

ECOLE NATIONALE DE LA STATISTIQUE  
ET DE L'ANALYSE DE L'INFORMATION



---

Projet Informatique 2A :  
Webservice de génération de données (sujet n°2)

---

*Etudiants :*

Abdoul-Aziz TOURE

Nathan SANDJA

Laurène VILLACAMPA

Isaac GANIYU

Adrien CORTADA

*Professeur :*

Rémi PÉPIN

*Encadrant :*

Antoine BRUNETTI

# Sommaire

<b>Introduction</b>	<b>2</b>
<b>1 Conception générale de l'application</b>	<b>3</b>
1.1 Description du sujet et analyse des besoins . . . . .	3
1.2 Diagramme de cas d'utilisation . . . . .	4
1.3 Planification des tâches . . . . .	4
1.4 Outils de développement . . . . .	5
<b>2 Modélisation UML et architecture du code</b>	<b>7</b>
2.1 Modélisation en classes d'objets . . . . .	7
2.1.1 Règles de la génération de jeux de données . . . . .	7
2.1.2 Importation des règles de génération des données . . . . .	9
2.1.3 Génération du jeu de données . . . . .	9
2.1.4 Export . . . . .	10
2.2 Diagramme d'activité : un exemple . . . . .	10
2.3 Modèle physique de base de données . . . . .	12
2.4 Structuration du code par couches . . . . .	13
2.4.1 Diagramme de package . . . . .	13
2.4.2 Le package de Business Object . . . . .	14
2.4.3 Le package tests . . . . .	14
2.4.4 Les packages DAO, factory et utils . . . . .	15
<b>3 Fonctionnalités de l'application et Guide d'utilisation</b>	<b>16</b>
3.1 Règles de génération des données . . . . .	16
3.1.1 Création des types . . . . .	16
3.1.2 Création des modalités . . . . .	16
3.1.3 Création des méta-types . . . . .	18
3.1.4 Génération et sauvegarde des données . . . . .	19
3.2 Import . . . . .	20
3.3 Export . . . . .	21
3.4 Data Access Object . . . . .	21
3.4.1 Gestion des modalités et des types . . . . .	21
3.4.2 Gestion des méta_types . . . . .	22
3.4.3 Gestion des données . . . . .	22
<b>4 Tests</b>	<b>24</b>
4.1 Tests des modules du packages business object . . . . .	24
4.1.1 Test de la classe generation_donnee . . . . .	24
4.1.2 Test de classe export_to_csv . . . . .	25
4.1.3 Test de la classe export_to_xml . . . . .	26
<b>5 Difficultés rencontrées et pistes d'amélioration</b>	<b>27</b>
<b>Conclusion</b>	<b>29</b>
<b>Annexe</b>	<b>30</b>

# 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. Enfin, la mise en place d'une application permet de rendre plus concrète la finalité de notre développement.

Comme le 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. Nous présenterons la mise en place du projet dans cette dynamique de groupe dans une première partie.

Ce rapport explicite le travail que nous avons réalisé relativement au sujet proposé par M. Antoine Brunetti : création d'un webservice de génération de données. La première partie est consacrée à la définition précise du travail à réaliser, puis à l'identification des outils à mettre en oeuvre pour le mener à bien. La seconde partie présente la modélisation UML réalisée, à l'aide de diagrammes adaptés qui résument les éléments que nous avons développés par la suite. Ces diagrammes ont été utiles à la compréhension précise par tous des objets manipulés, et à la cohérence des notations utilisées lors de développements en parallèle. Ils permettent également d'avoir une vue globale de la structure de l'application, de son architecture ainsi que de celle du code. Dans la suite, les fonctionnalités de l'application sont décrites précisément et un guide d'utilisation est proposé afin de permettre aux utilisateurs une bonne utilisation de l'API. Enfin, nous présentons les tests réalisés afin de s'assurer du bon fonctionnement de l'application relativement aux usages prévus.

Ce travail a été mené collectivement et dans une bonne dynamique. Faute de temps ou d'idées venues trop tardivement, certains points d'amélioration sont restés en suspens et pourraient mériter des développements supplémentaires. Nous concluons ce rapport par un bref aperçu des ouvertures possibles de ce projet.

# 1 Conception générale de l'application

## 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.

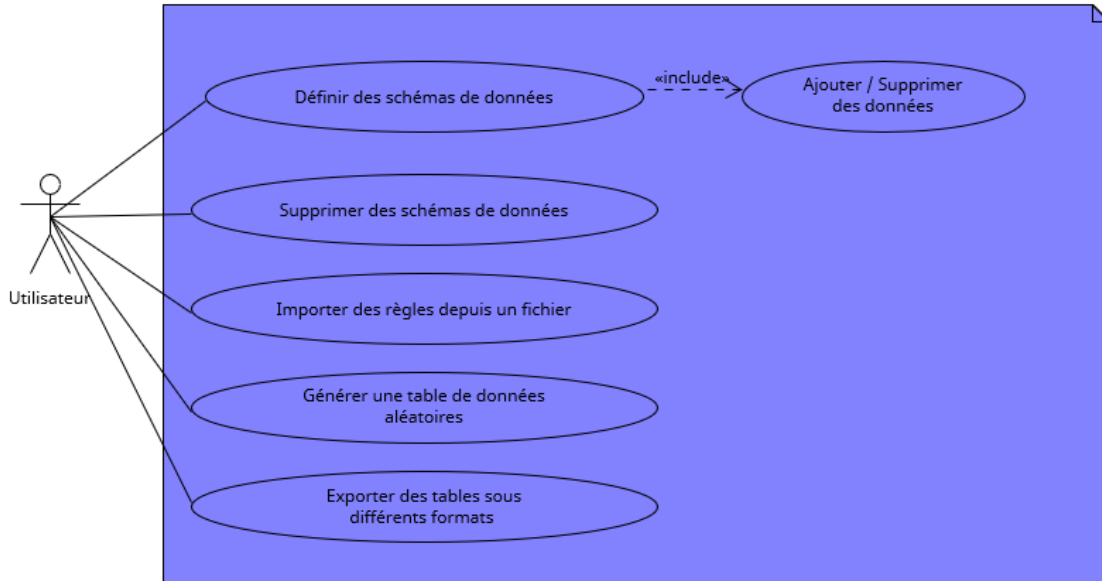
Une fois la génération d'un jeu de données effectuée, la table obtenue doit pouvoir être stockée en vue de sa récupération par l'utilisateur et de son exploitation future. Ce stockage s'effectue via une base de données. L'utilisateur doit donc pouvoir d'une part générer une base de données et d'autre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

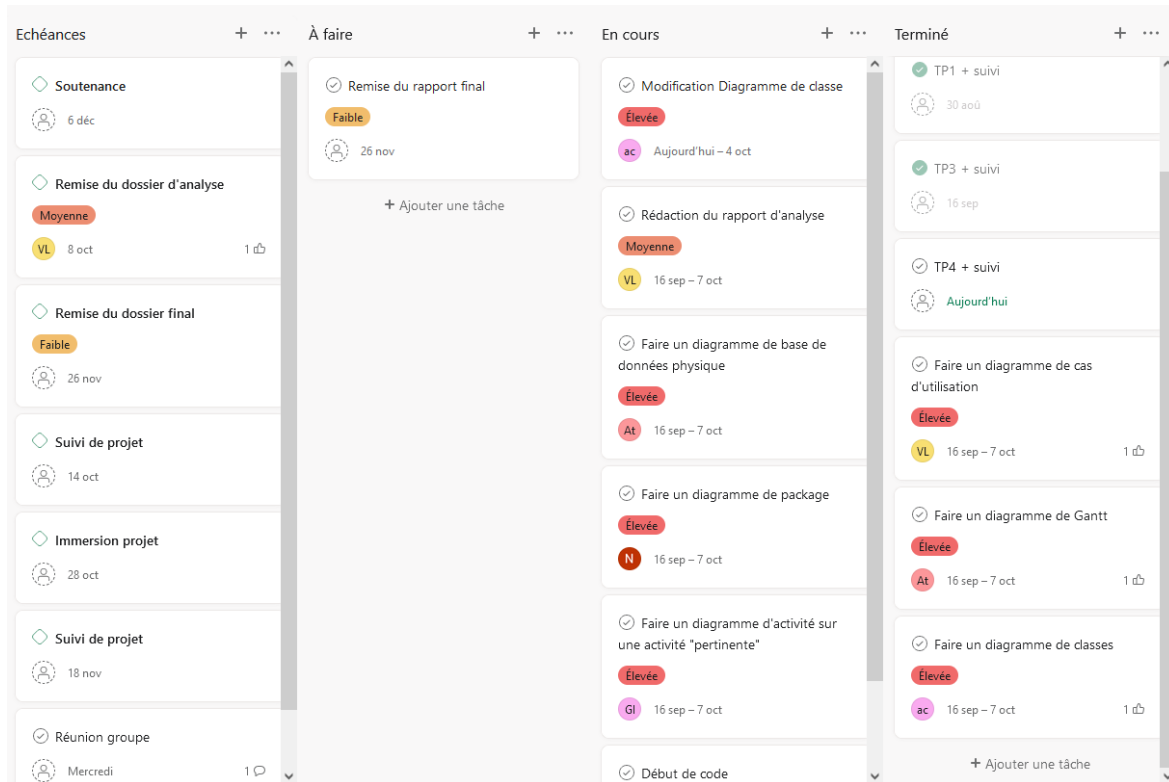
Afin de coordonner ce travail, un tableau Kanban est réalisé via le webservice Asana. Quatre catégories de tâches sont définies :

- 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 des diagrammes :

- **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 **PostgreSQL15** est utilisée. La connexion au serveur et l'exploitation de la base de données sont réalisées grâce à l'application **pgAdmin4**. Des extensions python sont exploitées dans la perspective de faire le lien entre python et le SGBD :

- **psycopg2\_binary** : pour la connexion à la base de données
- **requests** : pour la réalisation de requêtes

D'autres librairies sont utilisées pour le développement et recensées dans le fichier **requirements.txt** à la racine du projet. On peut citer notamment :

- **abc** pour gérer l'héritage entre les classes
- **unittest** pour la réalisation de tests unitaires
- **pandas** pour manipuler plus facilement les fichiers json notamment
- **fastAPI** pour créer l'application

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.

## 2 Modélisation UML et architecture du code

### 2.1 Modélisation en classes d'objets

#### 2.1.1 Règles de la génération de jeux de données

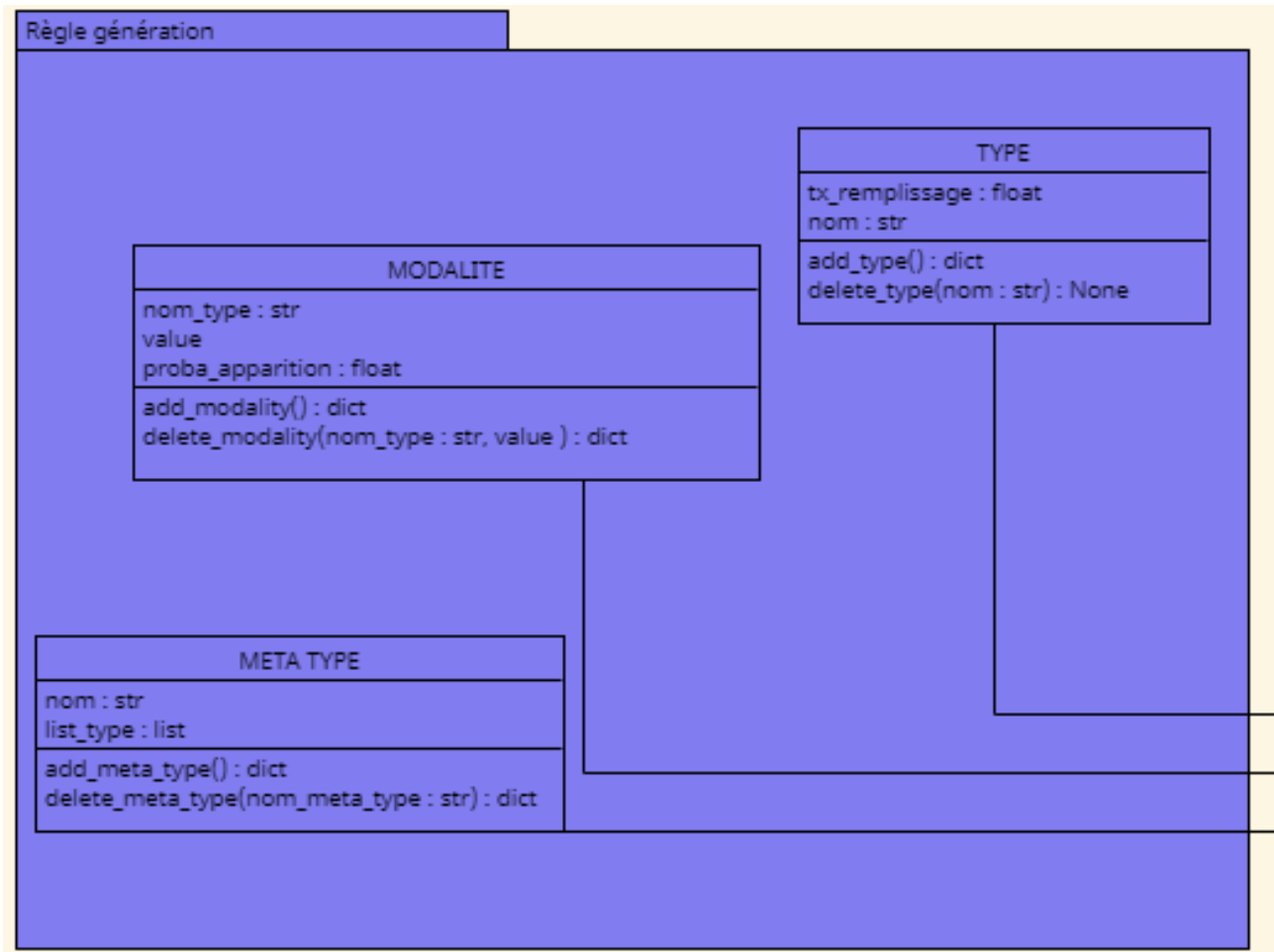


FIGURE 3 – Classes associées à la création des règles permettant la génération du jeu de données

- **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)
  - **tx\_remplissage** : un réel compris entre 0 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 taux de remplissage, un nom et qui ajoute un nouveau type au dictionnaire des types
  - **Delete\_type** : une méthode qui prend en argument un nom d'un type et qui enlève le type correspondant du dictionnaire de types
- **La classe MODALITE** : permet à l'utilisateur de créer de nouvelles modalités correspondant à un certain type



- **nom\_type** : une chaîne de caractères 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 le nom du type associé, 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 la valeur d'une modalité ainsi que le nom du type associé et qui enlève la modalité correspondante du dictionnaire des modalités
- **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
- **list\_type** : une liste de chaînes de caractères qui permet d'identifier une liste de types
  - **nom** : une chaîne de caractères qui permet de savoir comment nommer le méta-type en question
  - **Add\_list\_meta** : une méthode qui prend en argument une liste de type, le nom du 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 le nom du meta\_type et qui retire le méta-type correspondant du dictionnaire

### 2.1.2 Importation des règles de génération des données

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.

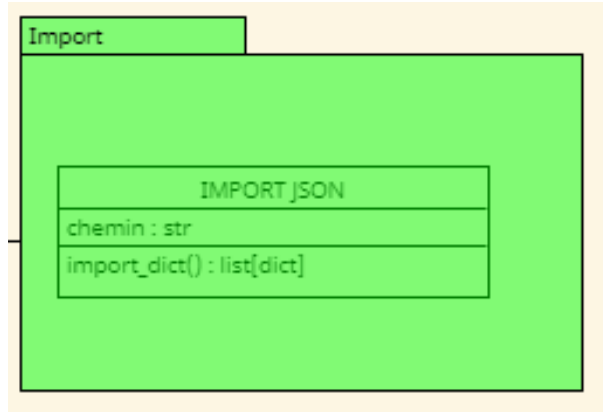


FIGURE 4 – Importation des règles de génération

- **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 et qui retourne une liste de dictionnaires

### 2.1.3 Génération du jeu de données

Maintenant que les règles de génération sont fixées il faut générer les données

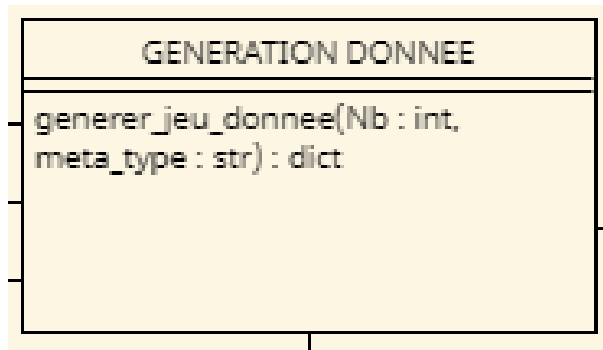


FIGURE 5 – Génération des données

- **La classe GENERATION DONNEE** : permet, à partir des règles de génération, de générer un nombre de données, défini 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

- **meta\_type** : une chaîne de caractères permettant d'indiquer à quel méta-type les données que l'on souhaite générer doivent correspondre. Cette chaîne de caractères renvoie à l'attribut `nom_meta_type` des objets métiers.
- **generer\_jeu\_donnee** : une méthode qui prend en argument `Nb` et `meta_type`, qui stocke dans un dictionnaire un nombre `Nb` de données générées correspondant au méta-type indiqué et qui retourne ce dictionnaire

#### 2.1.4 Export

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

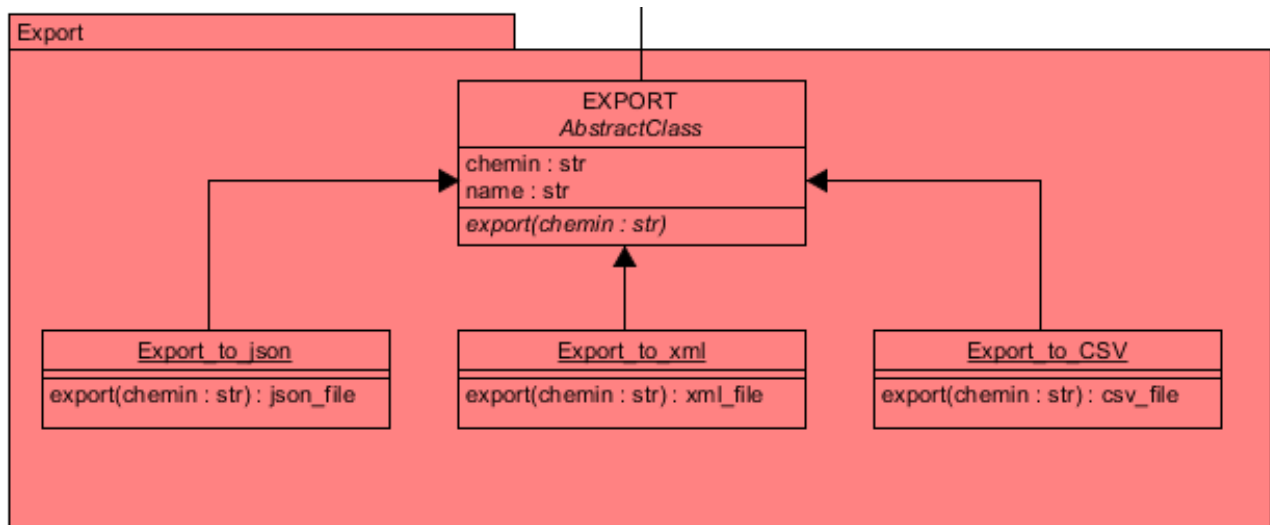


FIGURE 6 – Export des données

De la classe abstraite **EXPORT** héritent 3 classes filles qui fonctionnent de manière similaire : Les classes **Export\_to\_json**, **Export\_to\_xml** et **Export\_to\_CSV**. Ces classes permettent de stocker localement les tables générées, à partir d'un chemin donné par l'utilisateur. Les attributs et méthodes de ces classes sont ainsi identiques :

- **chemin** : une chaîne de caractères qui permet d'avoir l'emplacement où la table doit être stockée
- **name** : une chaîne de caractères qui permet de nommer le fichier pour le sauvegarder
- **export** : une méthode 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.

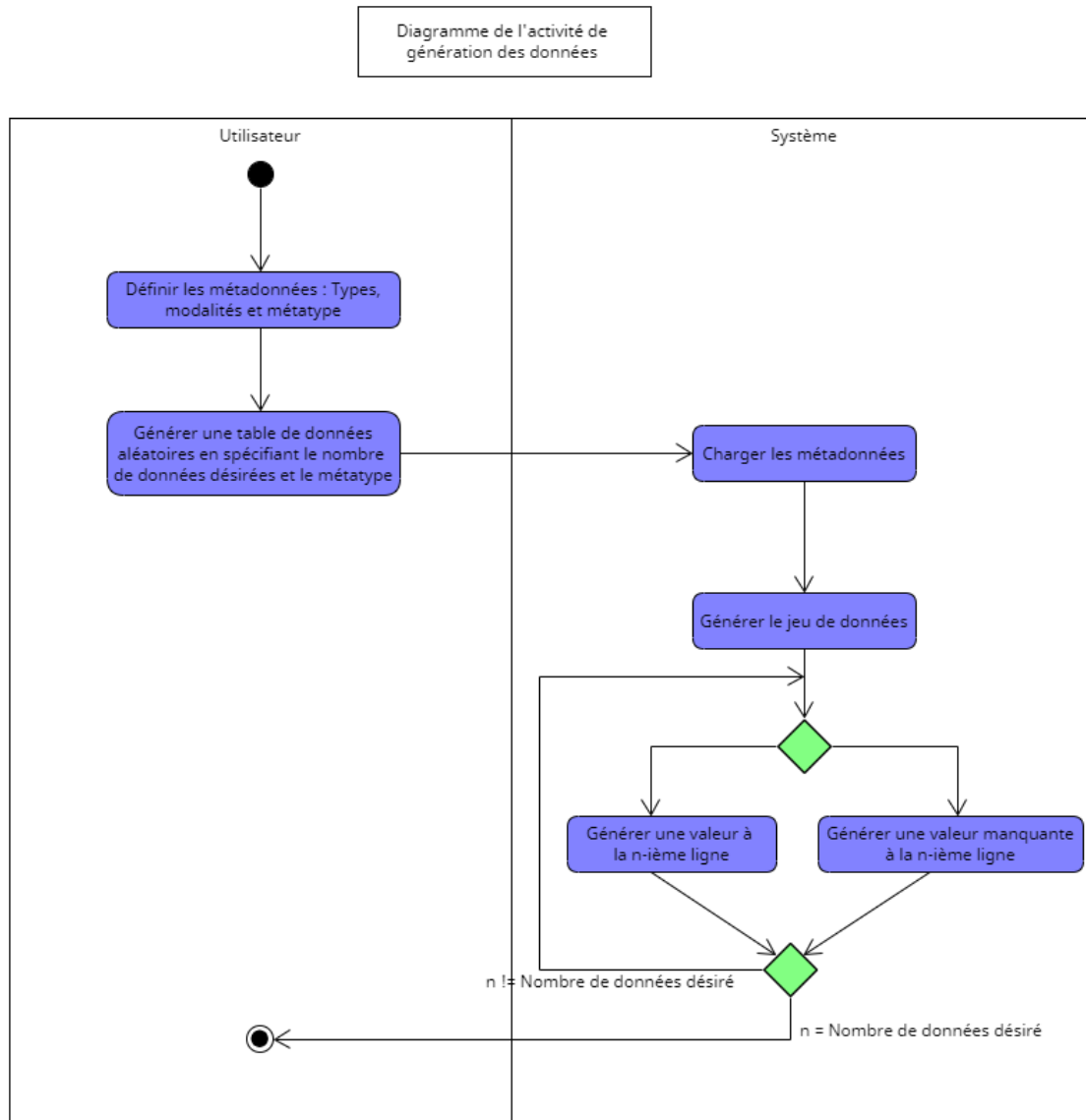


FIGURE 7 – Diagramme de l'activité de génération de données aléatoires avec des métadonnées définies par l'utilisateur

Pour utiliser cette fonctionnalité centrale de l'API, l'utilisateur doit au préalable définir les métadonnées qui régissent la génération des données. Il précise les différents types, les modalités associées à ces types puis il choisit le métatype qui représente la liste des types à considérer pour la génération des données. Une fois cette étape franchie, il renseigne le nombre de données à générer. Après avoir chargé les métadonnées renseignées par l'utilisateur, l'API génère automatiquement le jeu de données, c'est à dire la structure des données à générer, sur la base des informations contenues dans les métadonnées. Dès lors l'API est capable de générer aléatoirement les données en fonction du taux de remplissage de chaque modalité pour chaque type. Ainsi, pour chaque type, elle génère aléatoirement soit une modalité renseignée pour ce type, soit une valeur manquante. Une fois que le nombre de données désiré est atteint, l'algorithme prend fin.

## 2.3 Modèle physique de base de données

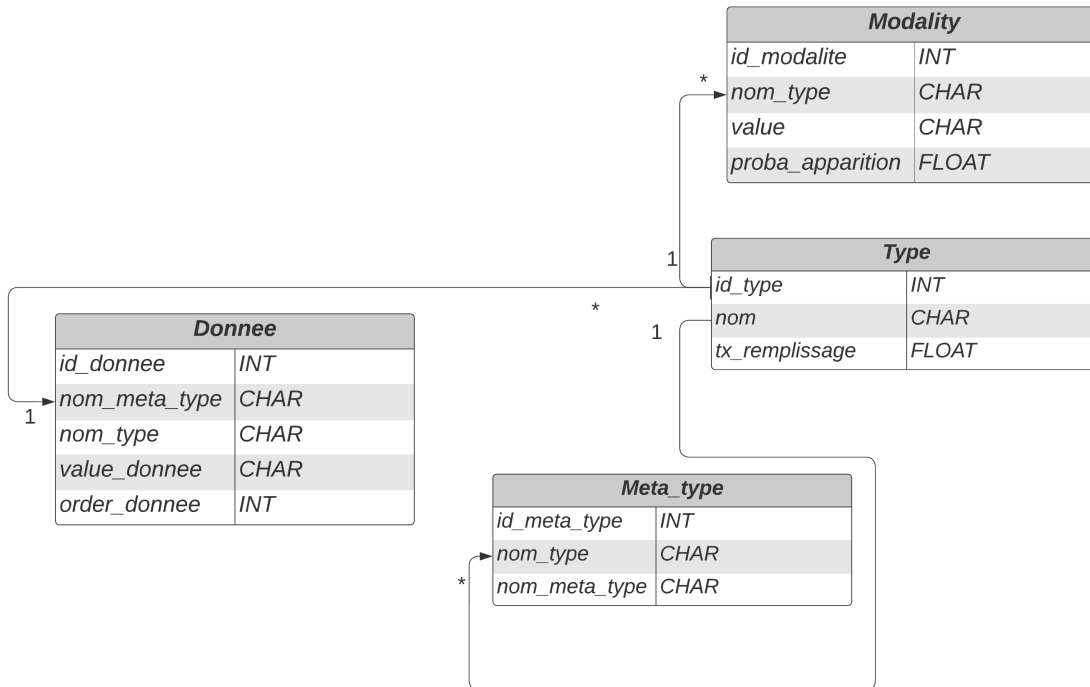


FIGURE 8 – Diagramme de base de données physique

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 , tx\_remplissage)

La table **modality** regroupe les valeurs qui peuvent être prises par les différents types ainsi que leur probabilité d'apparition.

Exemple : Modality(id\_modality, nom\_type, value, proba\_apparition)

La table **Meta\_type** permet de définir les "types composés". Un même méta-type sera associé à plusieurs types donc à plusieurs *id\_meta\_type*. Par exemple, un méta-type **individu** composé de plusieurs types qui sont le nom, prénom, âge, etc. Ce dernier point nous a conduit à ne pas définir les noms des types et des méta\_types comme des clés étrangères, ce qui posait des problèmes d'unicité de clés que nous n'avons pas eu le temps de résoudre. L'unicité des types, des modalités et des méta-types est assurée par des contrôles en amont de la création et la suppression des doublons au moment de la sauvegarde en base de données.

Enfin, la table **donnee** recense toutes les données générées. 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, et la récupération des "individus statistiques", c'est-à-dire des lignes entières de la table finale.

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

## 2.4 Structuration du code par couches

### 2.4.1 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.

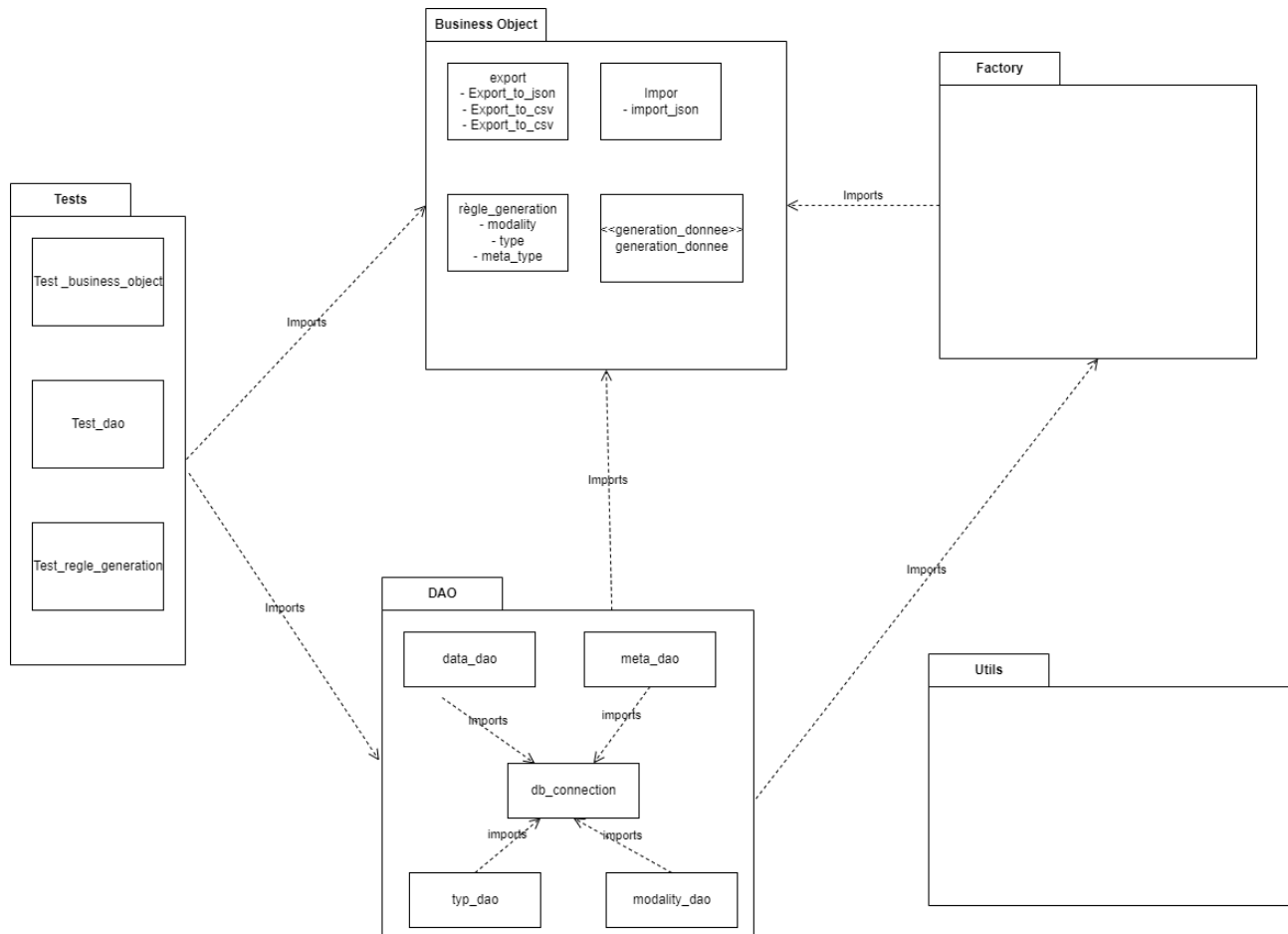


FIGURE 9 – Diagramme de paquetage

Nous avons décidé de regrouper les classes en cinq packages :

- Le package **Business Object** qui contient les classes métiers
- Le package **DAO** qui contient toutes les classes liées à la communication entre l'API et la base de données
- Le package **Factory** qui fait le lien entre la DAO et les objets métiers
- Le package **Utils** qui définit la classe Singleton, support à la connexion à la base de données
- Le package **Tests** qui réunit les tests des objets métiers et quelques tests de dao

### 2.4.2 Le package de Business Object

Les quatre sous-packages correspondant se présentent ainsi dans notre projet (cf Figure 10a) :

— Le module **Export** qui contient les classes du type export c'est à dire :

- **Export\_to\_json**
- **Export\_to\_xml**
- **Export\_to\_csv**

Ces modules implémentent les méthodes pour exporter le jeu de données sous format json, xml et csv.

— Le module **Import** qui contient le module import, ce module implémente les méthodes pour importer un fichier sous format json.

— Le module **regle\_generation** qui contient les modules **modality**, **type** et **meta\_type**.

— Le module **generation\_donnee** qui ne contient que la classe **generation\_donnee**

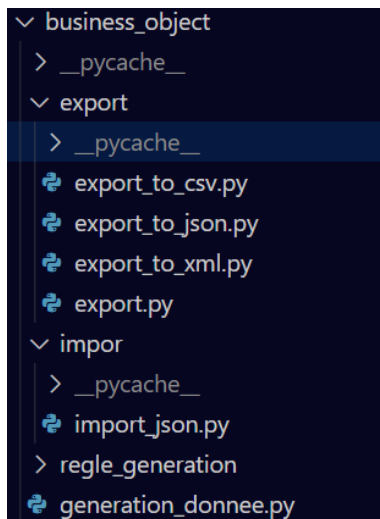
### 2.4.3 Le package tests

Ce package regroupe trois sous-packages, qui se présentent ainsi dans notre projet (cf Figure 10b) :

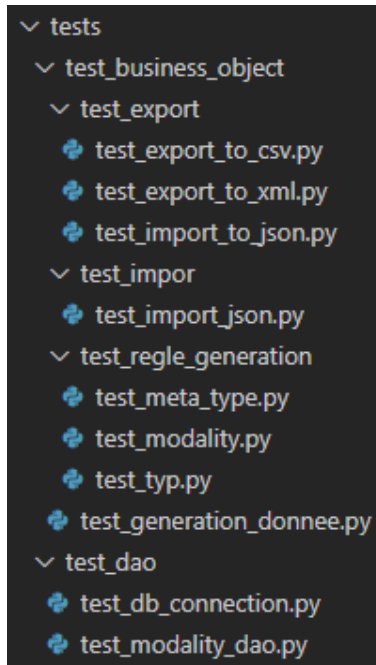
— Le package **test\_business\_object** qui regroupe les tests liés aux objets métiers, c'est-à-dire liés à l'export, à l'import et aux règles de génération des données (types, modalités et méta-types).

— Le package **test\_regle\_generation** qui teste la génération d'un jeu de données.

— Le package **test\_dao** qui propose un test de connexion et quelques tests sur la classe **ModalityDao**.



(a) package business\_object



(b) Package tests

#### 2.4.4 Les packages DAO, factory et utils

La DAO utilise en fait 3 packages, qui se présentent ainsi dans notre projet :

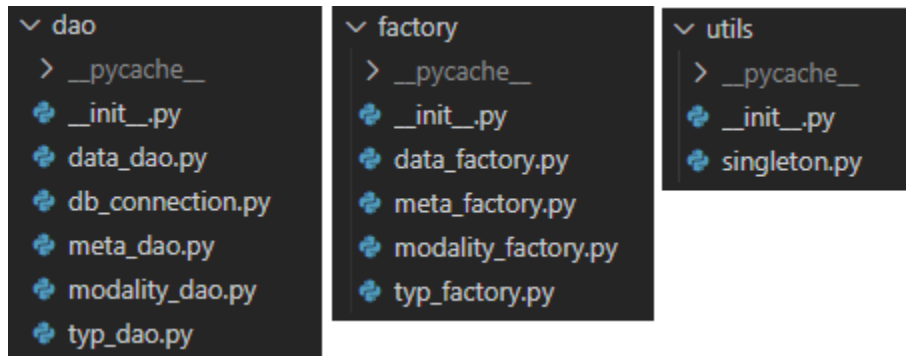


FIGURE 11 – Les packages liés à la DAO

- Le package **utils** : nous avons réinvesti la classe **Singleton** vue en TP de compléments informatiques. Elle nous permet d'assurer que la connexion à la base de données se fait de manière unique.
- Le package **dao** qui recense les méthodes permettant de stocker, récupérer, modifier ou supprimer les données en persistance, sur une base de données distante.
- Le package **factory** qui fait le lien entre les données brutes récupérées dans la base de données et les objets métiers, définis pour notre projet, que nous utilisons ou créons dans les méthodes des classes de dao.

La connexion à la base de données distante est assurée par la classe **DBConnection** définie dans le module **db\_connection.py** et qui hérite de la classe **Singleton**. Les autres classes associées à la DAO utilisent la fonction **connection** de cette classe pour communiquer avec la base de données.

A chaque objet métier (**Modality**, **Type**, **Meta\_Type**) correspond une classe DAO (**Modality-DAO**, **TypeDAO**, **MetaDAO**). La dernière classe DAO (**DataDAO**) correspond à celle qui interagit avec la table stockant les données générées.

Les méthodes implémentées dans ces classes sont de plusieurs types, qui permettent notamment :

- d'enregistrer des données dans la base : méthodes **save\_object(...)**
- de trouver des données, (par leur nom, leur(s) id(s) ou autre) : méthodes **find\_object\_by...(...)**
- de modifier des données : méthodes **update\_object(...)**
- de supprimer des données : méthodes **delete\_object(...)**

Dans les noms de méthodes génériques ci-dessus, **object** est remplacé par **modality**, **type**, **meta\_type** ou **data**. Ces méthodes sont appelées par l'API via des commandes **Get**, **Put** ou **Delete**, définies dans le module **api.py** à la racine du projet.



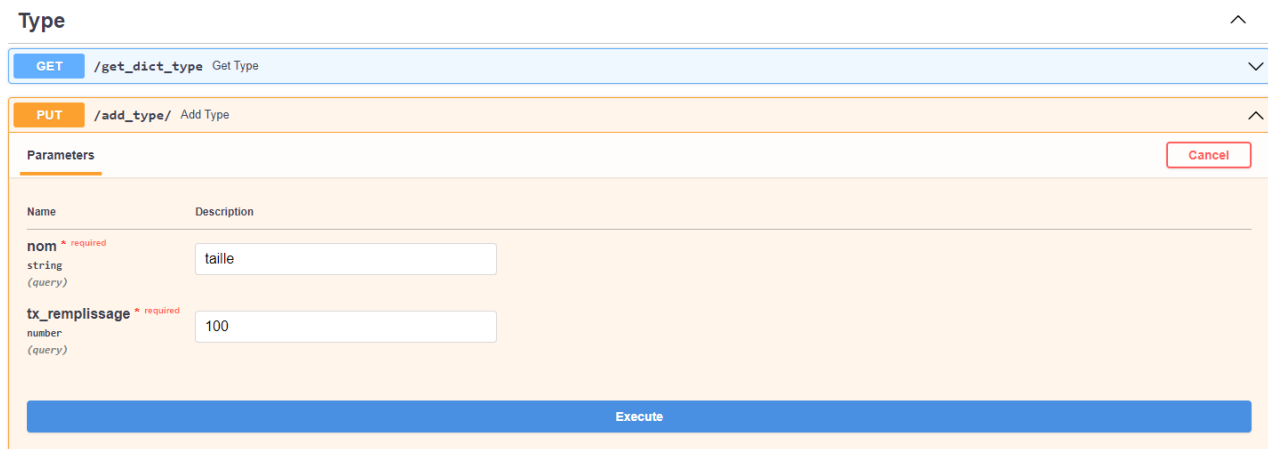
## 3 Fonctionnalités de l'application et Guide d'utilisation

### 3.1 Règles de génération des données

#### 3.1.1 Création des types

La première fonctionnalité de l'application nécessaire pour générer les données est la définition de règles de génération et notamment les types qu'il veut pouvoir générer.

L'utilisateur doit tout d'abord définir les différents types qu'il veut créer. Il a pour cela sur l'API accès à une fonction `add_type`, dans laquelle il pourra renseigner un nom, pour nommer le type en question et il devra aussi définir un taux de remplissage. Le taux de remplissage est un nombre compris entre 0 et 100 permettant de connaître le pourcentage de chance qu'une fois les données générées, il y ait bien une valeur dans le champ correspondant et non une donnée manquante.



The screenshot shows a web interface for the 'Type' API endpoint. At the top, there's a header 'Type' with a collapse icon. Below it, a blue bar shows the GET endpoint '/get\_dict\_type' with the description 'Get Type'. The main section is for the PUT endpoint '/add\_type/' with the description 'Add Type'. It features a 'Parameters' section with two input fields: 'nom' (required, string, query) with the value 'taille', and 'tx\_remplissage' (required, number, query) with the value '100'. A 'Cancel' button is in the top right of the parameters section. At the bottom, there is a large blue 'Execute' button.

FIGURE 12 – Ajout d'un type

Par exemple dans la figure ci-dessus l'utilisateur a ajouté le type *taille* avec un taux de remplissage de 100%

L'utilisateur peut aussi décider de supprimer des types déjà existant : il utilise la fonction `delete_type` de l'API. Pour cela, il doit renseigner un nom correspondant au type qu'il souhaite retirer du dictionnaire des types. Si le type existe, il sera alors retiré, sinon l'utilisateur recevra un message lui signalant qu'il a peut-être mal orthographié le type qu'il voulait retirer. L'API dispose également de la fonctionnalité `get_type` : elle permet d'afficher l'ensemble des types dans le dictionnaire des types à ce moment-là, au cas où l'utilisateur ne saurait plus ce qu'il a créé exactement.

#### 3.1.2 Création des modalités

Une fois que les types sont définis il est important de pouvoir y associer des modalités.

L'utilisateur peut ajouter des modalités avec la fonction `add_modality`. Cette fonction demande à l'utilisateur de renseigner : le nom du type associé à la modalité, la valeur de la modalité (un réel ou une chaîne de caractères) ainsi que la probabilité d'apparition. La probabilité d'apparition doit être comprise entre 0 et 100 et peut être calculée au prorata. Par exemple, si l'on veut associer 3 modalités au même type et que l'on rentre comme probabilités d'apparition 50, 50 et 100, alors les

modalités ont respectivement 25% , 25% et 50% de chance d'apparaître. Cela permet notamment de ne pas devoir modifier toutes les probabilités d'apparition associées à un type si l'utilisateur a oublié d'en ajouter une.

Il est aussi possible d'enlever des modalités du dictionnaire des modalités avec la fonction `delete_modality`. Il suffit de renseigner le nom du type associé à la modalité ainsi que la valeur de celle-ci pour la retirer du dictionnaire des modalités. Si par erreur la modalité en question n'existe pas ou que le type associé n'est pas dans le dictionnaire des types alors l'API renvoie un message d'erreur signalant à l'utilisateur de vérifier l'orthographe de ce qu'il a renseigné. Il est également possible de simplement afficher le dictionnaire des modalités avec la fonction `get_modality`.

L'utilisateur peut également créer des types dont les modalités sont continues. Il doit alors renseigner des éléments spécifiques dans les modalités associées. Il est possible de générer des variables continues suivant 3 lois :

- **normale**
- **uniforme**
- **exponentielle**

The screenshot shows a web interface for the 'Modality' API. At the top, there's a header 'Modality' with a dropdown menu showing 'GET /get\_dict\_modality Get Modality'. Below this is a section for 'PUT /add\_modality/ Add Modality'. Inside this section, there's a 'Parameters' table with three rows:

Name	Description
nom_type * required string (query)	taille
proba_apparition * required number (query)	1
value * required (query)	uniform

At the bottom of the form is a blue button labeled 'Execute'. There is also a 'Cancel' button in the top right corner of the parameters section.

FIGURE 13 – Ajout d'une variable uniforme

Tout d'abord il faut renseigner une modalité dont la valeur doit contenir une chaîne de caractères correspondant à "uniform", "normal" ou "exponential" selon la loi choisie. Lorsque cette modalité est renseignée la probabilité d'apparition n'a pas d'importance. Il faut ensuite que l'utilisateur entre les paramètres associés à cette loi.

- **Pour la loi normale** : Il y a deux paramètres à rentrer, "mean" et "variance". L'utilisateur doit donc renseigner le type associé puis dans la valeur de la modalité mean ou variance selon le paramètre qu'il veut renseigner et enfin dans probabilité d'apparition il doit rentrer la valeur de ce paramètre.
- **Pour la loi uniforme** : Les deux paramètres à renseigner sont "borne1" (correspondant à la borne inférieure) et "borne2" (correspondant à la borne supérieure). Puis comme pour la loi normale l'utilisateur renseigne le type associé, puis dans valeur de la modalité il renseigne "borne1" ou "borne2" et dans probabilité d'apparition la valeur de celle-ci.

- **Pour la loi exponentielle** : Il y a un seul paramètre à renseigner, le paramètre lambda. L'utilisateur implémente ce paramètre de la même manière que pour les lois précédentes.

The screenshot shows a web interface for adding a modality. At the top, there's a 'Modality' header. Below it, a blue bar indicates the current endpoint is 'PUT /add\_modality/' with the description 'Add Modality'. Underneath, a 'Parameters' section is visible, containing three input fields:

- nom\_type** (string, required): The value 'taille' is entered.
- proba\_apparition** (number, required): The value '160' is entered.
- value** (number, required): The value 'borne1' is entered.

A red 'Cancel' button is located in the top right of the parameters section. At the bottom of the interface is a large blue 'Execute' button.

FIGURE 14 – Ajout d'un paramètre

Dans les exemples précédents l'utilisateur a ajouté une variable taille qui suit une loi uniforme, et un paramètre qui est la borne inférieure qui vaut ici 160 cm. S'il ajoute le deuxième paramètre, borne2 avec une valeur de 190cm, alors la variable taille suivra une loi uniforme entre 160 et 190 cm.

Si un des paramètres vient à manquer alors la variable ne sera pas générée comme une variable continue. Ces règles de création des types et des modalités peuvent être directement importées depuis un fichier json comme nous le verrons par la suite.

### 3.1.3 Création des méta-types

Une fois que l'utilisateur a créé les types et les modalités il lui faut encore implémenter les méta-types avant de pouvoir générer des données.

Pour cela l'utilisateur doit utiliser la fonction `add_meta_type`. Il doit renseigner le nom du méta-type qu'il veut créer ainsi que la liste des types correspondant au méta-type. Si les types sélectionnés n'existent pas, l'API retourne une erreur en disant à l'utilisateur que tous les types sélectionnés n'existent pas.

**Meta-Type**

GET /get\_dict\_meta\_type Get Meta Type

PUT /add\_meta\_type/ Add Meta Type

Parameters

Name Description

nom \* required  
string  
(query) individu

Request body required application/json

```
[
  "Prénom", "Age", "Taille"
]
```

FIGURE 15 – Ajout d'un méta-type

Dans la figure ci-dessus l'utilisateur ajoute le méta-type "individu", qui est composé des variables "Prénom", "Age" et "Taille".

L'utilisateur peut également retirer des méta-types existant avec la fonction `delete_meta_type` et il peut voir le dictionnaire des méta-types avec la fonction `get_dict_meta_type`.

### 3.1.4 Génération et sauvegarde des données

La génération de données ne peut avoir lieu qu'une fois le méta-type correspondant créé. L'utilisateur peut alors générer des données de ce méta\_type, en indiquant le nombre de données souhaité (Nb) et le méta\_type à considérer. L'utilisateur devra donc générer autant de jeux de données que de méta\_types voulus.

**Génération et sauvegarde**

PUT /generation\_de\_donnee/ Generation Donne

Parameters

Name Description

Nb \* required  
integer  
(query) 50

meta\_type \* required  
(query) individu

Execute

FIGURE 16 – Génération d'un jeu de données

**Avertissement :** Si l'utilisateur souhaite stocker ces données dans la base de données, il doit les sauvegarder après chaque génération. Sinon, seulement les données correspondant au dernier méta\_type entré seront sauvegardées.

Une fois les données générées, l'utilisateur peut donc :

- voir le dictionnaire des données avec la fonction `get_dict_data`

- réinitialiser la base de données avec la fonction `reinitialiser_base_de_donnees`
- sauvegarder les données dans la base avec la fonction `sauvegarder_en_base_de_donnees`

L'application dispose d'un bouton pour chacun de ces usages, comme indiqué ci-dessous :

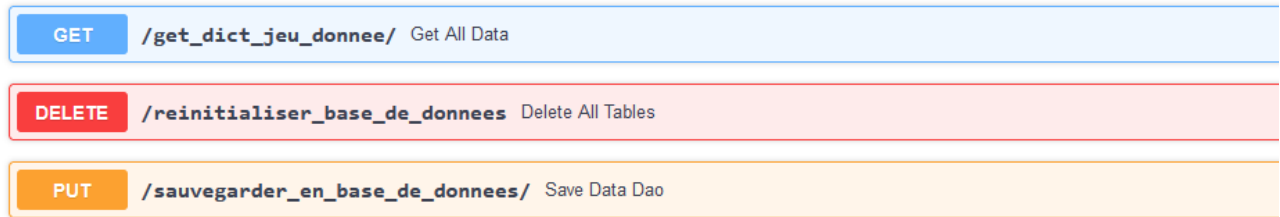


FIGURE 17 – Génération et sauvegarde de jeux de données

### 3.2 Import

Il est possible d'importer un fichier Json contenant les règles de génération plutôt que de tout créer manuellement avec l'API. En effet si le fichier Json a une forme bien précise (détaillée dans le code de la classe import) alors il est possible d'importer directement les dictionnaires des types et des modalités. L'utilisateur n'aura alors plus qu'à créer les meta-types avec L'API avant de pouvoir générer les données.

```
{
  "sexe": {
    "type": "M|F",
    "remplissage": 100,
    "proba d'apparition": ["same"]
  },
  "age": {
    "type": "uniform|borne1|borne2",
    "remplissage": 100,
    "proba d'apparition": [1,0,100]
  },
  "prenom": {
    "type": "Adrien|Abdoul|Laurène|Isaac|Nathan",
    "remplissage": 88.4,
    "proba d'apparition": ["same"]
  },
  "nom": {
    "type": "Cortada|Toure|Villacampa|Sandja",
    "remplissage": 85,
    "proba d'apparition": [20,20,20,20]
  },
}
```

FIGURE 18 – Exemple de fichier Json permettant l'import des règles de génération

Pour cela l'utilisateur doit utiliser la fonction `import_json` dans l'API et renseigner le chemin permettant l'accès au fichier Json en question. Tous les types et modalités contenus dans le fichier en question seront alors ajoutés aux dictionnaire des types et au dictionnaire des modalités.

### 3.3 Export

Une fois le jeu de données généré il est possible de l'exporter et de le sauvegarder dans trois formats : json, xml et csv. Pour cela l'utilisateur doit utiliser la fonction `Export`.

The screenshot shows the API interface for the `/export/` endpoint. The 'Parameters' section lists two required query parameters: `chemin` (string) and `name` (string). Both parameters have input fields. A 'Cancel' button is located in the top right corner, and an 'Execute' button is at the bottom.

FIGURE 19 – Exportation d'un jeu de données

L'utilisateur aura la possibilité de choisir où est ce qu'il va sauvegarder son jeu de données en choisissant le chemin ainsi que le nom. De plus l'application dispose de trois fonctions différentes selon le format sous lequel l'utilisateur veut sauvegarder son fichier : `export_to_json`, `export_to_xml` et `export_to_csv`.

The screenshot shows three API endpoints for exporting data: `/export_donnees_to_json/` (Export Json), `/export_donnees_to_xml/` (Export Xml), and `/export_donnees_to_csv/` (Export Csv). Each endpoint has a 'GET' method and a dropdown arrow.

FIGURE 20 – Exportation d'un jeu de données sous plusieurs format

### 3.4 Data Access Object

L'application permet à l'utilisateur de procéder à de multiples actions sur les tables stockées en base de données. Ces fonctionnalités ne sont opérationnelles qu'après avoir sauvegardé tous les éléments (types, modalités, méta\_types et données) correspondant à la génération de données en base de données. Cela se fait synthétiquement via la méthode `sauvegarder_en_base_de_donnees`. Ces différentes fonctionnalités font appel aux méthodes déjà vues dans la partie 2.4.2 à propos de la DAO. Nous ne les détaillerons pas toutes ici mais les présenterons de manière synthétique.

#### 3.4.1 Gestion des modalités et des types

L'utilisateur peut accéder à l'ensemble des modalités et des types via les fonctions `find_all_...`. Il peut également accéder à des modalités ou types particuliers via des fonctions `find_object_by...` en les recherchant par :

- leur identifiant (`_by_id`)

- leur nom (`_by_name`)
- leurs caractéristiques complètes, pour vérifier par exemple qu'une modalité ou un type figure bien dans la table correspondante. (`_by_modality` ou `_by_type`)

Les identifiants sont générés automatiquement dans la base de données au moment de la sauvegarde, à l'aide de séquences auto-incrémentées. Il est possible pour l'utilisateur de retrouver l'identifiant d'un type en spécifiant ses caractéristiques (taux de remplissage et nom) avec la fonction `find_id_type`. Nous nous sommes assurés qu'il n'y ait pas de doublons dans les tables `modality` et `type`.

L'utilisateur peut également modifier des modalités et des types en à partir de leur identifiant en utilisant les méthodes `update_modality_by_id` et `update_type_by_id` respectivement.

Enfin, il peut supprimer des modalités et des types, là encore en les recherchant soit par leur identifiant (id), soit par leurs caractéristiques, via les fonctions `delete_object_by...` ou `delete_all_object`.

### 3.4.2 Gestion des méta\_types

Le stockage des méta\_types est particulier. Dans la base de données, un même méta\_type correspond à plusieurs lignes comme l'illustre la figure ci-dessous :

	<code>id_meta_type</code> [PK] integer	<code>nom_meta_type</code> text	<code>nom_type</code> text
1	1	individu	nom
2	2	individu	prenom
3	3	individu	sexe
4	4	individu	age

FIGURE 21 – Exemple de table `meta_type`

Un méta\_type correspond donc à plusieurs identifiants : un pour chaque type dont il est constitué.

L'utilisateur peut accéder à tous les méta\_types via la fonction `find_all_meta`. Il peut accéder à un méta\_type particulier par son nom via la méthode `find_meta_by_name`. Il peut obtenir la liste des identifiants qui lui sont associés (également générés par une séquence auto-incrémentée) via la méthode `find_ids_meta`.

Enfin, il peut supprimer un méta\_type par son nom, ou l'ensemble des méta\_types déjà stockés avec les fonctions `delete_meta_by_name` et `delete_all_meta_type`.

### 3.4.3 Gestion des données

Un "individu statistique" (une "ligne" de la table finale) correspond en fait à plusieurs "données" (plusieurs lignes dans la base de données). Dans la Figure 22 ci-dessous, deux "individus statistiques" du méta\_type `individu` sont stockés, correspondant à huit "données" :

	id_donnee [PK] integer	nom_meta_type text	nom_type text	value_donnee text	order_donnee integer
1	1	individu	nom	Toure	1
2	2	individu	prenom	Nathan	2
3	3	individu	sexe	M	3
4	4	individu	age	20.46	4
5	5	individu	nom	Villacampa	1
6	6	individu	prenom	Isaac	2
7	7	individu	sexe	M	3
8	8	individu	age	60.67	4

FIGURE 22 – Exemple de table `donnee`

L'utilisateur peut récupérer la liste de dictionnaires correspondant à l'ensemble des données stockées via la fonction `find_all_data`. Il peut accéder à toutes les données associées à un `meta_type` particulier en renseignant son nom via la fonction `find_data_by_meta`. Il peut également accéder à une donnée particulière et la modifier par son identifiant grâce aux fonctions `find_data_by_id` et `update_data_by_id`. La fonction `find_id_donnee` permet de récupérer l'identifiant d'une donnée en spécifiant, le nom du `meta_type`, le nom du type, l'ordre du type et la valeur. Les méthodes de suppression permettent de supprimer une donnée par son identifiant pour `delete_data_by_id`, supprimer plusieurs données correspondant à un même `meta_type` (`delete_data_by_meta`), ou supprimer toutes les données (`delete_all_data`).

Des méthodes supplémentaires sont prévues afin de manipuler directement plusieurs éléments de la table `donnees` à la fois :

- Récupérer un individu statistique (une "ligne" complète de la table de données générée) : via la fonction `find_row_data`. Elle permet de récupérer tous les éléments de la table `donnees` faisant référence à un même individu de la table générée. Il suffit de spécifier le numéro de l'élément à récupérer . Par exemple, dans la figure 22, elle pourrait renvoyer les lignes 1 à 4 en renseignant qu'on veut le premier individu (1) ou les lignes 5 à 8 en renseignant qu'on veut le deuxième individu (2)
- Supprimer un individu statistique par son numéro, c'est à dire le numéro de ligne dans la table de données générée avec la fonction `delete_row_data`.
- Récupérer l'ensemble des valeurs d'une "colonne" de la table de données générée (les valeurs prises par un des types du `meta_type` considéré) : via la fonction `find_col_data`. Il suffit de spécifier le nom du `meta_type` et le nom du type. Par exemple, dans la figure 22, elle pourrait renvoyer les lignes 1 et 5 en renseignant comme nom du `meta_type` *individu* et comme nom du type *nom* ou les lignes 2 et 6 en renseignant comme nom du `meta_type` *individu* et comme nom du type *prenom*



## 4 Tests

### 4.1 Tests des modules du packages business object

#### 4.1.1 Test de la classe generation\_donnee

```
if __name__ == "__main__":
    t = Type(100, "age")
    t.add_type()
    t2 = Type(100, "prénom")
    t2.add_type()
    m = Modality("age", 100, 22)
    m.add_modality()
    m2 = Modality("prénom", 100, "Rémi")
    m2.add_modality()
    mt = Meta_type("individu", ["prénom", "age"])
    mt.add_meta_type()
    générer = Generation_donnee(10, "individu")
    if générer.generer_jeu_donnee() == {0: {'prénom': 'Rémi', 'age': 22}, 1: {'prénom': 'Rémi', 'age': 22},
    2: {'prénom': 'Rémi', 'age': 22}, 3: {'prénom': 'Rémi', 'age': 22},
    4: {'prénom': 'Rémi', 'age': 22}, 5: {'prénom': 'Rémi', 'age': 22},
    6: {'prénom': 'Rémi', 'age': 22}, 7: {'prénom': 'Rémi', 'age': 22},
    8: {'prénom': 'Rémi', 'age': 22}, 9: {'prénom': 'Rémi', 'age': 22}}:
        print("True")
    else:
        print("False")
    t3 = Type(100, 'adresse')
    t3.add_type()
    m3 = Modality('adresse', 100, 'SDF')
    m3.add_modality()
    mt2 = Meta_type("ind", ["prénom", "adresse"])
    mt2.add_meta_type()
    generer2 = Generation_donnee(10, "ind").generer_jeu_donnee()
    data = Generation_donnee.jeu_donnee
    liste = Generation_donnee.meta_type1
    if len(data) == 20:
        print("True")
    else:
        print("False")
    if len(liste) == 20:
        print("True")
    else:
        print("False")
```

FIGURE 23 – Test de la classe generation\_donnee

Ce test permet de voir le bon fonctionnement de la classe `generation_donnee`, la plus importante de notre API.

```
PS D:\Projet_2A\Projet-info-2A> & C:/Users/Toure/AppData/Local/Microsoft/WindowsApps/python3.10.exe d:/Projet_2A/Projet-info-2A/tests/test_business_object/test_generation_donnee.py
True
True
True
{0: {'prénom': 'Rémi', 'age': 22}, 1: {'prénom': 'Rémi', 'age': 22}, 2: {'prénom': 'Rémi', 'age': 22}, 3: {'prénom': 'Rémi', 'age': 22}, 4: {'prénom': 'Rémi', 'age': 22}, 5: {'prénom': 'Rémi', 'age': 22}, 6: {'prénom': 'Rémi', 'age': 22}, 7: {'prénom': 'Rémi', 'age': 22}, 8: {'prénom': 'Rémi', 'age': 22}, 9: {'prénom': 'Rémi', 'age': 22}, 10: {'prénom': 'Rémi', 'age': 22}, 11: {'prénom': 'Rémi', 'adresse': 'SDF'}, 12: {'prénom': 'Rémi', 'adresse': 'SDF'}, 13: {'prénom': 'Rémi', 'adresse': 'SDF'}, 14: {'prénom': 'Rémi', 'adresse': 'SDF'}, 15: {'prénom': 'Rémi', 'adresse': 'SDF'}, 16: {'prénom': 'Rémi', 'adresse': 'SDF'}, 17: {'prénom': 'Rémi', 'adresse': 'SDF'}, 18: {'prénom': 'Rémi', 'adresse': 'SDF'}, 19: {'prénom': 'Rémi', 'adresse': 'SDF'}}
```

FIGURE 24 – Résultat du test

On observe que le test a bien fonctionné et que la génération de la base de données est conforme à celle souhaitée.

#### 4.1.2 Test de classe export\_to\_csv

```
class testexport_to_csv(TestCase):

    def test_export_to_csv(self):
        table = {
            "sexe": {
                "type": "SEXE",
                "remplissage": 100
            },
            "age": {
                "type": "18|19|20",
                "remplissage": 100
            },
            "prenom": {
                "type": "NAME",
                "remplissage": 88.4
            },
            "nom": {
                "type": "NAME|'dupont'",
                "remplissage": 85
            }
        }

        # Vérifier si le fichier existe ou non
        if os.path.isfile("D:/Projet_Informatique_2A/table_csv.csv"):
            print("Fichier trouvé")
            tablecsv = Export_to_csv("D:/Projet_Informatique_2A/Projet-info-2A", "table_csv1.csv")
            csvfile = pd.read_csv("D:/Projet_Informatique_2A/Projet-info-2A/table_csv.csv")
            table_csv1 = tablecsv.export(json.dumps(table))
            table_csv = pd.read_csv("D:/Projet_Informatique_2A/Projet-info-2A/table_csv1.csv")
            print(table_csv==csvfile)
        else:
            print("Fichier non trouvé")
if __name__ == "__main__":
    unittest.main()
```

FIGURE 25 – Test de la classe export\_to\_csv

Le test de la classe `export_to_csv` a pour but de vérifier que le fichier existe bien et que les éléments qui s'y trouvent sont ceux attendus.

```
test.py
fichier trouvé
  Unnamed: 0  type  remplissage
0         True  True         True
1         True  True         True
2         True  True         True
3         True  True         True
.
-----
Ran 1 test in 0.263s

OK
PS D:\Projet_2A\Projet-info-2A> █
```

FIGURE 26 – Résultat du test de la classe

On observe que le test a bien fonctionné. Le fichier a bien été trouvé et les éléments sont bien ceux voulus.

### 4.1.3 Test de la classe export\_to\_xml

```
class TestExport_to_xml(TestCase):
    def test_export_to_xml(self):
        table = {
            "sexe": {
                "type": "SEXE",
                "remplissage": 100
            },
            "age": {
                "type": "18|19|20",
                "remplissage": 100
            },
            "prenom": {
                "type": "NAME",
                "remplissage": 88.4
            },
            "nom": {
                "type": "NAME|'dupont'",
                "remplissage": 85
            }
        }

        # Vérifier si le fichier existe ou non
        if os.path.isfile("D:/Projet_Informatique_2A/Projet-info-2A/output.xml"):
            print("fichier trouvé")
            tablexml = Export_to_xml("D:/Projet_Informatique_2A/Projet-info-2A", "table_xml1")
            xmlfile = pd.read_xml("D:/Projet_Informatique_2A/Projet-info-2A/output.xml")
            table_xml1 = tablexml.export(json.dumps(table))
            table_xml = pd.read_xml("D:/Projet_Informatique_2A/Projet-info-2A/table_xml1")
            print(table_xml==xmlfile)
        else:
            print("Fichier non trouvé")

if __name__ == "__main__":
    unittest.main()
```

FIGURE 27 – Test de la classe export\_to\_xml

Le test de la classe export\_to\_xml fonctionne de la même façon.

```
Object: test_export / test_export_e
fichier trouvé
   index  type  remplissage
0   True  True          False
1   True  True           True
2   True  True           True
3   True  True           True
.
-----
Ran 1 test in 0.108s
OK
```

FIGURE 28 – Résultat du test de la classe

Là encore on observe que le test a bien fonctionné. Le fichier a bien été trouvé. Cependant, les éléments ne sont pas ceux voulus car pour élément d'identifiant 0, le taux de remplissage est différent dans les deux fichiers.

## 5 Difficultés rencontrées et pistes d'amélioration

Durant ce travail nous avons rencontré certaines difficultés :

- Le stockage des données en base de données s'est avéré difficile. Un "individu statistique" (une ligne de la table finale à générer) a été enregistré sous la forme de plusieurs lignes dans la table `donnee` : chaque ligne de cette table correspond donc seulement à une cellule de la table finale. La manipulation d'une ligne ou d'une colonne de la table finale générée s'est avérée délicate. Les méthodes de DAO mises en place pour effectuer différentes manipulations de lignes ou de colonnes (et non de cellules) ont été particulièrement difficiles à mettre en place.
- Du fait de la structure de nos données, nous avons également abandonné les liens entre les tables de la base de données : les arguments `nom_type` et `nom_meta_type` n'ont finalement pas été définis comme des clés étrangères et nous nous sommes contentés d'assurer l'unicité des types et des méta\_types via des méthodes de suppression des doublons et de messages d'erreurs en cas de redondances ( `nom_type` ou `nom_meta_type` déjà présents dans la base de données).
- Les tests sur les exportations des tables de données se sont révélés difficiles. Cela nécessite de vérifier que les tables ont bien été exportées et que les éléments (type et modalité) ont été bien renseignés. Pour cela, on a essayé de faire des `assertEqual` et `assertIs` : le code nous renvoyait des erreurs, car le code ne vérifiait pas le contenu des tables, mais leurs types.
- Lors de l'utilisation de l'api, nous avons eu des difficultés à utiliser nos fonctions **export** : ceci était assez surprenant car elle fonctionnait lorsque l'on utilisait directement sur Visual Studio Code. Le code nous renvoyait que l'objet de type `coroutine` n'avait pas d'attribut `export` or nous nous connaissions pas les objets de type `coroutine`. Le problème venait du fait que les noms des fonctions `export_to_json`, `export_to_csv`, `export_to_xml` de l'API étaient exactement les mêmes que pour nos classes. De ce fait au même titre qu'une fonction récursive la fonction s'appelait elle-même et donc nous avons mis une majuscule au début des noms de nos modules.

Compte tenu de l'état actuel de l'application, plusieurs aspects de l'application peuvent être améliorés :

- **Taille des données générées** : Il serait intéressant d'optimiser le code afin de pouvoir générer de plus grands volumes de données. En effet, certains tests nous font penser que l'api pourrait rencontrer des difficultés à générer plus de 50000 données, en comptant le nombre de lignes multiplié par le nombre de colonnes. Le temps de génération de ces données pourrait alors être de plusieurs minutes (ajustable en fonction de la capacité de la machine). Un autre problème lié aux grands volumes de données viendrait du stockage de ces données en mémoire vive. Elles pourraient causer des problèmes de mémoire vive à la machine avant même qu'elles soient persistées en base de données. Ce problème prend de plus en plus d'importance lorsqu'on lance plusieurs fois la génération de données consécutivement, sans fermer l'api, car la mémoire vive n'est alors pas vidée. Un moyen intuitif de corriger cela serait donc de fermer l'api puis de la relancer afin de libérer de la mémoire vive.
- **Requêtage des données** : Au niveau de la DAO (Data Access Object), il pourrait être intéressant de trouver un moyen de mieux requêter les données. Notre système stocke toutes

nos données générées dans une même table de la base de données. Dans cette table, chaque ligne correspond en fait à une cellule des données réelles générées. Ainsi, la taille de la table augmente exponentiellement à chaque génération de données ce qui rend le requêtage de cette table de plus en plus ardu. Une piste de résolution de ce problème serait de discrétiser la table en de plus petites tables de telle sorte que lorsqu'on requête la grande table on fasse implicitement plusieurs requêtes sur les petites tables.

- **Traitement de statistique descriptive** : Ajouter au niveau des modules de services clients des classes permettant d'effectuer des analyses descriptives (moyennes, écart-type, quantile, etc.) de la base de données créée.
- **Affichage tableau jeu de données** : Pour rendre le visuel des tableaux appréciables pour l'utilisateur, il serait bien de créer un module view qui permettrait à l'utilisateur d'avoir une représentation du tableau détaillé.
- **Documentation du code** : Pour faciliter la lecture de la documentation du code, il serait intéressant d'effectuer une documentation générée sur HTML à partir d'un logiciel comme doxygen qui permet de générer une documentation de code en HTML.

## Conclusion

Ce travail a été l’occasion de découvrir ou d’approfondir la création d’un webservice via la réalisation d’une application avec le module python fastapi. Dans ce rapport, nous nous sommes attachés à expliquer aussi bien comment le système de génération de données que nous avons mis en place fonctionne et quelles en sont ses limites. Le guide d’utilisation explique ainsi à l’utilisateur comment prendre en main cette application. Il peut définir des types de données et les modalités prises par chaque type de données, créer des types composés de données, appelés méta-types, et ainsi générer des données associées, en respectant des contraintes en termes de probabilité d’apparition, de taux de remplissage etc. Il peut stocker ces données dans une base de données, et les exporter dans différents formats (json, csv, xml). Il peut également faire appel à des fichiers json pour définir les contraintes sur les types et les modalités manipulées.

Un des intérêts principaux de ce projet a été de mener une réflexion collective sur la manière de répondre à une demande utilisateur, depuis le besoin exprimé jusqu’à la réalisation intégrale de l’application. La transversalité des outils utilisés, depuis les diagrammes de modélisation UML jusqu’au système de gestion de base de donnée distante, ont rendu ce projet très instructif dans une perspective professionnalisante : usage de moyens de communication adaptés, planification des tâches et répartition des rôles au sein de l’équipe, mais également travail sur le code, les tests, la documentation. Chacun a pu participer au projet selon ses compétences et ses aptitudes, tout en conservant une vision globale du travail réalisé. L’expérience collective a été agréable et, nous l’espérons, relativement productive.

Des pistes d’amélioration demeurent évidemment, apparues tout au long du projet mais non abordées soit par manque de temps soit par le niveau de difficulté soulevé. Les pistes principales d’amélioration concernent la complexité de calcul et le stockage des données.

Nous remercions notre tuteur Antoine Brunetti qui nous a guidés tout au long de ce projet. Il a su rester à notre écoute et nous a apporté de nombreux conseils utiles à la clarification et à la formalisation des objets que nous manipulions.

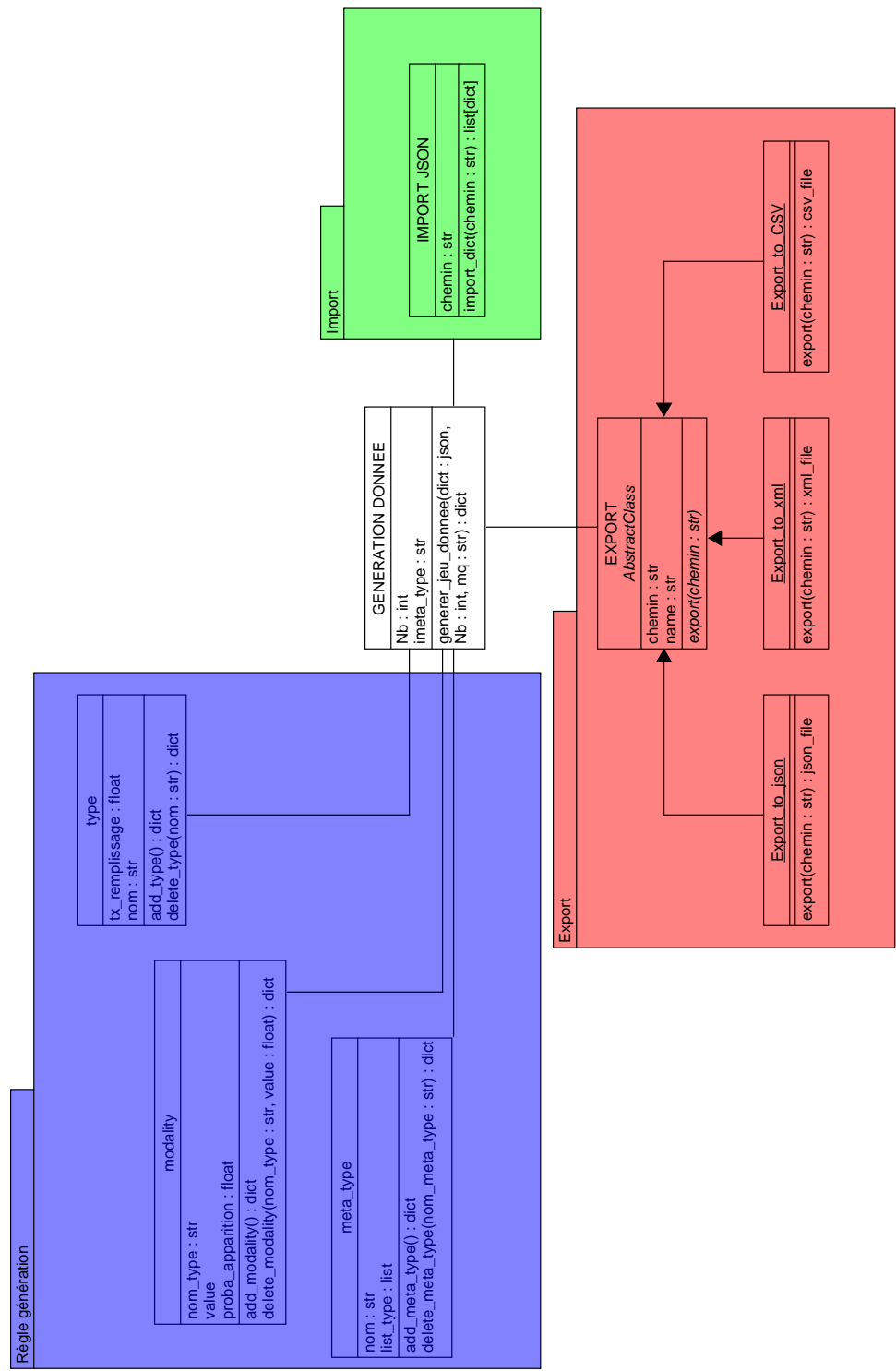


FIGURE 29 – Diagramme de classe

## Projet Informatique 2A : Note individuelle

Laurène Villacampa

**Sujet : création d'un webservice de génération de données**

**Tuteur : Antoine Brunetti**

### Première partie : l'entrée dans le projet

Le projet informatique me faisait un peu peur car je n'ai que peu de recul dans le domaine. Les exigences visées me paraissaient assez hautes en regard du nombre de cours et de TP réalisés. En outre, j'ai bénéficié pour le projet de 1A de l'aide d'un camarade interne qui avait déjà travaillé dans l'informatique et qui avait pris le leadership du projet, en faisant un soutien précieux. Suite à une mauvaise expérience en projet statistique, j'appréhendais également de devoir être leader sur le projet informatique alors que je ne me sentais pas armée pour cela.

#### Choix du sujet et répartition des rôles

Le choix du sujet s'est fait assez facilement et j'ai senti rapidement que l'expérience allait mieux se passer que prévu. J'ai eu quelques difficultés à entrer dans le sujet et à bien comprendre ce qui était attendu. Les mots « webservice », « endpoint » et autres me laissaient relativement perplexe. Mais nous avons bénéficié d'un soutien plein d'allant de la part de notre tuteur Antoine Brunetti et les rôles se sont définis très naturellement au sein du groupe, qui est apparu plutôt équilibré. Adrien a accepté de tenir le rôle du « chef de projet » (et a fourni un gros travail !). Abdoul et Isaac ont montré de sérieuses compétences en informatiques et en modélisation, ce qui a permis d'entrer assez vite dans le vif du sujet. Nathan et moi-même avions peut-être moins de connaissances initiales mais nous sommes entrés dans le sujet assez rapidement et avons bien participé il me semble tout du long.

#### Difficultés rencontrées

A titre personnel, j'ai été un peu en difficulté au début car les garçons, plus à l'aise, avaient tendance à travailler un peu « chacun dans leur coin », mais une fois que je leur ai dit que j'avais besoin d'être un peu drivée au début, ils ont très bien réagi et la suite du projet s'est très bien passée.

### Deuxième partie : le déroulement du projet

#### Modélisation

La partie modélisation a été réfléchi collectivement, puis chacun a réalisé un type de diagramme particulier. J'ai hérité du plus simple il me semble, à savoir le diagramme de cas d'utilisation. Nous pouvons remercier Adrien d'avoir « déblayé le terrain » des classes d'objets que nous allions manipuler pendant tout le projet : j'ai pu participer à la réflexion sur les classes à mettre en œuvre grâce à son travail initial, et le soutien du tuteur a été précieux également en la matière.



## **Ecriture de code**

Au cours du TP à propos de la DAO, j'ai senti une appétence personnelle pour ce domaine. J'avais envie de comprendre comment fonctionnait la liaison entre du code Python et une base de données. J'ai donc naturellement proposé de gérer la DAO, et j'ai peu touché à la partie métier du code. Grâce au document fourni par Rémi Pépin sur Moodle et à l'aide de Noël Debarles, j'ai réussi à installer sur mon ordinateur personnel un serveur local et une base de données PostgreSQL15. J'ai fait ce choix car j'utilise très peu la VM pour travailler depuis chez moi. Isaac est assez rapidement venu en renfort pour la DAO liée aux méta\_types et aux jeux de données. Nous avons travaillé en collaboration étroite pour régler tous les problèmes rencontrés et je le remercie grandement pour son aide et sa bienveillance, c'était très agréable de travailler avec son support. Il n'a malheureusement pas réussi à installer de serveur sur son propre ordinateur donc j'étais en charge des tests sur la base de données, ce qui s'est révélé un peu lourd sur la fin du projet

## **Rédaction du rapport**

Tout le monde a participé à la rédaction du rapport, en lien avec sa réalisation du code. Etant un peu plus à l'aise sur l'aspect rédactionnel, j'ai fait un travail assez conséquent de relecture et de corrections orthographiques.

## Troisième partie : Bilan

Je suis globalement très satisfaite de l'expérience de ce projet informatique. Tout le monde a joué le jeu et c'était très agréable de travailler dans ces conditions : les garçons s'entendent très bien entre eux et ont été à l'écoute de ce que je leur disais, ou des questions que je leur posais.

J'ai regretté qu'un TP ne nous ait pas permis de disposer tous d'un serveur local sur nos ordinateurs personnels, l'identification ayant posé problème à Isaac comme à moi-même. Plus globalement, j'aurais aimé pouvoir aller un peu plus loin car il me semble que de nombreuses fonctionnalités auraient pu être réfléchies relativement à ce webservice, mais à titre personnel j'ai déjà mis pas mal de temps à bien entrer dans le projet et à en comprendre les tenants et les aboutissants donc cela n'était pas très réaliste au vu du temps imparti.

La rédaction du rapport s'est bien passée et nous avons su éviter le « stress de la dernière minute », ce qui a rendu l'expérience relativement agréable !

Un bilan donc très positif et je remercie sincèrement mes quatre camarades pour leur entrain et leur maturité, ainsi qu'Antoine notre tuteur pour sa disponibilité.

## Projet informatique 2A : Note individuelle

Adrien Cortada

**Sujet : création d'un webservice de génération de données**

**Tuteur : Antoine Brunetti**

### **1. Entrée dans le projet:**

Mon groupe de projet informatique est composé de 5 personnes, Sandja Nathan, Villacampa Laurène, Toure Abdoul-Aziz, Ganiyu Isaac ainsi que moi même. Avant même le début du projet vers fin août j'ai créé un de discussion afin de nous permettre de communiquer et choisir un sujet. Nous avons finalement obtenu le sujet Webservice et génération de données.

### **2. Répartition des rôles:**

Au départ nous avons passé un moment tous ensemble pour bien comprendre le sujet, afin de bien tous se mettre d'accord sur le travail à réaliser. Après plusieurs séances et de multiples aller-retours avec notre encadrant Antoine Brunetti, nous avons enfin une vision claire du sujet. Ainsi pour le dossier d'analyse nous nous sommes réparti les tâches. Je me suis occupé du diagramme de classe, Nathan du diagramme de package, Isaac du diagramme d'activité, Abdoul-Aziz de celui de base de données et enfin Laurène s'est chargé du diagramme de cas d'utilisation. Nous avons tous ensemble écrit les parties du rapport correspondant tout en relisant le travail de tous les autres membres du groupe afin d'être tous en adéquation sur le rendu. Ainsi la rédaction du dossier d'analyse s'est bien déroulée.

Une fois la vue d'ensemble réalisée grâce aux différents diagrammes nous sommes passés à l'implémentation de l'application. Encore une fois nous avons séparé le travail entre les membres du groupe. Je me suis occupé de l'implémentation des parties règles de génération ainsi que import et génération de données. Abdoul-Aziz m'a aidé sur la fonction génération de données même si Abdou-Aziz et Nathan se sont majoritairement occupés de la partie export local des données. Enfin Isaac et Laurène se sont quant à eux majoritairement occupés de la partie data access object.

### **3. Ressenti de l'expérience:**

L'implémentation de l'application s'est bien déroulée notamment parce que lors des 3 jours dédiés au projet informatique nous sommes tous venus en présentiel, cela nous a permis de travailler de manière plus efficace et de nous entraider lorsque nous rencontrions une erreur. Ainsi, en présentiel tous les 5 la communication était facile, nous n'avons pas rencontré de point de tension, c'est pourquoi j'ai bien vécu le fait de faire le projet informatique au sein de ce groupe.

Je me suis proposé pour être chef de projet car je pense avoir les compétences et le leadership pour mener à bien ce projet. Ainsi tout au long du projet j'ai œuvré pour organiser les différents rendez-vous, personnes n'a jamais rechigné à venir et les séances de travail se sont toutes bien passées et ont été enrichissantes. Je n'ai jamais eu à relancer un des membres de mon groupe sur un travail qu'il n'aurait pas fini, nous avons toujours terminé le travail sans pression et dans les temps. J'ai également assuré une bonne communication entre le tuteur et l'ensemble du groupe.

La majorité de cette expérience s'est ainsi déroulée sans problème, mais si j'avais un conseil à donner ce serait de bien penser à l'ensemble des fonctionnalités avant de commencer à coder. En effet nous avons décidé, alors que le code était déjà bien avancé, d'ajouter la possibilité de générer des données selon une loi continue. Il a fallu ainsi trouver un moyen d'ajouter cette possibilité en changeant le code au minimum car tout fonctionnait presque déjà, l'insertion de cette fonctionnalité fut la chose la plus délicate que j'ai effectué lors de ce projet. Ainsi il vaut mieux réfléchir en amont à toutes les fonctionnalités.

## **PROJET INFORMATIQUE 2A: Note Individuelle de Toure Abdoul-Aziz**

**Sujet : création d'un webservice de génération de données**

**Tuteur : Antoine Brunetti**

**Membres d'Équipe : Toure Abdoul-Aziz, Sandja Nathan, Cortada Adrien,  
Villacampa Laurène, Ganiyu Isaac**

Tout d'abord, ce projet fut l'occasion de nous placer dans un cadre réel et professionnel avec des exigences et des attentes d'un tuteur. Nous avons donc été confrontés à une demande précise, établie par un cahier des charges. Afin de réaliser le meilleur travail possible, il a été important d'adopter une méthode de travail rigoureuse et organisée. Pour cela, nous avons beaucoup discuté pour que tous les membres du groupe aient la même vision du projet.

### **Le choix du sujet et répartition de travail**

Le choix du sujet a été effectué de manière à ce que tous les membres du groupe soient d'accord. On a tous effectué des choix de sujet qu'on mènerait à traiter. Ainsi, nous avons croisé nos choix et le thème qui est ressorti, est le webservice génération de base de données.

Pour s'imprégner du sujet, nous devons effectuer des diagrammes UML pour avoir des bases pour mener à bien ce projet. Pour cela, on a réparti les différentes tâches à réaliser, c'est-à-dire les différents diagrammes à effectuer :

- Le diagramme de cas d'utilisation a été effectué par Laurène
- Le diagramme d'activité a été réalisé par Isaac
- Le diagramme des classes réalisé par Adrien
- Le diagramme de packages construite par Nathan
- Le diagramme de physique de base de données créé par Abdoul

Ces diagrammes nous ont permis de faire un dossier d'analyse à notre tuteur Antoine Brunetti.

Après la réalisation des différents diagrammes qui ont été un support pour que notre code soit bien structuré. Par la suite, pour effectuer un début de code, la répartition a été comme suite, la partie DAO (Data Access Object) a été réalisée par Laurène & Isaac, puis la partie Business Object par Adrien & Abdoul & Nathan.

La rédaction du rapport a été faite par rapport au travail réalisé. Et chaque membre de l'équipe a effectué une relecture du rapport pour corriger des différentes coquilles présentes dans le rapport.

### **Tâches réalisé durant l'accomplissement du projet informatique**

Pour ma part, je me suis occupé d'abord à la réalisation du diagramme de base de données pour donner un aperçu de notre base de données à la fin de notre projet ce qui a permis à Isaac & Laurène de réaliser la DAO. Ensuite, j'ai apporté mon aide à Adrien pour construire le code sur la génération de bases de données de manière aléatoire. Après avoir généré les bases de données, nous devons permettre l'exportation de ces bases de données, pour cela moi & Nathan avons codés des classes pour permettre les exportations sous différents formats comme JSON, XML, et CSV.

Après la création de ces différentes classes, on devait tester les différentes classes pour s'apercevoir de leur bon fonctionnement.

### **Ressenti sur l'ensemble du projet**

Le projet informatique de cette année était plus complexe et riche que celui de l'année dernière. Ce projet a été un moyen pour chaque membre du groupe de progresser dans sa compréhension des différents diagrammes UML et aussi dans notre manière de coder. Usage de GIT nous a permis de coder en simultané ce qui nous fait progresser plus vite. Ce fut un plaisir pour moi de travailler avec cette équipe durant tout le long du projet, car on était tous impliqués quant à la réalisation de ce projet. Avec un chef d'équipe (Adrien), qui motivait l'équipe, et un tuteur (Antoine Brunetti) présent pour permettre et suivre l'avancée de notre projet.

## PROJET INFORMATIQUE 2A : NOTE INDIVIDUELLE

Membre : GANIYU Isaac

Sujet : Création d'un webservice de génération de données

Tuteur : Antoine Brunetti

### **1) Entame du projet**

Une fois avoir pris connaissance de la répartition des groupes du projet informatique, nous sommes très vite rentrés en contact. Nous avons discuté et avons collectivement fait nos choix de sujets. Fort heureusement, il nous a été attribué notre premier choix de sujet. J'ai été bien intégré au groupe dès le début du projet bien que je sois un admis sur titre (Et donc je ne connaissais personne dans mon groupe avant de les rencontrer). Je remercie donc les membres de mon groupe pour cela.

### **2) Déroulement du projet**

Tout d'abord, Adrien qu'on avait choisi comme « chef de groupe » nous a montré l'exemple et nous a permis d'avancer en nous présentant très tôt un diagramme de classe. Nous en avons profité pour distribuer les tâches concernant la modélisation UML du projet. Je me suis occupé du diagramme d'activité. Ensuite, Adrien nous a encore permis de faire un bond en avant en nous présentant une API qui fonctionnait avant même qu'on ait fini la modélisation UML. Je m'en suis donc inspiré pour concevoir le diagramme d'activité puis je l'ai affiné avec l'aide de notre encadreur Antoine Brunetti. Chacun de nous a ensuite contribué à la rédaction du rapport de mi-parcours en décrivant son diagramme et en relisant le rapport obtenu. Laurène a été d'une grande aide en tant que coordonnatrice en ce qui concerne le rapport. Nous nous sommes par la suite tous mis au code. Le partage des tâches m'a conduit au codage de la classe « import ». J'ai pu rapidement terminer le code de cette classe. Je suis donc parti en soutien de Nathan pour le code de la classe « export ». Notre duo été très efficace grâce à une bonne entente et une bonne coopération. Ainsi, nous avons rapidement fourni un code fonctionnel. J'étais donc encore une fois libre. J'ai donc formé un duo de travail avec Laurène qui travaillait sur la persistance des données et la DAO. Pour moi, cette dernière tâche a été largement plus compliquée que les tâches précédentes que j'ai effectuées. Tout le reste de mon temps a donc été consacré à la mise en place de la DAO. Je devais aussi constamment communiquer avec le duo constitué de Adrien et de Abdoul-Aziz qui eux s'occupaient des objets métiers et de la classe de génération de données. En effet, les classes DAO étant inspirées des objets métiers, je devais être informé de tous les détails du code des objets métiers et du code de génération de données. Abdoul-Aziz m'a beaucoup aidé dans ce sens.

Une fois le code de base de la DAO en place, la période d'essai de la DAO a débuté. Je travaillais étroitement avec Laurène. Je ne disposais pas de la base de données pour effectuer les essais, elle me faisait donc parvenir les différents problèmes qu'elle soulevait. Pour moi, cette période a été la plus difficile. Malgré le soutien de Laurène, j'ai passé plusieurs dizaines d'heures à débbugger, compléter, modifier ou affiner du code SQL et/ou Python sans même avoir de base de données pour tester directement moi-même.

Entre temps toutes les pièces du puzzle se mettaient en place pour notre API. Chacun de nous a donné le meilleur de lui-même pour mettre en place cette API, fruit de nos trois mois d'efforts. Et je suis particulièrement fier de notre travail vu le temps imparti.

Après la phase « code ». Nous avons eu environ 5 jours pour la rédaction du rapport en se basant bien sûr sur le rapport à mi-parcours. Personnellement, je préfère coder que rédiger le rapport mais ça s'est encore une fois bien passé. Avec la même formule, chacun de nous a contribué à la rédaction du rapport final.

### **3) Ressenti final**

Pour ma part, je tiens à remercier nos encadreurs Rémi Pepin et Antoine Brunetti pour leur encadrement. Je tiens également à remercier les membres de mon groupe pour leur investissement et leur sérieux.

Mon but dans ce projet était vraiment d'apprendre. Je pense avoir atteint cet objectif, dans la mesure où j'ai travaillé sur pratiquement chaque aspect du projet de l'import à la DAO en passant par l'export et la génération des données.

Ce projet a également été une bonne expérience dans le sens où notre groupe était très convivial, on s'entendait bien (et on continuera à bien s'entendre c'est sûr) et on a beaucoup rigolé (notamment beaucoup de fous rires sur le problème de coroutine). Cela a beaucoup facilité le travail je pense.

## Projet informatique 2A : Note individuelle

Nathan Sandja

Tuteur : Antoine Brunetti

Le projet de cette année m'a paru bien plus complexe que celui de l'année précédente mais beaucoup plus réalisable étant donné que le temps accordé était beaucoup plus important. L'avantage de ce projet était de mieux comprendre les notions qui nous ont été enseignés en cours et en TP car nous sommes retournés assez souvent dessus lorsque nous rencontrons une difficulté.

Participation effective au projet :

Ne pensant pas avoir les capacités j'ai préféré ne pas me proposer pour être désigné chef du groupe, de fait Adrien a été désigné (par vote à majorité). Au début du projet, nous nous sommes naturellement réparti les tâches selon nos capacités et notre motivation, le but étant de ne pas prendre une tâche si on ne se sentait pas à la hauteur. J'ai donc choisi de m'occuper du diagramme de paquetage. Lors de la remise du rapport intermédiaire j'étais chargé de la rédaction en rapport avec ce diagramme. Au niveau de l'organisation j'ai particulièrement bien aimé la nôtre, c'est-à-dire que chacun pouvait se concentrer de son côté tout en pouvant demander l'aide d'une personne en cas de difficulté. Chaque problème rencontré était signalé très vite, ceci était essentiel en particulier pour moi car je ne m'étais pas assez approprié du sujet au début. Par exemple, lorsque nous avons établi notre diagramme de classes, nous avons réparti les tâches et ainsi Isaac a pu m'aider pour les fonctions export lorsqu'il avait fini avec la fonction import. De même lorsqu'il est parti aider Laurène avec la DAO.

Conclusion :

D'un point de vue global j'ai bien apprécié le projet, ceci m'a permis de développer mes compétences en informatique et surtout à confronter des problèmes. La période d'immersion totale en est un très bon exemple car j'ai bloqué sur une erreur de mon code pendant plusieurs heures mais les efforts que j'ai fournis m'ont permis de mieux comprendre le problème de mon code. En revanche, les notions des diagrammes ne m'étaient pas assez claires au début du projet, notamment le diagramme de paquetage à faire lors du rendu du rapport final où j'ai dû me rendre compte assez tardivement que ceci avait un lien direct avec la programmation en couches.



