Ecole nationale de la statistique
et de l'analyse de l'information

Projet de Traitement de Données

1ère Année

Etudiants :
Ludovic Deneuville
Jean-Philippe Trotta
Laurène Villacampa

Professeur :
Benjamin Girault
Encadrant :
Natacha Njongwa Yepnga

2021 - 2022

Sommaire

Introduction

Ce projet s'inscrit dans le cadre du cours d'introduction à la Programmation Orientée Objet (POO). L'objectif principal est donc de mobiliser les notions acquises dans ce cours, en utilisant le langage de programmation Python. Ce type de programmation est centré autour de la notion d'objet et donc de classe : définissant des attributs et des méthodes, la création d'une classe autorise le programmeur à avoir l'entier contrôle de ce que peut réaliser l'utilisateur avec les éléments de cette classe (les objets). Ce principe d'encapsulation est au fondement de la POO et nous nous sommes attachés à le rendre explicite dans notre travail.

Le thème général du projet est l'étude du lien entre climat et énergie, à travers la réalisation d'une application permettant l'analyse statistique de données météorologiques et de consommation d'électricité. L'ouverture et la complexité du sujet proposé nous ont d'abord amenés à réaliser un cahier des charges afin de mieux cerner les attendus du projet. La réalisation du diagramme UML de cas d'utilisation nous a aidés à définir les classes que nous allions implémenter par la suite, et à les organiser : chacune dans un module, et articulées dans des paquets que nous détaillerons ultérieurement. Nous avons recensé toutes les classes dans un diagramme UML de classe que nous présenterons également. Nous avons ainsi pu mettre à profit les relations entre les classes que nous avons vues en cours : association, agrégation, ou héritage. Nous avons essayé d'utiliser au maximum la relation d'héritage pour faciliter l'implémentation de nouvelles fonctionnalités et donc la réutilisation du code. Nous avons également tenté de proposer des noms explicites et neutres, tant pour les attributs que pour les méthodes, de façon à accroître sa reproductibilité. Nous avons construit l'architecture de notre application en vue de pouvoir répondre à certaines questions :

— Comment mettre en forme les données fournies pour permettre leur traitement ?

— Comment enchaîner une série d'opérations sur les tables de données ? A ce sujet, nous avons découvert la notion de pipeline et implémenté une classe du même nom.

— Quelles fonctionnalités permettraient de répondre à des questions d'analyses statistiques sur les liens de corrélation entre deux variables ? Nous emploierons parfois le terme de "variable" au sens statistique, qui n'est pas à interpréter au sens homonyme de variable informatique.

— Comment « nettoyer » et combiner nos données en leur faisant subir diverses transformations ?

Dans ce rapport, nous présentons d'abord le cahier des charges avec la description des jeux de données à notre disposition, les contraintes techniques que nous devions respecter et les fonctionnalités attendues pour l'utilisateur de notre application avec le diagramme de cas d'utilisation. Nous expliquons ensuite comment nous avons géré l'import des données selon les différents formats (.csv et .json) fournis en compression (.gz), ainsi que l'export. Nous présentons à cette occasion la classe `TableDonnees` que nous avons implémentée pour conceptualiser les tables de données comme « objets ». Nous proposons enfin une présentation plus globale de l'ensemble des paquets de l'application, en détaillant les classes qui les composent, l'objectif fonctionnel des classes utilisées, leurs attributs et leurs méthodes, ainsi que leur relation avec d'autres classes.

1 Cahier des charges

Le but du projet est de réaliser un programme permettant d'analyser des données. Dans notre cadre d'étude, il s'agit de mettre en place un processus de traitement de fichiers de données. En sortie de notre programme, l'utilisateur obtiendra des données propres qu'il pourra utiliser pour des traitements statistiques. Dans un premier temps, les données utilisées sont les relevés météorologiques et les relevés de consommation électrique en France et sur les dix dernières années.

1.1 Jeux de données

Nous disposons de deux jeux de données :

- Jeu de données issu du site de Météo France :
  Ce jeu de données recense des données relatives à la météo mesurées dans les stations françaises (hors Corse) entre janvier 2023 et mars 2022. Il contient un fichier .csv par mois :
  - Maille géographique : station
  - Pas temporel : 3 Heures
  - Identifiants : instant (Date-heure) et station (ID)
- Jeu de données issu du site de Réseaux Energie :
  Ce jeu de données donne la consommation quotidienne brute régionale en électricité, exprimée en MW, en France (hors Corse) sur la période allant de janvier 2013 à décembre(!) 2022. Il contient un fichier .json par mois :
  - Maille géographique : région
  - Pas temporel : 1/2 Heure
  - Identifiants : instant (Date-heure) et région (code INSEE region)

Le programme issu de ce travail doit donc implémenter des fonctionnalités permettant l'analyse des données proposées. Il doit en outre pouvoir être réutilisé assez facilement, et être aisément adaptable à de nouvelles utilisations.

1.2 Contraintes techniques

Plusieurs contraintes techniques sont imposées pour ce projet.

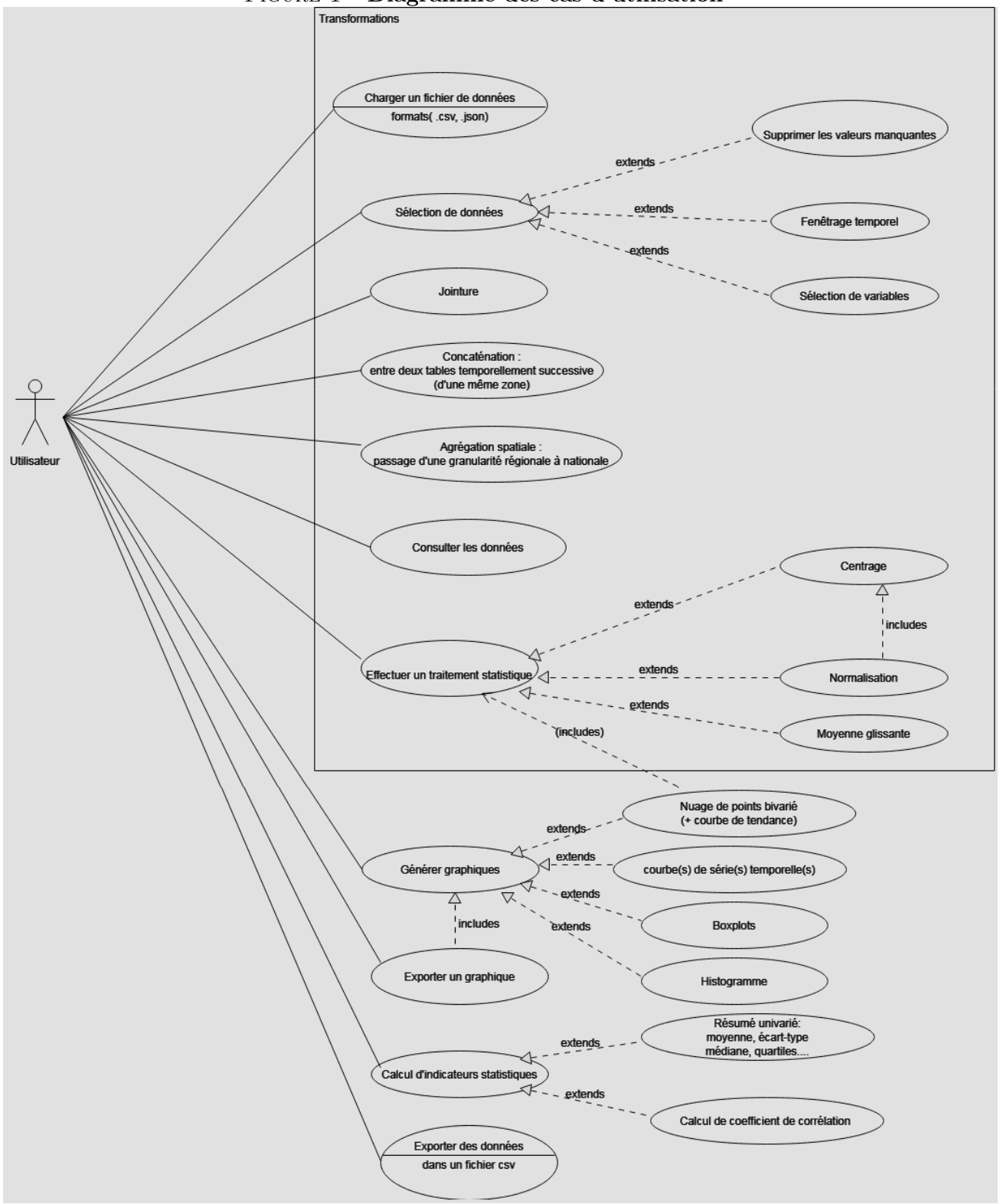
— Le travail demandé doit être réalisé dans le langage Python. Certains packages sont autorisés, à savoir `numpy`, `matplotlib` et `scipy`. A contrario, l'utilisation du package `pandas` est interdite. D'autres packages peuvent éventuellement être utilisés sur autorisation. Le module `datetime` qui fait partie de Python peut être utilisé.

— Dans une logique de programmation orientée objet, le programme doit être modulaire, réutilisable et évolutif. Toutes les classes devront être documentées et les méthodes testées.

1.3 Résultats attendus

Les utilisations attendues de ce programme sont résumées dans le diagramme de cas d'utilisation ci-dessous :

FIGURE 1 – **Diagramme des cas d'utilisation**

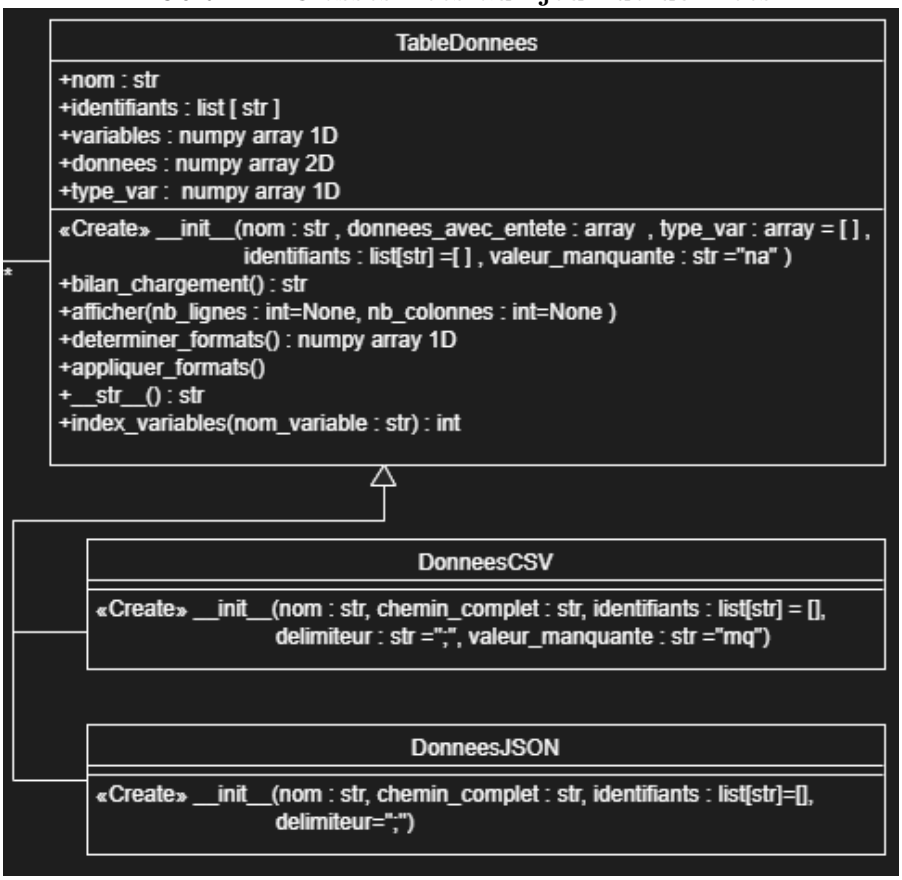


2 Gestion des données

2.1 Import et stockage des données

Comme précisé dans le cahier des charges, les données utilisées sont issues de deux types de fichiers : des données météorologiques issues de fichiers .csv et des données de consommation d'électricité issues de fichiers .json. Pour tenir compte de cette diversité des sources, et afin de faciliter l'implémentation du chargement de données dans de nouveaux formats, nous avons implémenté une classe nommée `TableDonnees`, dont deux classes filles héritent : `DonneesCSV` et `DonneesJSON`. Ci-dessous le diagramme des classes correspondant :

FIGURE 2 – **Classes liées aux jeux de données**



2.2 La classe `TableDonnees`

Un objet de `TableDonnees` possède les attributs suivants :

— `nom` : une chaîne de caractères pour l'identifier

— `identifiants` : la liste des variables définies comme identifiants. Cette liste sert à figer le format de ces variables à « str » ou « date ». Nous avons en effet des méthodes de détection et d'application des formats qui pourraient transformer une variable servant d'identifiant en « float ». Cela permet de mettre dès le début un garde-fou pour éviter par exemple, qu'une Moyenne soit appliquée sur ces variables.

— `variables` : de type numpy array 1D, il contient la liste des variables. Celle-ci est obtenue lors de l'instanciation en récupérant la première ligne du paramètre `donnees_avec_entete`

— `donnees` : tableau de type numpy array 2D contenant les données. Cet attribut est alimenté dans le constructeur en prenant le contenu du paramètre `donnees_avec_entete` privé de la première ligne.
— `type_var` : la liste des types de chaque variable. Celle-ci est soit renseignée par l'utilisateur, soit obtenue après application d'une méthode qui détermine le type (le "format") de chaque variable.

Le choix du recours au package `numpy`

Nous avions commencé par définir les attributs des classes avec des listes ou des listes de listes. Cependant, nous nous sommes rapidement rendus compte que les `numpy array` offrent d'avantage de possibilités, particulièrement dans le cas de tableaux en 2D. Si nous avions conservé les listes de listes, nous aurions dû développer des méthodes déjà disponibles avec le package `numpy`.
Un exemple de méthode très utile fournie par `numpy` est l'extraction d'une colonne d'un tableau en deux dimensions et sa transformation en liste.

Les méthodes suivantes sont définies pour un objet :

— `__init__(nom, donnees_avec_entete, type_var = [], identifiants = [], valeur_manquante = "na")` : constructeur qui initialise par défaut l'attribut `list_var` comme liste vide, et la liste des identifiants également à une liste vide. Le constructeur prend en paramètre la table `donnees_avec_entete` qui permet de définir les attributs `donnees` et `variables`
— `bilan_chargement()` : récapitule les informations chargées en donnant notamment le nom de la table, le nombre de lignes et de colonnes de la table obtenue.
— `afficher(nb_lignes = None, nb_colonnes = None)` : permet d'afficher la table de données, en précisant un nombre de lignes et de colonnes voulues. Par exemple, `afficher(10,5)` va afficher dix lignes et cinq colonnes et `afficher(nb_colonnes = 8)` va afficher toutes les lignes et huit colonnes.
— `determiner_formats()` : méthode permettant, à partir des données observées, d'attribuer un type à chaque variable
  ❍ Si la variable contient le mot « date », le type de cette variable sera `date`
  ❍ Si la variable fait partie de la liste des identifiants, celle-ci reste de type `str`
  ❍ Si toutes les données d'une variable sont de type `float`, la variable sera de type `float`
  ❍ Sinon la variable reste de type `str`
— `appliquer_format()` : méthode permettant de transformer les données associées à une variable de type `float` en `float`. Concernant les variables de type date, celles-ci sont converties en `int` au format « YYYYMMDDHHMISS ». Nous aurions pu utiliser le type `datetime` mais le format choisi répondait déjà à notre principal besoin, c'est à dire comparer des dates.
— `index_variable(nom_variable)` : méthode pour récupérer l'index d'une variable (c'est-à-dire l'indice de sa colonne dans la table de données).

2.3 Gestion des formats .csv et .json

Les deux classes `DonneesCSV` et `DonneesJSON` héritent des attributs et des méthodes précisés pour la classe `TableDonnees`. Nous avons introduit deux paramètres supplémentaires dans leurs constructeurs :

— `chemin_complet` : une chaîne de caractère précisant le chemin d'accès au fichier voulu

— `delimiteur` : le délimiteur pris en compte lors de l'import. Le delimiteur est paramétré par défaut à « ; » pour les fichiers de type .csv

- Format .csv ou .csv.gz :
  L'import de la table fonctionne également si le fichier .csv est contenu dans une archive .gz. Les valeurs manquantes sont recherchées par défaut sous la forme « mq » (valeur utiliée dans les fichiers météo). L'attribut `type_var` est instancié grace au résultat de la méthode `determiner_format()`.
- Format .json ou .json.gz :
  L'import est également possible si le fichier est archivé.

Le chargement d'un fichier .json est plus complexe qu'un fichier .csv. En effet, un fichier .json étant construit sous la forme de dictionnaire de dictionnaires, les variables ne sont pas les mêmes selon les individus. Ainsi il y a une étape préliminaire qui consiste à parcourir tous les individus pour obtenir la liste de toutes les variables.

Dans un second temps, le dictionnaire des individus est de nouveau parcouru. Pour chaque individu, l'algorithme essaie de récupérer les valeurs pour chacune des variables listées lors de l'étape précédente. Si la variable n'est pas trouvée, ce champs est renseigné à « nan » pour cet individu dans l'attribut `donnees`. Au final le constructeur le la classe DonneesJSON permet de convertir le contenu d'un fichier .json en objet de type TableDonnees.

2.4 Gestion des exports

Initialement nous avions prévu de gérer les exports de tables par un booléen en fin de pipeline, mais cela n'autorisait qu'un export potentiel en fin de procédure. Nous avons finalement décidé de créer une classe `Export` que nous détaillerons dans la partie sur la classe `Transformations`. Cela permet de considérer l'export comme une des opérations de la liste lancée via le pipeline, et ouvre ainsi la possibilité d'exporter plusieurs résultats au cours du traitement. Nous nous sommes limités à des exports au format .csv.

3 Présentation de l'application

3.1 Architecture globale de l'application

L'application est composée de plusieurs paquets :

— `estimateur` : paquet composé de modules permettant de calculer des estimateurs statistiques sur une variable ou sur toutes les variables d'une table. On trouve notamment dans ce paquet les deux modules `moyenne.py` et `ecarttype.py`, consacrés respectivement aux classes `Moyenne` et `EcartType`. Ces deux classes héritent d'une classe abstraite `AbstractEstimateur` définie dans le module `estimateur.py`.

— `lienvar` : paquet permettant d'étudier les corrélations entre deux variables, selon leurs types (qualitatif ou quantitatif). Ce paquet regroupe ainsi un module `coefficientcorrelation.py` (pour deux variables quantitatives), un module `testchisquare.py` (pour deux variables qualitatives), et un module `anova.py` (pour une variable qualitative et l'autre quantitative). On y a défini des classes `CoefficientCorrelation`, `TestChiSquare` et `Anova` qui héritent d'une classe abstraite `LienVar` définie dans le module `lienvar.py`. Ces classes prennent également en charge la réalisation de graphiques associés à chaque type de couple de variables, ainsi que son export. En conséquent, nous y avons également inclus le module `temporel.py` contenant la classe `Temporel` (également en héritage de `LienVar`) pour gérer la réalisation de graphiques temporels (pour une variable quantitative en fonction d'une variable `date`).

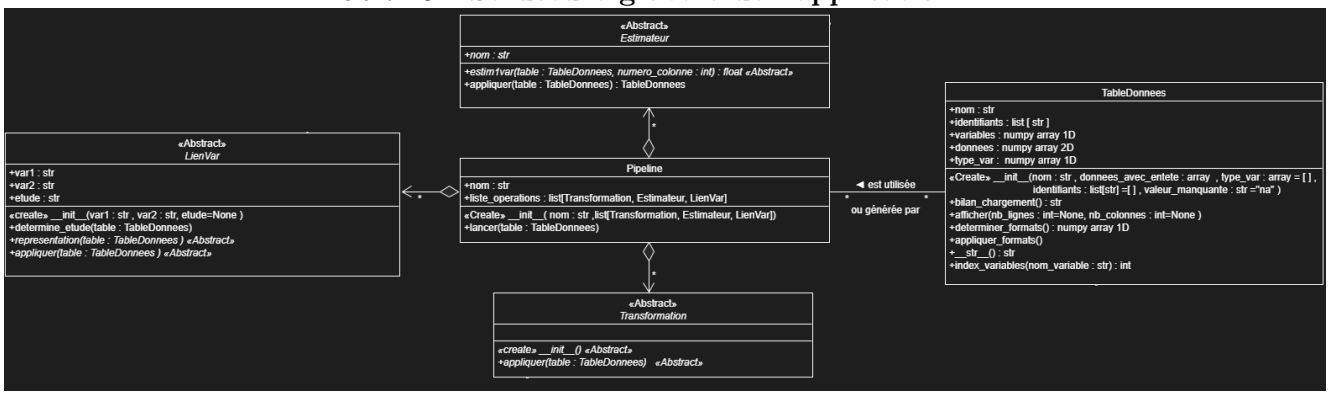
— `table` : ce paquet regroupe les modules définissant les classes abordées dans la section Gestion des données : `donneescsv.py` et `donneesjson.py` définissant respectivement les classes `DonneesCsv` et `DonneesJson`, toutes deux héritant de la classe `TableDonnees` définie dans le module `tabledonnees.py`.

— `transformation` : paquet composé des modules définissant des classes permettant de réaliser une « transformation » des données. Toutes ces classes héritent de la classe abstraite Transformation définie dans le module `transformation.py`. Nous étudions les classes implémentées plus en détail par la suite.

— `pipeline` : ce paquet est constitué d'un unique module `pipeline.py`, pierre angulaire de l'application puisqu'il définit la classe `Pipeline`, permettant de d'appliquer une suite d'opérations (des transformations, des calculs d'estimateurs, des études de liens entre variables avec représentation graphique à l'appui) à une table de données en vue de réaliser un traitement statistique, et de décider notamment de l'export des résultats obtenus.

Cette organisation générale est résumée par le diagramme des classes partiel suivant :

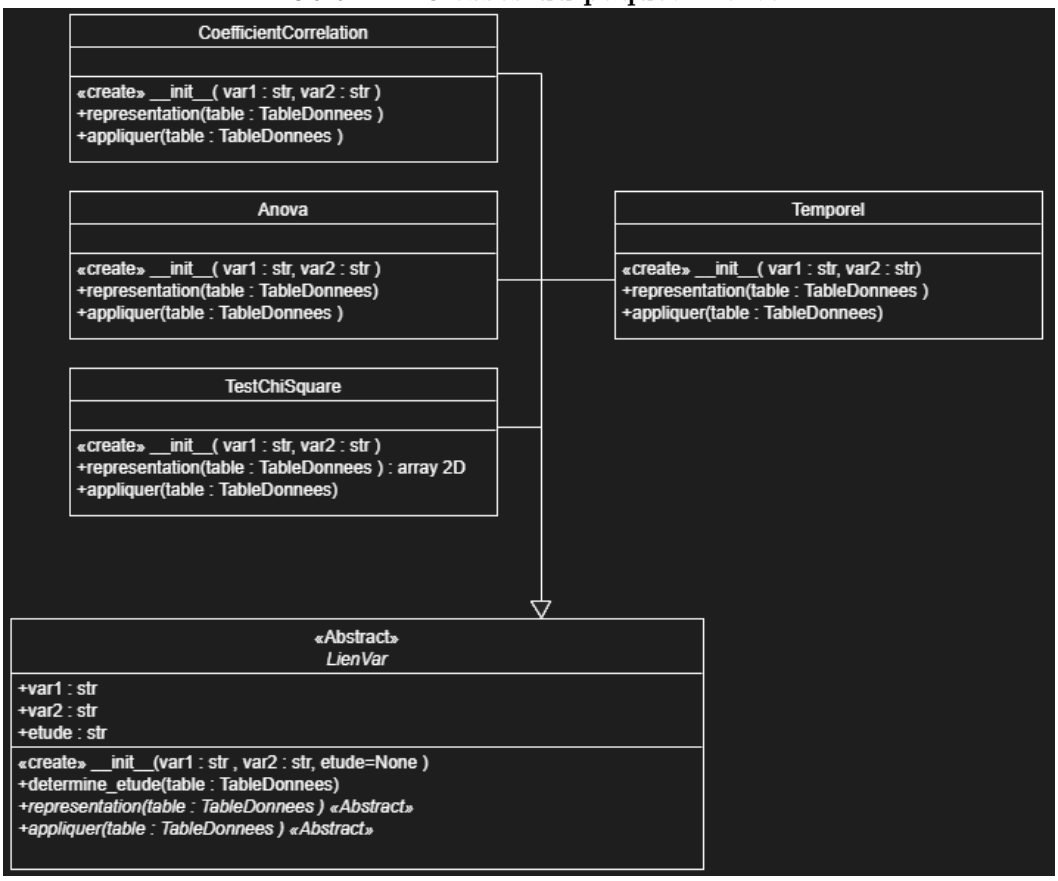
FIGURE 3 – **Structure globale de l'application**



3.2 Les classes du paquet `lienvar`

Nous avons implémenté une classe abstraite `LienVar` dont héritent les classes `CoefficientCorrelation`, `TestChiSquare`, `Anova` et `Temporel`. Voici le diagramme de classe correspondant :

FIGURE 4 – **Classes du paquet `lienvar`**



La classe mère : `LienVar`

La classe `LienVar` a trois attributs :

— `var1` et `var2` (de type str) : pour définir le nom des variables de la table de données sur lesquelles on étudiera la relation

— `etude` (de type str) : pour préciser la nature de l'étude qui va être réalisée entre les deux variables, selon si elles sont qualitatives, quantitatives ou de type `date`.

Dans le constructeur de cette classe, le paramètre `etude` est initialement défini par `None` avant d'être modifié dans la méthode `determine_etude()`. Cette méthode prend en entrée un objet `table` (d'où sont issues `var1` et `var2`), de type `TableDonnees`. En fonction de l'attribut `type_var` correspondant aux colonnes de `var1` et `var2`, elle attribue à `etude` une des modalités suivantes : "quanti/quanti", "quali/quanti", "quali/quali" ou "date/quanti".
Cette classe a enfin deux méthodes abstraites, qui ont le même paramètre `table` que la méthode précédente :

— `representation()` : pour faire un graphique ou un tableau de contingence selon ce qui est contenu dans `etude`. Pour les graphiques, un export est implémenté au format .png.

— `appliquer()` : exécute la méthode précédente en la complétant par un affichage d'informations caractérisant la relation entre les deux variables statistiques étudiées.

Cette dernière méthode est celle appliquée par le Pipeline. Par contre, elle ne modifie par la table donnée, donc elle peut s'appliquer à n'importe quelle(s) étape(s) du Pipeline.

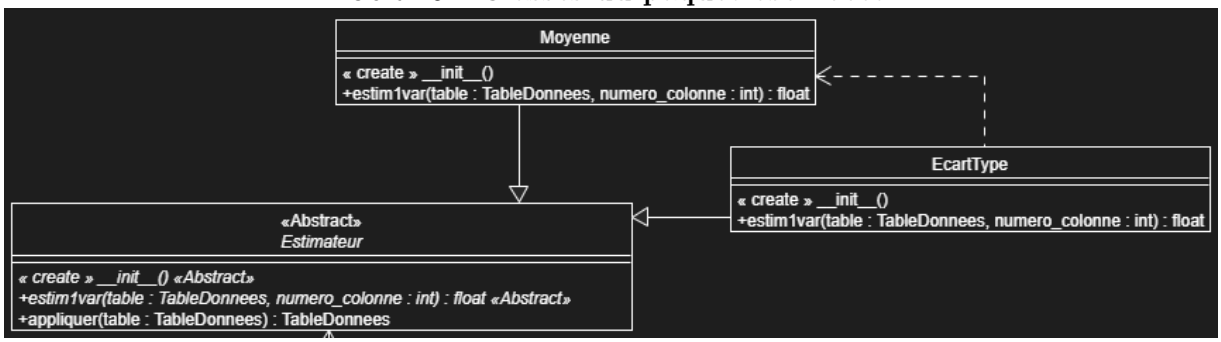
Les classes filles : `CoefficientCorrelation`, `TestChiSquare`, `Anova`, `Temporel`

L'ensemble des classes filles héritent des trois attributs de `LienVar` sans en définir d'autres. Le constructeur fait donc appel à celui de la classe mère. Pour chaque classe fille, la méthode `representation()` commence par un test sur la « valeur » de `etude` et vérifie que le type des variables est le bon pour l'étude concernée, sinon elle retourne un message d'erreur.
Par exemple, la classe `CoefficientCorrelation` ne s'applique que lorsque `etude` a la valeur « quanti/quanti ». Dans ce cas, la méthode `reprensentation()` affiche un nuage de points explicitement légendé et exporte son image sous un nom de fichier spécifique. Enfin, la méthode `appliquer()` rappelle la précédente et complète par l'affichage du coefficient de corrélation entre les deux variables et un commentaire d'interprétation du résultat.
Sur le même principe, dans la classe `Anova` la méthode `representation()` affiche un boxplot après avoir vérifié que `etude` est égal à « quali/quanti ». Dans la classe `TestChiSquare` l'affichage sera remplacé par un tableau de contingence après avoir testé que `etude` est égal à « quali/quali ». Enfin, lorsqu'une variable est quantitative et l'autre une date, l'affichage sera aussi un nuage de points selon la classe `Temporel`.

3.3 Les classes du paquet `estimateur`

Le but ici est de pouvoir calculer la moyenne ou l'écart-type, d'une variable ou de toutes les variables d'une table. Afin de faciliter l'implémentation de nouveaux calculs du même type, nous avons généré une classe abstraite et des classes filles pour chaque type de calcul. Voici le diagramme de classe correspondant :

FIGURE 5 – **Classes du paquet `estimateur`**



Ainsi la classe `Estimateur` possède une méthode abstraite `estim1var(table, numero_colonne)` qui permet de calculer l'estimateur sur la variable de l'objet `table` (de type `TableDonnees`) dont l'index est précisé par `numero_colonne`. La méthode `table_estimateur(table)` automatise quant à elle l'application de l'estimateur à toutes les colonnes de `table` (pris en paramètre).

Les classes `Moyenne` et `EcartType` héritent de cette classe abstraite. Dans chaque cas la méthode `estim1var(table, numero_colonne)` est spécifiée.

3.4 Les classes du paquet `table`

Cette partie a déjà été traitée dans la section Gestion de données. Nous n'y revenons pas plus en détails ici.

3.5 Les classes du paquet `transformation`

Nous avons identifié plusieurs types de transformations, certaines permettant de réaliser un calcul sur les données d'une table ou une modification des données, d'autres permettant une combinaison de tables. Toutes nos classes héritent donc de la classe abstraite `Transformation`. Chaque classe fille de `Transformation` contient une méthode `appliquer()`, prenant en paramètre une instance de la classe `TableDonnees`. Nous avons décidé d'ajouter à cette classe une transformation un peu à part permettant l'export des tables de données au format csv, comportant une unique méthode statique. Cela permet de réaliser plusieurs exports au cours du pipeline en l'intégrant dans la liste des transformations.

L'application implémente ainsi les classes suivantes :

— `Centrage` : centrage des variables de type `float`. Utilise notamment la classe Moyenne.

— `Concatenation` : concaténation des lignes de deux tables de données dont les variables sont identiques.

— `Export` : extraction d'un jeu de données vers un fichier au format .csv

— `Filtre` : application d'un filtre de modalités ou d'un fenêtrage temporel.

— `JointureInterne` : jointure interne entre deux tables.

— `MoyenneGlissante` : remplace les valeurs numériques d'une table par la moyenne des valeurs de lignes voisines selon un pas donné.

— `Normalisation` : centrage et réduction des variables de type float sans modifier les autres.

— `Reduction` : réduction d'une ou de plusieurs variables de type float.

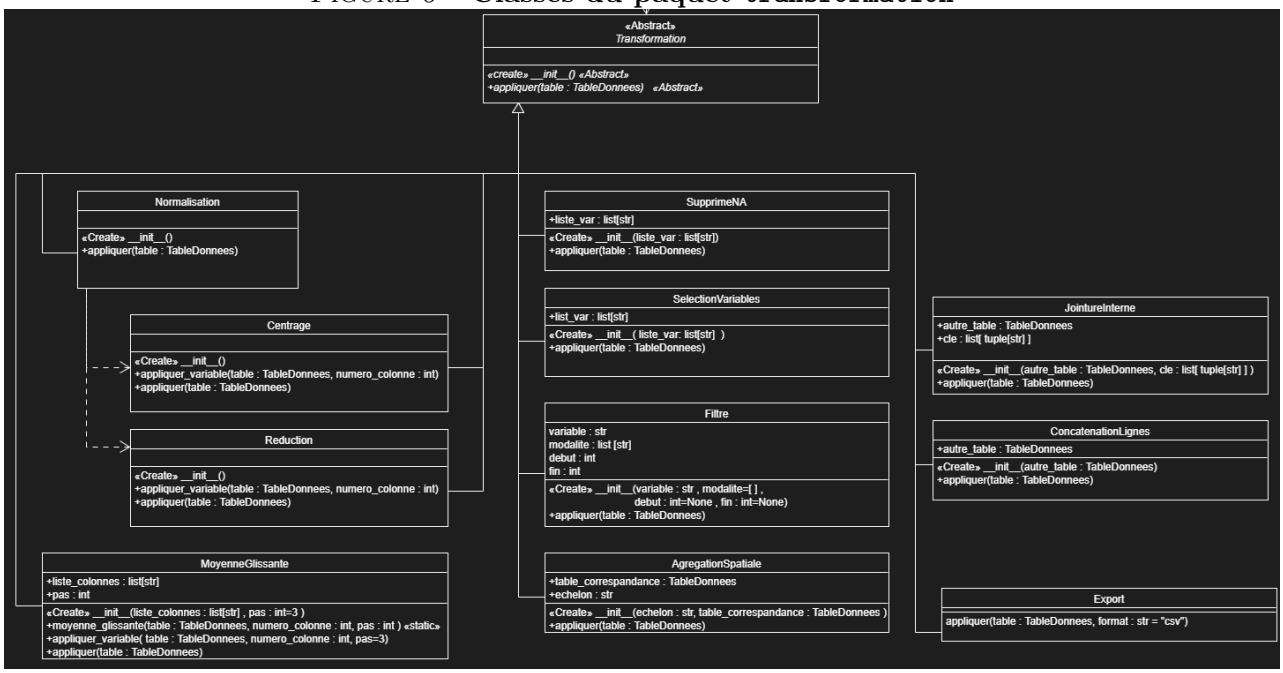
— `SelectionVariables` : extraction d'une sous-table en conservant certaines variables spécifiées.

— `SupprimeNA` : suppression des valeurs manquantes.

— `AgregationSpatiale` : agrège les données numériques d'une table (à l'aide d'un calcul de moyenne) pour passer des données par station aux données par région.

Voici le diagramme de classe correspondant :

FIGURE 6 – **Classes du paquet `transformation`**

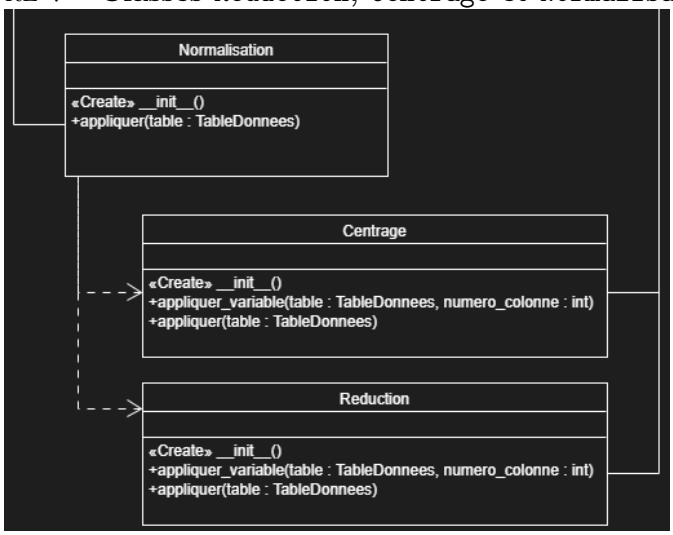


Nous donnons ici quelques précisions concernant les classes, avec des zooms sur les portions de diagramme UML correspondantes :

— Toutes les classes possèdent une méthode `appliquer` : c'est ce qui nous permet de faire appel à toutes ces transformations dans le pipeline dont nous traiterons juste après. Le seul paramètre de cette méthode de type `TableDonnees`.

— Les classes `Reduction` et `Centrage` permettent de réaliser ces opérations soit sur une variable (via la méthode `appliquer_variable`) soit à la table entière (via la méthode `appliquer`).

FIGURE 7 – **Classes `Reduction`, `Centrage` et `Normalisation`**



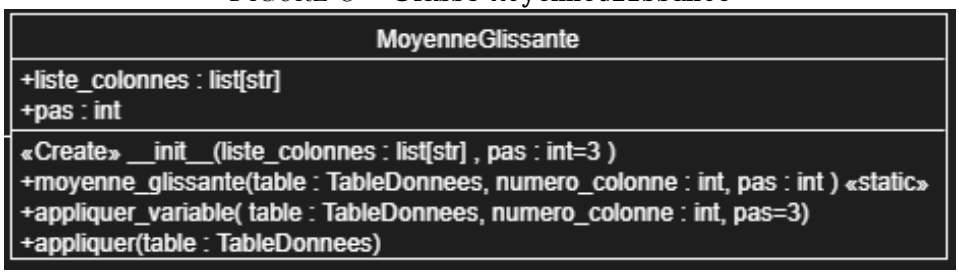
Même si cela n'apparaît pas explicitement sur le diagramme UML par soucis de lisibilité, ces classes utilisent les méthodes des classes `Moyenne` et `EcartType` définies dans le paquet

`estimateur` et décrites plus haut. Ainsi, la modification des données d'une table n'impacte que les variables de type `float`. La classe `Normalisation` (qui a recours aux deux précédentes) permet seulement de normaliser toutes les variables d'une table. Ces trois classes n'ont aucun attribut.

— La classe MoyenneGlissante a pour attributs :

- `liste_colonnes` : une liste de chaînes de caractères correspondant aux noms des variables statistiques sur lesquelles appliquer la transformation
- `pas` : un nombre entier (type int) qui donne le nombre de lignes autour de la valeur à remplacer par leur moyenne. Par exemple, si on l'initialise à pas=3 dans le constructeur, la valeur de la 10ème ligne sera remplacée par la moyenne des lignes 9, 10 et 11. Nous avons laissé cette initialisation par défaut dans le constructeur si l'utilisateur ne choisit pas un autre pas, mais nous avons aussi traité le cas d'un pas pair dans le code.

FIGURE 8 – **Classe** `MoyenneGlissante`



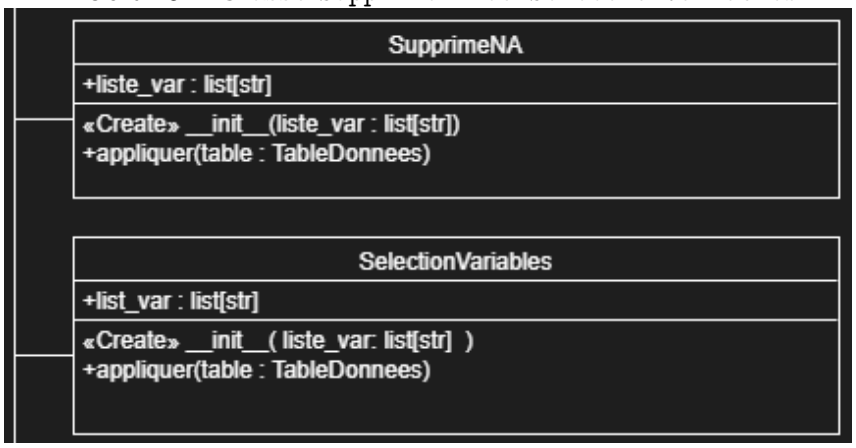
La classe `MoyenneGlissante` possède donc trois méthodes en plus de son constructeur :

- La méthode `moyenne_glissante()` est statique (pour plus de lisibilité, sinon elle aurait pu être intégrée comme une première partie du code de la méthode suivante), elle s'applique avec trois paramètres : une instance de `TableDonnees`, un `numero_colonne` sur laquelle appliquer les calculs, et le `pas`. Cette méthode retourne la liste des moyennes glissantes de la variable correspondant à `numero_colonne` pour l'utiliser dans la méthode suivante.
- La méthode `appliquer_variable()` a les mêmes paramètres que la précédente, et elle remplace les données de la colonne correspondant à `numero_colonne` par la liste des moyennes glissantes retournée par la méthode précédente.
- Enfin, la méthode `appliquer()` qui a toujours pour unique paramètre une instance de `TableDonnees`, applique la méthode précédente à toutes les variables numériques contenues dans l'attribut `liste_colonnes`.

— Les classes `SupprimeNA` et `SelectionVariables` ont pour attribut `liste_var`, une liste de variables. Leurs méthodes `appliquer` prennent en paramètre `table`, un objet de type `TableDonnees`.

- La classe `SupprimeNA` permet de supprimer les lignes de l'attribut `donnees` (d'une instance de TableDonnees) dont les variables spécifiées dans `liste_var` comportent des valeurs manquantes. Nous avons ainsi fait le choix de pouvoir cibler les variables qui nous intéressaient.
- La classe `SelectionVariables` modifie les attributs `donnees`, `variables` et `type_var` d'une instance de TableDonnees, en ne conservant que les variables qui sont spécifiées dans `liste_var`. `liste_var` est le seul attribut de la classe `SelectionVariables` : une liste de chaîne de caractères, correspondant à des noms de variables de la table de données.

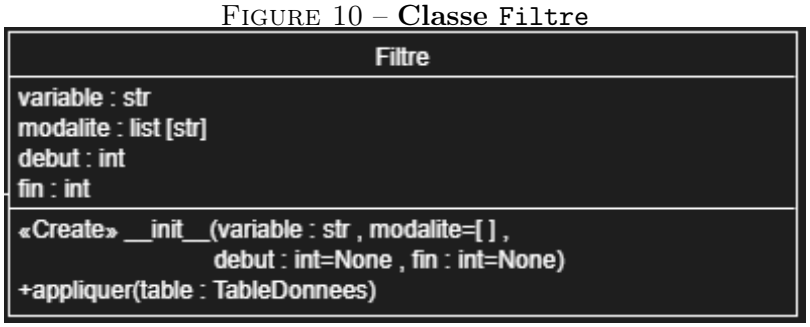
FIGURE 9 – **Classe** `SupprimeNA et SelectionVariables`



— La classe `Filtre` a quatre attributs :

- `variable` (de type str) : le nom de la variable sur laquelle on veut appliquer le filtre
- `modalite` (liste de str) : la liste des modalités de cette variable à conserver
- `debut` : une date au format YYYYMMDDHHMMSS, date à partir de laquelle on conserve les données
- `fin` : une date au format YYYYMMDDHHMMSS, date où le filtre s'arrête.

FIGURE 10 – **Classe** `Filtre`

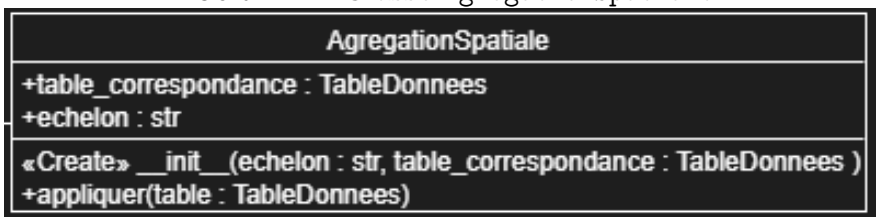


Dans le constructeur, le filtrage par modalité est initialisée à `modalite=[]`, de même pour le filtrage temporel avec `debut=None` et `fin=None`. Ceci permet de ne pas empêcher de créer le filtre quand l'une ou l'autre des informations n'est pas renseignée. La méthode `appliquer()` a encore pour paramètre `table` (instance de `TableDonnees`). Elle propose un traitement différencié selon si la variable n'est pas trouvée dans la table (message d'erreur), si elle est de type `date` ou si elle est de type `float` ou `str`. La méthode récupère le type de la variable via l'attribut `type_var` de la classe `TableDonnees` et réalise le cas échéant un filtre sur les modalités ou un filtre temporel.

— La classe AgregationSpatiale : Cette classe reste à finaliser. Elle répond à notre besoin de changer d'écehelon géographique, notamment pour passer des stations aux régions. A priori, elle comporte deux attributs :

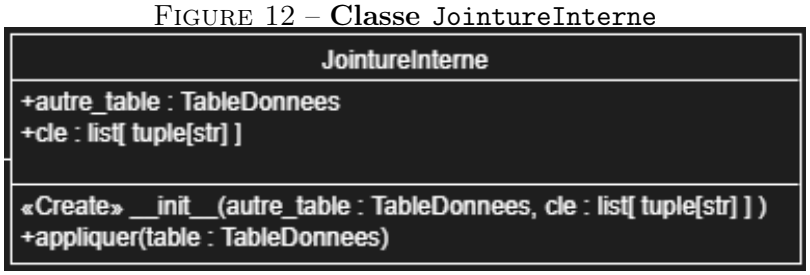
- `table_correspondance` : un objet de type `TableDonnees` qui fait le lien enre les deux niveaux géographiques
- `echelon` : une chaîne de caractères qui précise l'échelon géographique souhaité en sortie.

FIGURE 11 – **Classe** `AgregationSpatiale`



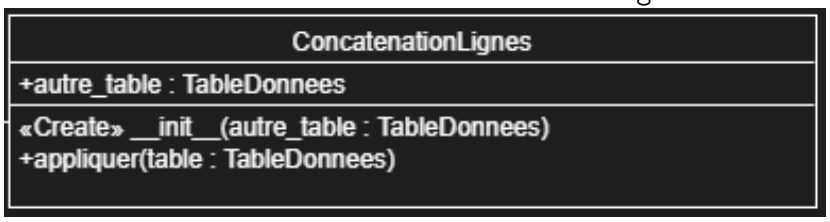
— Concernant la classe `JointureInterne`, la clé de jointure est sous la forme d'une liste de tuples (exemple : `[(table1_id, table2_id),(table1_date, table2_date)]`). La jointure se fait en mode « INNER JOIN », c'est à dire que si la clé n'existe pas dans l'une des tables, la ligne n'apparaît pas dans la sortie. Cette clé de jointure est l'attribut `cle` de cette classe, et le deuxième attribut est `autre_table`(l'autre instance de `TableDonnees` avec laquelle se fait la jointure).

FIGURE 12 – **Classe** `JointureInterne`



— La classe `ConcatenationLigne` vérifie en amont de l'exécution que les tables en entrée ont les mêmes variables. Elle possède un unique attribut `autre_table`, un objet de type `TableDonnees`. Sa méthode `appliquer` permet de concaténer les lignes de `table` (objet de type `TableDonnees` pris en paramètre) et `autre_table`.

FIGURE 13 – **Classe** `ConcatenationLignes`



— La classe Export possède une unique méthode statique `appliquer` qui prend en paramètre `table`, objet de type `TableDonnees`, et un format sous forme de texte ici fixé à « csv ».

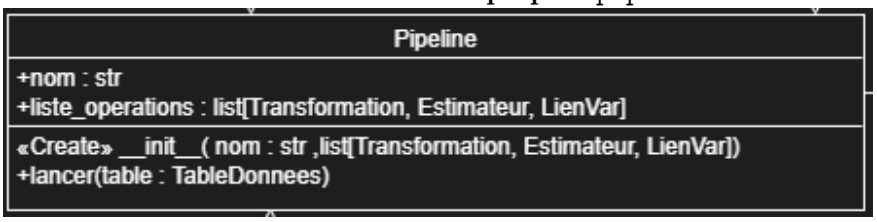
FIGURE 14 – **Classe** `Export`

L'application enregistre la table dans un dossier spécifiquement réservé aux exports, et nomme le fichier avec le nom de la table, la date et l'heure.

3.6 Les classes du paquet `pipeline`

Le paquet pipeline ne contient que la classe Pipeline décrite en Figure 15 ci-dessous. Cette classe est centrale dans notre application puisqu'elle permet de faire appel à une liste d'opérations en vue de réaliser un traitement de données complexe.

FIGURE 15 – **Classe du paquet `pipeline`**



Un objet de la classe Pipeline est défini par les attributs suivants :

— `nom` : nom du pipeline

— `liste_opérations` : liste contenant des objets des classes Transformation, Estimateur ou LienVar.

En plus du constructeur, le pipeline intègre la méthode `lancer` qui parcourt dans l'ordre la liste des opérations et les applique à l'objet `table` pris en paramètre. Initialement nous n'avions prévu d'insérer dans le pipeline que des objets de type `Transformation`. Nous avions d'abord également envisagé un booléen en sortie de pipeline pour gérer l'export éventuel de la table finale : cette solution nous est vite apparue limitante et finalement peu pertinente donc nous avons créé la classe `Export`. Nous avons par la suite hésité à regrouper les trois types d'opérations (`Transformation`, `Estimateur` et `LienVar`) dans une catégorie mère « `Operation` » mais finalement nous n'avons pas jugé cela vraiment nécessaire : nous avons plutôt uniformisé nos classes avec une méthode `appliquer`. L'autre choix aurait été possible et peut-être plus généraliste, mais nous avons décidé de ne pas le prioriser.

3.7 Utilisation de l'application

L'utilisation des fonctionnalités que nous avons créées implique une connaissance du langage Python. L'utilisateur a la possibilité d'appeler les classes implémentées. Par exemple, le processus le plus simple consiste à :

— charger une table depuis un fichier .csv (classe `donneesCsv`)

— créer un pipeline contenant des `Transformations` et en particulier un `Export`

— lancer le pipeline et récupérer le fichier .csv généré

Pour aller plus loin, il serait possible de créer un menu interactif ou une interface graphique pour faciliter l'expérience utilisateur. Néanmoins cela dépasse le cadre de notre projet.

Conclusion

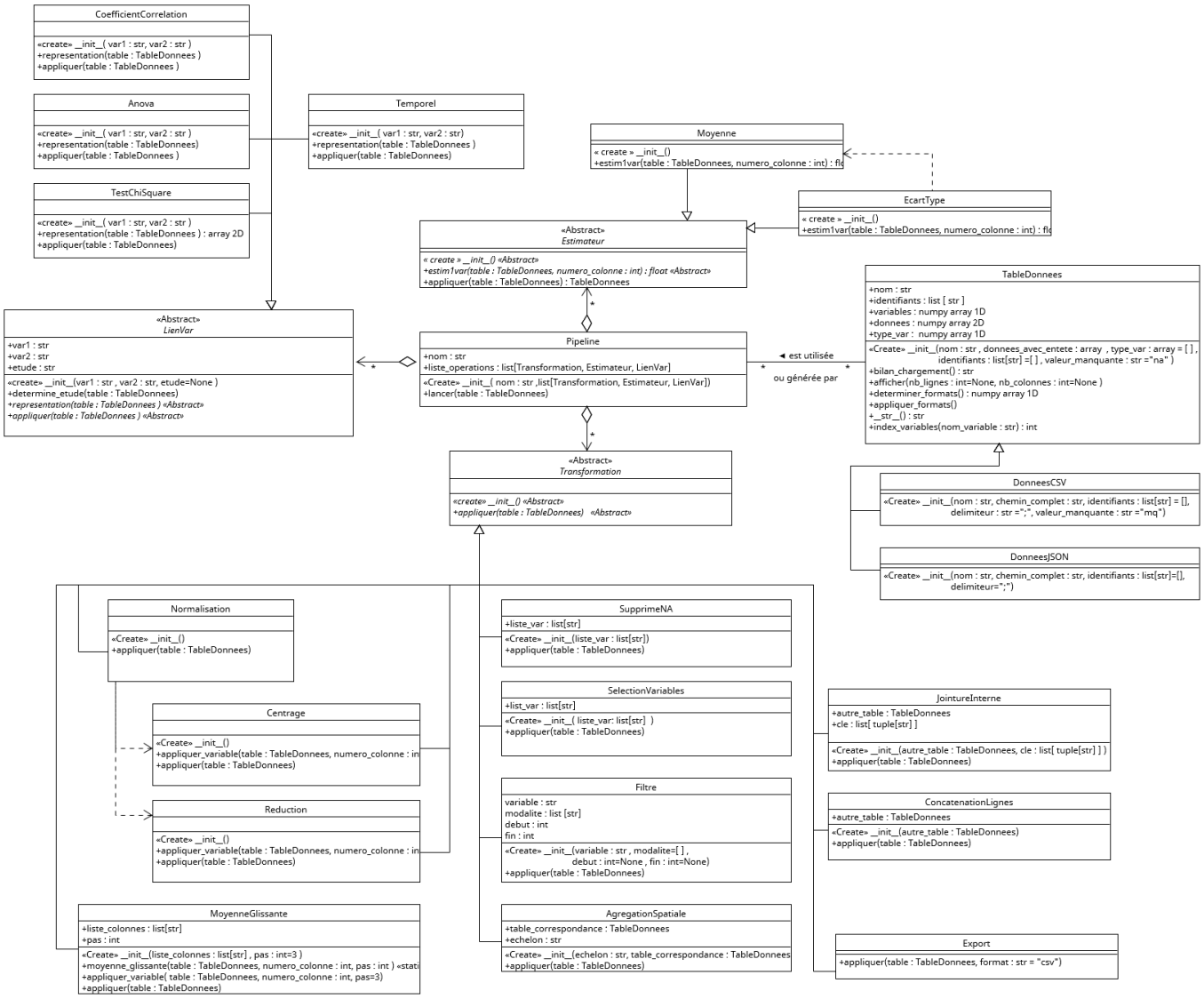
La réalisation de ce projet de traitement de données nous a permis à la fois de réinvestir nos compétences développées précédemment lors du projet statistique et d'autre part de mettre en application les apprentissages du cours de programmation orienté objet (POO). En effet, le traitement de données dans le contexte d'études statistiques nous a fait réfléchir à l'implémentation de fonctions parfois déjà utilisées avec le langage R. Essayer de reconstruire l'algorithme par nous-même et le coder dans un langage de programmation (ici Python) était une expérience formatrice. Cela nous a également poussé à organiser efficacement les relations entre les classes construites à l'aide d'un diagramme UML qui a évolué au cours de la réflexion et en interaction avec la phase de codage qui a suivie. L'UML a été un guide de route pour préparer la phase de programmation avec cohérence (par exemple, sans attributs ni méthodes redondantes, grâce à la notion d'héritage). De plus, l'interaction en équipe nous a facilité un échange horizontal de compétences afin de progresser en partageant les atouts de chacun (utilisation de Github, logique mathématique, maitrise du langage Python, etc...). Par chance, notre groupe était assez diversifié car il aurait pu nous manquer une formation à l'utilisation de Git, la maitrise de `Array` ou plus généralement du package `numpy`, ou encore une introduction à l'utilisation d'un Pipeline.

Malgré les délais assez courts, notre phase de programmation a bien avancé, et nous pensons que certaines classes encore incomplètes pourront être finalisées d'ici le rendu du code. Il nous reste à pouvoir traiter d'avantage d'exemples d'utilisation pour en déduire avec plus de recul de réelles interprétations statistiques qui soient exploitables. Les classes qui sont déjà fonctionnelles sont : celles des packages `table`, `pipeline`, `estimateur` ; puis la plupart de celles de `transformation` et quelques-unes de `lienvar`. Dans le package `lienvar`, les classes en construction `Anova` et `TestChiSquare` sont plutôt là à titre de prolongement possible pour généraliser l'application à d'autres tables de données que celles du projet, car à priori ces tables contiennent peu de données qualitatives (en opposition aux nombreuses données quantitatives). L'export en format `.json` pourrait également faire l'objet d'un prolongement.

Les limites rencontrées sont pour l'instant au niveau des classes `AgregationSpatiale` et `MoyenneGlissante` qui nous demandent de prendre encore du recul sur la gestion du format des dates et sur la nature de la table qui fait l'association entre les stations et les régions (que nous n'avons pas encore chargée).

Annexe 1 - Diagramme de classe UML

FIGURE 16 – **Diagramme de classe**



Annexe 2 - Utilisation de Git sur notre projet

Git est un logiciel de gestion de versions. Il facilite, pour chaque développeur, la synchronisation entre le dépôt local (le code qui est sur notre ordinateur) et le dépôt distant sur GitHub (le code commun). Il est ainsi beaucoup plus simple de travailler sur un même projet et d'éviter les décalages de versions. Dans le cadre de notre projet, nous utiliserons les fonctionnalités basiques de Git. Nous avons mis en place le protocole ci-dessous.

Notre dépôt est organisé avec les dossiers suivants :

— src : contient les fichiers python

— doc : fichiers de documentation (cahier des charges, diagrammes UML...)

— donnees : contient des fichiers de données, des fichiers tests et un dossier `exports` où seront générés tous les exports.

Avant de commencer à coder

— Dans Visual Studio Code, j'ouvre un terminal Git Bash

— Menu View > Terminal (ou CTRL+ù)

— je clique sur la petite flèche vers le bas à coté du +, puis sur Git Bash

— Normalement je suis placé directement dans le bon dossier et dans la console s'affiche :

```
idxxxx yyyyyy /p/projet-info-sources/Projet-info (main)
git pull      # permet de mettre à jour le dépôt local avec le dépôt distant
```

Je code

— je crée/modifie/supprime des fichiers python (ou autre)

— je teste que ça fonctionne bien

— une fois que j'ai un morceau de code qui fonctionne, je crée un **commit** (point de sauvegarde)

— il est très important de faire régulièrement un commit dès que quelque chose fonctionne bien. Cela évitera de perdre beaucoup de temps si ensuite par une action malheureuse, le code ne fonctionnait plus du tout. Dans ce cas, un simple retour arrière au dernier commit et l'on peut repartir sur de bonnes bases

```
git add .    # permet d'ajouter tous les nouveaux fichiers créés
git status   # pour voir les changements en cours
git commit -am «˜message explicite˜»     # Pour créer un commit
```

Si par la suite je fais une erreur, il est facile de revenir en arrière (au dernier commit) :

```
git reset --hard
```

Attention cette commande supprime toutes les modifications effectuées depuis le dernier commit. Par sécurité, je crée une copie du dossier **Projet-info** avant de lancer la commande.

Je partage mon travail

- Si personne n'a poussé du code entre temps, tout va bien, je vais pouvoir faire un **push**
- Par contre si le dépôt distant a été modifié, je dois synchroniser mon dépôt local avec les mises à jour effectuées sur le dépôt distant par d'autres membres de l'équipe
- Si vous n'avez pas touché aux mêmes fichiers, Git va effectuer la fusion tout seul lors du **git pull**
- Si vous avez touché au même fichier, ça se complique un peu. Git va dire qu'il n'a pas réussi de fusion automatique (*CONFLICT (content) : Merge conflict - Automatic merge failed*). Il faut ouvrir les fichiers en conflit et ceci apparaît :

```
<<<<<<< HEAD
« Les modifications que j'ai faites »
=======
« Les modifications faites par un autre membre de l'équipe »
>>>>>>>
```

- je modifie le fichier pour choisir quelle modification je garde
- je teste et je vérifie que tout est ok
- je recrée un commit `git commit -am « merge manuel »`
- et enfin, je peux faire `git push`

```
git pull           # pour récupérer les éventuelles modifications du dépôt distant
git status         # pour voir s'il n'y a pas de conflits
git push           # pousser son code vers le dépôt distant
```

Commandes Git : ce qu'il faut retenir

```
git status               # Voir ce qui est en cours
git pull                 # Copier dépôt distant vers dépôt local
git push                 # Copier dépôt local vers dépôt distant
git add .                # Avant un commit pour que git identifie les nouveaux fichiers
git commit -am « message » # Créer un point de sauvegarde
git diff                 # Avant de faire un commit, voir les différences
```