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

Projet Informatique 2A

Dossier d'analyse

Etudiants :
Abdoul-Aziz Toure
Nathan Sandja
Laurène Villacampa
Isaac Ganiyu
Adrien Cortada

Professeur :
Rémi Pépin
Encadrant :
Antoine Brunetti

2022 - 2023

Sommaire

Introduction

Ce projet informatique s'inscrit dans la continuité du projet de première année, basé sur le cours de programmation orientée objet. La spécificité de ce nouveau travail est d'articuler plusieurs des apprentissages de première année : non seulement le cours de programmation orientée objet avec la modélisation UML et l'implémentation sous Python, mais également l'utilisation d'une base de données distante et son interrogation via des requêtes SQL. De nouvelles compétences sont ainsi visées concernant la réalisation du lien entre un langage de programmation et l'appel à un service distant de gestion de base de données.

Dans la continuité du projet de première année, ce travail est réalisé en groupe. Dans une perspective professionnalisante, le groupe de 3 de première année est remplacé par une équipe de 5 personnes, avec un chef de projet désigné au sein du groupe. Cette nouvelle organisation conduit à renforcer l'usage d'outils collaboratifs de travail et de coordination : des outils de planification des tâches, des outils de communication, et des outils de gestion de versions en vue d'une programmation collective efficace.

La première partie de ce dossier est centrée sur l'analyse du sujet proposé par M. Antoine Brunetti et l'identification des outils à mettre en oeuvre pour mener à bien ce travail. Par la suite, une modélisation UML est réalisée à l'aide de diagrammes adaptés à la compréhension par tous des implémentations qui seront réalisés dans la suite du projet. La conception générale de l'application est ainsi finalisée en vue d'amorcer la phase de réalisation de l'application.

1 Phase d'étude et cahier des charges

1.1 Description du sujet et analyse des besoins

Le présent projet vise à élaborer une API (Application Programming Interface) pour proposer un webservice de génération de données. Il s'agit de permettre à un utilisateur de générer des jeux de données respectant certaines contraintes. Ce besoin peut être rencontré par des testeurs de programmes, ou des statisticiens qui cherchent à modéliser une situation.

Les tables de données à générer doivent répondre au besoin utilisateur et donc respecter des contraintes sur la nature et la taille des données. L'application doit donc permettre de définir :

- **Des types des données** :
  - **Type simple** : le nom du champ de la variable, et les modalités possibles pour cette variable. L'utilisateur doit en outre pouvoir attribuer un poids à chaque modalité.
    Exemple : champ SEXE avec pour modalités "M", "F", "A", généré avec 48% de "F", 45% de "H" et 7% de "A".
  - **Type composé** : le nom du champ de la variable, composé lui-même de plusieurs variables de type simple.
    Exemple : champ VOITURE composé de champs simples tels que NOMBRE_ROUES ou COULEUR préalablement définis.
- **Des schémas de données** :
  À partir des types de données définis, l'utilisateur doit pouvoir générer les "lignes" du jeu de données en précisant les types de données attendus, et éventuellement un taux de remplissage pour chaque champ. Ce dernier point suppose la génération de données manquantes pour compléter le jeu de données générées
- **La taille du jeu de données** :
  L'utilisateur doit pouvoir préciser le nombre d'observations attendues (le nombre de lignes du jeu de données).

L'utilisateur doit pouvoir définir ces contraintes soit directement par l'API soit par l'intermédiaire d'un fichier traité par l'API, au format JSON. L'application doit donc proposer une procédure de récupération des règles liées aux contraintes de la table à générer tenant compte de ces deux possibilités.

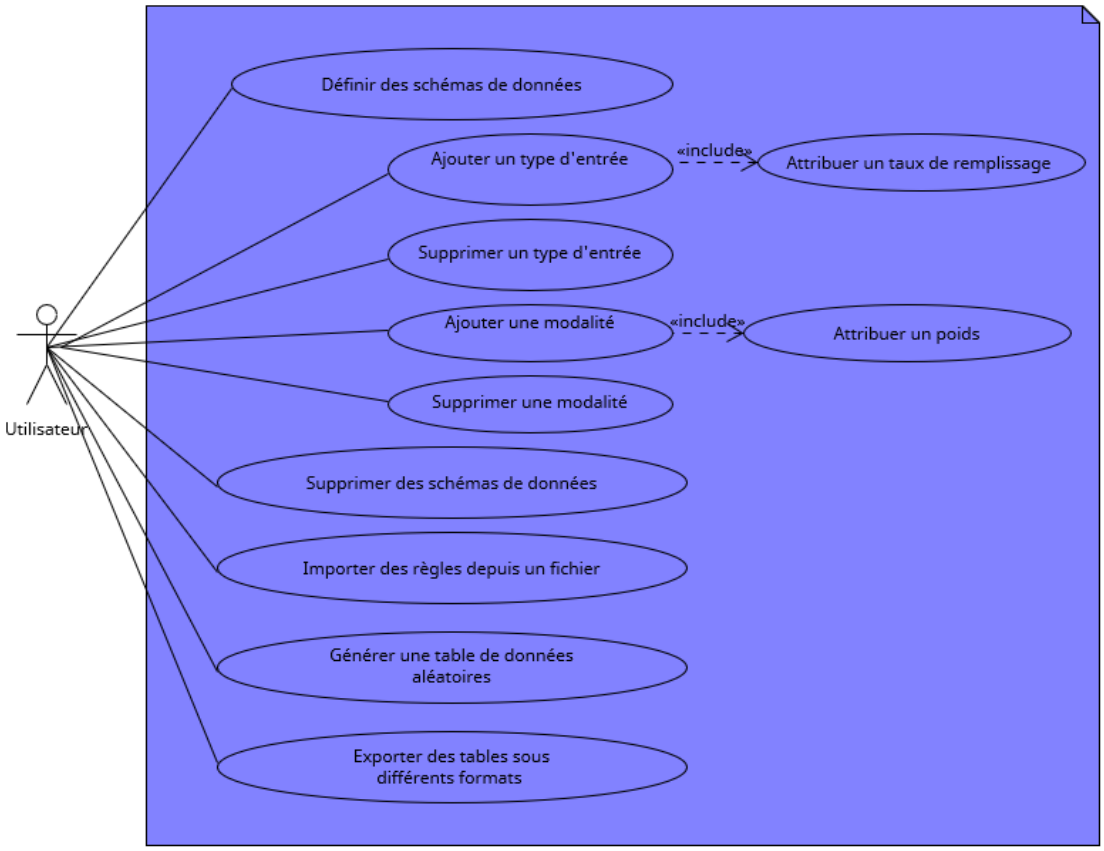
Une fois la génération d'un jeu de données effectuée, la table obtenue doit pouvoir être stockée en vue de sa récupération par l'utilisateur et de son exploitation future. Ce stockage s'effectue via une base de données. L'utilisateur doit donc pouvoir d'une part générer une base de données et d'autre part interroger cette base de données. L'application doit donc permettre la connexion à une base de données distante.

En outre, l'utilisateur doit pouvoir être capable de récupérer en externe les jeux de données générées : l'API doit donc prévoir une fonction d'export, éventuellement dans plusieurs formats pour faciliter l'exploitation future des tables (formats JSON, CSV...).

1.2 Diagramme de cas d'utilisation

L'analyse précédente est articulée dans le diagramme de cas d'utilisation ci-dessous (Figure 1).

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



Cette première analyse conduit à considérer le problème dans une perspective de programmation orientée objet. Les données à générer sont ainsi vues comme des instanciations de classes d'objets de plusieurs natures : des types d'entrées et des modalités qui leur sont associées. Ces types peuvent être simples ou composés pour répondre aux cas évoqués par le sujet.Pour chaque modalité on définit une probabilité d'apparition (son "poids") et pour chaque type on définit un taux de remplissage. Ces deux éléments sont donc considérés comme des attributs pour chacune des classes. Les données ainsi définies sont assemblées dans des schémas de jeux de données. La gestion des exports se fait dans un second temps, à partir de ces schémas.

L'application doit également permettre la gestion des modalités et des types (ajout, suppression, modification) et des méthodes spécifiques seront donc définis. Un diagramme de classes est réalisé dans la phase de conception pour préciser ces éléments.

1.3 Planification des tâches

Le projet est réalisé en trois phases :

— Une phase d'analyse et d'interprétation du sujet. Les contraintes techniques sont listées dans le cahier des charges, les utilisations sont représentées dans un diagramme de cas d'utilisation.

— Une phase de conception de l'application pour décrire plus précisément ses fonctionnalités et son organisation. La modélisation s'effectue à l'aide de diagrammes UML (diagramme de classes, diagramme de package, diagramme d'activité, diagramme de base de données).
ces deux premières phases aboutissent à la production du dossier d'analyse.

— Une phase de réalisation de l'application avec la mise en place d'une base de données et le développement en python3. Cette phase comporte la réalisation de tests et l'élaboration d'un scénario d'utilisation en vue de la soutenance.

Afin de coordonner ce travail, un tableau Kanban est réalisé via le webservice Asana. Quatre catégories de tâches sont définies :

— Échéances : mémo rappelant les dates de TP et de suivi de projet, de rendus intermédiaire et final, des jours d'immersion projet info et de soutenance. Les tâches de cette catégorie sont définies comme des jalons du projet.

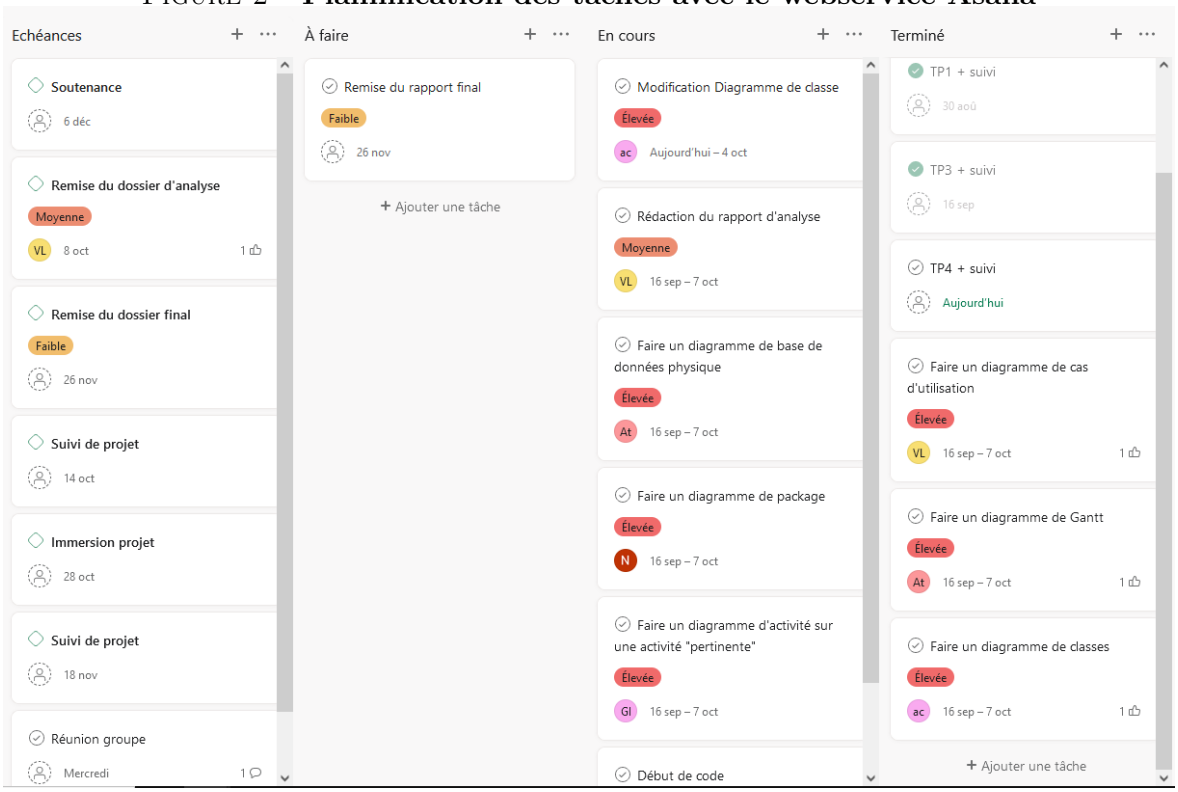
— À faire : tâches définies au fur et à mesure de l'avancement du projet et des nouvelles idées.

— En cours : tâches en cours de réalisation.

— Terminé : les tâches déjà réalisées pour garder une trace de ce qui a été fait.

Chaque tâche (en cours ou terminée) est attribuée à un membre du groupe, ce qui permet à chacun de suivre l'avancement du projet et de savoir à qui s'adresser pour chaque élément du projet. Lui est affecté également une date butoir et un niveau de priorité. Un code couleur est utilisé pour la priorisation des tâches (cf Figure 2). Chaque membre du groupe peut ajouter des tâches, les supprimer et les éditer.
La répartition des tâches en trois catégories "à faire", "en cours" et "terminé" a semblé pertinente pour ce projet de par sa clarté d'utilisation et son aspect professionnalisant. Toutefois, une vue chronologique sous forme de diagramme de Gantt est également disponible via le webservice Asana.

FIGURE 2 – **Plannification des tâches avec le webservice Asana**



1.4 Outils de développement

Le développement de l'application se fait dans le langage Python, via l'environnement VScode. Ce langage est en effet particulièrement adapté à la programmation orientée objet. Plusieurs extensions sont utilisées pour la réalisation du dossier d'analyse :

- `UMLet` : outil de génération de diagrammes de classe, de cas d'utilisation et de diagramme d'activité.
- `Drawio` : outil de génération de diagramme de package

Le webservice `lucidchart` est également utilisé pour réaliser le diagramme de base de données

La base de données `PostgreSQL` est utilisée comme système de gestion de base de données (SGBD). Des extensions python sont exploitées dans la perspective de faire le lien entre python et le SGBD :

- `psycopg2` : pour la connexion à la base de données
- `requests` : pour la réalisation de requêtes

D'autres libraires seront utilisées pour le développement et la réalisation des tests : `abc` et `unittest` à minima.

Afin de rendre plus efficace le travail de groupe, un système de versionnage de code source est utilisé. Git est privilégié, via l'hébergeur Github. Les dates clés du projet ainsi que les commentaires du tuteur sont stockés dans le fichier `ReadMe`.
Le webservice `fastAPI` sera utilisé ultérieurement pour la création du webservice associé à l'application.

2 Conception générale de l'application

2.1 Diagramme de classes

Pour coder de manière plus efficace ainsi que pour se donner une direction à suivre et formaliser nos idées nous avons donc décidé de mettre l'architecture de notre futur webservice sous la forme d'un diagramme de classes (cf Figure 10 en Annexe).

2.1.1 Règles de la génération de jeu de donnée

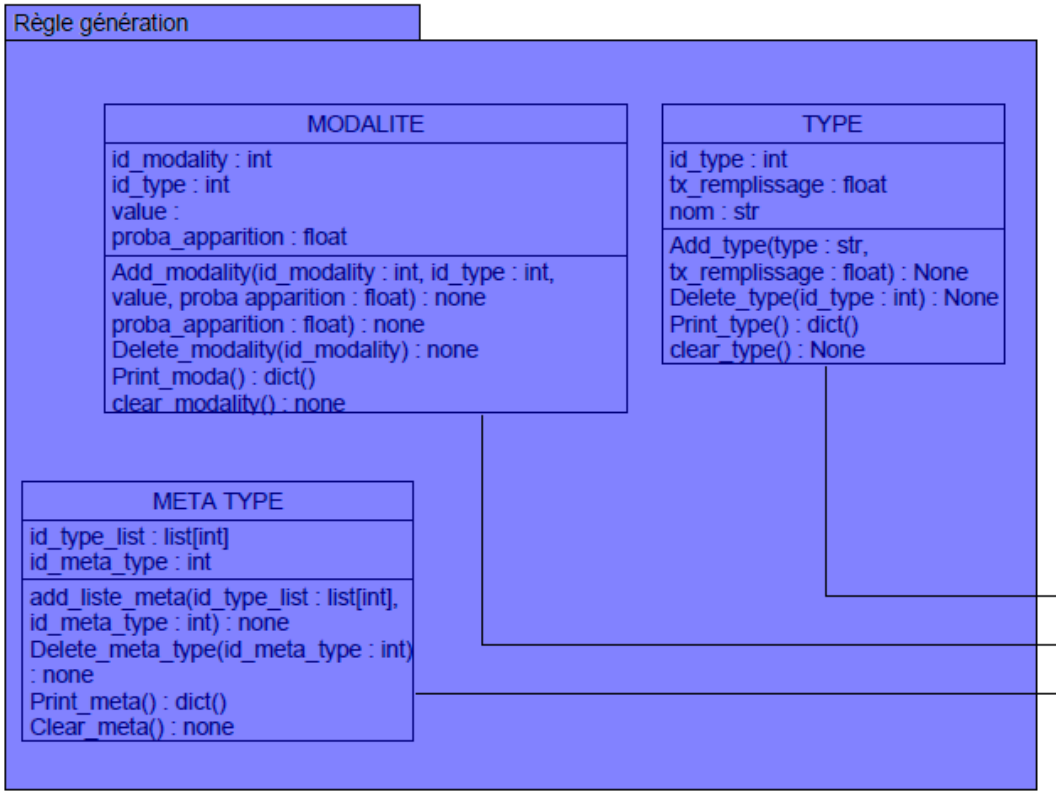


FIGURE 3 – Classes associées à la création des règles permettant la génération du jeu de donnée

- **La classe TYPE** : permet à l'utilisateur de créer un nouveau type (une variable pouvant prendre différentes modalités ou différentes valeurs si elle suit une certaine loi)
  - **id_type** : un entier qui permet d'identifier le type au sein du dictionnaire des types
  - **tx_remplissage** : un réel compris entre 1 et 100 qui indique la probabilité (en pourcentage) qu'une valeur ne soit pas manquante
  - **nom** : une suite de caractères qui permet à l'utilisateur de nommer le type en question afin de mieux pouvoir se repérer dans la table
  - **Add_type** : une méthode qui prend en argument un id type, un taux de remplissage et un nom et qui ajoute un nouveau type au dictionnaire des types
  - **Delete_type** : une méthode qui prend en argument un ID type et qui enlève le type correspondant du dictionnaire de types

- **Print_type** : une méthode qui affiche le dictionnaire contenant l'ensemble des types entrés par l'utilisateur
- **Clear_type** : une méthode qui efface tous les types du dictionnaire, le dictionnaire est alors vide

- **La classe MODALITE** : permet à l'utilisateur de créer de nouvelles modalités correspondant à un certain type
  - **id_modality** : un entier qui permet d'identifier le type au sein du dictionnaire des modalités
  - **id_type** : un entier qui permet de savoir à quel type est relié la modalité en question
  - **value** : une suite de caractères ou un réel contenant la valeur de la modalité en question
  - **proba_apparition** : un réel permettant de savoir quelle est la probabilité d'apparition de chacune des modalités
  - **Add_modality** : une méthode qui prend en argument un ID modality, un ID type, une valeur et une proba d'apparition et qui ajoute une nouvelle modalité au dictionnaire des modalités
  - **Delete_modality** : une méthode qui prend en argument un ID modality et qui enlève la modalité correspondante du dictionnaire des modalités
  - **Print_moda** : une méthode qui affiche le dictionnaire contenant l'ensemble des modalités entrées par l'utilisateur
  - **Clear_modality** : une méthode qui efface toutes les modalités du dictionnaire, le dictionnaire est alors vide

- **La classe META TYPE** : permet à l'utilisateur de créer un méta type, c'est à dire un nouvel objet contenant une liste de types
  - **id_type_list** : une liste d'entiers qui permet d'identifier une liste de types
  - **id_meta_type** : un entier qui permet de savoir à quel type est relié la modalité en question
  - **add_list_meta** : une méthode qui prend en argument id_type_list et id_méta_type et qui ajoute un méta type ainsi que la liste des types qui le composent au dictionnaire des méta types
  - **delete_meta_type** : une méthode qui prend en argument l'id_méta_type et qui retire le méta type correspondant du dictionnaire
  - **Print_meta** : une méthode qui affiche le dictionnaire des méta types
  - **Clear_meta** : une méthode qui efface tous les méta types du dictionnaire, le dictionnaire est alors vide

2.1.2 Importation de règle de génération de donnée

Le but de notre webservice étant de générer des données il y a besoin de règles, que l'utilisateur peut créer (cf classe 2.1.1) ou décider d'importer via des fichiers au format JSON.

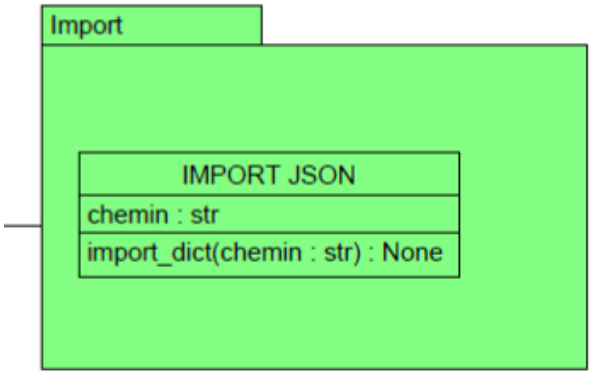


FIGURE 4 – Importation des règles de génération

- **La classe IMPORT JSON** : permet d'importer des fichiers au format JSON contenant les règles de génération de données
  - **chemin** : une chaîne de caractères qui permet d'indiquer où sont situés les fichiers des règles de génération
  - **import_dict** : une méthode qui prend en argument le chemin afin d'importer les dictionnaires JSON contenant les règles de génération

2.1.3 Génération du jeu de donnée

Maintenant que les règles de génération sont fixées il ne reste plus qu'à générer les données

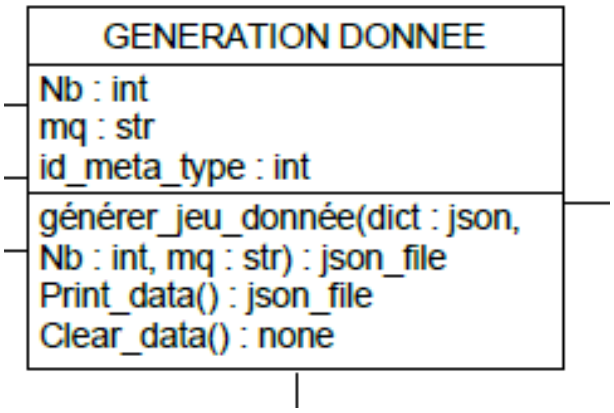


FIGURE 5 – Génération des données

- **La classe GENERATION DONNEE** : permet à partir des règles de génération de générer un nombre de données sélectionné par l'utilisateur pour n'importe quel méta type et de les stocker dans un dictionnaire

— **Nb** : un entier correspondant au nombre de données que l'on souhaite générer

— **mq** : une chaîne de caractères correspondant à la façon dontles données manquantes doivent apparaître dans la table

— **id_metatype** : un entier permettant d'indiquer à quel méta type correspondent les données que l'on souhaite générer

— **générer_jeu_donnée** : une méthode qui prend en argument Nb, mq et Id méta type et qui stocke dans un dictionnaire le nombre Nb de données générées correspondant au méta type indiqué

— **Print_data** : une méthode qui affiche l'ensemble des données générées

— **Clear data** : une méthode qui efface l'ensemble des données générées, le dictionnaire des données est alors vide

2.1.4 Export

Une fois le jeu de données généré il est prévu que le jeu de données soit sauvegardé en ligne mais également que l'utilisateur puisse sauvegarder le jeu de données sur son ordinateur avec le format de son choix.

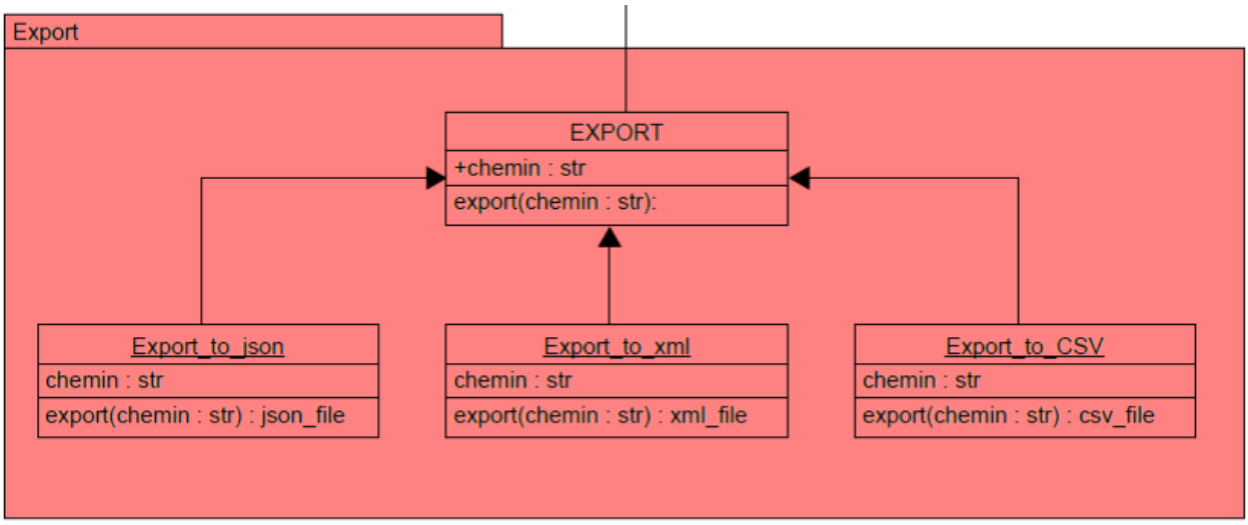


FIGURE 6 – Export des données

De la classe EXPORT héritent 3 classes filles Export to json, Export to xml, Export to CSV qui fonctionnent de manière similaire :

- **Les Classes Export to json, xml, csv** : permettent à partir des règles d'un chemin donné par l'utilisateur de stocker localement les tables générées
  — **chemin** : une chaîne de caractères qui permet d'avoir l'emplacement où la table doit être stockée
  — **export** : un module qui prend en argument le chemin indiquant où la table doit être stockée et qui la sauvegarde au format souhaité

2.2 Diagramme d'activité : un exemple

Dans l'optique de faire fonctionner efficacement l'activité principale de notre API, il nous a fallu mettre au point un diagramme d'activité autour de cette activité. Ce diagramme détaille les différentes actions élémentaires exécutées lors de l'appel de notre activité, qui représente ici un cas d'utilisation : la génération de la table de données aléatoires.Ce diagramme permet surtout de voir comment est-ce que ces différentes actions élémentaires sont agencées entre elles afin d'aboutir au résultat final.

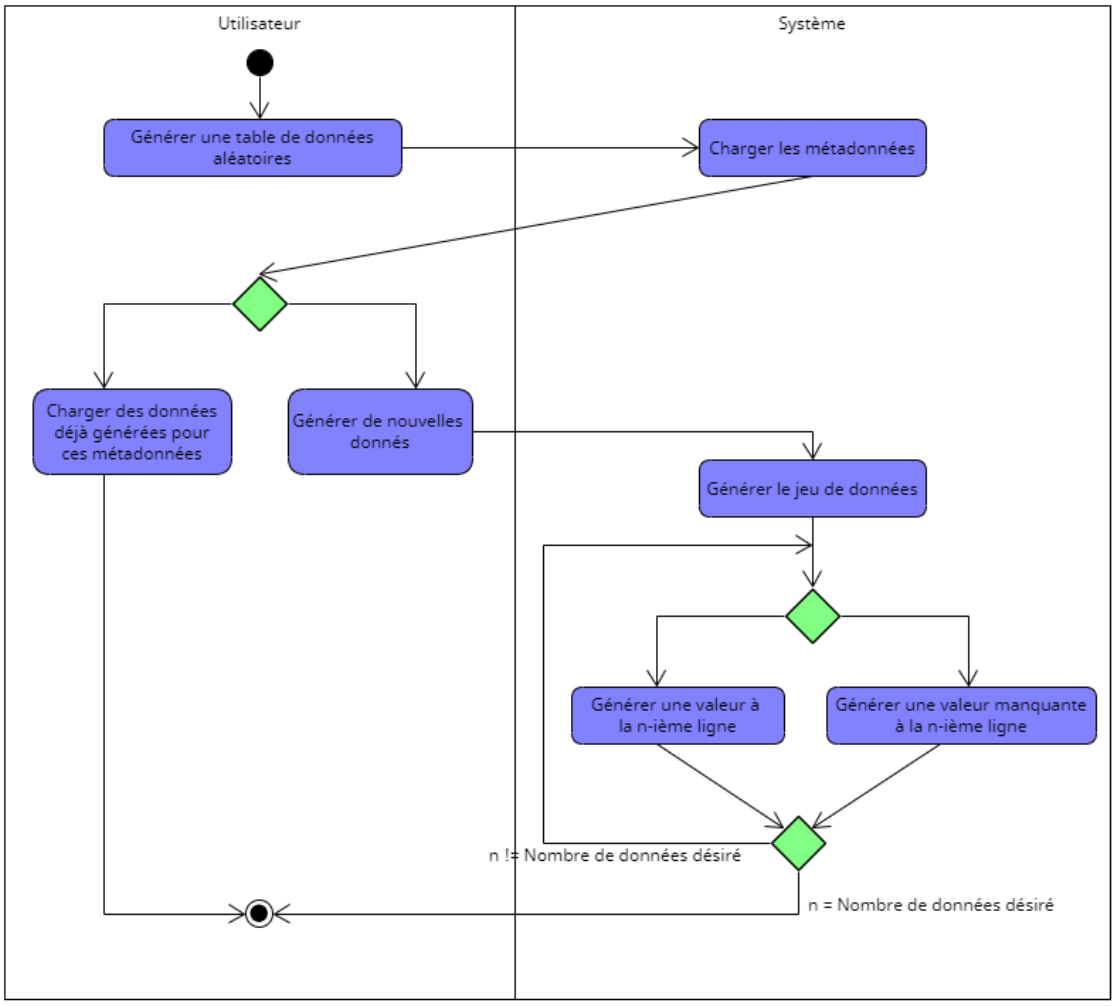


FIGURE 7 – Diagramme de l'activité de génération de données aléatoires avec des métadonnées définies par l'utilisateur

Pour ce cas d'utilisation, après avoir chargé les métadonnées renseignées par l'utilisateur, l'API lui donne l'opportunité de choisir entre charger des données qui auraient déjà été générées pour ces métadonnées et générer de nouvelles données. La première option met fin à l'algorithme dans la mesure où il suffirait juste de charger des données stockées. La seconde fait appel au système. Ce dernier génère dans un premier temps le jeu de données adéquat. Puis, en fonction du taux de remplissage, génère aléatoirement des données ou des valeurs manquantes. Une fois que le nombre de données désiré est atteint, l'algorithme prend fin.

2.3 Diagramme de package

Pour illustrer les différentes couches de notre application, nous avons dû modéliser ceci avec un diagramme de paquetage afin de regrouper les classes fortement couplées et identifier les différents liens entre les paquetages.

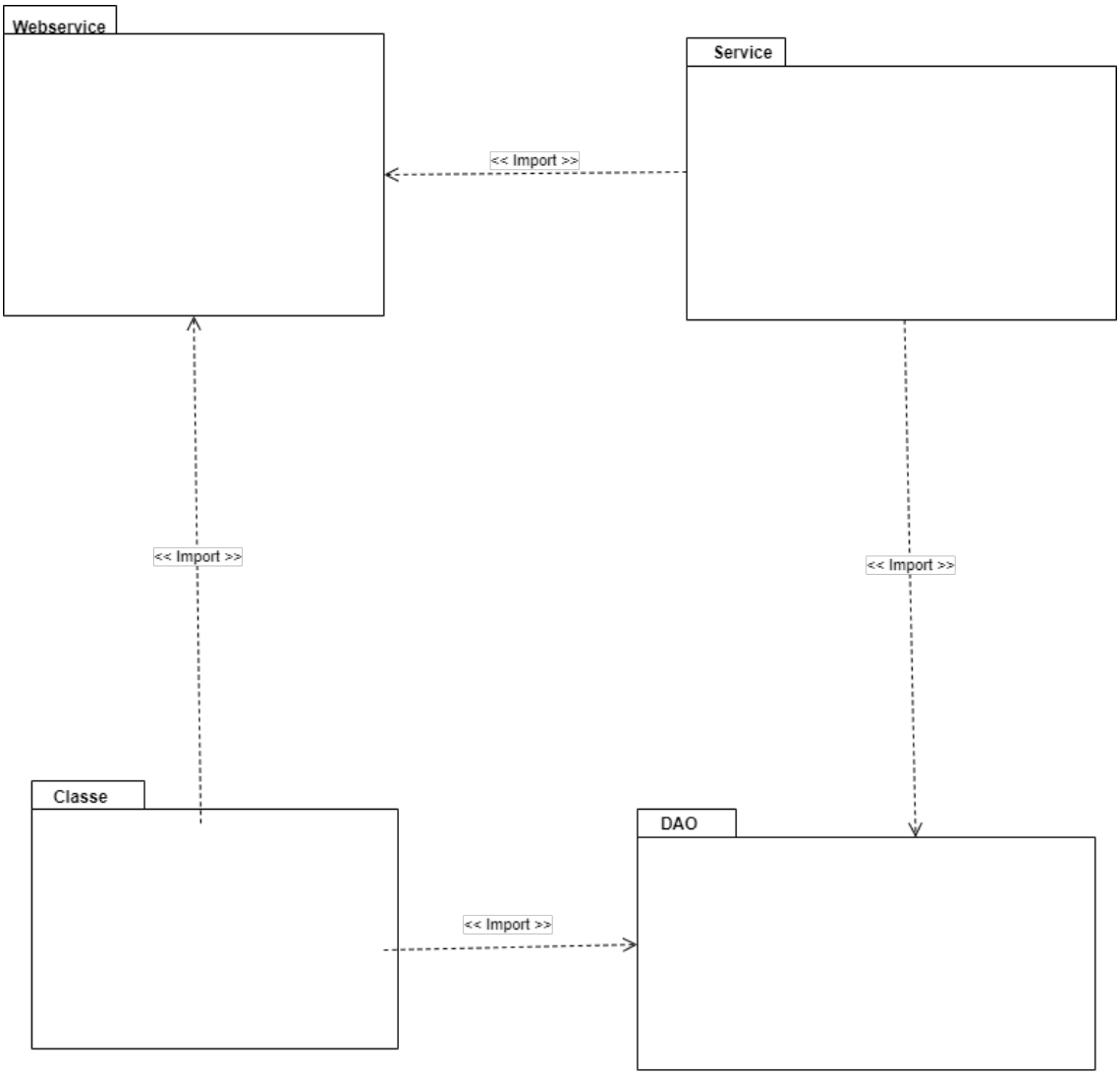


FIGURE 8 – Diagramme de paquetage

Nous avons décidé de regrouper les classes en quatre paquetages :

— Le paquetage **Classe** qui contient les classes métiers

— Le paquetage **DAO** qui contient toutes les classes liées a la DAO

— Le paquetage **Webservice**

— Le paquetage **Service**

Les paquetages Classe et Service importent les paquetages webservice et DAO afin d'accèder aux bases de données créées par les utilisateurs et leur proposer différentes fonctionnalités.

2.4 Modèle physique de base de données

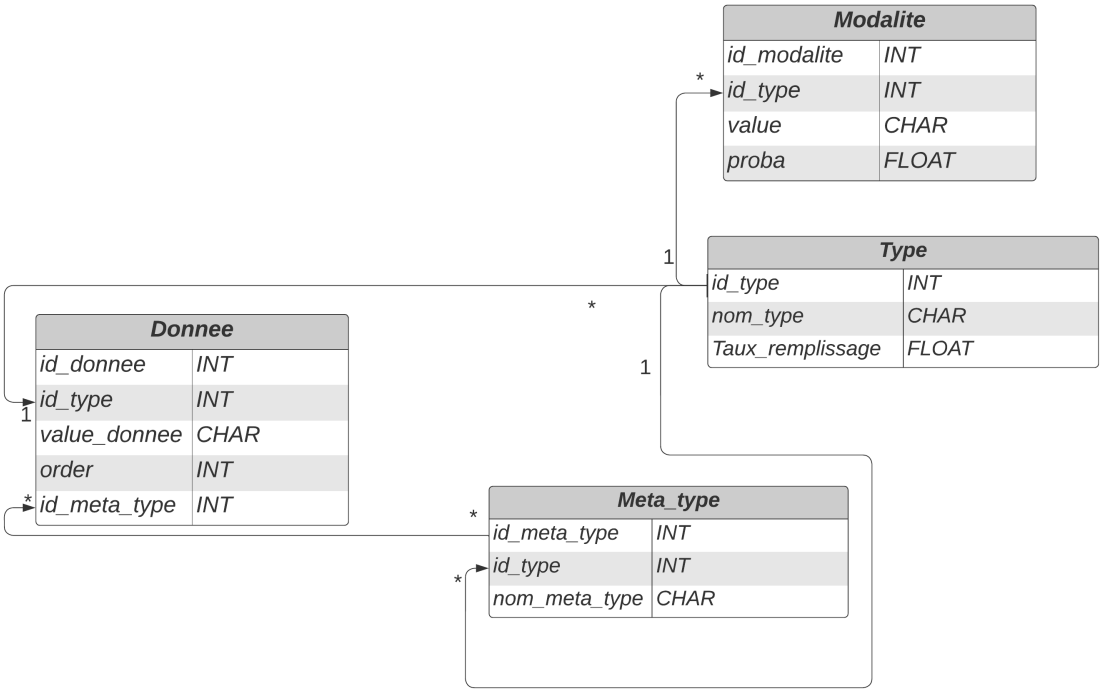


FIGURE 9 – Diagramme de base de données physique

Le diagramme de notre base de données est centré autour de la table "type" : cela rend possible la connexion entre chacun des tableaux. La table "type" comprend les différents types d'objets qui composent notre base de données et chaque type est accompagné d'un taux de remplissage.
Exemple : Type(id_type, nom_type , Taux_remplissage)

Grâce à la table "Type", on crée deux nouvelles tables "Meta_type" qui permettent de regrouper les informations identifiant un type bien défini. Par exemple, un type mécanicien composé de plusieurs métas-type qui sont le nom, prénom, salaire, âge, etc.
Exemple : Meta_type(id_meta_type, #*id_ type*, nom_meta_type)

Puis on crée la table "Modalite" qui regroupe les valeurs prises par les différents types et aussi leur probabilité d'apparition.
Exemple : Modalite(id_modalite, #*id_ type*, value, proba)

En créant toutes ces tables, nous pouvons maintenant créer la table "Donnee" qui est la jonction entre la table "Modalite" et la table "Meta_type". On retrouvera donc les valeurs prises par les modalités de chaque type de données et l'ordre des types pour faciliter la lecture des données pour le méta type.
Exemple : Donnee(id_donnee, *#id_ type, #id_ meta_ type*, value_donnee, order)

Ces éléments de modélisation UML vont maintenant servir au développement de l'application.

Annexe

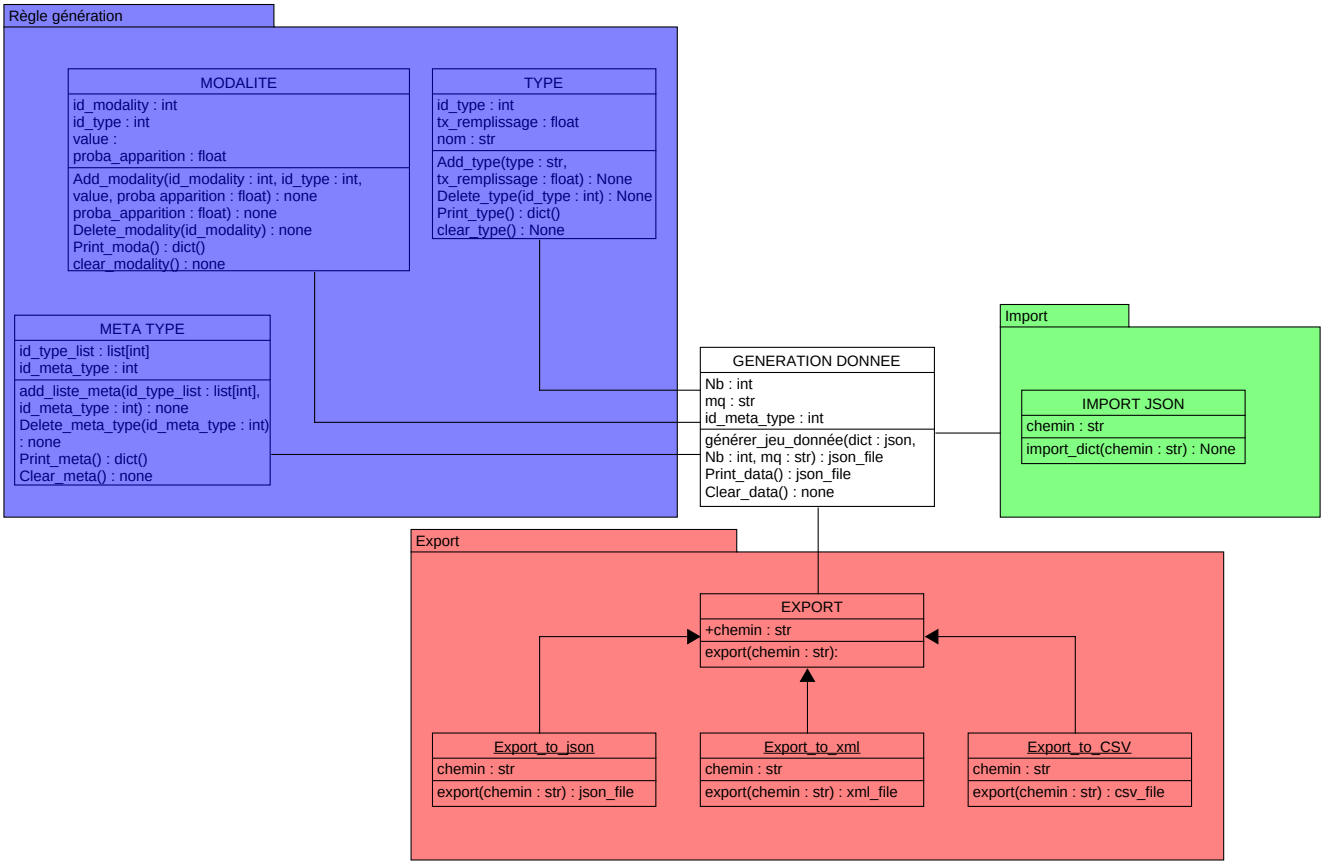


FIGURE 10 – Diagramme de classe