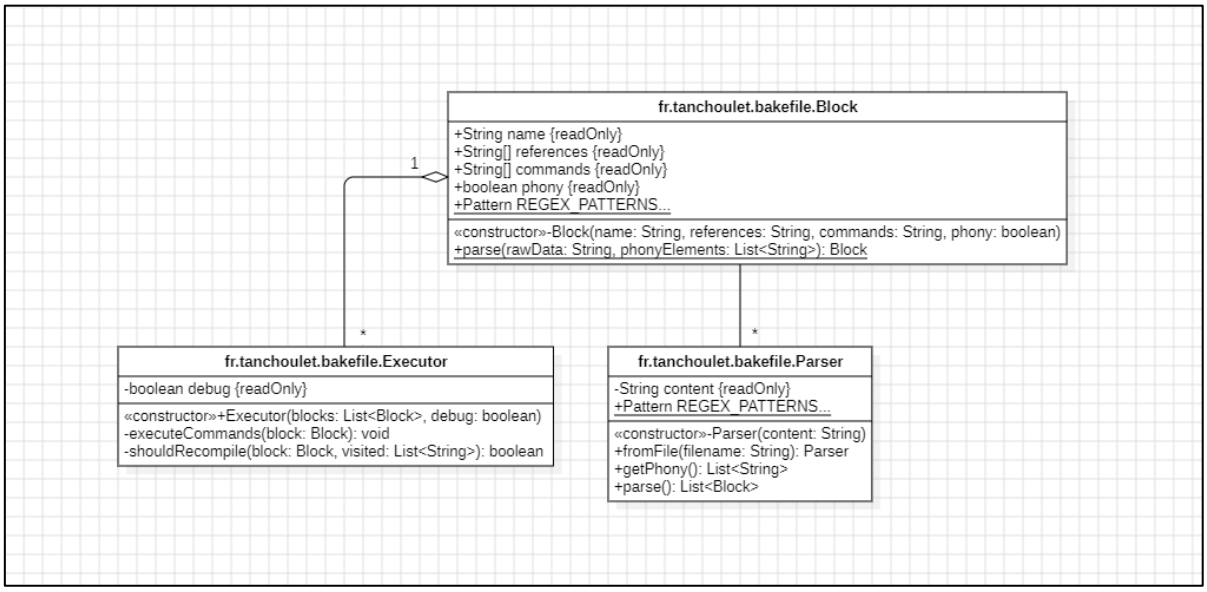

Figure 2.0 - Diagramme de classes simplifié

III. Les dépendances circulaires

Afin d'empêcher les dépendances circulaires, la classe `fr.tanchoulet.bakefile.Executor` utilise une méthode récursive utilisant un parcours préfix de l'arbre permettant de ne pas rencontrer deux fois les mêmes dépendances dans l'historique de parcours de chacun des blocs.

Une pile vide est créée à l'invoquation de la méthode sur la racine du bloc que l'on souhaite exécuter. Pour chacune des dépendances de ce même bloc, la méthode est invoquée en copiant la pile et en rajoutant le bloc actuel. Le processus est réitéré par la suite pour chacune des dépendances, cependant si jamais il retombe sur un des blocs qu'il à noté dans sa pile de blocs visités, il détecte une dépendance circulaire et ne va pas plus loin dans l'arbre (voir figure 3.0).

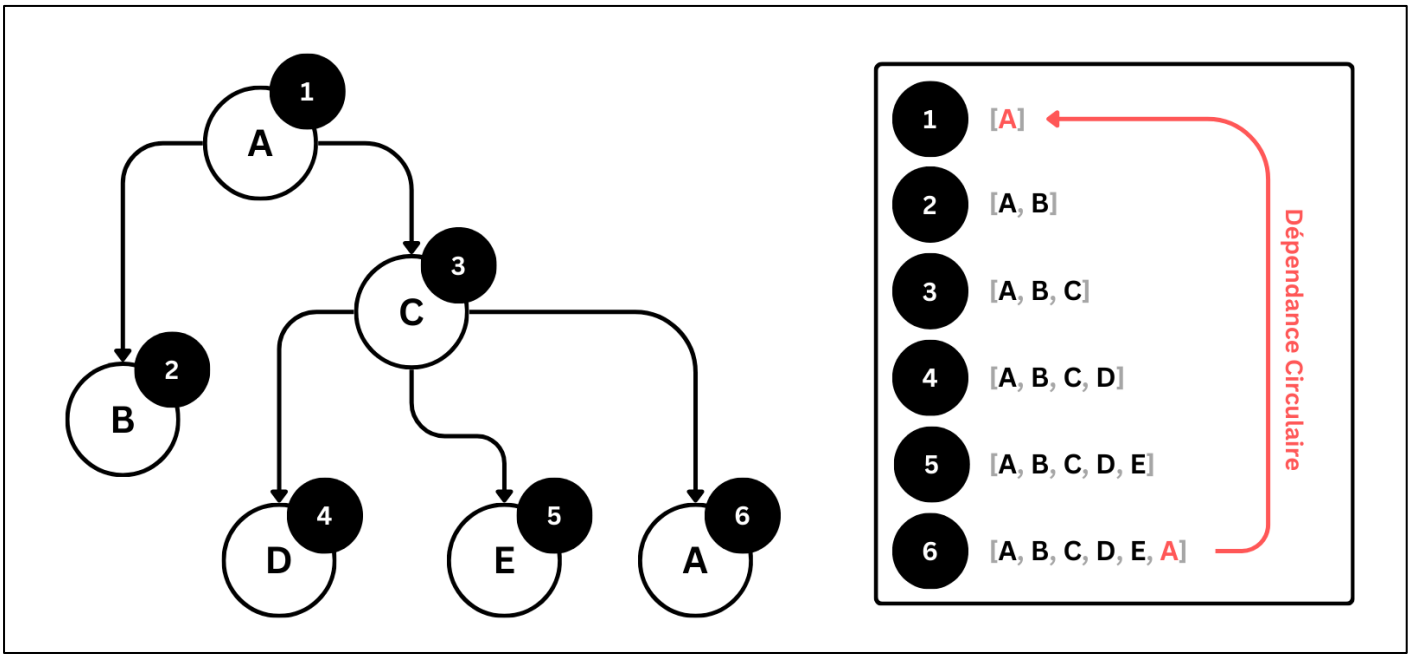

Figure 3.0 - Schéma explicatif de l'algorithme de détection des dépendances circulaires

IV. Enumération des structures de données abstraites

a) Les Patterns / Matchers

Les pattens (Expressions régulières) ont été utilisées afin de pouvoir récupérer plus facilement des parties de texte qui ont par la suite pu être nettoyées et interprétées par le `fr.tanchoulet.bakefile.Parser` de donner une structure de données cohérente et compréhensible par la machine.

b) Les Arbres

Un arbre est utilisé dans les classes `fr.tanchoulet.bakefile.Executor` (qui représente la racine de l'arbre) et `fr.tanchoulet.bakefile.Block` (qui représente les feuilles, nœuds). Il permet de faire une hiérarchisation de l'ordre d'exécution des blocs en fonction de leur dépendences. Le parcours de l'arbre se fait en infixe car les dépendences d'un fichier doit se compiler avant celui qui les utilise.

c) Les Piles

Dans la méthode `fr.tanchoulet.bakefile.Executor#shouldRecompile(Block block);`, une liste de chaine de caractères est créée et est utilisée comme Pile dans laquelle chacun des blocs qui ont été visités sont rajoutés afin d'empêcher les dépendances circulaires.

V. Conclusions personnelles

a) Conclusion de Louis

Ce projet m'a permis d'apprendre correctement le fonctionnement de Make. Il m'a également beaucoup appris au niveau du lexer/parser en discutant avec mon camarade et enfin, il m'a aussi permis d'approfondir mes connaissances en récursivité.

b) Conclusion de Gaston

Ma participation dans ce projet m'a permis d'approfondir mes compétences quant à l'organisation d'un projet en répartissant les taches au préalable et en améliorant la qualité des travaux préparatifs (diagrammes de classes, organisation des taches à faire, implémentation de la javadoc et mises à jour progressives du Makefile). Cette SAÉ à m'aussi été bénéfique pour mieux comprendre le fonctionnement concret d'un Makefile et de la commande « make ». Sur le point de vue coopératif, l'utilisation d'un système de peer programming m'a permis de voir des possibles bugs et opportunités d'optimisation de manière précoce.