Projet « Analyseur de données »

Préambule : Lire attentivement les points suivants.

1. Ce travail est un **travail individuel**. Vous pouvez échanger entre vous sur la compréhension du sujet et sur la manière de le traiter mais vous devez chacun écrire vos réponses et votre code.
2. Vous devez répondre aux questions dans un document. Les diagrammes UML peuvent être dessinés à la main ou avec un outil tel que `https://app.diagrams.net/` ou `https://plantuml.com`.
3. Les fichiers fournis sont accessibles via Moodle.
4. Le code et le document doivent être déposés sur votre dépôt Git (dossier projet).

1 Cahier des charges

L'objectif de ce projet est d'analyser des données issues de diverses sources sur lesquelles ont doit appliquer divers traitements. De nouveux traitements devront pouvoir facilement être ajoutés. Ici, les données se limitent à une valeur réelle repérée par une position (abscisse et ordonnée). La valeur pourrait représenter une mesure de pluviométrie, de température, etc.

1.1 Les sources

Les sources peuvent être des fichiers textes, fichiers XML, etc.

Un premier type de fichier texte organise les données ligne par ligne. Chaque ligne contient dans l'ordre, séparés par des blancs, une abscisse, une ordonnée, un numéro d'ordre (ignoré) et une valeur réelle. L'abscisse et l'ordonnée sont des entiers et définissent une position modélisée par la classe Position fournie. Le fichier `donnees.txt` est un exemple de ce format.

Un deuxième type de fichier texte organise aussi les données ligne par ligne. Sur chaque ligne apparaissent dans l'ordre un identifiant (ignoré), une abscisse (entier), une ordonnée (entier), un mot (ignoré), une valeur (réel) et, enfin, une lettre (ignorée). Les fichiers `donnees-f2.txt` et `donnees-erreurs.txt` sont des exemples de ce format.

Les données peuvent aussi être dans un fichier XML. Plusieurs DTD sont envisagées, numérotés de 0 à 8. Vous traiterez celle qui correspond à votre numéro modulo 8 (numero % 8). Par exemple, si vous avez le numéro 22, vous traiterez la DTD donnees6.dtd.

1.2 Les traitements

On veut pouvoir facilement réaliser plusieurs traitements qui pourront être combinés de différentes manières. De nouveaux traitements doivent aussi pouvoir être ajoutés. Aussi, on s'inspire du patron de conception *chaîne de responsabilité* [1], largement adapté.

La classe Traitement, abstraite, modélise la notion de traitement et l'enchaînement entre traitements. Chaque traitement peut avoir plusieurs traitements suivants. Une donnée est traitée par un traitement puis, potentiellement, par tous les traitements suivants. On implante donc une version du patron *chaîne de responsabilité* sous forme d'arbre de responsabilités.

Voici un exemple d'une telle chaîne :

```
GenerateurXML("brut--genere.xml")
    --> Positions --> Max --> Somme --> SommeParPosition
    --> SupprimerPlusPetit(< 0.0) --> SupprimerPlusGrand(> 10.0)
        --> GenerateurXML("valides--genere.xml")
        --> Positions --> Max --> Somme --> SommeParPosition
    --> Normaliseur(début=0.0, fin=100.0)
        --> GenerateurXML("normalisees--genere.xml")
        --> Positions --> Max --> Somme --> SommeParPosition
```

Chaque mot qui commence par une majuscule correspond à une classe de traitement et dans les parenthèses qui suivent apparaissent les paramètres de ce traitement (fournis lors de la construction du traitement). Le détail des traitements est décrit dans la suite. Les comprendre individuellement n'est pour l'instant pas nécessaire. Les flèches `-->` symbolisent l'enchaînement des traitements. Si un même traitement a plusieurs traitements suivants, les flèches sont à la ligne, indentées de la même manière pour chaque traitement suivant.

Ainsi, le premier traitement est `GenerateurXML` (il écrit un document XML dans le fichier `brut--genere.xml`). Ce traitement a trois chaînes de traitements suivantes. La première enregistre les positions, calcule le max des valeurs, la somme des valeurs et la somme des valeurs par position. La seconde chaîne de traitements supprime les valeurs négatives, puis les valeurs supérieures à 10 puis continue avec deux chaînes, la première qui produit le fichier XML `valides--genere.xml` et la seconde qui enregistre les positions, calcule le max, la somme et la somme par position (comme tout à l'heure). Enfin la troisième chaîne de traitements ressemble à la deuxième chaîne de traitements mais les données sont normalisées au lieu d'être supprimées et le fichier engendré s'appelle `normalisees--genere.xml`.

Dans le patron d'origine, la responsabilité de traiter la donnée est réalisée par un et un seul traitement et la donnée n'est alors pas transmise aux traitements suivants. Dans notre exemple, c'est le cas de `SupprimerPlusPetit` et `SupprimerPlusGrand`. Cependant, contrairement au patron d'origine, plusieurs traitements peuvent traiter les données. Ceci peut être en agrégeant des données localement comme `GenerateurXML`, `Positions`, `Somme`, `Max` qui calculent des données locales (production d'un fichiers XML, les positions vues, sommes de valeurs vues ou plus grande valeur) mais transmettent les données aux traitements suivants qui pourront aussi traiter ces données. Le traitement peut aussi transmettre aux traitements suivants une donnée modifiée. C'est par exemple le cas de `Normaliseur`.

1. Pour en savoir un peu plus sur le patron *chaîne de responsabilité*, on peut lire : `https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern`

La classe Traitement spécifie trois méthodes qui sont appelées par l'analyseur. Les méthodes `gererDebutLot` et `gererFinLot` sont appelées quand le traitement d'une source (un lot) commence et se termine, respectivement. En paramètre de ces deux méthodes une chaîne de caractères identifie le lot. La méthode `traiter` est appelée sur chaque donnée de la source avec comme paramètre la position et la valeur.

Les méthodes `gererDebutLot` et `gererFinLot` se propagent toujours sur les traitements suivants. C'est le traitement qui décide de propager la méthode `traiter` sur les traitements suivants. Par exemple un traitement qui consiste à supprimer les valeurs trop grandes ne transmettra aux traitements suivants que les données inférieures à un seuil.

Les traitements à faire sont :

1. Somme : somme de toutes les valeurs traitées, quelque soit le lot. On affichera la somme actuelle à la fin du traitement de chaque lot.

   ```
   lot1: somme = 77.39999999999999
   ```

2. Positions : toutes les positions traitées, dans l'ordre, quelque soit le lot.
3. Donnees : toutes les données traitées (position et valeur), dans l'ordre, quelque soit le lot.
4. Multiplicateur : transmet au traitements suivants la valeur multipliée par un facteur fourni en paramètre du constructeur.
5. SommeParPosition : somme les valeurs correspondant à une même position, quelque soit le lot. Voici un exemple d'affichage (qui sera fait à la fin du traitement de chaque lot).

   ```
   SommeParPosition lot1 :
     - Position@400(2,1) -> 18.0
     - Position@3e1(1,1) -> 39.4
     - Position@3e2(1,2) -> 20.0
   Fin SommeParPosition.
   ```

6. SupprimerPlusGrand (et SupprimerPlusPetit) : ne transmet pas aux traitements suivants les valeurs plus grandes (ou plus petites) qu'un seuil fourni en paramètre du constructeur.
7. Max : Le maximum des valeurs traitées, quelque soit le lot. Voici un exemple d'affichage (qui sera fait à la fin du traitement de chaque lot).

   ```
   lot1: max = 18.0
   ```

8. Normalisateur : normalise les valeurs de chaque lot en les ramenant dans un intervalle début..fin (paramètres du constructeur) en appliquant une transformation affine. Le principe est le suivant, si on note m et M la plus petite et la plus grande valeur, d et f le début et la fin de l'intervalle, alors chaque valeur x est remplacée par $a \times x + b$ avec $a = (f - d)/(M - m)$ et $b = f - a \times M$.

   Notons que l'on peut obtenir le min à partir du max : c'est l'opposé du max des opposés des valeurs. Par exemple, le min(2, 3) est -max(-2, -3).

9. GenerateurXML : produit à chaque fin de lot, un document XML avec toutes les données traitées (de ce lot et des précédents) organisées par lot (i.e. le lot d'appartenance d'une donnée doit se retrouver dans le document XML). Le nom du fichier dans lequel le document

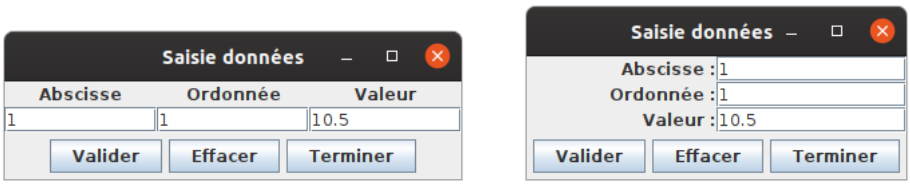
sera écrit sera un paramètre du constructeur de ce traitement. La DTD de ce document est à définir. Elle doit s'appeler `generateur.dtd`.

10. Vous choisirez ce dernier traitement. Soyez inventif !

Enfin, on contrôlera qu'aucun cycle n'est créé quand on ajoute des traitements à la suite d'un traitement existant. Il s'agit de vérifier que le traitement auquel on va ajouter des traitements n'est pas déjà dans les suivants de ces traitements. En cas de détection de cycle, on lèvera l'exception non vérifiée `CycleException` qui devra être définie.

1.3 Swing

Une interface graphique sera définie pour engendrer un fichier de type 1. Deux IHM sont envisagées figure 1(a) (pour les numéros impairs, par exemple 21) et 1(b) (pour les numéros pairs, par exemple 22). Le bouton « Effacer » vide les trois zones de saisies. Le bouton « Valider » enregistre les données (Abscisse, Ordonnée et Valeur). Les deux premières doivent être entières, la dernière réelle. Si ce n'est pas le cas, l'enregistrement n'aura pas lieu et la zone de saisie passera en fond rouge (`setBackground` de `JTextField` avec la couleur `Color.RED`). L'identifiant sera le numéro d'ordre de la donnée saisie (1 pour la première saisie). Enfin, le bouton « Terminer » enregistre les données dans un fichier (précisé lors du lancement de l'application comme argument de la ligne de commande) et quitte l'application. Fermer la fenêtre (ici, cliquer sur la croix rouge), ferme l'application sans enregistrer les données.



(a) en ligne (b) en colonne

FIGURE 1 – IHM possibles pour saisir les données

1.4 Introspection

Construire la chaîne de traitements peut être assez fastidieux. Aussi, nous proposons de pouvoir définir une chaîne de traitements sous forme textuelle (chaîne de caractères ou fichier). Pour ce faire, nous allons utiliser la classe `Scanner` du paquetage `java.util` et l'introspection. `Scanner` permet de lire un texte « mot » à « mot » grâce aux méthodes `next`, `nextInt()`, etc. On peut aussi savoir si le prochain « mot » est un entier avec `hasNextInt()`, etc.

Nous choisissons une syntaxe facile à analyser (et donc un peu moins facile à écrire). Pour chaque traitement, on indiquera son nom, ses paramètres, et ses suivants. Pour les paramètres, on indiquera le nombre de paramètres (un entier), le type du paramètre (un mot) puis le paramètre effectif. Pour les suivants, on indiquera le nombre de suivants puis la description de chaque

suivant. On constate donc que cette syntaxe est récursive, récursivité que l'on aura intérêt à mettre en œuvre dans le programme.

Voici un exemple qui correspond à la chaîne de la section 1.2 :

```
1  String calculs = "Positions 0 1 Max 0 1 Somme 0 1 SommeParPosition 0";
2  String generateur = "GenerateurXML 1 java.lang.String NOM--genere.xml";
3  String traitement1 = generateur.replaceAll("NOM", "brut") + " 3"
4      + " " + calculs + " 0"
5      + " " + "SupprimerPlusPetit 1 double 0.0 1 SupprimerPlusGrand 1 double 10.0 2"
6          + " " + generateur.replaceAll("NOM", "valides") + " 0"
7          + " " + calculs + " 0"
8      + " " + "Normaliseur 2 double 0.0 double 100.0 2"
9          + " " + generateur.replaceAll("NOM", "normalisees") + " 0"
10         + " " + calculs + " 0";
```

La chaîne `calculs` correspond au traitement `Positions` qui n'a pas de paramètre (0) suivi d'un traitement (1) qui est `Max` aussi sans paramètre (0) avec un traitement suivant (1) qui est `Somme` sans paramètre et avec pour traitement suivant `SommeParPosition` sans paramètre. La chaîne est incomplète car le nombre de suivants du dernier traitement n'est pas indiqué. Notons que `calculs` représente la chaîne de traitements que nous avions rencontrée trois fois dans notre exemple de la section 1.2. Elle nous permet donc de la factoriser.

La chaîne `generateur` décrit un générateur XML (`GenerateurXML`) qui prend un paramètre (1) dont le type est `java.lang.String` et la valeur est `NOM--genere.xml`. Là encore, il manque le nombre de traitements suivants.

Enfin, la chaîne `traitement1` correspond à l'exemple complet de la section 1.2 dans lequel on réutilise les deux chaînes précédentes plusieurs fois. Ainsi, sur la ligne 3, on réutilise `generateur` en remplaçant « `NOM` » par « `brut` » et on indique qu'il y a trois traitement suivants qui sont donnés sur les lignes suivantes (3, 5 et 8).

Conseils : Il est conseillé d'adopter une approche ADR (Analyse Descendante Récursive) : on écrit une méthode par « morceau » de la grammaire. Dans le code fourni, la méthode `traitement` est complète. Elle appelle la méthode `analyserTraitement` qui analyse un traitement complet (y compris ses paramètres et ses suivants), `analyserCreation` analyse la création d'un objet et retourne l'objet instancié, `analyserSignature` analyse les paramètres d'un traitement et retourne les paramètres formels et les paramètres effectifs (regroupés dans un objet de type `Signature`), etc. Ces méthodes sont présentes dans la classe `TraitementBuilder`.

Remarque : On supposera que les types donnés pour les paramètres permettent d'identifier le constructeur à utiliser. On ne traitera donc pas les cas de sous-typage, de conversion implicite, de nombre variable d'arguments, etc.

Référencer les traitements : *(facultatif)* Pour exploiter ces traitements par la suite (par exemple pour récupérer le max, la somme, la somme des valeurs pour une position donnée), il serait souhaitable de garder un accès sur les traitements ainsi construits.

Une solution consiste à compléter notre description de la chaîne de traitement en ajoutant la définition d'un identifiant devant un traitement. Le traitement sera alors stocké dans un tableau associatif avec comme clé l'identifiant. Bien sûr, pour pouvoir exploiter ses méthodes spécifiques, il faudra transtyper le traitement (ou utiliser l'introspection).

Dans l'exemple suivant, on a mis un identifiant devant chaque traitement. C'est le mot-clé `id` qui dit que l'on définit l'identifiant et le mot qui suit est l'identifiant lui-même. Avec `Scanner`, on pourra utiliser l'appel `hasNext("id")` pour savoir si le mot suivant est "id" ou pas, et donc savoir si un identifiant a été défini.

```
id X-p Positions 0 1 id X-m Max 0 1 id X-s Somme 0 1 id X-S SommeParPosition 0 0
```

Ici, on a choisi un identifiant qui commence par « `X-` » car cette chaîne sera utilisée plusieurs fois. On pourra alors faire un `replaceAll("X-", ...)` pour avoir des identifiants différents pour ses différentes utilisations (comme pour la chaîne `generateur`).

1.5 L'existant

Une version très préliminaire de l'application a été créée. Elle se limite à calculer la somme des valeurs issues d'un fichier texte de type 1. Il s'agit de la classe `AnalyseurInitial`. Conscient que cette approche n'était pas viable pour traiter la multitude des sources (d'autres pourraient être ajoutées) et des traitements (il y en aura d'autres aussi), quelques classes de la nouvelle architecture ont été définies.

La classe centrale est `Analyseur` qui fait le lien entre les traitements et les sources. Les sources sont abstraites sous la forme d'un itérable qui permet de récupérer successivement chacun des éléments de la source, éventuellement sans avoir à tous les consulter si l'information attendue est au début. Le paramètre de généricité de `Iterable` est :

```
AbstractMap.SimpleImmutableEntry<Position, Double>
```

Nous utilisons `AbstractMap.SimpleImmutableEntry` car l'API Java ne fournit pas de classe pour représenter une paire. Il y en a une [2] dans JavaFX (`javafx.util.Pair`) et une dans la bibliothèque Apache Commons Lang mais nous ne voulons pas « tirer » ces bibliothèques pour une seule classe.

Certains traitements ont été spécifiés plus avant (`SommeAbstrait`, `PositionsAbstrait`, etc.).

Les classes modélisant les sources d'information sont à faire.

La classe `Position` est définie. Mais comme vous le montreront les programmes de tests (`PositionsTest` en particulier), l'auteur n'a pas été assez attentif en cours (ou plutôt en TD) et ne l'a donc pas définie correctement.

L'interface `FabriqueTraitement` et sa réalisation `FabriqueTraitementConcrete` permettent de s'abstraire de la classe qui réalise vraiment une spécification. Ceci est particulièrement vrai pour `Somme` et `Positions`, traitements pour lesquels une spécification est fournie au moyen des classes `SommeAbstrait` et `PositionsAbstrait`.

2 Livrables

Les livrables sont :

2. Voir `https://stackoverflow.com/questions/5303539/didnt-java-once-have-a-pair-class`.

1. Le fichier `questionnaire.txt` complété.
2. Le code source de l'application.
3. Les fichiers d'exemples, en particulier XML et DTD associées.

3 Calendrier

Les principales dates sont :

1. 22 avril : mise en ligne du sujet.
2. 23 avril : création des dépôts git avec accès donné aux enseignants.
3. 23 avril : 1/2 séance sur le projet (comprendre).
4. 22 mai : séance de TP sur le projet.
5. 1er juin : remise première version (via Git).
6. 1er juillet : remise version finale (via Git). D'autres modifications pourront être poussées ensuite mais elles ne seront pas forcément prises en compte dans la correction.