Projet : Simplexe

DUT Informatique 2ème année
Formation initiale
IUT de Vélizy
Université de Versailles-St-Quentin-en-Yvelines

Étudiant(s) :

Claire BAUCHU, Louis BIZOT, Damien CHANCEREL, Victor CHARDERON, Martin NIOMBELA, Adam SERRAKH

Enseignant(s) :

M. AUGER, M. MARTEL

Cours :

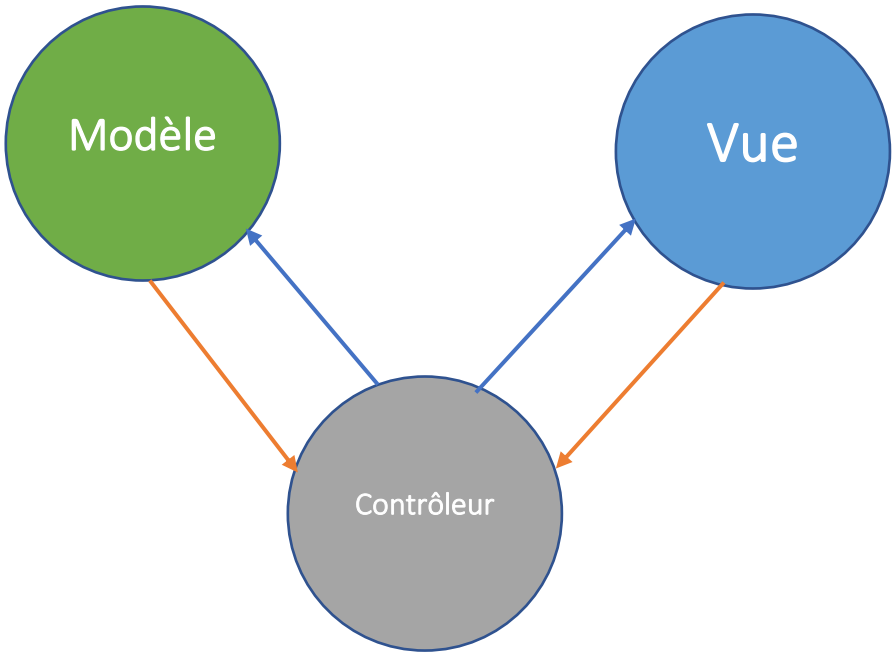
Conception

Sommaire

1 Introduction

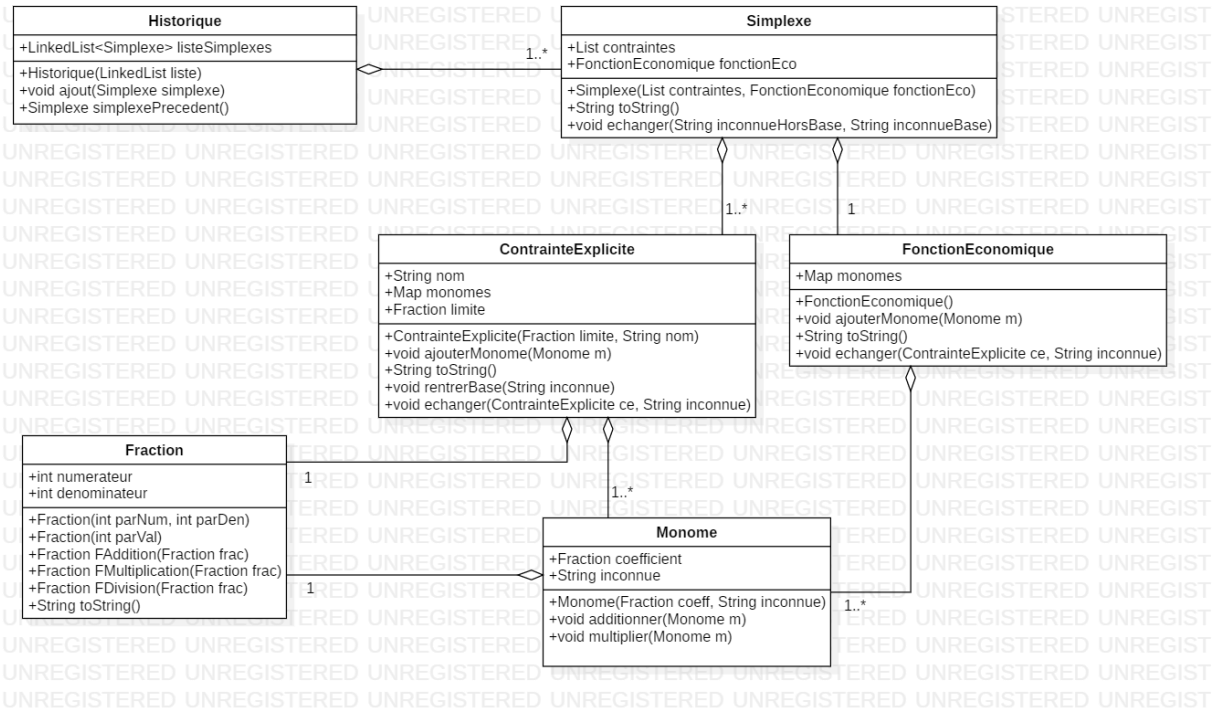
Ce document présente notre conception détaillée de notre projet tutoré à propos de la méthode du simplexe. Ce projet est commandé par M. Martel et M. Auger.

Nous avons décidé d'utiliser l'architecture MVC pour notre projet, à savoir Modèle Vue Contrôleur. Ainsi, le modèle contient le simplexe en lui-même et les structures permettant de stocker les données. La vue contient les panels, et tous les accessoires visuels des panels (boutons, barres de menu, …). Le contrôleur permet de lier ces deux parties du programme : par exemple, afficher un simplexe à l'écran, réagir au clic sur un bouton agissant sur le modèle lui-même.



2 Spécification détaillée de la structure du modèle

2.1 Diagramme de classe



2.2 Responsabilité de chaque classe

2.2.1 Fraction

2.2.1.1 Attributs

`int numerateur` : Ce premier attribut est un nombre signé tenant lieu de numérateur pour une fraction. Ainsi, il peut être négatif ou positif

`int denominateur` : Cet attribut est un nombre signé tenant lieu de dénominateur pour une fraction. Il peut être négatif ou positif

2.2.1.2 Méthodes

2.2.1.2.1 Constructeurs

Nous avons implémenté 2 constructeurs pour cette classe. Le premier prend comme paramètres deux nombres, représentant le numérateur et le dénominateur de la fraction créée. Le second constructeur ne prend qu'un paramètre, un nombre, tenant lieu de numérateur. Le dénominateur est automatiquement instancié à 1. Cela nous permet d'instancier une fraction égale à un nombre entier. Ainsi, nos calculs dans notre application produisent toujours un résultat avec une valeur exacte, grâce à l'utilisation de fractions plutôt que de nombres flottants.

2.2.1.2.2 Opérations mathématiques

Cette classe comporte toutes les opérations sur les fractions possibles (addition/soustraction, multiplication, division). Tous les calculs contiennent à la fin de la méthode un appel à une méthode de simplification `(Fraction Simplifier(Fraction f)`), renvoyant une fraction irréductible.

2.2.1.2.3 toString()

Notre méthode `toString` pour cette classe nous permet d'afficher les fractions selon plusieurs cas. Si la fraction comporte un numérateur et un dénominateur différents de 1 et 0, la fraction s'affiche sous la forme : numérateur / dénominateur.

Si le dénominateur vaut 1, elle n'affiche que le numérateur.

Si le numérateur vaut 0, elle n'affiche que 0.

2.2.2 Monôme

2.2.2.1 Attribut

`Fraction coefficient` : Le coefficient du monôme est une fraction, nous permettant ainsi de toujours obtenir des valeurs exactes (1/3 au lieu de 0.3333… avec un nombre flottant). (cf. rubrique précédente sur la classe Fraction).

`String inconnue` : L'inconnue d'un monôme est une chaîne de caractère de la forme « x_n », où n vaut un nombre.

2.2.2.2 Méthodes

2.2.2.2.1 Constructeur

Cette classe ne comporte qu'un constructeur, prenant en paramètres une fraction et une chaîne de caractères et créant un monôme avec ces deux attributs.

2.2.2.2.2 Opérations mathématiques

Les opérations sur les monômes sont l'addition et la multiplication. Ces méthodes appellent en fait les méthodes correspondant à l'addition ou la soustraction sur les coefficients (Fraction) des monômes concernés par l'opération.

2.2.3 Contrainte Explicite

2.2.3.1 Attributs

`String nom` : le nom d'une contrainte explicite est similaire à l'attribut « inconnue » d'un monôme. C'est une chaîne de caractère de la forme « x_n », où n vaut un nombre.

`Map monome` : cette Map contient des couples clé/valeurs de la forme `<inconnueDuMonome(String), monome(Monome)>.` Ce mode de stockage de données nous permet de trouver facilement le monôme recherché lorsque nous souhaitons effectuer une opération sur le monôme « x_n » précis. L'avantage d'une Map est également l'unicité des objets contenus. Cela nous permet de ne pas avoir deux monômes d'inconnue « x_2 » dans une même ligne de contrainte explicite.

De plus, nous n'avons pas besoin dans le cadre de notre application de conserver les indices ou l'ordre exact des monômes dans une contrainte explicite. Par ailleurs, le nombre de monômes dans notre contrainte explicite peut être variable.

C'est pourquoi le choix d'une `Map` s'est révélé plus adéquat qu'une liste chaînée ou un tableau.

`Fraction limite` : cette fraction représente la limite à ne pas dépasser dans une contrainte explicite.

$$\begin{cases} 2x_1 + 4\,x_2 + 5x_3 + 7x_4 \leq 42 \\ x_1 + x_2 + 2x_3 + 2x_4 \leq 17 \\ x_1 + 2x_2 + 3x_3 + 3x_4 \leq 24 \end{cases}$$

Ici, les limites de chaque contrainte sont : 42, 17 et 24. Cependant, avant de commencer à manipuler le simplexe, ces valeurs sont « rentrées » dans le simplexe de cette manière :

$$\begin{cases} x_5 = 42 - 2x_1 - 4\,x_2 - 5x_3 - 7x_4 \\ x_6 = 17 - x_1 - x_2 - 2x_3 - 2x_4 \\ x_7 = 24 - x_1 - 2x_2 - 3x_3 - 3x_4 \\ z = 7x_1 + 9\,x_2 + 18x_3 + 17x_4 \end{cases}$$

2.2.3.2 Méthodes

2.2.3.2.1 Constructeur

Le constructeur de cette classe prend en paramètres une `Fraction` limite et un nom de type chaîne de caractères. La `map` contenant les monômes est instanciée et est vide à la création de la `ContrainteExplicite`.

2.2.3.2.2 `ajouterMonome(Monome m)`

Cette fonction ajoute le `Monome` à la `Map monomes`, sous la forme du couple <clé, valeur> suivant : `<inconnueDuMonome(String), monome(Monome)>`. Si l'on veut ajouter un nombre sans inconnue dans la contrainte, on place une `String` vide à la place de `inconnueDuMonome`.

2.2.3.2.3 Fonctions pour l'échange

$$\begin{cases} X_5 = 42 - 2\,x_1 - 4x_2 - 5x_3 - 7x_4 \\ X_6 = 17 - 1x_1 - 1x_2 - 2x_3 - 2x_4 \\ X_7 = 24 - 1x_1 - 2x_2 - 3x_3 - 3x_4 \end{cases}$$

Simplexe de départ

$$Z = 7x_1 + 9x_2 + 18x_3 + 17x_4$$

$$X_5 = 42 - 2(17 - x_2 - 2x_3 - 2x_4 - x_6) - 4x_2 - 5x_3 - 7x_4$$
$$X_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$$
$$X_7 = 24 - (17 - x_2 - 2x_3 - 2x_4 - x_6) - 2x_2 - 3x_3 - 3x_4$$

Echange de x_1 et x_6

$$\begin{cases} X_5 = 8 - 2x_2 - x_3 - 3x_4 - 2x_6 \\ X_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6 \\ X_7 = 7 - x_2 - x_3 - x_4 + x_6 \end{cases}$$

$$Z = 119 + 2x_2 + 4x_3 + 3x_4 - 7x_6$$

Résultat à obtenir

$$X_1 = 3 + x_2 - 3x_6 + 2x_7$$
$$X_5 = 1 - 1x_2 - x_4 + x_6 - x_7$$
$$X_3 = 7 - x_2 - 2x_4 + x_6 + x_7$$

$$Z = 147 - 2x_2 - x_4 - 3x_6 - 4x_7$$

2.2.3.2.4 rentrerBase(String inconnue)

Cette fonction calcule la contrainte explicite en fonction de l'inconnue rentrée en paramètre (si la contrainte explicite contient un monôme ayant pour inconnue la chaîne de caractères entrée en paramètre). Dans l'illustration ci-dessus, le résultat de cette fonction est (en vert) la ligne $x_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$ (dans ce cas, on a appelé la fonction ainsi : `x6.rentrerBase(« x1 »)`. On a calculé x_6 en fonction de x_1).

2.2.3.2.5 echanger(ContrainteExplicite ce, String inconnue)

Cette fonction échange dans une contrainte explicite l'inconnue passée en paramètre (si la contrainte explicite contient un monôme ayant pour inconnue la chaîne de caractères entrée en paramètre) par l'expression passée en paramètres (qui est aussi une `ContrainteExplicite`). Dans l'illustration ci-dessus, cette fonction effectue les calculs présentés en vert (lignes x_5 et x_7, dans lesquelles x_1 a été remplacée par la valeur $x_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$.) Le résultat renvoyé par cette fonction est la contrainte explicite simplifiée (les monômes de même inconnue ont été additionnés ou soustraits selon leurs signes).

2.2.3.2.6 toString()

L'affichage d'une `ContrainteExplicite` se fait sous la forme :

```
Nom = [Monômes de la Map monomes]
```

Cette fonction est créée de sorte que les erreurs d'affichage soient évitées. Cette fonction n'affiche pas $x_2 + -x_3$ par exemple, elle affichera $x_2 - x_3$. De même, si le monôme $1x_5$ est contenu dans la `Map`, alors cette fonction va n'afficher que x_5.

2.2.4 FonctionEconomique

2.2.4.1 Attributs

`Map monomes` : cette `Map` contient des couples clé/valeurs de la forme `<inconnueDuMonome(String), monome(Monome)>`. Ce mode de stockage de données nous permet de trouver facilement le monôme recherché lorsque nous souhaitons effectuer une opération sur le monôme « x_n » précis. L'avantage d'une `Map` est également l'unicité des objets contenus. Cela nous permet de ne pas avoir deux monômes d'inconnue « x_2 » dans une même fonction économique.

De plus, nous n'avons pas besoin dans le cadre de notre application de conserver les indices ou l'ordre exact des monômes dans la fonction économique. Par ailleurs, le nombre de monômes dans notre fonction économique peut être variable.

C'est pourquoi le choix d'une `Map` s'est révélé plus adéquat qu'une liste chaînée ou un tableau.

2.2.4.2 Méthodes

2.2.4.2.1 Constructeur

Le constructeur de `FonctionEconomique` ne prend pas de paramètre. Il crée simplement une `FonctionEconomique` dont la `Map monomes` est instanciée mais vide.

2.2.4.2.2 ajouterMonome(Monome m)

Cette fonction ajoute le `Monome` à la `Map monomes`, sous la forme du couple <clé, valeur> suivant : `<inconnueDuMonome(String), monome(Monome)>`. Si l'on veut ajouter un nombre sans inconnue dans la contrainte, on place une `String` vide à la place de `inconnueDuMonome`.

2.2.4.2.3 toString()

L'affichage d'une `FonctionEconomique` se fait sous la forme :

```
z = [Monômes de la Map monomes]
```

L'appellation « z » est une convention pour le nom d'une fonction économique dans un simplexe, et ce nom n'est pas paramétrable dans notre application.

Cette fonction est créée de sorte que les erreurs d'affichage soient évitées. Cette fonction n'affiche pas $x_2 + -x_3$ par exemple, elle affichera $x_2 - x_3$. De même, si le monôme $1x_5$ est contenu dans la `Map`, alors cette fonction va n'afficher que x_5.

2.2.4.2.4 echanger(ContrainteExplicite ce, String inconnue)

Cette fonction échange dans la fonction économique l'inconnue passée en paramètre (si la fonction économique contient un monôme ayant pour inconnue la chaîne de caractères entrée en paramètre) par l'expression passée en paramètres (qui est une `ContrainteExplicite`). Dans l'illustration ci-dessus, cette fonction effectue les calculs présentés en vert (lignes x_5 et x_7, dans lesquelles x_1 a été remplacée par la valeur $x_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$.) Le résultat renvoyé par cette fonction est la fonction économique simplifiée (les monômes de même inconnue ont été additionnés ou soustraits selon leurs signes).

2.2.5 Simplexe

2.2.5.1 Attributs

`List contraintes` : Cet attribut est une liste contenant autant de contraintes explicites que désiré. Nous avons choisi ce type de schéma de données pour pouvoir accéder aux contraintes par indice. En effet, puisque les noms des contraintes changent au fur et à mesure des échanges, utiliser une `Map` aurait rendu le changement de nom plus difficile (dans le cadre d'un couple <clé, valeur>). De plus, l'ordre des contraintes doit être respecté, puisque nous avons besoin de retenir les indices des contraintes (voir fonction échanger)

`FonctionEconomique fonctionEco` : cet attribut est la fonction économique associée au simplexe.

2.2.5.2 Méthodes

2.2.5.2.1 Constructeur

Le constructeur prend en paramètres une `List contraintes` et une `FonctionEconomique`. Le simplexe créé est composé de ces deux paramètres.

2.2.5.2.2 toString()

L'affichage du simplexe est fait sous la forme suivante :

$$x_5 = 8 - 2x_2 - 1x_3 - 3x_4 - 2x_6$$
$$x_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$$
$$x_7 = 7 - x_2 - x_3 - x_4 - x_6$$
$$z = 119 + 2x_2 + 4x_3 + 3x_4 - 7x_6$$

La méthode appelle les fonctions `toString()` des contraintes explicites et de la fonction économique.

2.2.5.2.3 echanger(String inconnueHorsBase, String inconnueBase)

Cette fonction appelle la fonction `rentrerBase(inconnueBase)` sur la contrainte explicite de la `List` du simplexe ayant pour nom `inconnueHorsBase`). Ensuite, la nouvelle valeur de la contrainte explicite modifiée est remplacée dans les autres contraintes explicites et dans la fonction économique (dans l'illustration, on calcule d'abord x_1 puis on remplace tous les x_1 par la nouvelle valeur). C'est pour cela que nous avons besoin de conserver l'indice auquel est située la contrainte explicite sur laquelle on utilise `rentrerBase()`. Ainsi, on garde ensuite cet indice en mémoire et on parcourt toute la liste de `contraintesExplicites` sauf la case de cet indice, pour appeler sur toutes les contraintes (et sur la fonction économique après) la fonction échanger.

2.2.5.2.4 String echangePertinent()

Cette fonction calcule les différents échanges possibles à partir d'un simplexe, et détermine quel échange est le plus pertinent pour se rapprocher le plus possible du résultat optimal. Cette fonction renvoie une chaîne de caractères de la forme « x_1 -> x_6 » si l'échange à effectuer est entre x_1 et x_6, par exemple.

2.2.6 Historique

2.2.6.1 Attributs

`LinkedList<Simplexe> listeSimplexes` : cet attribut est une liste chaînée contenant les simplexes successifs que l'on obtient au fur et à mesure des échanges. Nous avons choisi une telle structure de données puisque l'ordre de cette liste est crucial. Par ailleurs, la taille de cette liste est variable et évolue avec le temps, et les suppressions et ajouts ne se font qu'à la fin de la liste. Cela nous est donc apparu comme la meilleure structure de données pour stocker notre historique de simplexes.

2.2.6.2 Méthodes

2.2.6.2.1 Constructeur

Le constructeur d'`Historique` instancie seulement une liste chaînée vide pour l'attribut `listeSimplexes`.

2.2.6.2.2 ajout(Simplexe simplexe)

Cette fonction ajoute un simplexe à la fin de la liste chaînée `listeSimplexes`.

2.2.6.2.3 Simplexe simplexePrecedent()

Cette fonction efface le dernier simplexe ajouté à la fin de la liste chaînée `listeSimplexes`.

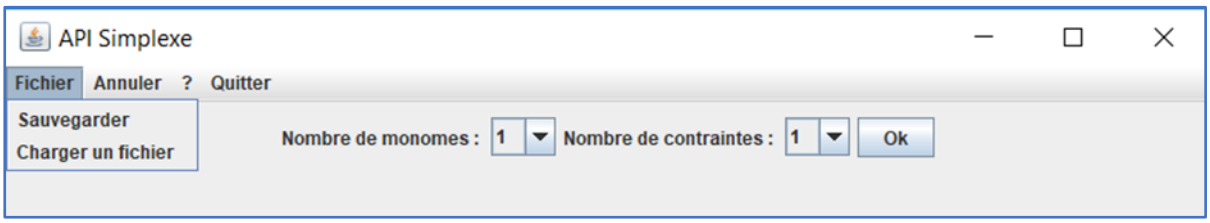
3 Spécification détaillée des interfaces utilisateurs

3.1 Formulaires

3.1.1 Formulaire de saisie d'un simplexe

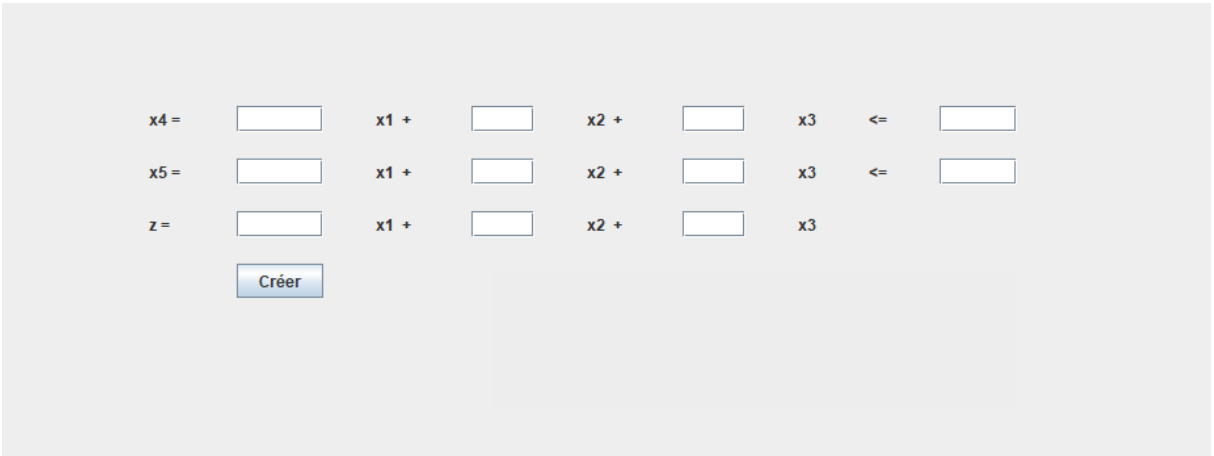
3.1.1.1 Choix du nombre de contraintes et de monômes

Ce formulaire, situé dans le `panelNouveauSimplexe`, propose de choisir le nombre de contraintes et de monômes désiré pour le simplexe. Nous avons fixé une limite de 10 pour ces nombres, en accord avec nos clients.



3.1.1.2 Choix des coefficients et des limites

Ce formulaire comporte des champs de texte organisés selon les valeurs sélectionnée à l'étape précédente (à savoir choisir le nombre de contraintes et de monômes). Par exemple, si l'utilisateur sélectionne 2 contraintes explicites et 3 monômes, le formulaire aura cette forme :



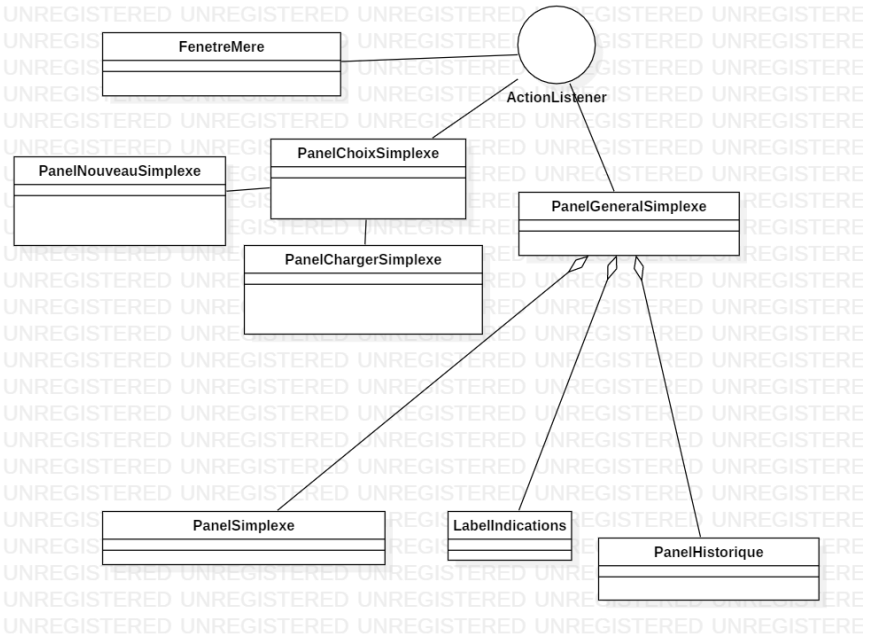
Tous ces champs de texte doivent être remplis par des nombres, négatifs ou positifs. Ils deviendront les coefficients des différents monômes. (Voir Modèle)

3.1.2 Formulaire de chargement d'un fichier

Ce formulaire sera simplement un bouton pour parcourir ses dossiers et sélectionner un fichier à charger.

3.2 Hiérarchie et gestion des panels

Le diagramme suivant est un diagramme de classes simplifié permettant de comprendre la hiérarchie des panels de notre application.



La fenêtre mère est gérée par un `CardLayout` où le `panelChoixSimplexe` est empilé avec le `PanelGeneralSimplexe`. Le `panelChoixSimplexe` est composé de deux panels eux-aussi empilés par un `CardLayout` : `PanelNouveauSimplexe` et `PanelChargerSimplexe`. Ces deux panels fils contiennent les formulaires évoqués dans la rubrique précédente.

Le `PanelGeneralSimplexe` contient au centre le `PanelSimplexe`, où figure le dernier simplexe obtenu. Ce simplexe est affiché ainsi :

$$x_5 = 8 - 2x_2 - x_3 - 3x_4 + 2x_6$$
$$x_1 = 17 - x_2 - 2x_3 - 2x_4 - x_6$$
$$x_7 = 7 - x_2 - x_3 - x_4 + x_6$$
$$z = 119 + 2x_2 + 4x_3 + 3x_4 - 7x_6$$

Les inconnues des monômes (de la forme « x_n ») à droite du signe « = » des contraintes explicites sont des boutons (cela permettra d'échanger les valeurs. Cliquer sur le x_2 de la première ligne correspond à vouloir échanger x_2 et x_5. Cliquer sur le x_3 de la 3ème ligne correspond à vouloir échanger x_3 et x_7). Les actions de ces boutons seront développées plus tard lorsque nous aborderons le contrôleur.

Le `panelHistorique` sera situé à l'est du `panelGeneralSimplexe` et contiendra tous les simplexes contenus dans l'Historique (dans lequel on ajoute tous les simplexes obtenus au fur et à mesure des échanges).

Le `labelIndication` sera situé sous le `panelHistorique` et contiendra l'échange le plus judicieux à effectuer à partir du dernier simplexe obtenu.

3.3 Charte graphique

Organisation commune avec le groupe matrice :

Notre organisation des panels et l'allure de notre barre de menu au haut de l'écran doit être commune avec le groupe Matrice (chef de projet : Ludivine Ducamp), puisque ces deux applications seront rassemblées en un seul et même logiciel.

Nous avons opté pour une organisation intuitive et sobre, puisque le but premier de notre application est la pédagogie.

4 Spécification détaillée du contrôleur

4.1 Champ d'action

Le Contrôleur, dans l'architecture MVC (que nous avons choisie), s'occupe des liens entre la vue et le modèle.

Les liens entre différents panels sont pour leur part gérés dans la vue. Ainsi, les changements de panels dans les `CardLayout` mentionnés pour la vue sont gérés dans la fenêtre mère et non dans le Contrôleur, puisque changer de panel ne touche aucunement le Modèle.

Ainsi, notre Contrôleur va s'occuper de gérer les envois des formulaires évoqués dans la partie « Formulaires», les échanges de variables, et l'obtention d'indications.

4.2 Actions détaillées

4.2.1 Création d'un simplexe

Le Contrôleur va récupérer les valeurs des différents champs de texte remplis par l'utilisateur (voir « Formulaire de saisie d'un simplexe»). Le Contrôleur va vérifier que les valeurs entrées sont bien des `int`, puis constituer des monômes avec ces coefficients transformés en Fraction, les ajouter à des `Map` et ainsi créer des contraintes explicites (ayant pour limite les valeurs entrées par l'utilisateur) et une fonction économique. Ces contraintes explicites seront ajoutées à une liste, et le constructeur de `Simplexe` va être appelé avec pour paramètres cette liste et la fonction économique créée. Le simplexe ainsi créé va être renvoyé à la vue, au `PanelGénéralSimplexe`, qui va afficher le simplexe créé. Un `Historique` va être

créé, auquel le Contrôleur ajoute le simplexe créé. L'historique est ensuite envoyé au `panelHistorique`.

4.2.2 Charger un simplexe

Le Contrôleur va récupérer le nom du fichier ou le chemin du fichier entré par l'utilisateur, puis charger le fichier en question et le transformer en `Historique` contenant tous les simplexes générés grâce à une fonction de lecture/écriture dans les fichiers. Le dernier simplexe de l'historique va être envoyé à la vue, au `PanelSimplexe`, et l'`Historique` lui-même va être envoyé au `panelHistorique`.

4.2.3 Échange dans un simplexe

Le Contrôleur va récupérer les noms des contraintes du `panelGénéralSimplexe` selon le système évoqué dans la partie « Hiérarchie et gestion des panels ». Le Contrôleur va donc effectuer l'échange des deux variables dans le simplexe en appelant la fonction `echanger(String inconnueBase, String inconnueHorsBase)` sur le dernier simplexe de l'`Historique`. L'échange ainsi effectué donne lieu à un nouveau simplexe, qui est ajouté à l'`Historique`. L'`Historique` est envoyé au `panelHistorique`, et le simplexe issu de l'échange au `panelSimplexe`.

4.2.4 Annuler un échange

Le bouton annuler échange sera situé en haut, dans une barre de menu. Le Contrôleur va alors supprimer le dernier simplexe de l'Historique grâce à la méthode `simplexePrecedent()`. Il va ensuite envoyer le nouveau dernier simplexe au `panelSimplexe` (qui est en fait le simplexe précédent) et le nouvel `Historique` au `panelHistorique`.

4.2.5 Obtenir des indications

Au clic d'un bouton « obtenir indication » placé sous le `panelHistorique`, le Contrôleur va appeler `String echangePertinent()` sur le dernier simplexe de l'`Historique`, puis renvoyer le résultat à `labelIndication`.

5 Réalisation

5.1 Stockage de données : modalités et politiques

L'historique des simplexes effectués n'est pas enregistré automatiquement. Seul l'utilisateur peut décider s'il souhaite ou non enregistrer dans un fichier cet historique.

Il peut choisir l'emplacement de ce fichier, mais pas son extension (pour éviter les erreurs en cas de chargement ultérieur du fichier).

L'utilisateur a tous les droits sur ses fichiers, il peut les déplacer, les supprimer, les partager … Notre application n'aura aucune incidence sur ces fichiers, il ne fera que les écrire et les lire.

5.2 Sécurité

Un historique de simplexe n'est pas un bien sensible ou à caractère personnel. Par conséquent, nous n'assurons pas de sécurité spécifique autour des fichiers créés. Notre application n'étant présente sur aucun réseau, il n'y a pas de risque venant de notre application pouvant concerner les mélanges, partages, suppressions inopinées de fichiers. Tout est sous la responsabilité de l'utilisateur.

Notre méthode d'enregistrement d'historique de simplexe n'utilise et n'enregistre pas de données relatives à l'utilisateur, et n'accède à aucun autre fichier extérieur à l'application.

5.3 Choix des outils

Nous avons fait différents choix concernant les outils à utiliser pour la création de cette application. Certains de ces choix devaient être communs avec le groupe Matrice, puisque nos deux applications seront rassemblées dans un seul logiciel.

Nous avons tout d'abord opté pour le langage Java, puisque ce langage nous est familier. De plus, nous avons étudié l'architecture MVC avec Java, que nous avons choisie pour notre projet. Par ailleurs, la gestion des classes par package avec Java et l'IDE Eclipse nous a parue adéquate pour un projet avec 6 développeurs impliqués (une première durant notre cursus).

Pour l'aspect graphique, nous avons choisi la bibliothèque graphique Java Swing pour sa clarté et ses outils pratiques et sobres. Les possibilités de personnalisation étant vastes, ce choix nous a paru pertinent pour une application telle que la nôtre.

Pour la conception du projet, et notamment les diagrammes à réaliser (diagramme de cas d'utilisation, de séquence, …), nous avons utilisé le langage UML (*Unified Modeling Language*) et l'outil graphique StarUML utilisant UML, pour constituer et exporter les diagrammes dans différents formats (PDF, PNG …).

Enfin, pour réaliser les tests de nos classes et de l'ensemble des méthodes de notre application, nous avons choisi d'utiliser JUnit, un framework de test unitaire pour Java. Nous allons donc créer un package dans notre application comportant des tests implémentés grâce à JUnit.

6 Conclusion

Ce document constitue donc notre étude détaillée du projet simplexe. Nous avons pu déterminer les différentes classes à coder, leur hiérarchie, et les outils que nous allions utilise. De plus, nous avons choisi les différentes structures de données utilisées, afin de répondre au mieux aux besoins de M.Auger et M.Martel, nos clients. Ces structures de données sont cruciales puisqu'elles assurent, si elles sont bien choisies, une efficacité de traitement certaine.

Nous avons également déterminé notre politique de stockage de données, ce qui représente actuellement un enjeu important pour tout développeur.

Ce document sera donc la base de la documentation de notre code, et permet donc dès maintenant de nous organiser pour le développement de notre application, puisque nous pouvons dorénavant estimer la charge de travail et les compétences nécessaires à l'implémentation des différentes fonctionnalités.