labellabel4.0.427

Projet HbVar

Kathleen Favre
Elham Amin Mansour

Semestre de printemps 2019

Contents

1 Introduction

Le projet présenté dans ce rapport est proposé dans le cadre du projet collaboratif HbVar, aspirant à développer une méthode par spectométrie de masse et analyse de données permettant de détecter et de caractériser les différents types d'hémoglobine ainsi que leurs mutations, permettant ainsi de faciliter le diagnostic des différentes pathologies de l'hémoglobine.

Reposant sur le site `http://globin.bx.psu.edu/cgi-bin/hbvar/query_vars3`, le programme à réaliser dans le cadre de ce projet consiste donc à parser la page Web concernant les variants de l'hémoglobine ainsi que leurs mutations, pour en extraire la donnée et la rendre interrogeable. A partir de cette donnée ainsi traitée, le programme devra donc être à même d'établir une correspondance entre des caractéristiques fournies en entrée par l'utilisateur -décrivant un profil d'hémoglobine- et ces différents variants. Le programme prendra donc en entrée des paramètres tels que le rapport m/z, la charge, la tolérance, le type de masse -moyenne ou monoisotopique-, ou encore la chaine d'hémoglobine concernée, et fournira en sortie l'ensemble des mutations pouvant correspondre au profil décrit par les données expérimentales.

Dans la finalité, répondant aux critères imposés par le cahier des charges, le programme produit devra:

- parser la page du site Web fourni pour en extraire la donnée, c'est-à-dire les différents variants et leur descriptif.
- fonctionner avec ou sans connection Internet, c'est-à-dire parser aussi bien la page du site Web qu'un fichier .html
- stocker la donnée récupérée dans des structures flexibles et interrogeables
- afficher le descriptif des mutations possibles en sortie

Un descriptif du travail entrepris, ainsi que les différents choix d'usage de librairies ou encore un manuel présentant l'utilisation du programme sont donc proposés dans ce rapport.

2 Développement du projet

Le programme se structurant à l'usage en différentes étapes, celles-ci ont été reprises lors du développement du programme, divisant le travail à fournir en différentes phases, comme présenté dans la figure 1. Dans cette partie, il s'agira donc d'expliquer les différents choix de développement faits, tels que l'usage de librairies, mais aussi de documenter le code fourni en présentant la démarche optée.

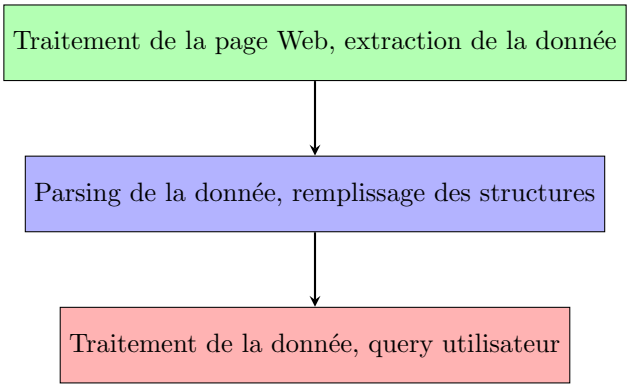


Figure 1: Schéma des étapes de traitement du programme

2.1 Etablissement d'une grammaire

Le projet portant essentiellement sur la notion de parsing,le travail concernant ce projet a débuté par l'établissement d'une grammaire permettant de décrire les différents descriptifs de mutations établis sur la page Web HbVar ayant été fournie.

2.1.1 Grammaire BNF

Compte tenu de la structure des données fournies sur le site, il a été vivement conseillé, dans une première approche, de mettre en forme une grammaire personnalisée, plutôt que d'utiliser une librairie ou une grammaire préexistante ne permettant pas autant de flexibilité quant aux différentes formes d'entrées pouvant être établies.

En utilisant la notation BNF ainsi que son formalisme, une grammaire permettant de décrire la structures des entrées de la page Web à parser a été écrite, permettant également d'établir une première approche de la grammaire à implémenter pour le parseur. Cette grammaire se présente comme suit:

$\langle oneLine \rangle ::= \langle entry \rangle \langle NEWLINE \rangle \langle EOF \rangle$

$\langle variantName \rangle ::= \langle text \rangle$

$\langle HGVSName \rangle ::= \langle text \rangle$

$\langle entry \rangle ::= \langle variantName \rangle \langle separator \rangle \langle mutation \rangle \langle separator \rangle \langle HGVSName \rangle$

$\langle mutation \rangle ::= \langle mutationType \rangle (\langle or \rangle \langle mutation \rangle)?$

⟨*type*⟩ ::= (⟨*insertion*⟩ | (⟨*header*⟩ (⟨*modify*⟩ | ⟨*addSet*⟩))) (⟨*and*⟩ ⟨*type*⟩)?

⟨*mutationType*⟩ ::= (⟨*insertion*⟩ ⟨*protein*⟩| (⟨*protein*⟩ ⟨*header*⟩ (⟨*modify*⟩ | ⟨*addSet*⟩))) (⟨*and*⟩ ⟨*type*⟩)?

⟨*header*⟩ ::= ⟨*number*⟩ (⟨*SYMBOLS*⟩)? (index1())? (⟨*HYPHENE*⟩ number()(index1())?)?

⟨*insertion*⟩ ::= ⟨*aminoAcidSequenceReverse*⟩`inserted between codons`⟨*number*⟩ ⟨*index1*⟩ ⟨*and*⟩ ⟨*number*⟩ ⟨*index1*⟩ (LETTER)+

⟨*modify*⟩ ::= ⟨*aminoAcidSequence*⟩ (⟨*replaceSet*⟩ | ⟨*deleteSet*⟩)

⟨*replaceSet*⟩ ::= (⟨*CHANGETO*⟩ ⟨*aminoAcidSequence*⟩ | `replaced with` ⟨*aminoAcidSequence*⟩)

⟨*deleteSet*⟩ ::= `->` (`and inserted` | `AND inserted`⟨ *aminoAcid*⟩)?

⟨*addSet*⟩ ::= `stop` | `STOP`|`Stop`⟨*CHANGETO*⟩⟨*aminoAcid*⟩(⟨*addChain*⟩)?

⟨*addChain*⟩ ::= ⟨*SYMBOLS*⟩ `modified C` ⟨ *HYPHENE*⟩ `terminal sequence`⟨*SYMBOLS*⟩ ⟨*index2*⟩ ⟨*aminoAcidSequenceReverse*⟩⟨*index2*⟩(⟨*aminoAcid*⟩⟨*HYPHENE*⟩)?`COOH`

⟨*protein*⟩ ::= ⟨*proteinChain*⟩ (⟨*or*⟩ ⟨*proteinChain*⟩)?

⟨*aminoAcidSequence*⟩ ::= ⟨*aminoAcid*⟩(⟨*HYPHENE*⟩⟨ *aminoAcid*⟩)+?

⟨*aminoAcidSequenceReverse*⟩ ::= (⟨*aminoAcid*⟩⟨ *HYPHENE*⟩)+ (⟨*aminoAcid*⟩)?

⟨*aminoAcid*⟩ ::= `Glu`|`Ala`|`Arg`|`Asn`|`Cys`|`Gln`|`Gly`|`His`|`Ile`|`Leu`|`Lys`|`Met`|`Phe`|`Pro`|`Ser`|`Thr`|`Trp`|`Tyr`|`Val`

⟨*proteinChain*⟩ ::= `Ggamma`|`Agamma`|`delta`|`beta`|`alpha1`|`alpha2`

⟨*number*⟩ ::= ⟨*NUMBER*⟩ | ⟨*DIGIT*⟩

⟨*index1*⟩ ::= ⟨*PARANTHESIS*⟩ ⟨*insideParanthesis*⟩ ⟨*PARANTHESIS*⟩

⟨*index1*⟩ ::= ⟨*PARANTHESIS*⟩ ⟨*number*⟩ ⟨*PARANTHESIS*⟩

La grammaire à été revue plusieurs fois pour en arriver à cette grammaire qui est adaptée aux différents cas d'exceptions.

Il est possible de voir des exemples pour les fonctions de la grammaire afin de rendre la grammaire plus compréhensible.

- entry :
  Hb F-Albaicin Ggamma 8(A5) Lys>Gln OR Ggamma 8(A5) Lys>Glu HBG2:c.[25A >C or 25A>G]
- variantName :
  Hb F-Albaicin
- HGVSName :
  HBG2:c.[25A >C or 25A>G]
- mutation :
  Ggamma 8(A5) Lys>Gln OR Ggamma 8(A5) Lys>Glu
- mutationType:
  beta 6(A3) Glu>Lys AND beta 83(EF7) Gly>Asp
- insertion:
  Lys-Val-Leu- inserted between codons 68(E12) and 69(E13) of beta
- replaceSet:
  Ggamma 5(A2) Glu>Gly
  beta 2 - 5 His-Leu-Thr-Pro replaced with His-Ser-Asp-Ser
- deleteSet:
  alpha2 137(H20) - 138(H21) Thr-Ser->0 AND inserted Thr alpha1 61(E10) Lys->0
- addSet:
  beta 147, Stop>Gln; modified C-terminal sequence: (147)Gln-Ala-Arg-Phe-Leu-Ala-Val-Gln-Phe-Leu-Leu- Lys-Val-Pro-Leu-Phe-Pro-Lys-Ser-Asn-(167)Tyr-COOH
  alpha2 142( ) Stop>His

2.1.2 Structure de la grammaire

Factorisation

Au fur et à mesure qu'on avançait dans la grammaire on s'est rendu compte que la grammaire imposait certaines factorisations afin de faire son choix entre les fonctions au cours du parsing. Donc certaines mutations qui débutaient de la même façon ont été factorisées comme par exemple replaceSet et deleteSet qui commencent, les deux, par une séquence d'acides aminés.Ces deux derniers ont été regroupés dans la même fonction qui s'appelle modify.
Il a aussi été remarqué que tous les types de mutations contenaient une entête commune, contenant une ou deux protéines accompagnées d'une seule position ou un interval entre deux positions.En fin nous avons décidé d'y consacrer une fonction(à voir header)et de la factoriser.

Symboles portants un certain sens

Ce qui est particulièrement important dans la grammaire de la base de données, c'est que certains symboles sont propres à certains cas de figure. Le fait que ces derniers portent un sens, nous a inscité à les utiliser comme moyen de discrétisation des cas de differents types de mutation. Dans replaceSet le symbole CHANGETO(>), dans deleteSet le symbole "->", dans addSet le terme "stop" et encore dans insertion le terme "inserted between" sont toujours présents.

2.1.3 JavaCC

Après établissement d'une grammaire personnalisée permettant de décrire les différents cas de figure présents sur le site, il s'est avéré nécessaire de choisir un outil permettant l'implémentation de celle-ci, selon différents choix proposés tels que JavaCC handler, JParsec, JCombinator ou encore Parsec J. Après une phase de documentation permettant d'explorer ces différentes possibilités, il s'est avéré que JavaCC constituait l'un des outils les plus répandus pour générer un parseur, s'avérant donc relativement bien documenté.

De prime abord, de part sa complexité en temps linéaire, JavaCC est apparu comme l'outils permettant le mieux de répondre aux besoins imposés par ce projet, et notamment à celui relatif au temps d'exécution du programme.

De plus, lors de phases de documentation plus avancées, il s'est avéré que JavaCC semblait également le mieux convenir au modèle de grammaire établi, de part son utilisation d'une grammaire EBNF -reprenant donc le même formalisme-, mais également pour son principe se basant sur un algorithme LL(k) -c'est-à-dire une analyse lexicale descendante.

Ainsi, semblant répondre le mieux aux critères établis pour ce projet et apparraissant comme relativement flexible, JavaCC est apparu comme l'outil le plus approprié pour développer la grammaire permettant, par la suite, de parser la donnée.

2.2 Récupération de la donnée

Succédant à l'établissement d'une grammaire, la suite du travail s'est porté sur la récupération de la donnée présente sur la page Web HbVar `http://globin.bx.psu.edu/cgi-bin/hbvar/query_vars3` à parser, pour ensuite tester la première version du parseur établie sur celle-ci.

2.2.1 Jsoup

Dans une première approche du problème, il est apparu qu'une simple "lecture" de la page Web entrainait l'obtention du code HTML de celle-ci, ne permettant donc pas une extraction directe de la donnée. Une lecture seule de la page ne suffisant donc pas, il est apparu nécessaire de parser le résultat ainsi obtenu afin de retirer le balisage HTML et de n'extraire que la donnée. Le balisage HTML étant uniforme et respectant un standard, il s'est avéré judicieux d'utiliser un outil ou une librairie, telle que Jsoup.

Étant une librairie très connue, très bien documentée, et toujours entretenue (la dernière mise-à-jour remontant à quelques mois), cette dernière s'est imposée comme étant l'outil le plus adéquat pour parser la page Web et en extraire la donnée nécessaire.

2.2.2 Documentation de la classe ParsingWeb

Afin de réaliser le parsing de la page Web HbVar et d'en extraire la donnée, le package parsingWeb contenant la classe ParsingWeb -faisant appel à la librairie Jsoup- a été implémentée dans le programme. Une brève documentation de cette classe est fournie dans cette section.

Comme précisé dans les critères d'exigence du programme, il est nécessaire de dissocier le cas ou le parsing est effectué à partir de la page Web -nécessitant donc une connection Internet- ou directement sur un fichier .html fourni.Ces deux cas sont dissociés à partir de l'argument -l/r donné en entrée lors de l'exécution du programme -suivi du chemin du fichier .html à parser- comme décrit dans le paragraphe 2.5. Ces deux cas de figure sont traités respectivement par les méthodes parseWebPage(String html, File file) et parseHTML(String input, File output).

Afin d'établir la distinction entre ces deux cas, le programme vérifie donc dans un premier temps si l'argument fourni en entrée correspond à -l -décrivant un parsing local, c'est-à-dire directement sur un fichier .html fourni- ou -r -décrivant le parsing de la page Web, nécessitant donc une connexion Internet.

Suivant cette dissociation, l'existence du fichier .html donné en argument est vérifiée dans le cas du parsing de la page Web, permettant ainsi d'éviter l'écrasement involontaire de fichier. Ainsi, si le fichier existe, le programme termine avec un message d'erreur, sinon il poursuit son exécution et procède au parsing. Dans le cas du parsing d'un fichier .html, le programme se poursuit sans effectuer de vérification particulière.

Par ailleurs, dans le cas ou l'on parse directement la page Web contenant la donnée (et non pas un fichier .html), il est possible de remarquer que, celle-ci ne pouvant pas être parsée directement, le parsing de la page `http://globin.bx.psu.edu/cgi-bin/hbvar/counter` est nécessaire. Cette dernière étape permet la récupération des informations nécessaires à l'établissement d'une requête au serveur permettant d'aboutir à la création d'un fichier .html. Ce dernier fichier peut lui-même être parsé afin de récupérer la donnée. Ainsi, la méthode parseWebPage(String input, File output) réalisant le parsing de la page Web fait appel aux méthodes request(String output) -qui elle même fait appel aux méthodes parseCouterTable() et post()- et parseHTML(String input, String output).

Afin de détailler davantage ces différentes étapes relatives au parsing et à l'extraction de la donnée,

il est possible de fournir une brève documentation de la classe, détaillant l'ensemble des méthodes la constituant ainsi que leur descriptif, tel que présenté dans la figure 2.

Signature des méthodes	Descriptif
public boolean testConnexion()	Fonction qui teste la connection internet.
public void parseWebPage(String html, File file)	Fonction qui s'occupe du parsing de la page Web dans le cas d'une connexion internet. Fait appel aux méthodes: request(String output) parseHTML(String input, String output)
private void request(String output)	Fonction qui génère une requête au serveur. Fait appel aux méthodes: parseCounterTable() post()
private void parseCounterTable()	Fonction qui parse la page `http://globin.bx.psu.edu/cgi-bin/hbvar/counter` afin de constituer la requête.
private String post()	Fonction qui crée la requête au serveur
public void parseHTML(String input, File output)	Fonction qui parse un fichier HTML dont le chemin est donné dans input et dont le résulat est écrit dans un fichier dont le chemin est donné dans output.

Figure 2: Documentation de la classe ParsingWeb

2.2.3 Séparateur des données

Un problème aperçu dans la grammaire était le fait que toutes les entrées ne débutaient pas par une protéine qui ne nous permettait donc pas de l'utiliser commen moyen de discrétisation entre la partie VariantName et la partie mutation d'une entrée.
Par exemple la partie mutation du cas de figure <insertion>(section 2.1.1), débute par une séquence d'acides aminés. Il n'est pas non-plus possible de la voir comme separateur entre mutation et nom car certaines entrées possèdent des noms contenant un acide aminé(e.g. Hb J-Valencia qui contient 'VAL'). Donc il a été conclu qu'une formule de discrétisation globale entre la partie variantName et mutation n'existait pas.

La solution trouvée était de séparer ces trois colonnes par une chaîne(inexistante dans la base de donnée), au moment du parsing du fichier html(parseHTML()). Donc au moment de la concaténation, dans la classe parsingWeb, chaque élément d'une ligne est collé au prochain élément

par la chaîne separator(définie dans la classe User).

Les espaces du premier élément de chaque ligne(c'est à dire la première colonne : nom de mutation) sont remplacés par le character spaceSeparator (défini dans la classe User). Car pour la simplicité de la grammaire, tous les espaces sont ignorés dans la grammaire. Donc pour garder les espaces compris dans le nom de mutation, ils ont été remplacés avant d'appliquer la grammaire. (ATTENTION: separator et spaceSeparator sont définis séparément dans la grammar.jj)

2.3 Etablissement des structures de stockage de la donnée

Áfin de stocker ces données il a fallu construire des structures qui se trouvent dans le package mutationStructures:

2.3.1 Mutation

Celui-ci est une classe contenant les attributs suivants:

- name:
  le nom de la mutation dans la base de données
- protein:
  Chaque type de mutation contient une seule protéine
- le HGVSname:
  le nom hgvs marqué sur la base de donées
- massMonoisos, massAverages:
  deux variables de type double pour stocker la masse de la mutation sous les unités monoisotopique et moyenne
- deltaMonoiso,deltaAverage:
  les deux deltas qui consistent à la différence entre la masse de la protéine et sa mutation(les unités monoisotopique et moyenne)
- listOfTypes:
  une liste de mutationType(une autre structure).
  exemple:
  Hb C-New Cross beta 6(A3) Glu>Lys AND beta 83(EF7) Gly>Asp HBB:c.[19G>A;251G>A]
  Deux types de mutation sont observés, séparés par des “and”

Au début du projet, la structure de la mutation ne contenait qu'un seul type de mutation mais en rencontrant certaines entrées comme le cas ci-dessus qui contiennent plus qu'un seul type de mutation, cela est apparu insuffisant.

Elle contient aussi les méthodes suivantes:

- calculateDeltaMonoiso(), calculateDeltaAverage():
  Calcule le delta de la mutation pour chaque unité de masse. Cela est fait en additionnant le delta de toutes les types de mutation.

- calculateMassAverage(),calculateMassMonoiso():
  Pour calculer la mass monoisotopique et la masse moyenne, il faut additionner la masse de la protéine et la masse du delta (d'où l'appel de calculateDelta()). Ces deux derniers sont appelés dans le constructeur de la mutation au moment de sa création.

- getProtein():
  renvoie la protéine associée à la mutation

- getDeltaMonoiso()
  renvoie deltaMonoiso

- getDeltaAverage()
  renvoie deltaAverage

- getMassMonoiso()
  renvoie massMonoiso

- getMassAverage()
  renvoie massAverage

2.3.2 MutationType

Après avoir comparé différents types de structures, il est apparu plus judicieux de construire des classes différentes pour chaque cas de figure c'est à dire les différents types de mutations. L'intérêt de ce choix est la possibilité d'avoir une implémentation propre au type de la mutation pour les méthodes getDeltaAverage() et getDeltaMonoisos().

Mais à fin d'avoir une structure générale commune entre tout type de mutation, l'interface MutationType avec les méthodes ci-dessus à été faite. À savoir que les méthodes ci dessus ne sont pas implémentées dans l'interface MutationType mais assure que tout type de mutation les implémente.

- getDeltaMonoiso():
  renvoie la différence entre la mutation et la protéine sous unité monoisotopique

- getDeltaAverage():
  renvoie la différence entre la mutation et la protéine sous unité moyenne

- print():
  affichage des données de la mutation

2.3.3 Type de mutation spécialisée

Chaque type de mutation contient une position ou un interval decrit par deux positions. À fin d'éviter la répétition, l'ensemble des ces champs ont été regroupé dans une classe de nom Header. Les types de mutations qui correspondent le mieux à la grammaire sont les suivantes:

- AddAcid:
  La structure pour une mutation où seulement un seul acide aminé est ajouté.
  - getdeltaMonoiso() et getdeltaAverage: renvoie la masse de l'acide aminé ajouté

  - getAmino: renvoie l'acide aminé ajouté print()
  - getHeader()

- AddChain:
  La structure pour une mutation où une séquence d'acides aminés est ajoutée.
  - getdeltaMonoiso() et getdeltaAverage: renvoie la masse de la séquence ajoutée
  - getSeqAmino(): renvoie la séquence d'acides aminés
  - print()
  - getHeader()

- DeleteSet:
  La structure pour une mutation où une séquence d'acides aminés ou un seul acide aminé est enlevé.Il est possible que à cette place, un acid aminé soit inséré.
  - getdeltaMonoiso() et getdeltaAverage: renvoie la masse de la séquence enlevée multipliée par -1 additionnée par l'acide aminé remplaçant si cela existe.
  - getSeqAmino(): renvoie la séquence d'acides aminés
  - getInserted(): renvoie vrai si un acide aminé remplacant existe sinôn renvoie faux
  - getAminoInserted(): renvoie l'acide aminé inséré
  - print()
  - getHeader()

- Insertion:
  La structure pour une mutation où une séquence d'acides aminés a été insérée.
  - getdeltaMonoiso() et getdeltaAverage: renvoie la masse de la séquence insérée.
  - getSeqAmino(): renvoie la séquence d'acides aminés insérée
  - print()
  - getHeader()

- Replacement:
  La structure pour une mutation où une séquence d'acides aminés a remplacé une autre séquence d'acides aminés.
  - getdeltaMonoiso() et getdeltaAverage: renvoie la masse de la séquence remplacante soustraite par la séquence enlevée.
  - getSeqOld(): renvoie la séquence d'acides aminés enlevée
  - getSeqOld(): renvoie la séquence d'acides aminés ajoutée
  - print()
  - getHeader()

2.3.4 Protéines et Acides

Servant au calcul des masses ainsi qu'à l'implémentation des méthodes getdelta() présentes dans les classes décrivant les mutations possibles, il a fallu inclure dans le programme des structures permettant de stocker les informations relatives aux différents acides - Ala, Arg, Asn, Asp, Cys, Glu, Gln, Gly, His, Ile, Leu, Lys, Met, Phe, Pro, Ser, Thr, Trp, Tyr, Val-, ainsi qu'aux différentes chaines de protéines - beta, alpha1, alpha2, Ggamma, Agamma, delta. Ces données s'avérant être des invariants biologiques, des classes et des énumérations ont été implémentées dans le package helpers, permettant de décrire ces structures dans le programme, facilitant ainsi le calcul des masses et des différences de masse delta à l'échelle des différentes classes représentatives des mutations.

Ainsi, venant compléter l'architecture des classes présentées, il est possible de fournir un bref descriptif de ces classes et énumérations permettant de stocker les informations relatives aux acides et protéines.

- AminoAcidEnum: énumération permettant de décrire les différents acides aminés pouvant composer une protéine, à savoir Ala, Arg, Asn, Asp, Cys, Glu, Gln, Gly, His, Ile, Leu, Lys, Met, Phe, Pro, Ser, Thr, Trp, Tyr, Val. Cette énumération contient en particulier la 'letter code', le nom, la composition chimique, la masse monoisotopique, ainsi que la masse moyenne pour chacun d'entre eux.

- AminoAcid: classe permettant de décrire des acides aminés à l'échelle du programme. Cette classe utilise l'énumération AminoAcidEnum, afin de restreindre le choix d'acides aminés possible. Cette classe permet, entre autres, d'accéder directement à la masse d'un acide aminé, qu'elle soit monoisotopique ou moyenne, et est utilisée dans certaines classes du package mutationsStructures, telle que par exemple AddAcid ou encore DeleteSet.

- AminoAcidSeq: classe permettant de décrire des séquences d'acides aminés. Utilisant la classe AminoAcid, cette classe permet notamment de faciliter le calcul de la masse d'une chaine de séquence d'acides aminés, qu'elle soit monoisotopique ou moyenne. Elle est également employée dans les classes de certains types de mutations, telles que par exemple addChain ou Replacement.

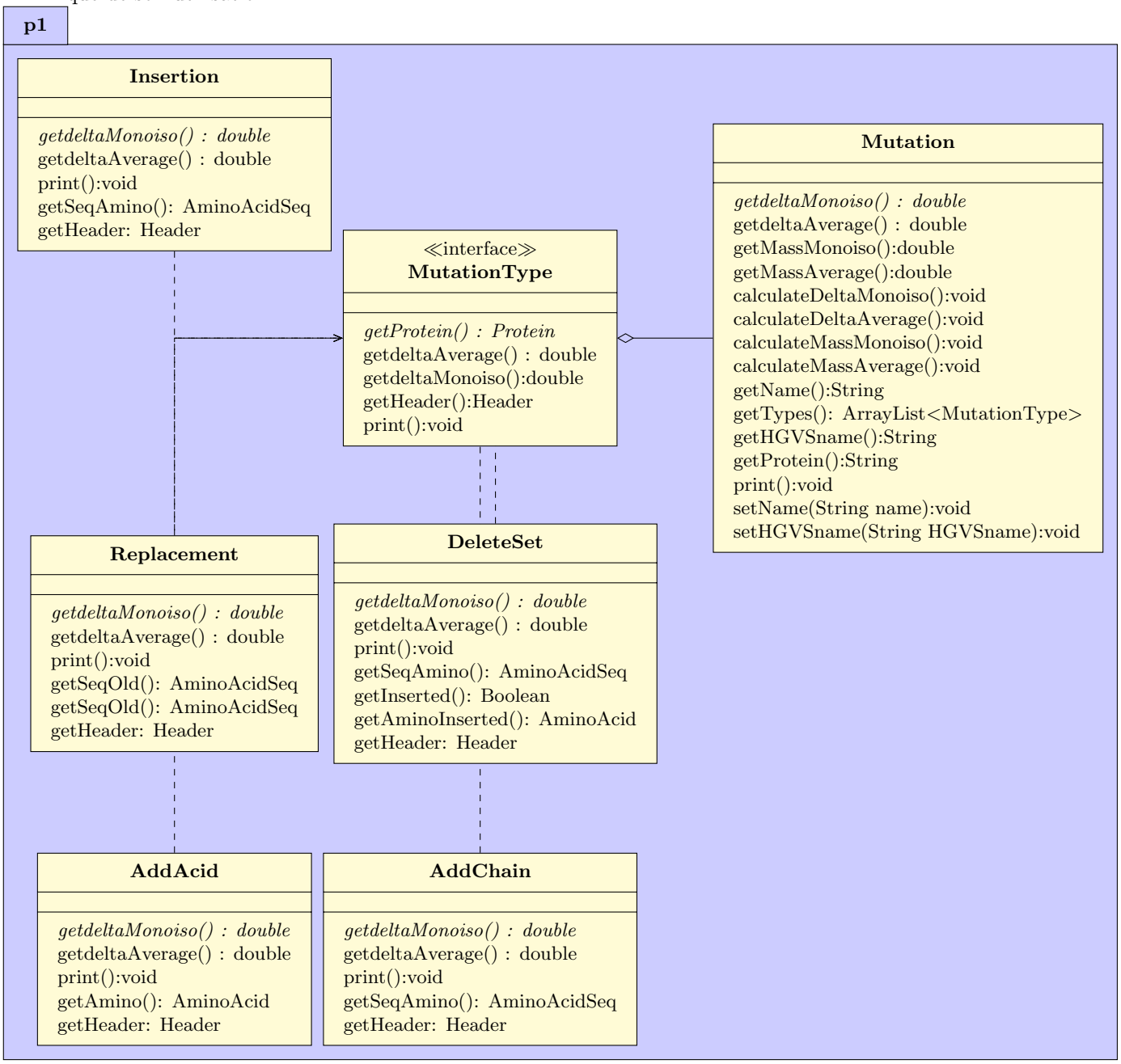
- ProteinEnum: énumération permettant de décrire les différentes chaines d'acides aminés pouvant composer l'hémoglobine, à savoir beta, alpha1, alpha2, Ggamma, Agamma, delta. Cette énumération contient en particulier le nom HbVar, le nom, l'identifiant SwissProt, le nombre d'acides aminés, ainsi que la séquence la caractérisant pour chacune d'entre elles.

- Protein: classe permettant de décrire des protéines à l'échelle du programme. Cette classe utilise l'énumération ProteinEnum, afin de restreindre le choix des chaines d'hémoglobine possible à celui imposé par la composition de l'hémoglobine. Cette classe permet, entre autres, d'accéder directement à la masse d'une protéine, qu'elle soit monoisotopique ou moyenne, et est utilisée à l'échelle du programme pour caractériser les chaines beta, alpha1, alpha2, Ggamma, Agamma, delta.

2.3.5 Description générale des classes

Ainsi, venant compléter les structures décrites dans les paragraphes précédents, il est possible de présenter de façon générale leur organisation et leur architecture au travers du diagramme de la

Figure 4, présentant donc une vue générale de l'organisation du package mutationStructures, ainsi que de son utilisation.



2.4 Parsing de la donnée

2.4.1 Récupération de la donnée

Regardons d'abord la forme générale d'une entrée potentielle:

Forme générale

nomDeMutation
protéine(1.1) or protéine(2.1) mutationType(1.1) AND protéine(1.1) or protéine(2.1) mutationType(1.2) AND...AND protéine(1.1) or protéine(2.1) mutationType(1.n)
OR
.
.
.
OR
protéine(n.1) or protéine(n.2) mutationType(n.1) AND protéine(n.1) or protéine(n.2) mutationType(n.2) AND...ANDprotéine(n.1) or protéine(n.2) mutationType(n.m)
nomHGVS

Donc nous avons une telle forme:

Forme générale compressée

nomDeMutation
BLOC(1)
OR
BLOC(2)
OR
.
.
.
OR
BLOC(n)
nomHGVS

ATTENTION: or ou OR sont considérés égaux!

Définition d'un BLOC

BLOC: protéine(1) or protéine(2) MutationType(1) AND protéine(1) or protéine(2) MutationType(2) AND... AND protéine(1) or protéine(2) MutationType(n)

ATTENTION: Il faut nécessairement avoir les mêmes deux protéines ou la même protéine pour tous les types de mutation d'un bloc!

Fonctionnement de la récupération

Comme il est possible de constater, certaines entrées peuvent avoir des protéines alternatives ou des types de mutations alternatives.
Du premier regard, une option envisageable est de laisser aux structures la possibilité d'avoir plusieurs protéines pour chaque mutation et même la possibilité d'avoir plusieurs types de mutations alternatives(comme sur la base de données). Mais cela nous amène à plusieurs valeurs de masse et protéines dans chaque mutation. Cela complique la recherche de toutes les mutations ayant une certaine protéine ou une masse comprise dans un interval donnée. Comme il n'a pas été demandé de modéliser la structure du stockage de la même façon exacte que la base de données, une deuxième option à été envisagée. Celle-ci consiste à casser chaque entrée en plusieurs mutations étant donné que chaque mutation contient une seule protéine et une seule mutation sur la chaîne d'hémoglobine. Donc en résumé à chaque fois qu'un "OR" ou "or" apparaît dans une entrée, la grammaire ajouté une nouvelle instance de mutation.
La récupération s'est fait dans le fichier Grammaire Hbvar de la manière suivante:

Démarragee du one_Line()

Si une ligne n'est pas un saut de ligne ou la fin du fichier, la fonction entry() est appelée. Ce dernier est appelé autant de fois nécessaire jusqu'à la fin du fichier.

Démarragee du entry()

Dans la fonction entry(), grâce aux fonctions de la grammaire, le nomMutation est le nomHGVS sont récupérés(section 2.1.1).
Par contre la partie correspondante à la mutation n'est pas apparu aussi simple. Pour récupérer cela, en restant toujours dans la fonction entry(), la fonction mutation() est appelée.

Démarrage du mutation()

La fonction mutation() trouve les mutations à créer en regardant chaque BLOC (indiqué dans la forme: séparé les uns des autres par "OR" ou "or") de façon récursive. Comme chaque structure de mutation contient une seule masse, c'est sur qu'en rencontrant un "or" il faut commencer à stocker dans une nouvelle mutation car il s'agit d'un nouveau delta. Regardons l'exemple ci-dessous:

Exemple

Entrée à récupérer: nomEntrée protéine1 42(CD1) - 44(CD3) Phe-Glu-Ser->0 OR protéine2 43(CD2) - 45(CD4) Glu-Ser-Phe->0 nomHGVS
Alors pour stocker cette entrée il faut la dupliquer en deux mutations comme indiqué ci-dessous:
nomEntrée protéine1 42(CD1) - 44(CD3) Phe-Glu-Ser->0 nomHGVS
nomEntrée protéine2 43(CD2) - 45(CD4) Glu-Ser-Phe->0 nomHGVS

La fonction mutation() appelle mutationType() sur chaque BLOC d'une façon récursive. Il est possible de voir son fonctionnement d'une manière plus détaillée.

Démarrage du mutationType()

La fonction mutationType() renvoie les mutations de chaque BLOC de la manière suivante: Elle récupère le premier type de mutation et elle stocke la liste de protéines qu'il y trouve. Ensuite s'il y a plus qu'un seul type de mutation elle appelle la fonction type() qui lui renvoie une liste de tous les types de mutations qui se suivent(elle les trouve récursivement). Puis, dans cette même fonction, une itération sur la liste de protéines est faite. Pour chacune de ces protéines, un objet du type mutation est instancié en lui passant la protéine et la liste des types de mutations. Donc à la fin de mutationType() on se retrouve avec autant de mutations que de protéines. Enfin ces mutations sont renvoyées.

Suite du mutation()

Ensuite mutation() réunit toutes les mutations trouvées dans chaque BLOC et les renvoie.

Suite du entry()

Dans entry(), un objet de type "Entry" est instancié et toutes les mutations trouvées par mutation() lui sont passées. Le constructeur de "Entry" initialise le nom et le nomHGVS de chacune de ces mutations par le nom et le HGVSnom récupérés dans entry(). Donc elles auront toutes le même nom et hgvsnom.

ATTENTION: le nom et hgvsnom ne sont pas propre à une mutation et donc ne peuvent pas être utilisés comme clé.

Suite du one line()

Enfin, le entry contenant toutes les mutations est renvoyé vers one line(). Si entry() a été bien récupéré, le one line envoie l'objet vers la méthode grammarParsing. Ensuite dans ce dernier toutes les mutations du entry sont ajoutées à une grande liste de mutations. À la fin du parsing du fichier, elle sera renvoyé vers la classe User.

2.4.2 Gestion des exceptions: argument -v

Ignorer les exceptions

Le calcul de la masse de certaines entrées de la base de données étant indéterminé, il a été demandé de les ignorer. Pour que le programme ne s'arrête pas en rencontrant ces cas d'exceptions non-inclus dans la grammaire,la paire catch et try ont été utilsés (à l'exécution du oneline()). Les exceptions sont attrapées et la fonction error skipto(int kind) est appelée. Ce dernier continue à parser et à prendre le token suivant jusqu'à ce qu'il arrive à un saut de ligne.À ce moment, un entry du nom "unexpected" est renvoyé depuis le entry() et la grammaire procède à la prochaine ligne.

Affichage des exceptions: argument -v

Pour mettre l'utilisateur au courant des lignes non-prises, il est apparu necessaire de les afficher sur la sortie d'erreur. Mais la possibilité de voir ces lignes a été laissée à l'utilisateur graçe à l'argument -v.

Pour cela, dans la méthode grammarParsing() de la classe Grammaire_HbVar, les numéros des lignes correspondantes aux entrée nommées "unexpected" sont gardées dans une liste. Celle-ci accompagnée de la grande liste de mutations, composent la structure MutationList qui est le type de retour de la méthode grammarParsing().

Ensuite dans la classe User où se trouve le main, les numéros de ces lignes sont récupérés. Dans le cas où l'argument -v a été inclus dans l'interrogation, la méthode handler(), se trouvant dans la classe ExceptionHandler(package exceptions), est appelée et ces lignes accompagnées par le fichier contenant toutes les entrée de la base de données lui sont passés. Enfin dans la méthode handler(), toutes les lignes non-prises sont affichées sur la sortie stderr.

2.5 Partie utilisateur et rendu interrogeable

Le but final du programme étant d'être interrogeable par un utilisateur et de fournir une réponse à sa demande sous forme d'affichage, le travail s'est finalisé par le développement d'une partie utilisateur permettant de gérer ces query sous forme de différents modes.

2.5.1 Rendu interrogeable

Le but final de ce programme étant de décrire un type de mutation possible en faisant coïncider des paramètres caratérisant un profil d'hémoglobine avec l'intégralité de la donnée récupérée, l'utilisateur se doit de fournir un certain nombre d'informations en entrée du programme. Comme spécifié dans le cahier des charges, ces informations seront fournies sous forme d'une ligne de commande se présentant comme suit:

hbvarextract -mz 950 -charge 18 -tol 1 [-chain beta] [-mass mono/avg] -r/l /path/.../fich.html [-v]

avec -chain, -mass et -v des options, -r/l permettant de spécifier s'il s'agit de parsing d'un fichier .html (-l) ou de la page Web (-r), et -v servant à spécifier l'affichage des entrées non prises en compte sur la sortie d'erreur. Les arguments peuvent être entrés sans ordre particulier sur la ligne de commande.

Relativement à ces arguments fournis sur la ligne de commande et en prenant en considération que par défaut la masse monoisotopique est utilisée, la partie utilisateur se décompose en quatre modes:

- Le mode 1 traite des entrées de la forme:

  hbvarextract -mz 950 -charge 18 -tol 1 [-mass mono] -r/l /path/.../fich.html [-v]

  avec [-mass mono] optionnel puisque la masse monoisotopique est utilisée par défaut.

Cette entrée établie la correspondance pour un rapport m/z, une charge, et une tolérance donnés.

- Le mode 2 traite des entrées de la forme:

  hbvarextract -mz 950 -charge 18 -tol 1 -chain beta -r/l /path/.../fich.html [-v]

  Cette entrée établie la correspondance pour un rapport m/z, une charge, et une tolérance donnés pour un type de chaine d'hémoglobine spécifiée.

- Le mode 3 traite des entrées de la forme:

  hbvarextract -mz 950 -charge 18 -tol 1 -mass avg -r/l /path/.../fich.html [-v]

  Cette entrée établie la correspondance pour un rapport m/z, une charge, et une tolérance donnée, établissant les calculs avec la masse moyenne plutôt que la masse monoisotopique.

- Le mode 4 traite des entrées de la forme:

  hbvarextract -mz 950 -charge 18 -tol 1 -chain beta -mass avg -r/l /path/.../fich.html [-v]

  Cette entrée établie la correspondance pour un rapport m/z, une charge, et une tolérance donnée, établissant les calculs avec la masse moyenne pour une chaine d'hémoglobine donnée.

2.5.2 Calcul théorique et mise en pratique

La correspondance entre le descriptif d'une mutation présent dans la base de données interrogeable établie et les paramètres fournis par l'utilisateur permettant de décrire un profil d'hémoglobine reposant sur le calcul de la masse d'une chaine en particulier, ce dernier calcul est rappelé ci-dessous:

$$\text{masse} = \frac{m}{z} \times \text{charge-charge}$$

Par exemple, en reprenant le calcul proposé dans le cahier des charges fourni,la masse de l'Hemoglobin beta + 19 (m/z = 836.1, chargée 19x) chargée 1x est calculée comme suit:

$$\text{masse} = \frac{m}{z} \times \text{charge-charge} = 836.1 \times 19\text{-}19$$

Ainsi, s'appuyant également sur les méthodes getMass() et getDelta() décrites dans le paragraphe 2.3, l'implémentation des des modes 1,2,3 et 4 repose essentiellement sur l'application de cette formule.

Pour chaque mode une itération est faite sur toutes les mutations.
Ensuite si une mutation satisfait les conditions propres au mode, alors le numéro de la mutation trouvée(correspondantes à l'interrogation), suivi des données correspondantes à la mutation est affiché. À fin d'afficher les données, la méthode print() implémentée dans la structure mutation est appelée.

- **mode1**: La masse monoisotopique est récupérée grâce à la méthode getMassMonoiso(). Ensuite il est vérifié que la valeur absolue de la différence entre cette masse et la masse calculée par setMass() (expliqué dans 2.5.2) est plus petite ou égale à la tolérance passée en paramètre dans l'interrogation. Cette tolérence a été gardée dans l'attribut tol.

$$|mutation.setMass() - mutation.getMassMonoiso()| <= tol \quad (1)$$

- **mode2**: La condition dans le mode1 est vérifiée et en plus il est vérifié que la protéine de la mutation est la même que celle indiquée dans l'interrogation. La protéine de la mutation est prise par getProtein() et la protéine passée dans l'interrogation a été gardée à l'avance dans l'attribut chain.

$$mutation.getProtein() == chain \quad (2)$$

- **mode3**: La masse moyenne est récupérée grâce à la méthode getMassAverage(). Ensuite il est vérifié que la valeur absolue de la différence entre cette masse et la masse calculée par setMass() (expliqué dans 2.5.2) est plus petite ou égale à la tolérance passée en paramètre dans l'interrogation. Cette tolérence a été gardée dans l'attribut tol.

$$|mutation.setMass() - mutation.getMassAverage()| <= tol \quad (3)$$

- **mode4**: La condition dans le mode3 est vérifiée et en plus il est vérifié que la protéine de la mutation est la même que celle indiquée dans l'interrogation. La protéine de la mutation est prise par getProtein() et la protéine passée dans l'interrogation a été gardée à l'avance dans l'attribut chain.

$$mutation.getProtein() == chain \quad (4)$$

2.5.3 Documentation de la classe Mode

La classe Mode implémentée permettant de décrire un mode décrit par l'entrée de l'utilisateur, une brève documentation de cette classe est fournie dans cette partie.

Les attributs de cette classe correspondent aux différentes caractéristiques d'un mode, renseignant la demande d'un utilisateur entrée sur la ligne de commande.

- L'attribut int mode: stocke le numéro du mode défini avec la méthode setMode(int length,boolean chainFlag, boolean massFlag).
- L'attribut double mz: stocke le rapport m/z fourni.
- L'attribut double charge: stocke la charge fournie.
- L'attribut double tol: stocke la tolérance fournie.
- L'attribut String chain: stocke le nom de la chaine si nécessaire.
- L'attribut String massOpt; stocke l'option concernant la masse.
- L'attribut double mass: stoke la valeur de la masse calculée avec la méthode setMass().

- L'attribut boolean vExists: permet de déterminer si l'argument -v a été entré par l'utilisateur et donc s'il est nécessaire d'afficher les entrées non prises en compte lors du parsing sur la sortie d'erreur.
- L'attribut boolean local: détermine si le parsing est local (true) -c'est-à-dire effectué sur un fichier .html- ou sur réseau (false) -c'est-à-dire effectué à partir de la page Web-.
- L'attribut String file: stoke le chemin du fichier .html à créer ou à parser pour récupérer la donnée stockée sur la page Web.

Signature des méthodes	Descriptif
public Mode(String[] args)	Constructeur qui définit un mode selon les arguments donnés sur la ligne de commande.
public void setMode(int length,boolean chainFlag, boolean massFlag)	Fonction qui définit la valeur de l'attribut mode, soit le numéro du mode.
public void setMass()	Fonction qui définit la valeur de la masse selon la formule mass=mz*charge-charge
public int getMode()	Fonction qui renvoie la valeur du mode
public double getMz()	Fonction qui renvoie la valeur du rapport m/z
public double getCharge()	Fonction qui renvoie la charge
public double getTol()	Fonction qui renvoie la tolérance
public String getChain()	Fonction qui renvoie la chaine
public String getMassOpt()	Fonction qui renvoie l'option relative à la masse
public double getMass()	Fonction qui renvoie la masse
public void mode1(ArrayList¡Mutation¿ mutations)	Fonction qui répond à la demande définie par le mode 1
public void mode2(ArrayList¡Mutation¿ mutations)	Fonction qui répond à la demande définie par le mode 2
public void mode3(ArrayList¡Mutation¿ mutations)	Fonction qui répond à la demande définie par le mode 3

Signature des méthodes	Descriptif
public void mode4(ArrayList¡Mutation¿ mutations)	Fonction qui répond à la demande définie par le mode 4

Figure 3: Documentation de la classe Mode

3 Validation du programme

Prenant compte de l'importance de la précision et validité des résultats pour diagnostiquer les maladies, il a été décidé de faire cértaines vérifications.

3.1 Vérification de calcule de masses

À fin de vérifier que le calcul de masse à été bien fait, Le calcul de plusieurs entrées à été fait manuellement à fin de s'assurer leurs légitimité. Il est possible de voir tout d'abord: la masse calculée pour chaque chaîne de protéine:

- Alpha1 de code P69905
  monoisotopique:15098.874219999982 moyenne:15108.340800000014
- Alpha2 de code P69905
  Monoisotopique: 15098.874219999982 moyenne:15108.340800000014
- Beta de code P68871
  Monoisotopique: 15839.238779999985 moyenne:15849.200300000017
- Ggamma de code P69892
  Monoisotopique: 15967.244419999986 moyenne:15977.233100000018
- Agamma de code P69891
  Monoisotopique: 15981.260069999986 moyenne:15991.260000000017
- delta de code P02042
  Monoisotopique:15896.238489999987 moyenne:15906.270400000016

Chaque entrée est prise de l'un des types de mutation de manière aléatoire:

- **Replacement**
  1,Hb F-Montchat,Ggamma,1(NA1)Gly>Ser
  masse monoisotipique trouvée:15997.254989999987
  masse moyenne trouvée:16007.259400000017

  - monoisotopique
    Delta = mono(Ser) - mono(Gly)= 87.03203 − 57.02146 = 30.01057
    mono(mutation) = Mono (Ggamma) + delta: 15967.244419999986 + 30.01057 = 15997.25499 **VALIDE**

- moyenne
  Delta = avg(Ser) - avg(Gly): $87.0782 - 57.0519 = 30.0263$
  Avg (Ggamma) + delta: $15977.233100000018 + 30.0263 = 16007.2594$ **VALIDE**

- **Insertion**
  780,Hb Bronx,beta,between 68(E12)-69(E13)inserted Lys-Val-Leu
  masse monoisotipique trouvée:16179.486209999985
  masse moyenne trouvée:16189.666400000016

  - monoisotopique
    Delta = mono(Lys) + mono(Val) + mon(Leu) = $128.09496 + 99.06841 + 113.08406 = 340.24743$
    mono(mutation)= mono(beta) + delta = $15839.238779999985+340.24743 = 16179.48621$
    VALIDE

  - moyenne
    Delta = avg(Lys) + avg(Val) + avg(Leu) = $128.1741 + 99.1326 + 113.1594 = 340.4661$
    avg(mutation)= avg(beta) + delta = $15849.200300000017 + 340.4661 = 16189.6664$
    VALIDE

- **DeleteSet**
  796,Hb M Dothan,beta,25(B7)-26(B8)Gly-Glu->0 and inserted Glu
  masse monoisotipique trouvée:15782.217319999985
  masse moyenne trouvée:15792.148400000016

  - monoisotopique
    Delta =-(mono(Gly) +mono(Glu)) + mono(Glu) = $-57.02146-129.04259+129.04259 = -57.02146$
    mono(mutation) = mono(beta) + delta = $15839.238779999985 + -57.02146 = 15782.21732$ **VALIDE**

  - moyenne
    Delta =-(avg(Gly) +avg(Glu)) + avg(Glu) =$-57.0519-129.1155+129.1155 = -57.0519$
    avg(mutation) = avg(beta) + delta = $15849.200300000017 + -57.0519 = 15792.1484$
    VALIDE

- **AddChain**
  1359,Hb Constant Spring (Hb CS),alpha2,142()Stop>Gln Gln-Ala-Gly-Ala-Ser-Val-Ala-Val-Pro-Pro-Ala-Arg-Trp-Ala-Ser-Gln-Arg-Ala-Leu-Leu-Pro-Ser-Leu-His-Arg-Pro-Phe-Leu-Val-Phe-Glu
  masse monoisotipique trouvée:18496.70271499998

  - monoisotopique
    Delta = $mono(Gln)+mono(Ala)+mon(Gly)+mono(Ala)+mono(Ser)+mono(Val)+$

$mono(Ala) + mono(Val) + mono(Pro) + mono(Pro) + mono(Ala) + mono(Arg) + mono(Trp) + mono(Ala) + mono(Ser) + mono(Gln) + mono(Arg) + mono(Ala) + mono(Leu) + mono(Leu) + mono(Pro) + mono(Ser) + mono(Leu) + mono(His) + mono(Arg) + mono(Pro) + mono(Phe) + mono(Leu) + mono(Val) + mono(Phe) + mono(Glu) + mono(COOH)$= 128.05858 + 71.03711 + 57.02146 + 71.03711 + 87.03203 + 99.06841 + 71.03711 + 99.06841 + 97.05276 + 97.05276 + 71.03711 + 156.10111 + 186.07931 + 71.03711 + 87.03203 + 128.05858 + 156.10111 + 71.03711 + 113.08406 + 113.08406 + 97.05276 + 87.03203 + 113.08406 + 137.05891 + 156.10111 + 97.05276 + 147.06841 + 113.08406 + 99.06841 + 147.06841 + 129.04259 = 3397.828495
mono(mutation)= $mono(alpha2) + delta$ = 15098.874219999982 + 3397.828495 = 18496.702715 **VALIDE**

- **AddAcid**
  11360,Hb Zurich-Altstetten,alpha2,142( ) Stop ¿His
  masse monoisotipique trouvée:15235.933129999981
  masse moyenne trouvée:15245.481900000015

  - monoisotopique
    Delta = mono(His) = 137.05891
    mono(mutation) = mono(alpha2) + mono(His)= 15098.874219999982 + 137.05891 = 15235.93313 **VALIDE**

  - moyenne
    Delta =avg(His) = 137.1411
    avg(mutation) = $avg(alpha2) + avg(His)$ = 15108.340800000014 + 137.1411 = 15245.4819 **VALIDE**

3.2 Vérification des interrogations en élargissant la tolérence

1. **-mz 850 -charge 17 -tol 300 -l "/chemin" -mass avg**
   $850 * 17 - 17 = 14433$

   Donc le programme trouve les mutations dans l'interval [14133,14733]
   5 mutations y correspondent:

   - Hb Lleida: 14688.860000000015
   - Hb Fenton: 14555.753200000014
   - Hb Goya: 14343.501500000015
   - Hb J-Biskra: 14343.501500000015
   - Hb J-Biskra: 14343.501500000015

   La tolérance est élargie de 300 à 350 et donc le nouvel interval est: [14083, 14783]
   Une mutation est ajoutée à la liste des résultats :

- Hb Zhanjiang: 14758.997400000015

Comme il peut être constaté cette mutation ne tombe pas dans le premier interval mais tombe bien dans le deuxième.

2. **-mz 1000 -charge 18 -tol 400 -l "/chemin" -mass avg**

   $1000 * 18 - 18 = 17982$
   Donc le programme trouve les mutations dans l'interval: [17582,18382]
   Une seule mutation y correspond:

   - Hb Zunyi: 18356.196740000018

   La tolérance est élargie de 300 à 500 et donc le nouvel interval est: [17482,18482]
   Une nouvelle mutation est rajoutée:

   - Hb Koya Dora: 18467.214440000014

   Comme il peut être constaté "Hb Koya Dora" ne tombe pas dane le premier interval mais tombe bien dans le deuxième.

4 Manuel utilisateur

Afin de compléter le descriptif du travail accompli, il est proposé dans cette partie de fournir un bref manuel utilisateur, décrivant l'intéraction de l'utilisateur avec le programme.

4.0.1 Compilation

Le programme reposant à la fois sur JavaCC et sur JCC, il est nécessaire de procéder la compilation en plusieurs étapes.

Dépendant de JavaCC, la grammaire, contenue dans le fichier .jj, doit tout d'abord être compilée, permettant ainsi de générer un parseur utilisable avec le reste du programme.

Par la suite, le programme peut être compilé avec la commande de compilation javac habituelle.

4.0.2 Exécution et entrée utilisateur

Après compilation, le lancement du programme peut-être fait avec la commande java habituelle, suivie des arguments à fournir tel que décrit dans la partie 2.5, traitant de la partie utilisateur. L'exécution du programme est donc produite par une commande de type:

java hbvarextract -mz 950 -charge 18 -tol 1 -l/r cheminDuHTML [-chain beta] [-mass mono/avg]

comme décrit dans les paragraphes précédents. Par ailleurs, il est possible de préciser que la méthode main du programme produit se trouve dans la classe User.

4.0.3 Format des résultats

Chaque mutation contenant 6 champs, est affichée sur une ligne séparée. Ces champs sont séparés par des virgules.
Exemple: 1507,Hb Aghia Sophia,alpha1,62(E11)Val->0,14999.805809999982
Il est possible de voir la signification de chacun des champs:

- champ1: le numéro de la mutation parmis les résultats trouvés
- champ2: le nom de la mutation
- champ3: la chaîne protéine de la mutation
- champ5: les types de mutation (séparés par des " AND " s'il y en a plusieurs)
- champ6: la masse(sous unité monoisotopique ou moyenne)

4.0.4 Fonctionnement général du programme

La figure 5, présente les étapes effectuées tout au long de l'exécution du program.

la figure 6, présente une carte qui indique où chaque classe se trouve. Dans le cas de besoin de changement, elle peut montrer rapidement quelle classe et quel package doit s'adapter aux modifications.

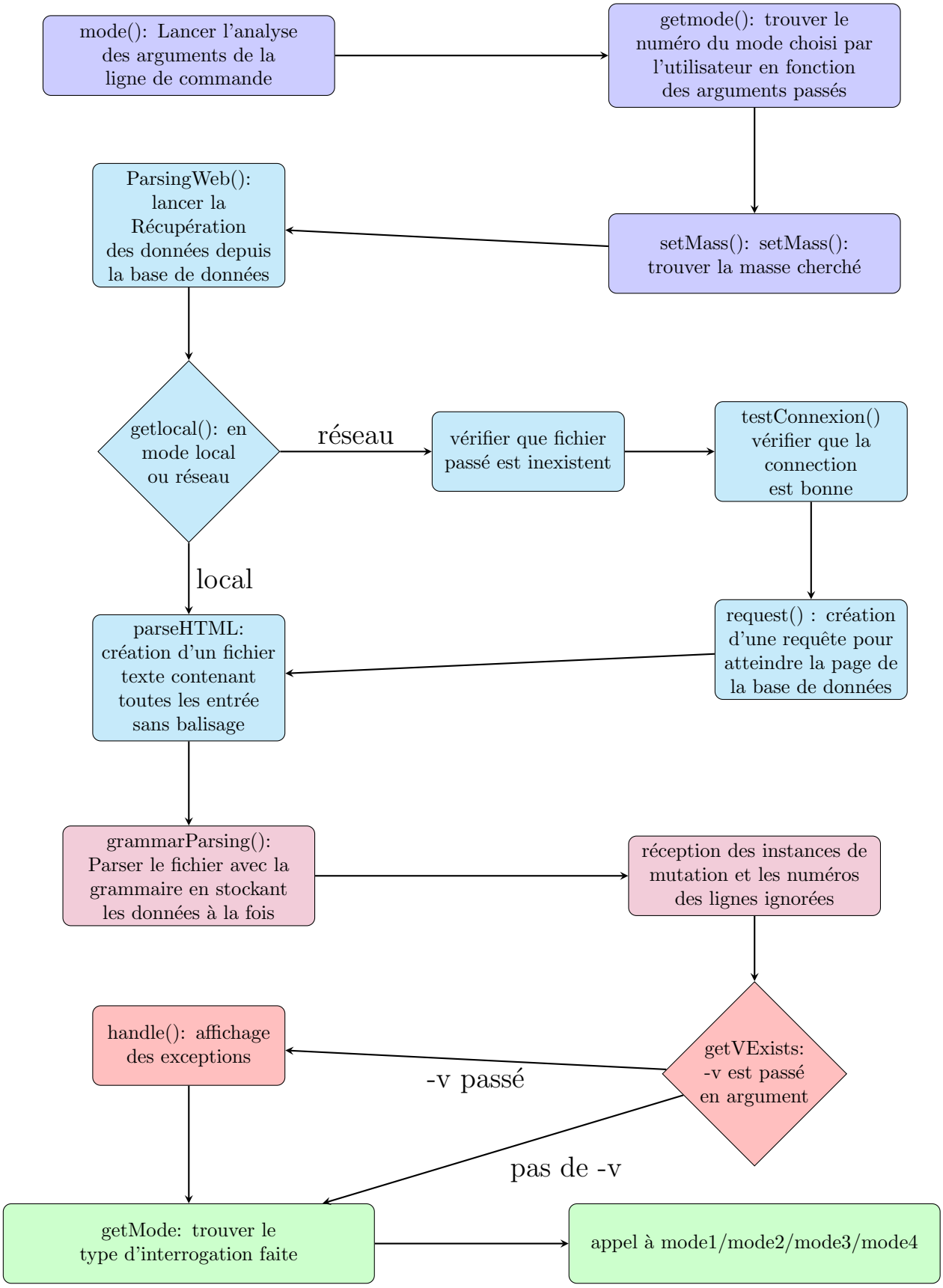


Figure 4: la fonction séquentielle du moment de l'exécution à l'affichage des résultats

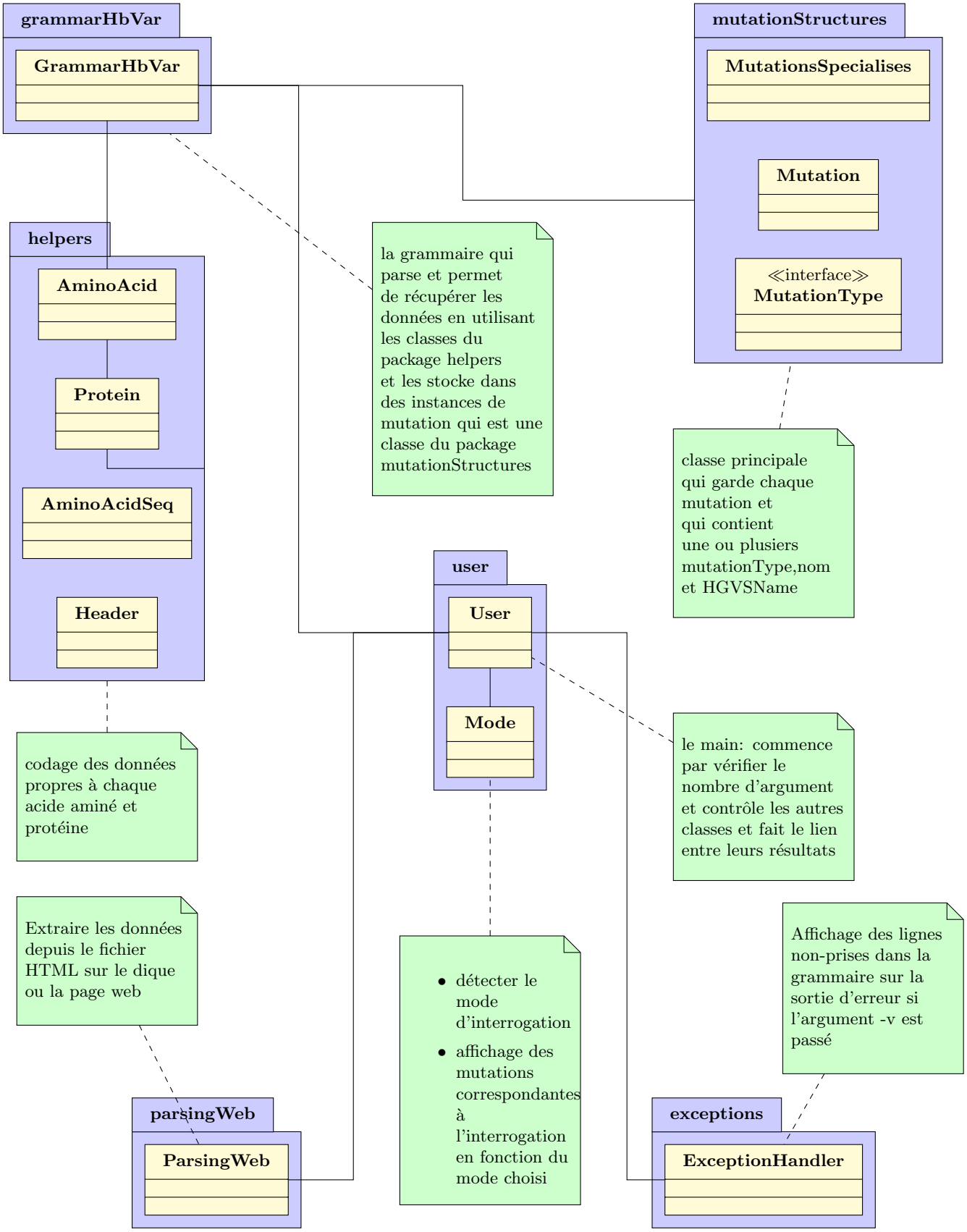


Figure 5: Une description de l'organisation du projet en expliquant le rôle de chaque package et classe

5 Conclusion

Ainsi, répondant au cahier des charges proposé, le programme hbvarxtract permet, à partir du parsing de la page Web `http://globin.bx.psu.edu/cgi-bin/hbvar/query_vars3` ou d'un fichier .html extrait de cette page, d'établir la correspondance entre des paramètres donnés en entrée par l'utilisateur et le descriptif d'une ou de plusieurs mutations possibles correspondant à ce profil.

le lien du dépôt Github: `https://github.com/elham123456/HbVarProject`