Projet Info II

Lairë ~ Éditeur de sonnet

Manon HILL et Enzo POGGIO

Plan

Introduction

Lairë est un éditeur de sonnets français. Il permet de compter le nombre de pieds de chaque vers, vérifier si les vers riment entre eux, et propose des mots pour compléter un vers. Il permet d'écrire des sonnets en octosyllabes, décasyllabes, ou en alexandrins, et apporte une aide à qui veut écrire un sonnet.

Le principe est de lancer un éditeur de texte en Java qui assiste l'utilisateur pour écrire un type de poème particulier. Cet éditeur respecte les règles de la poésie classique (voir **annexe 1**). Cet éditeur propose une interface graphique adaptée à la recherche de vocabulaire dans le dictionnaire, sur demande. Il indique aussi sur demande à l'utilisateur où il en est dans la versification, la construction et la rédaction de son poème avec des indications affichées dans l'interface. Pour finir l'utilisateur pourra exporter son œuvre au format texte .

Diagramme des cas d'utilisation UML du système

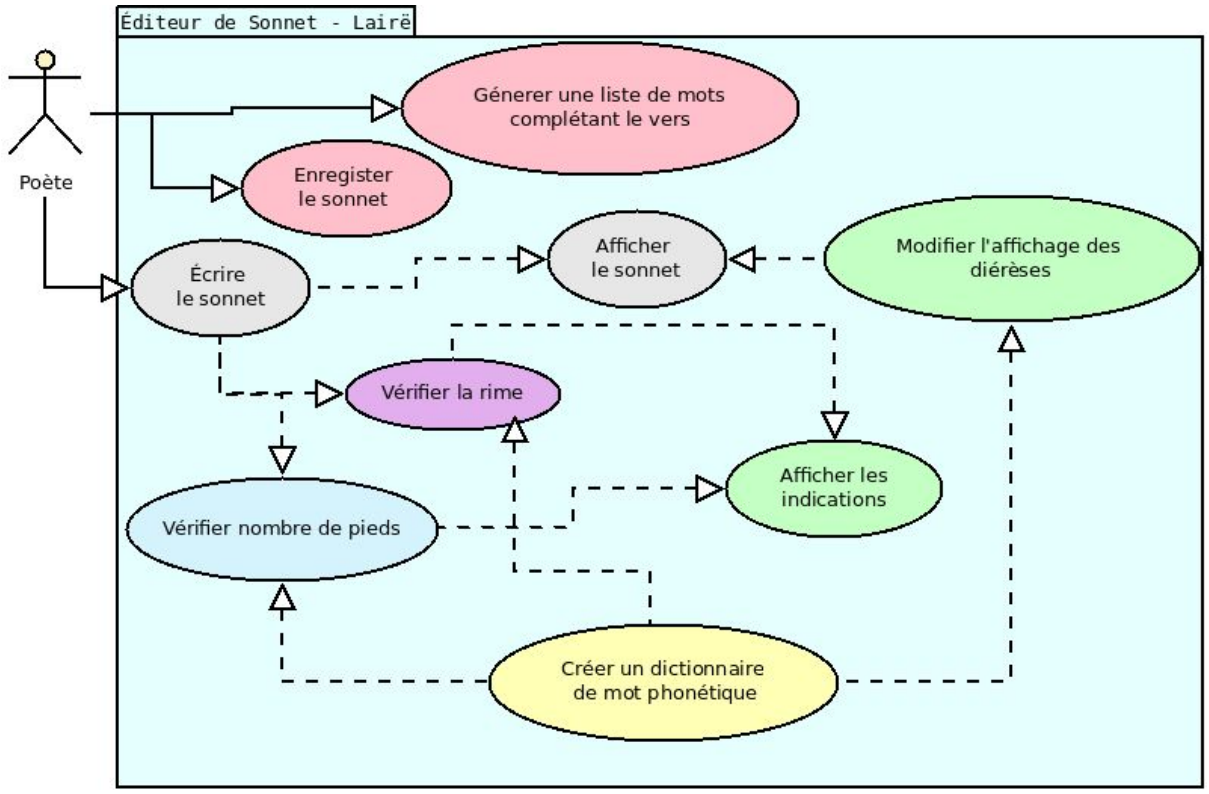


Fig.1: Use Case avant la programmation

L'ancien diagramme Use Case ne prend pas en compte toutes les possibilités. Il ne permet pas à un administrateur de modifier le dictionnaire et d'ajouter des mots. Il ne contient pas l'action permettant au poète de choisir quand son texte est vérifié. Il contient une action "Modifier l'affichage des diérèses" qui a été supprimée, car l'affichage des indications permet déjà à l'utilisateur de savoir si un vers contient ou non une diérèse.

Le diagramme Use Case final prend mieux en compte la structure du programme (Fig.2) , contient toutes les actions correspondant à des use cases en -rose-, contient toutes les actions correspondant à une modification de l'interface en -gris-, les méthodes en -violet-, et tout ce qui concerne le dictionnaire en -jaune-. Il prend en compte la nécessité d'un administrateur qui peut modifier le dictionnaire pour y ajouter ou supprimer des mots.

Scénarios:
1) L'administrateur remarque un mot manquant dans le dictionnaire. Il ouvre le fichier FPD.txt et ajoute une entrée sous la forme de mot;prononciation. (ex: rire;rir). Lairë est désormais capable de compter le nombre de pieds du mot rire, et de trouver son phonème final.
2) Le poète est bloqué: Il ne trouve pas de rime en deux pieds rimant avec "orange". Il entre quel vers il souhaite compléter, et écrit "1A" dans la zone de texte, puis clique sur le bouton

“Compléter vers”. Un message s’affiche “Veuillez entrer le nombre entre 1 et 14 correspondant au vers que vous souhaitez compléter.” L’utilisateur modifie son entrée et écrit “1”. Des mots proposés s’affichent, “ange, grange, vidange…” et l’utilisateur peut choisir celui qu’il veut pour compléter son poème.

3) L’utilisateur veut vérifier que le vers qu’il vient d’écrire contient bien 12 pieds. Il clique sur “Vérifier rimes et pieds”. Le programme affiche à côté de son sonnet le nombre de pieds de chaque vers déjà écrit, et si la rime est correcte, incorrecte, ou inconnue. L’utilisateur peut modifier son sonnet en conséquent.

4) L’utilisateur entre le titre “C’est bi1 là?” et appuie sur le bouton “Enregistrer”. Un message s’affiche: “C’est enregistré!”. L’utilisateur peut retrouver son sonnet sous le nom “Cestbi1l.txt” car Lairë a supprimé tous les caractères qui ne sont pas des lettres sans accents ou des chiffres.

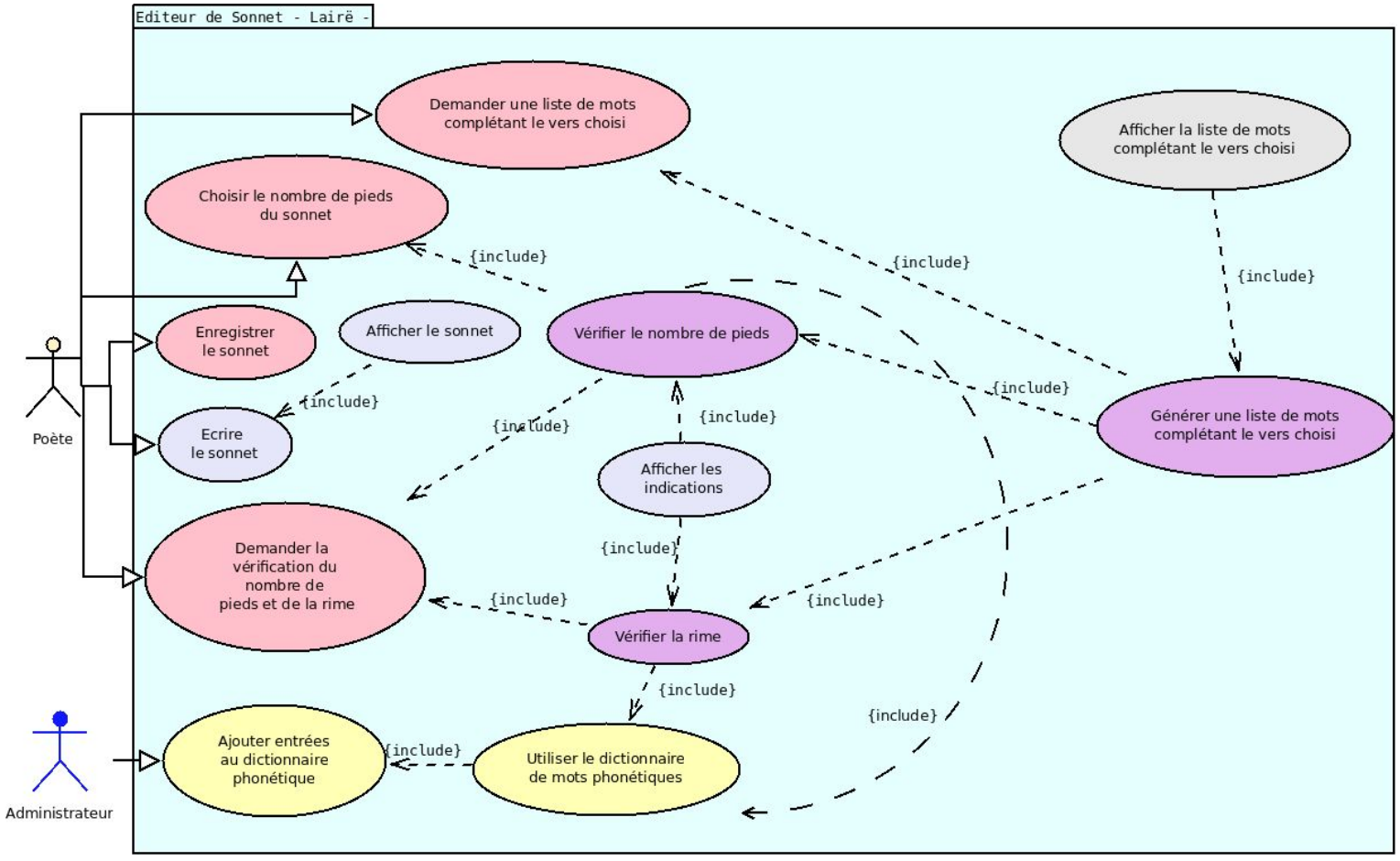


Fig.2: Use Case après la programmation

Diagramme des classes UML

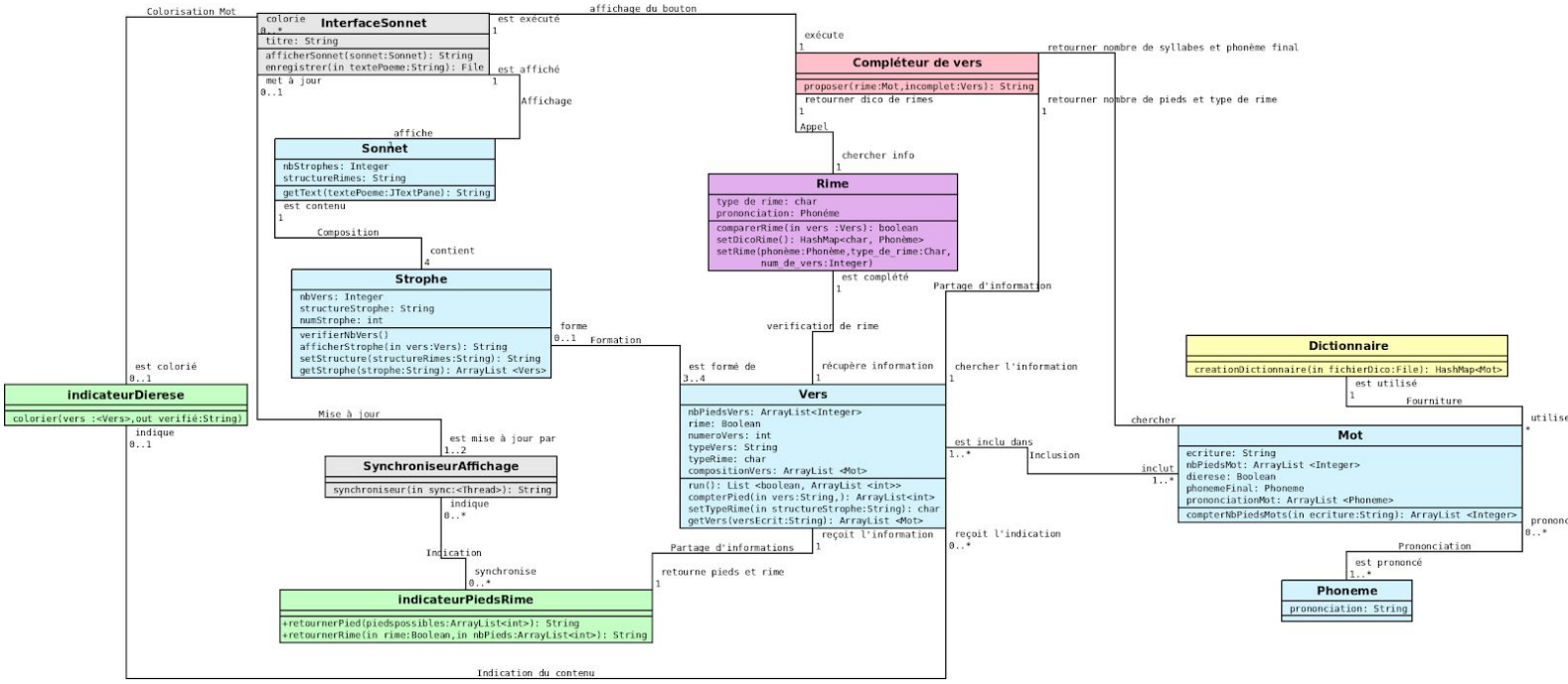
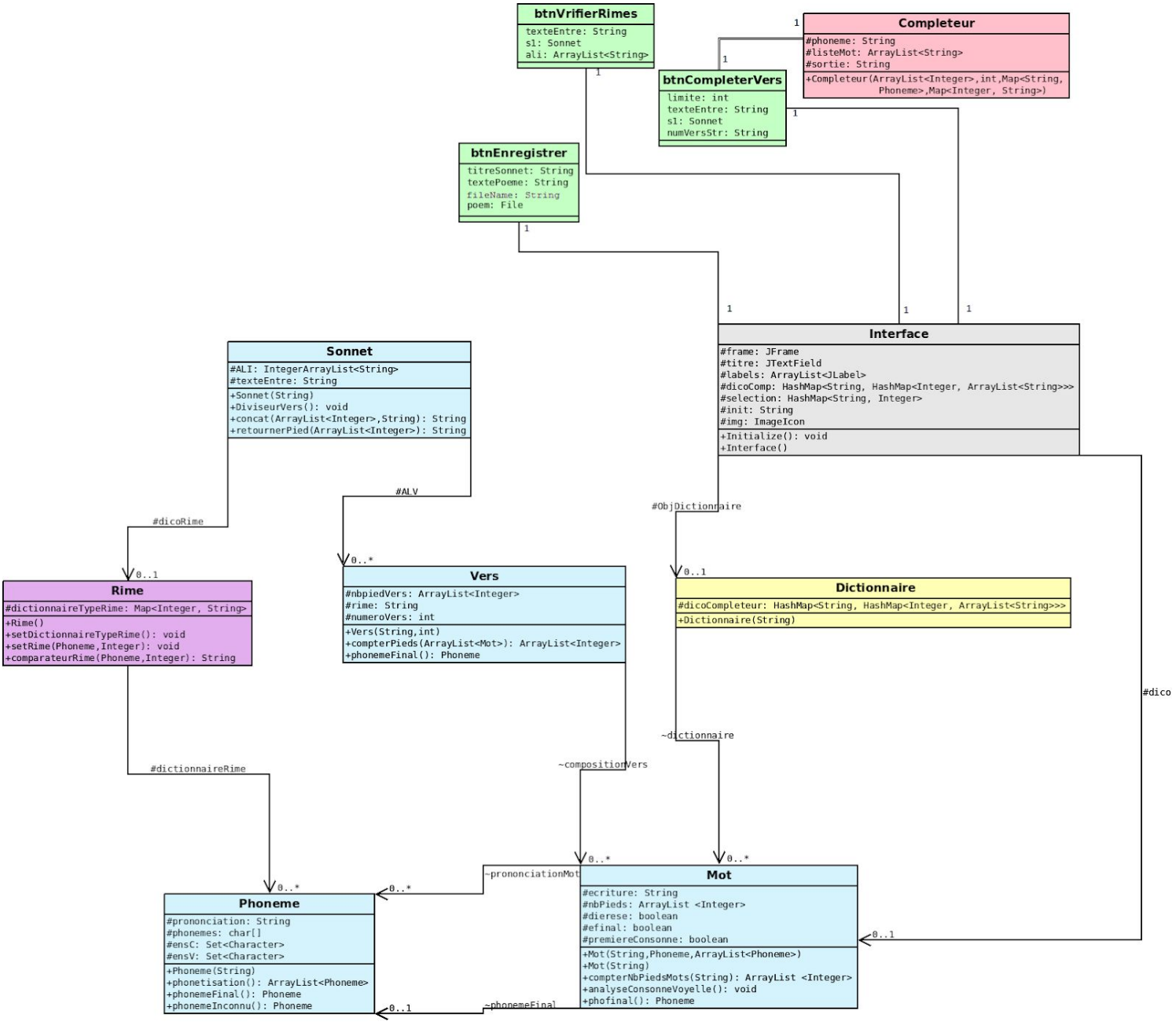


Fig.3: Diagramme des Classes avant la programmation

Ce diagramme des classes contient beaucoup de classes, objets, et méthodes inutiles. Nous avons simplifié la structure générale du projet, et n'avons pas implémenté de Threads, car une mise à jour constante de l'interface pourrait déranger le poète dans son travail et lui distraire l'oeil au lieu de l'aider.

Fig.4: Diagramme des Classes après la programmation

On voit mieux la structure, le programme est simplifié et épuré, les classes ne contiennent plus de méthodes fausses ou superflues.
On a supprimé les différents indicateurs (nombreDePieds et validité de la rime), car les messages retournés sont tout de suite produits dans **Sonnet**.



Structures de données choisies

Si on reprend le code couleur de la figure 4 du diagramme des classes après la programmation, on peut comprendre la structure de données générale:

1) En -bleu- sont nos classes d'objet. Ces classes représentent les différentes parties d'un sonnet; de quoi il est composé. Chaque classe a ses propres méthodes pour créer l'objet conformément à sa description.

La classe **Phoneme** contient la représentation phonétique d'un phonème ou de la prononciation d'un mot entier. Elle contient donc juste un String. Les autres variables d'instance sont juste des constantes ou des temporaires pour traiter les données.
La classe **Mot** contient plusieurs variables relatives au mot: le nombre de pieds de celui-ci dans une liste de Integer, l'orthographe du mot dans un String, et, des variables Boolean pour déterminer s'il contient une diérèse, un e final, ou une consonne au début. Elle contient aussi un phonème final et une prononciation de type **Phoneme**.
Vers contient des longueurs en liste de Integer calculées sur la liste de **Mot** qu'il contient, une rime de type String qui lui permet de savoir si le vers rime ou non, et un numéro pour le retrouver facilement et connaître sa place.
Sonnet contient le texte crée par l'utilisateur et la liste String des informations à retourner dans l'interface. Il initialise aussi plusieurs variables qui seront utiles pour les autres méthodes comme la liste des **Vers** ou le dicoRime qui seront utiles pour la complétion de vers ou la vérification de rimes.

2) En -jaune- notre classe de traitement de la données:
Dictionnaire est la classe qui va lire le fichier et créer deux Map qui vont servir à la création des différents objets cités ci-dessus.
Le premier est le dictionnaire du type HashMap<String, Mot> qui contient l'écriture du mot en majuscule et l'objet mot associé, crée à partir de l'écriture du mot et de l'écriture de sa prononciation.
Le second est du type HashMap<String,HashMap<Integer,ArrayList<String>>>. Ce dictionnaire sert à classifier les mots par leur longueur et par leurs phonèmes finaux. Ce dictionnaire est utile pour la classe **Completeur**.

3) En -violet- notre méthode de classe pour tester la rime:
Rime contient un HashMap<Integer,String> qui sert à initialiser le type de rime de chaque vers: 1,4,5,8 → A; 2,3,6,7 → B; 9,10 → C;11,13 → D; 12,14 → E.
Rime sert aussi à créer le dictionnaire des rimes qui sera du type HashMap<String,Phonemes> qui servira à savoir si deux vers riment entre eux, et/ou à compléter un vers.

4) En -rose- notre méthode de classe pour générer une liste:
La classe **Completeur** récupère l'écriture du phonème et la longueur restante à compléter que **BtnCompleterVers** lui envoie. Puis elle produit un String en sortie à partir de la liste produite par la lecture du dictionnaire.

5) En -gris- notre interface: Voir Interface utilisateur

6) En -vert- les boutons, nos classes d'action accessible par l'interface:
btnEnregistrer récupère le titre donné au sonnet dans le String titreSonnet, puis le modifie pour enlever les caractère spéciaux et les accents dans le String fileName, ce qui permettra

d'enregistrer un fichier txt. Il récupère ensuite le texte écrit par l'utilisateur dans le String textePoeme. Le programme crée ensuite un fichier txt contenant le titreSonnet, le textePoeme, et intitulé fileName.txt

btnVrifierRimes récupère le texte du sonnet dans le String texteEntre, crée un objet sonnet contenant ce texte, et le traite grâce à la classe **Sonnet**. **Sonnet** divise le texte en vers pour remplir l'ArrayList ALI (ArrayList Information), compte le nombre de pieds de chaque vers et détecte le phonème final de chaque vers, puis rentre les indications de chaque vers dans ALI. ALI est renvoyé à **btnVrifierRimes**, et son contenu est utilisé pour modifier les JLabels de l'interface qui donnent les informations à l'utilisateur.

btnCompleterVers trouve des mots dans le dictionnaire dont le phonème final correspond à la rime voulue, et dont le nombre de pieds est égal ou inférieur au nombre de pieds manquants dans le vers à compléter.

Description des principaux algorithmes développés

Les algorithmes principaux sont ceux du comptage de pied, celui de la vérification de la rime et celui de la complétion de vers. Ces algorithmes sont scindés à travers les différentes classes, et les tâches sont subdivisées afin de garder l'unité de cohérence de chaque classe d'objet.

1) Compter les pieds:

On compte d'abord les pieds dans la classe **Mot** où l'on traite l'écriture du mot grâce à la méthode compterNbPiedsMots(String ecriture): cette méthode nous permet de compter le nombre de pieds de chaque mot en remplaçant toutes les voyelles qui comptent par un V majuscule. Puis, cette méthode dénombre les voyelles dans une liste et si le mot contient une diérèse, le nombre de pieds est augmenté de 1 et ajouté à la liste pour chaque diérèse trouvée. Ici on ne compte évidemment pas la prononciation du e muet, car on traite les mots de manière individuelle, et le e muet dépend du mot suivant.

Ensuite, dans compterPieds(ArrayList <Mot> cV) de la classe **Vers** on parcourt la liste de vers avec deux itérateurs afin d'assurer une gestion du e muet.

Puis on crée une variable min et une variable max, qui vont correspondre à la longueur minimale et maximale du vers traité, auxquelles on ajoute les plus petites valeurs de longueurs de chaque mot et les plus grandes valeurs de longueurs de chaque mot, respectivement. Selon le résultat du test des deux itérateurs, on ajoute +1 à min et à max dépendamment de la présence ou non d'un e muet. Pour finir on ajoute tous les entiers entre min et max à la liste que l'on retourne.

2) Vérifier la rime:

À chaque vers lu dans le texte récupéré dans **Sonnet**, on initialise une rime de dictionnaireRime afin de connaître l'écriture du phonème final pour les vers 1, 2, 9, 11 et 12.

De plus, à chaque objet **Vers** créé on vérifie la rime grâce à comparateurRime(Phoneme p, Integer i). Dans cette méthode on récupère le numéro du vers et le dernier phonème. Avec le numéro du vers et les dictionnaires de Rime on obtient le phonème avec lequel le vers devrait rimer. On compare donc celui-ci avec l'écriture du phonème p entré. On retourne en conséquent la valeur de validité de la rime et si le phonème est inconnu on le retourne comme tel.

3) Compléter vers:

On génère dictionnaire dicoComp dans l'interface depuis **Dictionnaire** en se basant sur les nombres de pieds de l'écriture de chaque mot, et en se basant sur le phonème final. On ajoute chaque mot dans une liste à l'intérieur d'un dictionnaire de Integer, lui-même à l'intérieur d'un dictionnaire de Phoneme.
Puis, dans la classe **Completeur** on récupère l'écriture du Phoneme p et le nombre de pieds manquants au vers. On crée une liste de Mot vide, que l'on remplit avec toutes les valeurs inférieures au nombre de pieds manquants. Puis on transforme cette liste en String, que l'on retourne.

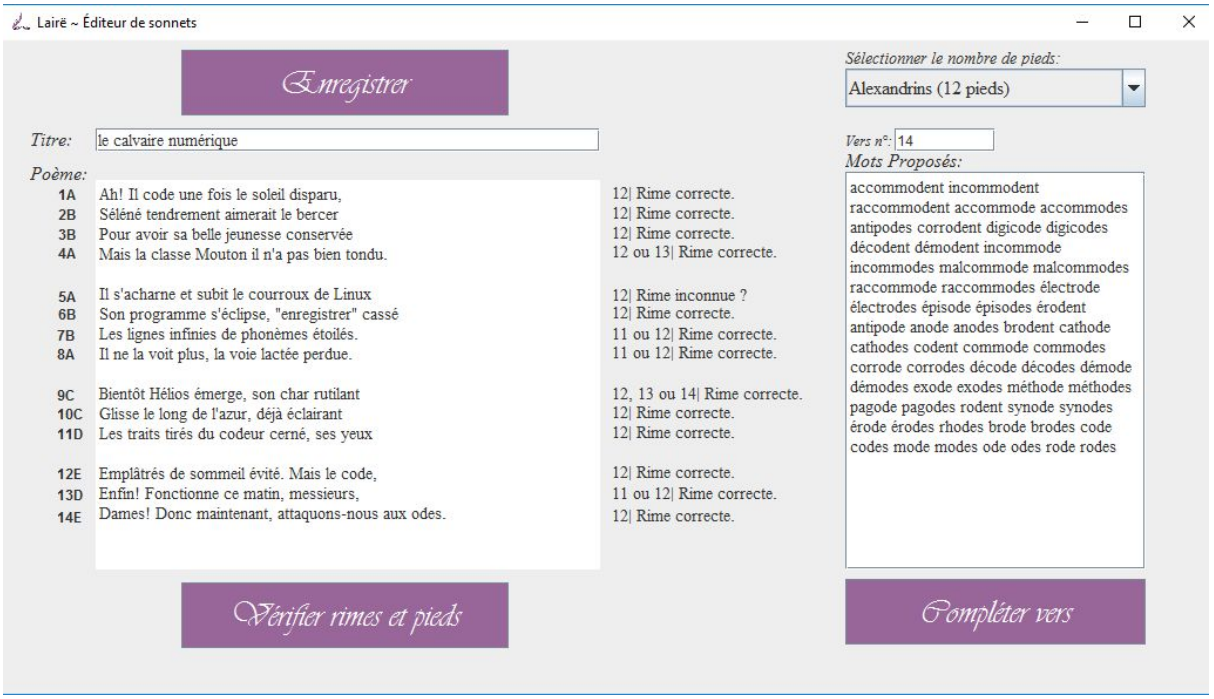
Interface utilisateur

Nous avons implémenté l'interface dans éclipse grâce à GUI. Elle contient trois boutons, **btnEnregistrer** qui permet d'enregistrer le sonnet, **btnVrifierRimes** qui modifie le texte affiché dans des JLabels pour indiquer le nombre de pieds de chaque vers et si la rime est correcte, incorrecte ou inconnue, et **btnCompleterVers** qui affiche dans un JPannel à droite du sonnet des mots qui pourraient compléter un vers. Les trois zones de texte les plus importantes sont **textPane**, où l'utilisateur écrit son sonnet, **versN**, où l'utilisateur indique quel vers il aimerait compléter, et **motsProposes**, où la liste des mots proposés pour compléter le vers s'affichent. Une ComboBox **nbPiedsChoisi** permet à l'utilisateur de choisir le nombre de pieds de son sonnet, ce qui affectera le nombre de pieds des mots proposés.
L'interface contient aussi plusieurs JLabels, dont la majorité restent inchangés et donnent simplement des indications sur le fonctionnement de Lairë à l'utilisateur. Certains JLabels,

A1pieds
B1pieds
B2pieds
A2pieds
A3pieds
B3pieds
B4pieds
A4pieds
C1pieds
C2pieds
D1pieds

E1pieds
D2pieds
E2pieds

sont modifiés quand l'utilisateur clique sur le bouton **btnVerifierRimes**, et affichent des informations à propos des vers écrits: leur nombre de pieds et s'ils riment entre eux.



Tests et les résultats

Pour tester et débugger Lairë, nous avons choisi des sonnets et les avons entrés dans Lairë pour tester toutes ses fonctions, en affichant toujours beaucoup de texte dans le log pour nous indiquer exactement ce que faisait le programme. Nous avons ainsi réussi à faire en sorte que Lairë marche exactement comme on le veut.

Difficultés rencontrées

Le dictionnaire bugge pour certains mots et on ne sait pas pourquoi. Par exemple, les mots haïe ou ouïe: Le programme ne trouve ni le mot, ni la rime dans le dictionnaire FPD.txt.
Nous ne savons pas comment résoudre ce genre de problème, il faudrait lire le dictionnaire mot à mot et tester chaque mot pour trouver les erreurs, ce qui n'est pas faisable considérant que le dictionnaire contient 201'517 mots.

Vers la fin du projet, eclipse s'est mis à refuser de fonctionner sur l'ordinateur d'Enzo, ce qui a posé des problèmes pour la programmation. Ce n'est ni facile ni souhaitable de coder sur un programme qui se ferme seul en permanence.

Une fois le programme terminé, nous avons failli perdre l'entièreté de notre programme en le re-modifiant sans le sauver sous forme de zip. Heureusement nous avons pu récupérer les données de quelques heures auparavant, et n'avons rien perdu.

Conclusion

Ce projet nous a confrontés à la complexité de traiter des données phonétiques et littéraires, peu souvent associées à l'informatique dans l'esprit commun. Il nous a fallu traduire des concepts de langage tels que compter les syllabes, prononcer ou non un e, qui n'ont rien à voir avec les sciences et beaucoup plus avec l'art. Trouver les outils et les instructions à donner au programme pour traduire ce genre de concept était, on peut le dire, complexe. De plus, nous avons étés confrontés à comment gérer intelligemment des ressources de données immenses. La langue Française, qui est la seule langue traitée par le programme, contient beaucoup de mots, et chaque mot a du être divisé en beaucoup d'informations. Il a fallu trouver un moyen de compacter tout ça de manière à ce que le programme ne soit pas trop lent. Finalement, la forme même du genre de texte que nous voulions traiter, le sonnet, a du être divisé de manière astucieuse en différents objets pour correspondre au but de cet exercice, ce que nous avons réalisé après une bonne session de gymnastique mentale.

Améliorations et potentiels développements futurs:

1) Nous pourrions faire en sorte que l'espace entre chaque strophe s'affiche automatiquement, que l'utilisateur soit forcé d'écrire dans la structure donnée et ne puisse pas faire n'importe quoi.
2) Nous pourrions fusionner notre Completeur avec le projet de Valentin Pouilly, pour que les mots proposés ne soient pas piochés dans le dictionnaire au hasard, mais classés selon leur fréquence d'utilisation en plus de l'ordre alphabétique et la longueur.
3) Nous pourrions ajouter un moyen d'importer un dictionnaire, pour que le programme soit utilisable dans plusieurs langues.
4) Nous pourrions ajouter des Threads et un bouton qui les lance et les arrête, pour que l'utilisateur puisse avoir du feed-back sur son poème pendant qu'il l'écrit s'il le souhaite et si cela ne le dérange pas.
5) Nous pourrions implémenter d'autres formes de poèmes que la forme sonnet, les JLabels se déplaceraient à l'emplacement parfait pour chaque type de poème, le schéma de rimes changerait, le nombre de pieds aussi.

Annexe 1: Règles de la poésie Classique

Un sonnet est composé de quatre strophes, deux quatrains et deux tercets.
Lairë programme propose trois types de vers:

8 pieds: un octosyllabe (Elle / a / pas / sé, / la / jeu / ne / fille)
10 pieds: un décasyllabe (Maî / tre / Cor / beau / sur / un / ar / bre / per / ché)
12 pieds: un alexandrin (Oh! / Com / bien / de / ma / rins, / com / bien / de / ca / pi / taines)

Chaque voyelle compte comme une syllabe, excepté le e et le y. Chaque groupement de voyelle compte comme une ou deux syllabes(voir plus bas, la diérèse), excepté s'il contient le e ou le y.

- le -e muet en fin de vers ne compte pas comme syllabe, à moins d'appartenir à un mot monosyllabique dont la seule voyelle est le e, comme "je".
- le -e muet suivi d'un son vocalique (voyelle) ne compte pas comme syllabe, à moins d'appartenir à un mot monosyllabique dont la seule voyelle est le e, comme "je".
- le -e muet suivi d'un son consonantique compte comme une syllabe à part entière.

-Le y est une lettre compliquée: elle agit comme une consonne lorsqu'elle est suivie ou précédée d'une voyelle, mais agit comme une voyelle si elle est entourée de consonnes.
Lo / yal.
cy / clone.

la diérèse

Le poète peut faire prononcer en deux sons ce que habituellement on ne prononce qu'en un seul : c'est une diérèse. Exemple: "un / vi / o / lon" (4) au lieu de "un / vio / lon" (3).