# Diagramme de classe « explication »

Le système de génération de datasets est conçu comme une architecture modulaire en 5 couches qui respecte rigoureusement les principes SOLID. L'objectif est de permettre à un utilisateur de définir formellement un projet de données, de générer automatiquement des datasets réalistes selon des contraintes paramétrables, puis de les exporter dans différents formats. L'ensemble est pensé pour être extensible sans modification du code existant, ce qui constitue le cœur de la philosophie Open/Closed.

Le cœur du système : la couche domaine

Au centre de l'architecture se trouve la classe Project, qui représente le point d'entrée principal du système. Un projet contient un identifiant, un nom, une description, et surtout une liste d'Entity (entités). C'est le Project qui orchestre la génération complète du dataset via sa méthode generateDataset(). Cette classe incarne le principe de Single Responsibility en se concentrant uniquement sur la gestion d'un projet de dataset dans son ensemble.

Chaque Entity représente une table ou une entité métier, comme "Utilisateur", "Produit" ou "Voiture". Une entité possède ses propres attributs (colonnes), peut contenir des sous-entités (par exemple, une entité "Adresse" imbriquée dans "Utilisateur"), et définit la quantité de lignes à générer. La relation entre Project et Entity est une composition forte (losange noir en UML), car les entités n'existent que dans le contexte d'un projet. L'Entity possède également une relation d'agrégation avec elle-même pour gérer les sous-entités, permettant ainsi de créer des structures hiérarchiques complexes.

Chaque Attribute définit un champ de l'entité avec son nom, son type de données (via l'enum DataType : STRING, INTEGER, FLOAT, BOOLEAN, DATE, ENUM) et ses contraintes. L'Attribute est lié par composition à la classe Constraints, qui encapsule toutes les règles de génération : valeurs minimales et maximales, moyenne, médiane, liste de valeurs possibles pour les énumérations, et surtout le type de Distribution (UNIFORM, NORMAL ou CUSTOM). Cette séparation entre Attribute et Constraints respecte le principe de Single Responsibility, car l'un gère la définition structurelle tandis que l'autre gère les règles métier.

Le résultat de la génération est stocké dans un Dataset, qui est essentiellement un conteneur organisant les données générées par entité. Il contient une map associant chaque nom d'entité à une liste de Record (enregistrements). Un Record représente une ligne de données sous forme de map clé-valeur, où chaque clé est le nom d'un champ et chaque valeur est la donnée générée. Cette structure permet une grande flexibilité dans la manipulation des données avant l'export.

La génération de données : le pattern Strategy

Pour générer les valeurs des attributs, le système utilise le pattern Strategy via l'interface IDataGenerator. Cette interface définit un contrat unique : generate(attribute: Attribute) qui retourne un objet. C'est ici que les principes Open/Closed et Dependency Inversion brillent particulièrement. L'Entity ne dépend pas d'une implémentation concrète de générateur, mais de l'interface IDataGenerator. Cela signifie qu'on peut ajouter autant de générateurs que nécessaire sans jamais toucher au code de Entity.

Trois implémentations principales sont proposées. Le RandomDataGenerator effectue une génération aléatoire simple en respectant les bornes définies dans les contraintes. Par exemple, pour un attribut "age" avec min=18 et max=65, il génère simplement un nombre aléatoire dans cet intervalle. Le DistributionDataGenerator va plus loin en appliquant des lois de probabilité. Si on définit un salaire avec une distribution normale (moyenne=3000, écart-type=500), environ 68% des valeurs générées se situeront entre 2500 et 3500 euros, créant ainsi un dataset plus réaliste statistiquement.

Le ExternalSourceGenerator est le plus sophistiqué : il récupère des données réalistes depuis des sources externes. Au lieu de générer aléatoirement "XYZ" comme prénom, il appelle une API ou lit un fichier pour obtenir de vrais prénoms français comme "Marie", "Jean" ou "Sophie". Ce générateur dépend lui-même d'une abstraction : l'interface IDataSource.

Les sources de données externes

L'interface IDataSource définit le contrat pour récupérer des données externes via une méthode fetchData(type: String) qui retourne une liste d'objets. Cette abstraction permet de créer différentes implémentations sans impacter les générateurs qui les utilisent. Le NameApiSource appelle une API REST pour récupérer des noms et prénoms réalistes, potentiellement filtrés par origine géographique (français, anglais, etc.). Le CityDataSource lit depuis un fichier local (CSV ou JSON) une liste de villes.

Cette architecture à deux niveaux d'abstraction (IDataGenerator → IDataSource) illustre parfaitement le principe de Dependency Inversion : les modules de haut niveau (ExternalSourceGenerator) ne dépendent pas des modules de bas niveau (NameApiSource), mais tous deux dépendent d'abstractions (IDataSource). On pourrait facilement ajouter un ProfessionDataSource, un ProductDataSource ou même un WikidataSource sans modifier une seule ligne de code existante.

L'export des données : encore Strategy

Une fois le dataset généré, il faut l'exporter dans différents formats. Ici encore, le pattern Strategy est appliqué via l'interface IDataExporter qui définit la méthode export(dataset: Dataset, outputPath: String). Le Dataset possède une méthode export(exporter: IDataExporter) qui délègue l'export à l'implémentation concrète fournie. C'est un exemple parfait du principe Open/Closed : le Dataset est fermé à la modification (on ne

le change jamais) mais ouvert à l'extension (on peut ajouter des formats d'export indéfiniment).

Quatre exporteurs sont implémentés. Le CsvExporter génère des fichiers CSV classiques avec des valeurs séparées par virgules et une ligne d'en-tête. Le JsonExporter produit un tableau JSON où chaque objet représente un enregistrement. Le XmlExporter crée une structure XML hiérarchique avec des balises pour chaque champ. Le SqlExporter génère des instructions INSERT SQL directement utilisables dans une base de données.

Le principe de Liskov Substitution est pleinement respecté : n'importe quel IDataExporter peut être utilisé de manière interchangeable. Le code client (Dataset ou DatasetService) ne sait pas et ne se soucie pas de savoir s'il utilise un CsvExporter ou un JsonExporter, il manipule uniquement l'interface. Si demain on veut ajouter un ParquetExporter ou un AvroExporter, on crée simplement une nouvelle classe implémentant IDataExporter, sans toucher au reste du système.

Les factories : création d'objets découplée

Pour créer les bons générateurs et exporteurs sans créer de dépendances directes, deux factories sont introduites. La DataGeneratorFactory possède une méthode createGenerator(type: String) qui retourne l'IDataGenerator approprié. Si on passe "random", elle instancie un RandomDataGenerator ; si on passe "normal", elle crée un DistributionDataGenerator avec distribution normale ; si on passe "external", elle instancie un ExternalSourceGenerator. Cette factory encapsule la logique de création et peut être facilement étendue.

De même, l'ExporterFactory crée les exporteurs via createExporter(format: String). Passer "csv" retourne un CsvExporter, "json" retourne un JsonExporter, et ainsi de suite. Ces factories respectent le principe de Single Responsibility : elles ont une seule raison de changer (l'ajout de nouveaux types de générateurs ou exporteurs), et isolent complètement la logique de création de la logique métier.

Le service : orchestration de haut niveau

Au sommet de l'architecture se trouve le DatasetService, qui agit comme une façade orchestrant l'ensemble du système. Cette classe possède une DataGeneratorFactory et une ExporterFactory en attributs, et expose trois méthodes publiques : createProject() pour initialiser un nouveau projet, generateDataset(project: Project) pour lancer la génération complète des données, et exportDataset(dataset: Dataset, format: String) pour exporter le résultat.

Le DatasetService incarne le principe de Dependency Inversion en dépendant des factories (abstractions) plutôt que des implémentations concrètes. Lorsqu'un utilisateur appelle exportDataset(dataset, "csv"), le service utilise l'ExporterFactory pour créer le

bon exporteur, puis délègue l'export à cet exporteur. Le service ne connaît jamais directement les classes CsvExporter, JsonExporter, etc.