Compte rendu du projet réseau

I – Introduction :

Dans le cadre de ce projet, nous avons mis l'accent sur l'exploitation de l'aspect orienté objet de Python, en structurant l'application de manière à réduire la complexité et à faciliter la compréhension des transferts de données parfois complexes. Notre objectif était de limiter le nombre de lignes par fonction et de créer une couche d'abstraction claire et compréhensible.

Pour atteindre cet objectif, nous avons largement utilisé des concepts tels que les classes abstraites, grâce au module natif ABC, l'héritage, l'encapsulation et la généricité. Ces approches nous ont permis de développer une architecture solide et cohérente pour le projet.

Ce document présente l'architecture du projet et rend compte des choix de conception et des décisions prises pour assurer la clarté, la modularité et l'efficacité de l'application.

II – Architecture :

Le projet se structure en deux grandes parties :

1. Les tests unitaires
2. Le code en lui-même

Les tests unitaires représentent une étape essentielle dans le processus de développement d'applications, en particulier lorsqu'il s'agit de programmation orientée objet. Conscients de leur importance, nous avons consacré un temps considérable à la réalisation de ces tests.

1. Les tests unitaires :

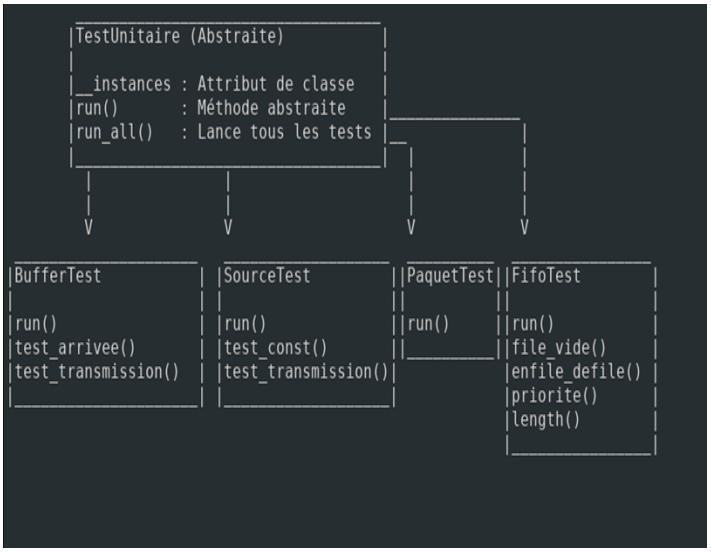

Diagramme de l'architecture des tests unitaires

Dans notre projet, toutes les classes représentant les tests unitaires héritent d'une classe de base appelée **« `TestUnitaire` »**. Cette classe implémente une méthode abstraite **« run() »**. Dans chaque fichier de définition de test, nous avons veillé à instancier un objet du test correspondant, ce qui permet d'exécuter le test en question.

Lorsque la méthode **« `TestUnitaire.run_all()` »** est appelée dans le fichier run_tests.py, tous les tests importés sont lancés simultanément. À l'issue de l'exécution des tests, le programme affiche les informations nécessaires pour déterminer si tous les tests ont été effectués avec succès ou si certains ont échoué.

En cas d'échec d'un test, son nom est affiché à l'écran grâce à la méthode **« `__repr__()` »**, qui est surchargée dans chaque classe héritant de **« `TestUnitaire` »**. Cette approche nous permet de repérer rapidement les tests ayant échoué et de faciliter le processus de débogage.

2. Le programme principal :

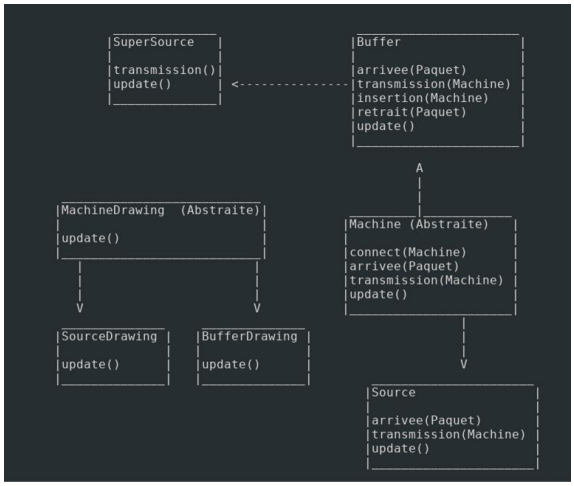

Diagramme de l'arborescence des classes principales

Pour commencer, le programme principal est structuré autour de deux arborescences de classes principales : les classes **« `Machine` »** et les classes **« `MachineDrawing` »**. Comme illustré dans le diagramme précédent, les différences entre les classes **« `Buffer` »** et **« `Source` »** résident principalement dans la manière dont elles redéfinissent les méthodes **« `arrivee()` »**, **« `transmission()` »** et **« `update()` »**.

Un élément notable est que la classe **« `SuperSource` »** ne dérive pas de la classe **« `Source` »**, mais de la classe **« `Buffer` »**. Cette particularité est due au fait qu'une **« `SuperSource` »** imite le comportement d'un **« `Buffer` »** en ce qui concerne le stockage des paquets, en utilisant une file d'attente.

Quant aux classes dérivant de **« `MachineDrawing` »**, la seule méthode redéfinie est la méthode **« `update()` »**. Cette dernière est responsable de la mise à jour des couleurs d'affichage des machines. Il est important de noter qu'il n'existe pas de classe **« `SuperSourceDrawing` »**. En effet, puisque la classe **« `SuperSource` »** dérive de la classe **« `Buffer` »**, il est approprié d'utiliser un **« `BufferDrawing` »** pour dessiner les **« `SuperSource` »**.

De plus, les méthodes **« `update()` »** sont appelées périodiquement et ont pour rôle de mettre à jour le contenu des objets auxquels elles se rapportent. Le fonctionnement fondamental du programme repose principalement sur ces méthodes **« `update()` »**. Elles sont responsables de la prise de décision concernant la transmission d'un paquet, le retrait d'un paquet de la file d'attente dans le cas des **« `Buffer` »** et des **« `SuperSource` »**, ainsi que la gestion du comportement des différentes stratégies de routage pour les **« `SuperSource` »**. En somme, les méthodes **« `update()` »** sont au cœur du fonctionnement du programme, assurant la mise à jour et la coordination des différents éléments du système.

Enfin, dans ce projet, la densité de génération des paquets, notée lambda dans le sujet, est représentée par la variable de classe **« `Source.generation_probability` »**. Cette variable régule le nombre de paquets produits par les sources et les **« `SuperSource` »**.

Petite parenthèse :

Les classes suivantes ne figurent pas sur le diagramme pour des raisons de simplicité, mais, voilà leur utilité :

→ La classe **« `Paquet` »** représente de manière simple un paquet d'information. Elle contient un identifiant en trois lettres majuscules, à partir duquel on peut déterminer une couleur associée au paquet, et le nécessaire pour calculer le temps de vie.

→ La classe **« `Fifo` »** représente une file d'attente, elle est principalement utilisée par les **« `Buffer` »**. Elle comprend les méthodes **« `defiler()` »** et **« `enfiler()` »** ainsi que quelques autres détails.

→ La classe **« `Tab` »** représente un onglet de l'application. Elle est dotée d'une liste de widgets Tkinter à afficher, ainsi que d'une "handle" qui est en réalité un bouton Tkinter. Ce bouton permet de montrer ou cacher les widgets de l'onglet en question, offrant ainsi une interface utilisateur organisée et facile à naviguer.

III – Le fichier « main.py » :

Le fichier "main.py" contient le code nécessaire pour exécuter le programme avec un réseau simple. Il comprend plusieurs fonctions clés, telles que :

- **`update_model()`** : fonction qui appelle les méthodes **« `update()` »** de toutes les instances de classes dérivant de **« `Machine` »** dans le réseau.
- **`update_drawings()`** : fonction qui appelle les méthodes **« `update()` »** de tous les objets dérivant de **« `MachineDrawing` »**.
- **`update_stats()`** : fonction qui met à jour les widgets chargés d'afficher les statistiques pertinentes concernant le fonctionnement du réseau.

En outre, vous pouvez trouver quelques autres fonctions courtes, dont le fonctionnement est relativement simple et facile à comprendre.

De plus, sous la ligne **« `if __name__ == '__main__':` »**, vous trouverez tout le code nécessaire à la création des onglets de l'interface, à l'initialisation du réseau et, enfin, aux appels initiaux des fonctions **« `update_model(), update_drawings()` »** et **« `update_stats()` »**.

IV – Conclusion :

En conclusion, ce projet a permis d'explorer et de mettre en pratique divers concepts de programmation orientée objet, tels que l'héritage, l'encapsulation et la généricité, en utilisant le langage Python. Grâce à une architecture bien structurée et une approche modulaire, nous avons développé une application simulant un réseau de transmission de données, comprenant des sources, des buffers et des stratégies de routage. L'interface graphique et les statistiques fournissent une expérience utilisateur agréable et des informations utiles sur le fonctionnement du réseau. Ce projet constitue une expérience enrichissante dans la conception et le développement d'applications complexes.