Licence Informatique

La Rochelle Université

GALACTIC - Framework

Yannis Rouhier

2023

Table des matières

Liste des figures

Abstract

Résumé en français

Durant mon stage de fin de Licence Informatique à La Rochelle Université, j'ai eu l'occasion de faire un stage au sein du laboratoire de recherche Laboratoire Informatique, Image et Interaction (L3i). J'ai rejoint l'équipe du projet **GALACTIC** à La Rochelle Université du 24 avril au 29 juin 2023 afin de contribuer à son projet de *framework*.

Le rapport de stage expose les différentes étapes que j'ai parcourues afin de concevoir une interface en ligne de commande (CLI) permettant de créer des modèles d'extensions pour la plateforme **GALACTIC**, ainsi que l'explication du fonctionnement de l'interface en ligne de commande du *framework* **GALACTIC**.

Il comprend également les outils qui ont été employés pour le développement optimal du projet *framework* **GALACTIC**. Grâce aux travaux d'un précédent stagiaire, qui s'est penché sur le sujet en 2022, une évaluation des outils les plus appropriés pour le projet a été réalisée. Les outils utilisés sont principalement `poetry` pour la gestion du projet, `cleo` pour la création d'interfaces en ligne de commande, et `Jinja2` comme moteur de génération de modèles/squelettes (Templates).

Je suis reconnaissant envers Monsieur Yacine Ghamri-Doudane, directeur du laboratoire, de m'avoir accepté au sein du L3i pour mon stage.

J'aimerais également exprimer ma gratitude envers mon maitre de stage, Monsieur Christophe Demko, enseignant-chercheur à l'Université de La Rochelle, pour son accompagnement tout au long du projet, ainsi que pour sa transmission de connaissances en gestion de projet de développement informatique. Je le remercie également pour les compétences pointues que j'ai acquises en *Python*.

Veuillez noter que ce rapport est incomplet en raison des dates de soutenances prévues avant la fin du stage, ce qui signifie qu'une partie du travail n'a pas encore été réalisée.

Abstract in English

During my final internship for my Bachelor's degree in Computer Science at La Rochelle University, I had the opportunity to intern at the research laboratory called Laboratory of Computer Science, Image, and Interaction (L3i). I joined the team of the **GALACTIC** project at La Rochelle University from April 24th to June 29th, 2023, to contribute to their *framework* project.

The internship report presents the various steps I went through to design a command-line interface (CLI) for creating extension models for the **GALACTIC** platform, as well as an explanation of the functionality of the **GALACTIC** framework's command-line interface.

It also includes the tools that were used for the optimal development of the **GALACTIC** *framework* project. Thanks to the work of a previous intern who worked on the subject in 2022, an evaluation of the most appropriate tools for the project was conducted. The tools used primarily include `poetry` for project management, `cleo` for creating command-line interfaces, and `Jinja2` as a template/skeleton generation engine.

I am grateful to Mr. Yacine Ghamri-Doudane, the director of the laboratory, for accepting me at L3i for my internship.

I would also like to express my gratitude to my internship supervisor, Mr. Christophe Demko, a teacher-researcher at the University of La Rochelle, for his support throughout the project, as well as for his knowledge transfer in computer software development project management. I also thank him for the advanced skills I have acquired in *Python*.

Please note that this report is incomplete due to the scheduled presentations before the end of the internship, which means that a portion of the work has not been completed yet.

1 L'entreprise

Le Laboratoire Informatique, Image et Interaction (L3i) a été crée en 1993, près de 100 personnes travaillent au sein du laboratoire. Il s'agit d'un laboratoire de recherche de Sciences du Numérique de l'Université de La Rochelle, il associe les chercheurs en Informatique de l'IUT ainsi que les chercheurs du Pôle Sciences de l'Université de La Rochelle. Il compte 33 chercheurs et chercheuses, 35 doctorants et doctorantes, 4 personnes permanentes de soutien à la recherchce et 25 personnes sur projets et est dirigé par Monsieur Yacine GHAMRI-DOUDANE.

Le laboratoire L3i fait parti de réseaux de recherches régionaux (Fédération CNRS MIRES, ERT " Interactivité Numérique "), nationaux (GDR I3 et GDR IRIS) et internationaux (IAPR) dans les secteurs de visibilité de son action scientifique, autour d'un projet stratégique lié à la gestion intelligente et interactive des contenus numériques. Cela est renforcé grâce à une politique de volontariste de participation ou de pilotages de projets de recherches labélisés (ANR, PCRD, …), au sein desquels le laboratoire occupe couramment une position de leadership.

Le Laboratoire est également membre de l'Alliance Big data lancée en 2013 pour favoriser le développement en France de nouveaux services et projets dans ce domaine.

Ses travaux sont menés en partenariat avec une dizaine de centres de recherches nationaux (dont l'INRIA, institut national de recherche en sciences et technologies du numérique). Le L3i entretient par ailleurs des liens privilégiés avec de nombreux centres de recherche à travers le monde. Il est également engagé dans près d'une vingtaine de partenariats industriels sur l'ensemble du territoire français.

Le laboratoire du L3i est divisé en 3 équipes :

- Modèles et Connaissances
- Images et Contenus
- Dynamique des systèmes et Adaptativité

Mon stage se déroule au sein de l'équipe Modèles et Connaissances et plus précisément autour du projet **GALACTIC** de cette équipe.

2 La plateforme GALACTIC

La présentation de la plateforme **GALACTIC** est issue du du diaporama de présentation de l'architecture de la plateforme (https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-

20min-complete.pdf) et il s'agit d'un travail fait en commun avec Éléonore THONNEAU, Kévin VOISIN, Alexandre BUFFARD, Martin EHLINGER et de Jules BRICOU.

GALACTIC est l'acronyme de **GA**lois **LA**ttices **C**oncept **T**heory **I**mplicational system and **C**losures. L'objectif de la plateforme est pouvoir mettre en place un ensemble d'outils permettant de travailler sur l'Analyse Formelle des Concepts ainsi que sur la Théorie des treillis. La plateforme **GALACTIC** est structurée de la façon suivante:

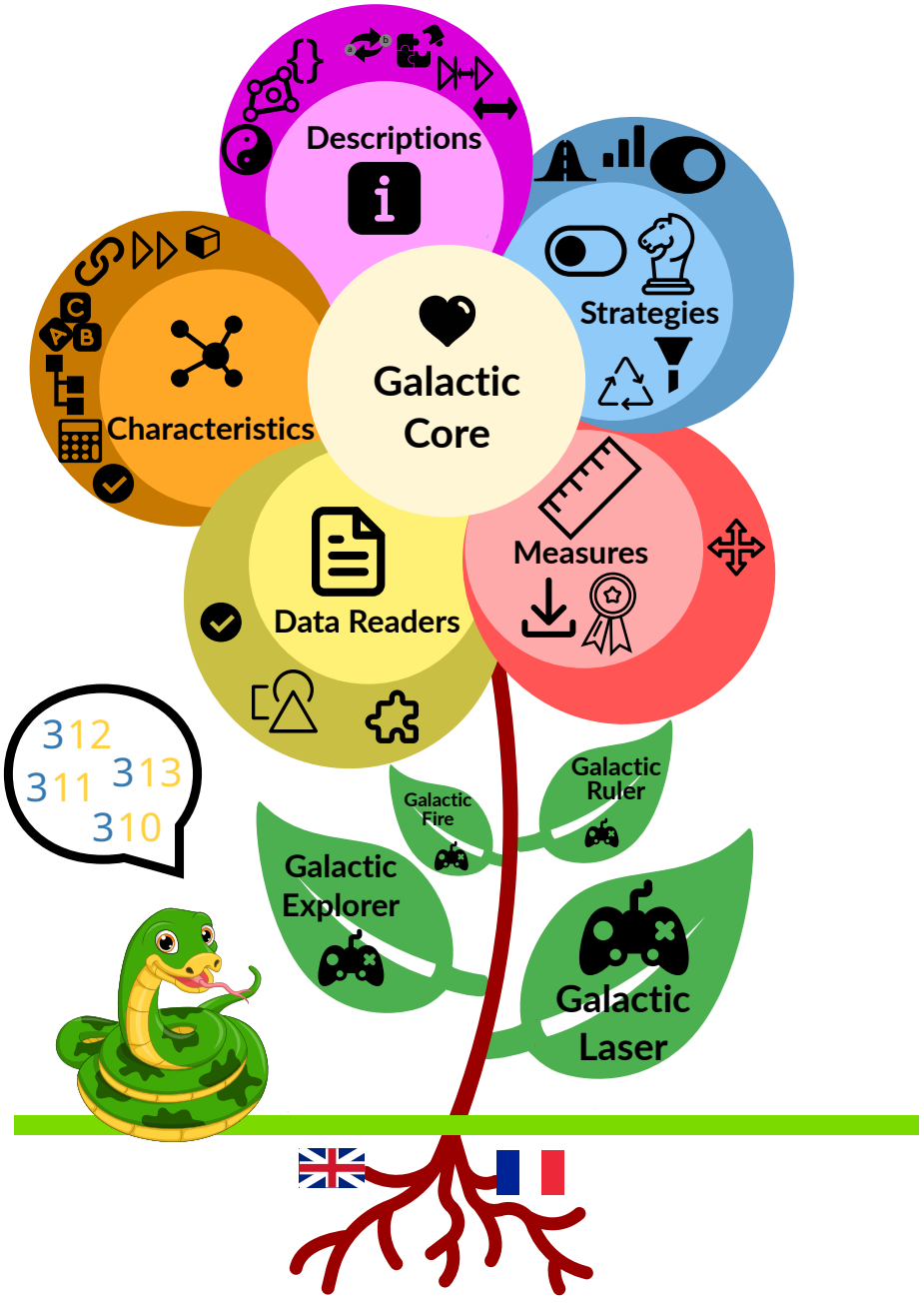

Figure 1: **GALACTIC** architecture

- Un noyau, représenté par le cœur de la fleur, qui contient les opérations de base et structures de données et qui implémente un nouvel algorithme (NextPriorityConcept) inspiré des pattern structures.

- Des *plugins*, représentés par les pétales de la fleur:
  - Caractéristiques : Booléen, Numérique, Catégorique, String, Séquentielle, Chain, Triadic. Ils permettent de définir les caractéristiques.
  - Descriptions : Booléen, Logique, Catégorique, String (regex), String (distance), Chain, Séquentielle, Séquentielle (distance), Triadic. Ils définissent les prédicats et les espaces de description utilisés pour représenter et définir les données avec précision.
  - Stratégies : Booléen, Logique, Catégorique, Numérique, String, String (distance), Chain, Séquentielle, Séquentielle (distance), Triadic. Ils définissent la manière utilisée pour explorer les données, il utilise des descriptions pour générer des prédécesseurs pour chaque concept dans le treillis.
  - Mesures : predecessor Cardinality, sucessor Cardinality, Confidence. Ce sont des paramètres des stratégies de filtrage prédéfinis dans la librairie core.
  - Data Readers : Ils sont utilisés pour lire différents types de fichiers de données. Le moteur de base détecte le type de fichier en utilisant son extension. Les *plugins* Data Readers existants sont: YAML, JSON, CSV, TOML, INI, TXT, SLF, DAT, CXT.
  - Localization: Ils sont utilisés pour traduire les applications dans d'autres langues.
- Des applications, représentées par les feuilles de la fleur, qui sont développées afin d'utiliser la librairie de la plateforme. Ce sont des interfaces pour l'utilisateur:
  - **GALACTIC** Laser : pour construire les treillis et explorer les données.
  - **GALACTIC** Explorer : pour explorer de façon interactive les treillis contruits.
  - **GALACTIC** Ruler : pour extraire les règles d'implication.
  - **GALACTIC** Fire : pour exécuter un système de règles.
- La bulle au dessus du serpent indique les différentes versions de *Python* avec lesquelles la plateforme **GALACTIC** est compatible.
- Au niveau des racines on peut voir les différentes langues disponibles sur la plateforme.

3 Le Stage

3.1 Le Sujet Du Stage

Mon stage s'est déroulé au sein du Laboratoire Informatique, Images et Interaction (L3i) de l'Université de La Rochelle. J'ai travaillé sur le projet de la plateforme **GALACTIC** présentée précédemment. Plusieurs *plugins* ont déjà été créés pour la plateforme **GALACTIC** afin de pouvoir travailler sur différents types de données tels que les données numériques, logiques, séquentielles, catégoriques ou encore de type `String`.

Le travail qui m'a été demandé consiste à développer une interface en ligne de commande en utilisant *Python*. Cette interface permettra de générer des modèles pour les extensions du *framework* **GALACTIC**.

Le projet sera divisé en deux parties distinctes. Dans la première partie, nous créerons l'application `framework GALACTIC` qui servira de point d'entrée pour l'utilisation des *plugins* qui pourront être développés pour enrichir l'application. Ensuite, dans la seconde partie, nous concevrons le *meta-plugin* `py-galactic-framework-characteristic` (characteristic). Ce *meta-plugin* servira de modèle pour créer de nouvelles caractéristiques.

Le sujet a été partiellement abordé en 2022 par un stagiaire de Licence Informatique, Jules Bricou, qui s'est penché sur les différentes comparaisons nécessaires pour mettre en place l'interface en ligne de commande du *framework* **GALACTIC**. J'ai pu rapidement m'approprier le sujet et commencer à travailler dessus. J'ai d'abord pris en main les outils individuellement, puis je les ai combinés pour concevoir les prémices du *framework*.

3.2 Conception de l'application GALACTIC *framework*

Avant de pouvoir débuter le travail sur l'application, il m'a été nécessaire de me familiariser avec les outils qui ont été choisis pour le projet. J'ai donc effectué des essais individuels pour me familiariser avec eux, puis lorsque j'ai bien pris en main les principaux outils, j'ai commencé à travailler sur le projet.

3.2.1 Présentation des outils

Ainsi, initialement, `setuptools` était utilisé comme gestionnaire de dépendances et de packages Python, mais il s'est avéré que `poetry` est un meilleur choix. Il offre de meilleures performances dans la gestion des versions de packages et des dépendances. De plus, `cleo` est un package utilisé pour créer une interface en ligne de commande (CLI) et permet également la création de commandes associées à des applications. La bibliothèque `importlib.metadata` est utilisée pour les fonctions `entry_points()`, qui permettent la détection de plugins destinés au framework, ainsi que pour `version()`. En outre, `configparser` est utilisé pour créer et gérer des fichiers de configuration, enregistrant ainsi les préférences de l'utilisateur, telles que le nom de l'auteur ou son adresse e-mail. Enfin, `Poeblix` est un plugin de `Poetry` utilisé pour installer les fichiers de données du plugin dans l'environnement de l'utilisateur.

3.2.2 La commande galactic

Le but de cette commande est de pouvoir créer, à partir d'un modèle de base/template à remplir, le squelette d'une extension pour le *framework* Galactic. Pour cela, lorsque la commande est exécutée,

```
$ galactic
```

est entrée par le biais d'un terminal, elle doit pouvoir soit remplir le modèle automatiquement grâce aux options ajoutées dans la commande entrée, soit pouvoir demander après exécution de la commande à pouvoir remplir chaque partie du modèle par le biais de questions répondues à travers le terminal. Voici un exemple du résultat lorsqu'on utilise `$ galactic`

```
$ galactic

GALACTIC framework (version 0.1.0.dev0)

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display help for the given command. When no
↪  command is given display help for the list command.
  -q, --quiet           Do not output any message.
  -V, --version         Display this application version.
      --ansi            Force ANSI output.
      --no-ansi         Disable ANSI output.
  -n, --no-interaction  Do not ask any interactive question.
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal
↪  output, 2 for more verbose output and 3 for debug.

Available commands:
  characteristic  Characteristic plugin
  help            Displays help for a command.
  list            Lists commands.
```

En première ligne de résultat, le nom et la version de l'application sont affichés, suivis des différentes options applicables à la commande galactic `$ galactic --help`, qui par défaut retourne l'aide de la commande `list`. Les différentes commandes disponibles dans galactic sont ensuite présentées, comprenant deux commandes par défaut, `help` et `list`, ainsi que tous les *meta-plugins* installés dans l'environnement de l'utilisateur qui ont été créés pour le *framework* **GALACTIC**. Nous pouvons voir ici que le *meta-plugin* "characteristic" est installé et disponible pour une utilisation ultérieure.

Le code de l'application *framework* **GALACTIC** est régi par de nombreux codes de style qui sont testés :

- `black`, un formateur de code *Python* ;
- `flake8`, qui nous permet d'utiliser `pyflakes` pour analyser le code et détecter les erreurs, ainsi que `pycodestyle`, un outil pour vérifier le code *Python* conformément aux conventions du "Python Enhancement Proposals 8" (PEP 8), un guide de style pour le code *Python* ;
- `pylint`, un analyseur de code statique pour *Python*. Il analyse le code sans l'exécuter, recherche des erreurs, applique une norme de codage, propose des modifications et repère les "code smells", qui sont des indications superficielles de problèmes plus profonds dans un système. Les "code smells" ne signalent pas toujours un problème en eux-mêmes, mais servent d'indicateurs. Ils sont faciles à repérer et peuvent conduire à des problématiques intéressantes. Même les personnes inexpérimentées peuvent les détecter. Aborder un "code smell" à la fois permet d'améliorer progressivement les compétences en programmation.

3.2.3 Fonctionnement de la commande `galactic`

Avec `cleo`, la création d'une application est simplifiée : il suffit d'instancier la classe `Application` en lui passant les arguments, c'est-à-dire le nom et la version de l'application. Ensuite, on définit le nom qui s'affichera lors de l'appel de l'application. En utilisant `application.run()`, notre application démarre lorsque la fonction `run()` est appelée. Grâce à `poetry`, nous pouvons configurer un point d'entrée (*entry point*) nommé `galactic`. Lorsque le *framework* **GALACTIC** est exécuté, il est conçu pour être extensible en recherchant les *meta-plugins* installés dans l'environnement de l'utilisateur qui appartiennent au groupe de *plugins* `galactic.apps.cli.framework`. Pour chaque *meta-plugin* détecté, nous chargeons, instancions sa classe et appelons la méthode `activate()`. Cette méthode, à laquelle nous fournissons notre application en argument, ajoute les commandes spécifiques du *meta-plugin* à l'application du *framework* **GALACTIC** (voir figure fig. 2).

```python
def run()->None:
    application = Application(name="galactic",
    ↪ version=version("py-galactic-framework"))
    application.set_display_name("GALACTIC framework")
    for plugin in entry_points(group="galactic.apps.cli.framework"):
        cls = plugin.load()
        instance = cls()
        instance.activate(application)
    application.run()
```

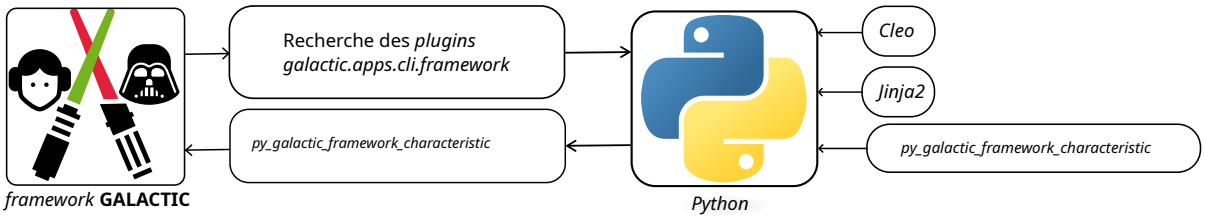

Figure 2: Schéma de recherche du *meta-plugins*

3.3 Conception du *meta-plugin*

Afin de développer le *meta-plugin*, tout comme l'application, j'ai utilisé `poetry`, `cleo`, `configparser`, `Poeblix`, `Jinja2` qui ont été présentés précédemment.

3.3.1 Le *meta-plugin* characteristic

L'objectif est clair : une fois installé comme n'importe quel package *Python*, il sera spécifiquement utilisé avec le *framework* **GALACTIC**. Cet outil sera dédié à la création de *meta-plugins* de caractéristiques, comprenant des fichiers générés à partir de modèles pré-remplis, prêts à être développés selon les besoins de l'utilisateur.

Avec le *framework* **GALACTIC**, on utilisera un *meta-plugin* de cette manière,

```
$ galactic characteristic [argument] [option(s)]
```

À partir d'ici, on peut donner les arguments et/ou options que le *meta-plugin* "characteristic" accepte.

```
$ galactic characteristic --help

Description:
  Characteristic plugin

Usage:
  characteristic [options] [--] [<action>]

Arguments:
  action                to manage a characteristic [default: "create"]

Options:
  -o, --output=OUTPUT   output for characteristic plugin
  -h, --help            Display help for the given command. When no
 ↪  command is given display help for the list command.
  -q, --quiet           Do not output any message.
  -V, --version         Display this application version.
      --ansi            Force ANSI output.
      --no-ansi         Disable ANSI output.
  -n, --no-interaction  Do not ask any interactive question.
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal
 ↪  output, 2 for more verbose output and 3 for debug.
```

On voit que le *plugin* de son nom complet 'py-galactic-framework-characteristic', peut recevoir un argument qui sera, par défaut, create, afin de créer une nouvelle caractéristique. De plus, nous avons la possibilité de spécifier facultativement l'option `--output` pour indiquer le chemin de destination de notre nouvelle caractéristique. Dans le cas contraire, lors de l'exécution de la commande, il nous sera demandé de le fournir.

3.3.2 Fonctionnement du *meta-plugin* `characteristic`

Le *meta-plugin* est initialement un projet `poetry` standard, mais ce qui le distingue en tant que *plugin* réside dans le fichier `pyproject.toml`. Ce fichier de configuration du projet contient des informations essentielles telles que le nom du projet, sa version et les dépendances requises pour son fonctionnement. Cependant, afin que `poetry` puisse identifier ce projet comme un *plugin* spécifique, il utilise la section `tool.poetry.plugins."galactic.apps.cli.framework"`.

Nous ajoutons "galactic.apps.cli.framework", ce qui est crucial pour informer l'environnement *Python* de l'utilisateur qu'il s'agit d'un *plugin* appartenant au groupe "galactic.apps.cli.framework". Ainsi, le *framework* **GALACTIC** peut correctement identifier les *meta-plugins* qui lui sont destinés.

Le cœur du *meta-plugin* est représenté par la classe `CharacteristicPlugin`, avec sa méthode `activate`. Lorsqu'elle est appelée avec l'instance de l'application, nous ajoutons les commandes

que le *meta-plugin* peut offrir.

```python
class CharacteristicPlugin:

    def activate(self, application:Application):
        command=CharacteristicCommand()
        application.add(command)
```

Le *meta-plugin characteristic* possède une seule commande, `characteristic`, qui peut prendre un argument et l'option facultative `--output`.

Lorsqu'on donne l'argument "create" à la commande `characteristic`, cela exécutera le code pour créer une nouvelle caractéristique. L'aspect le plus important à savoir est l'utilisation de `self.ask()` pour demander à l'utilisateur les informations de manière interactive pour la création de la caractéristique. On met également en place un fichier `config.ini` qui enregistre les préférences de l'utilisateur, telles que le nom de l'auteur ou son adresse e-mail, afin d'éviter de redemander les informations de manière répétée. Par exemple, on utilise `os.getlogin()` pour récupérer le nom de l'utilisateur de l'environnement, qui sera le nom par défaut lors de la première utilisation du *meta-plugin*. Grâce au package *Python* `platformdirs` et à la méthode `user_config_dir()`, on peut créer le fichier `config.ini` à l'emplacement suivant : `C:\\Users\\utilisateur\\AppData\\Local\\nom-projet` (exemple sur un environnement Windows). L'avantage est que `platformdirs` s'adapte à l'environnement de l'utilisateur, ce qui rend le *meta-plugin* compatible avec Windows, Linux ou Mac. Ensuite, pour la génération de nos fichiers via des fichiers modèles, on utilisera Jinja2. Voici un exemple de fichier modèle pour une classe *Python* pouvant prendre autant d'arguments que souhaité.

```python
class {{ className }}:
    def __init__(self{% for key,value in data.items() %}, {{ value }}{%
    ↪  endfor %}):
        {% for key,value in data.items() -%}
        self.{{ value }}={{ value }}
        {% endfor %}
```

Voici le résultat en donnant un argument nommé "name",

```python
class person:
    def __init__(self, name):
        self.name=name
```

Le *meta-plugin characteristic* produira un fichier *Python* contenant une classe, ainsi que les fichiers pyproject.toml et README.md. Ce travail est encore en cours, donc les fichiers générés par le *meta-plugin* ne sont pas définitifs, des modifications peuvent être apportées. Lorsque l'utilisateur fournit les informations nécessaires, Jinja2 les traitera pour les intégrer dans les fichiers générés à partir des modèles. Ainsi, une fois le *plugin* terminé, le projet de l'utilisateur sera créé. Il lui suffira alors d'exécuter les commandes suivantes,

```
$ poetry lock
$ poetry install
```

Et lorsque son *plugin* sera terminé, il pourra construire le projet avec

```
$ poetry blixbuild
```

Cela générera un fichier de type “wheel” (.whl) qui pourra être installé comme n’importe quel autre package *Python*.

Remarquez que nous n’avons pas utilisé la commande par défaut `poetry build` pour le projet. Nous utilisons plutôt la dépendance `poeblix`, d’où `poetry blixbuild`, qui nous permet de gérer les fichiers de données nécessaires au fonctionnement du *meta-plugin* et de les installer sur le système de l’utilisateur. En effet, le *meta-plugin characteristic* comprend des fichiers modèles indispensables, que nous installons dans un dossier `share` à la racine de l’environnement *Python*. Nous obtenons cet emplacement grâce à `sys.prefix`, qui nous fournit l’emplacement de *Python*.

Voici comment le *meta-plugin* est organisé : il se compose de deux parties. La première partie est le cœur, comprenant les différents fichiers *Python* nécessaires à son fonctionnement, tandis que la seconde partie contient les fichiers modèles.

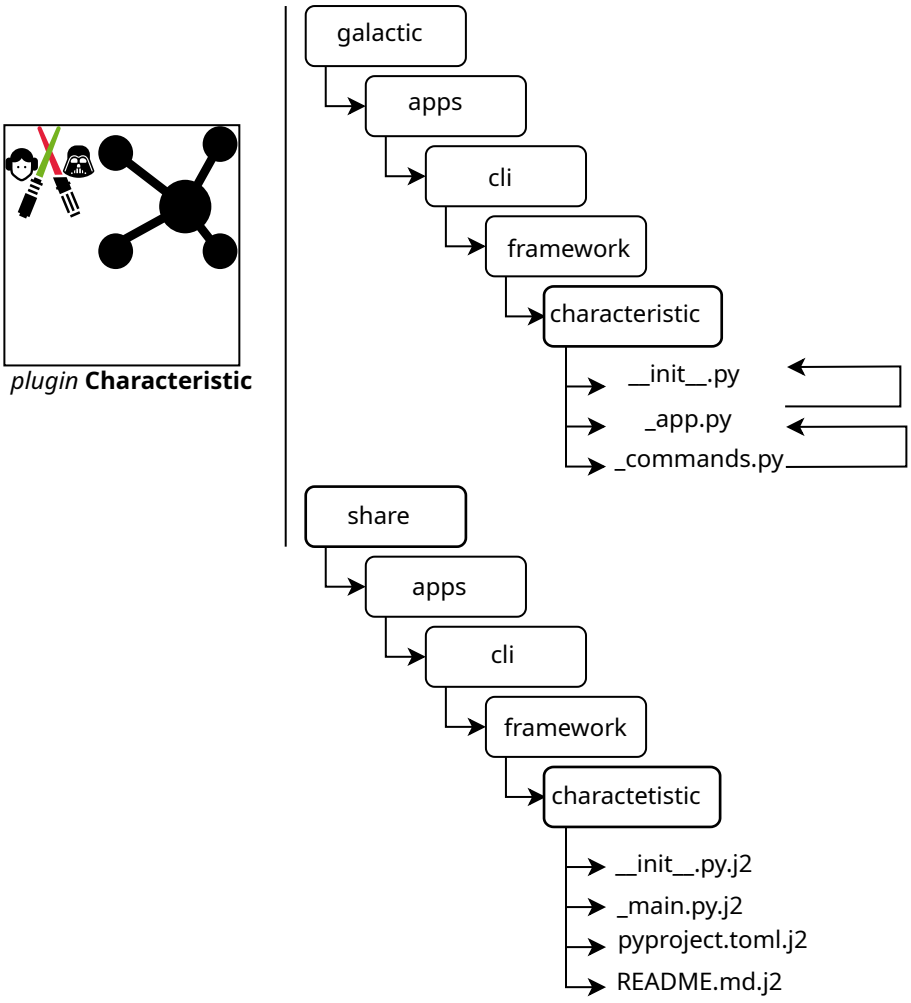

Figure 3: Schéma du *meta-plugin* Characteristic

On constate que `_app.py` a accès à `_commands.py` et que les méthodes ou classes mentionnées

dans `__init__.py` seront publiques. Cependant, ces méthodes ou classes seront exclusivement issues du fichier `app.py`.

3.4 Difficultés rencontrées

La principale difficulté que j'ai rencontrée concernait la documentation de la bibliothèque `cleo`. Elle est uniquement mise à jour sur le dépôt GitHub du projet et est plutôt succincte en termes de démonstration des fonctionnalités. Cependant, étant donné que le projet est accessible à tous sur GitHub et compte tenu de ma jeune expérience de développeur, je n'avais pas l'habitude de consulter directement les détails dans le code ni d'examiner d'autres projets disponibles librement sur GitHub qui utilisent `cleo`. Mon maître de stage, M. Demko, m'a bien conseillé à ce sujet. Une petite remarque : à l'origine, nous envisagions d'utiliser `appdirs` au lieu de `platformdirs` pour déterminer l'emplacement où le fichier de configuration serait créé. Cependant, j'ai réalisé que `appdirs` était plus ancien et donc obsolète. La raison en est que `platformdirs` est la version améliorée d'`appdirs`.

3.5 Installation

Une fois que les deux projets ont été *build*, on obtient deux fichiers .whl. Nous pouvons ensuite installer le *framework* **GALACTIC** ainsi que le *meta-plugin* à l'aide de pip.

Installation du *framework* **GALACTIC**

```
$ pip install py_galactic_framework-0.1.0.dev0-py3-none-any.whl
```

Installation du *meta-plugin* py_galactic_framework_characteristic

```
$ pip install py_galactic_framework_characteristic-0.1-py3-none-any.whl
```

3.6 Utilisation

Je vais illustrer l'utilisation du *meta-plugin characteristic* avec l'application *framework* **GALACTIC** à l'aide d'un exemple. Veuillez noter qu'il ne s'agit pas d'une version finale et qu'il y aura probablement des modifications car le développement est toujours en cours.

```
$ galactic characteristic create

Enter the name of the plugin, default "Hello_There_Project":
 ↪  Characteristic_Test
Write the description of the project:  This is a test to create a new
 ↪  Characteristic
Give the absolut path where the project will be create, default:
 ↪  C:\Users\yanni\OneDrive\Bureau\**GALACTIC**2\poetryProject:  C:\Us
ers\yanni\Onedrive\Bureau
Enter the author name, default author "yanni":  Yannis Rouhier
(Optional) Write your email, if no email leave blank
 ↪  yannis.rouhier@etudiant.univ-lr.fr
Setting up projects directories
```

```
Directories setuped
Setting up project files
Give the name of your python class, default: 'hello_there_class'
↪  CharacteristicTestClass
Enter the number of attributes the class will have, default "1":  3
Number 1 attribute name:  test1
Number 2 attribute name:  test2
Number 3 attribute name:  test3
Plugin project "Characteristic_Test." succesfully created at
↪  C:\Users\yanni\Onedrive\Bureau\Characteristic_Test !
```

3.6.1 Résultat

Voici donc le projet “Characteristic_Test” créé en utilisant le *meta-plugin characteristic* de l’application *framework* **GALACTIC**. Dans sa version de développement actuelle, le projet est composé de trois fichiers : un fichier *Python* nommé `CharacteristicTestClass.py` qui contient une classe *Python* appelée `CharacteristicTestClass` avec trois attributs : `test1`, `test2` et `test3`. Le projet inclut également le fichier `pyproject.toml`, qui est essentiel pour un projet `poetry`, ainsi qu’un fichier `README.md`.

Voici le contenu des fichiers,

Le fichier *Python* avec la classe `CharacteristicTestClass`

```python
class CharacteristicTestClass:
    def __init__(self, test1, test2, test3):
        self.test1=test1
        self.test2=test2
        self.test3=test3
```

Le fichier `pyproject.toml`

```toml
name = "Characteristic_Test"
version = "0.1.0"
description = "This is a test to create a new Characteristic"
authors = ["Yannis Rouhier <yannis.rouhier@etudiant.univ-lr.fr>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.10"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Le fichier `README.md`

```
# **Characteristic_Test**
Project created by Yannis Rouhier, <yannis.rouhier@etudiant.univ-lr.fr> the 22/0
```

```
# Installation
```bash
$ cd C:\Users\yanni\Onedrive\Bureau\Characteristic_Test
$ poetry check
$ poetry lock
$ poetry install
```
Now ready for developpement purposes
```

Et enfin le projet a une arborescence qui se présente ainsi

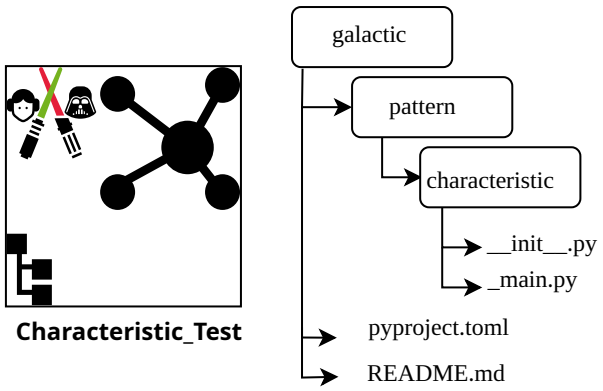

Figure 4: Arborescence du *plugin* characteristic test

4 Conclusion

J'ai effectué mon stage au Laboratoire Informatique, Image et Interaction dans le cadre du projet **GALACTIC**. Mon objectif principal consistait à développer une interface en ligne de commande permettant de générer des modèles pour les extensions du *framework* **GALACTIC**. Afin d'assurer le bon fonctionnement de l'application framework GALACTIC, j'ai également dû créer le *meta-plugin* py.galactic.framework.characteristic, qui permet à l'utilisateur de l'application *framework* **GALACTIC** de créer une nouvelle caractéristique.

L'application est opérationnelle, je ne dirais pas qu'elle est achevée car il reste à mettre en place les tests unitaires et il est possible qu'il y ait encore des ajustements ou des ajouts, étant donné que j'aurai encore quatre semaines de stage après la soutenance. Une amélioration envisageable pour l'application framework GALACTIC serait d'ajouter une option appelée –output permettant d'enregistrer dans le fichier de configuration et de partager aux *meta-plugins* un répertoire de sortie par défaut.

Le *meta-plugin* py.galactic.framework.characteristic est également opérationnel, mais il existe plusieurs points qui peuvent être modifiés ou des fonctionnalités ajoutées. Ce qui reste à faire, c'est d'implémenter les tests unitaires et les tests de convention de style.

Ce stage m'a permis d'enrichir mes connaissances en *Python* tout en apprenant. Par exemple l'utilisation de Poetry m'a permis d'apprendre comment il est possible de créer des packages *Python* pouvant être publiés sur PyPi, une collection de programmes pour *Python* notamment grâce au fichier d'extension wheel (.whl). Créer l'application ainsi que le *meta-plugin characteristic* a exigé de moi une rigueur l'organisation et la dénomnation des fichiers, afin de faciliter la compréhension par une personne externe. Il m'a également permis de découvrir la rigueur et le professionnalisme attendus pour un projet informatique, notamment les tests unitaires, le respect du style de code et l'organisation des fichiers. J'ai également acquis une solide compréhension de Git, par exemple avec la méthodologie que mon maitre de stage M.Demko m'a enseigné, son implication dans le développement de l'application a été essentielle: pour chaque correction il créait une *issue* sur *Git* exemple `Fix #8` et lorsque il avait terminé les modifications, il soumettait une *merge request* avec comme titre `Resolve "Fix #8"`. Grâce à toutes ces connaissances acquises pendant le stage, mes projets personnels en bénéficieront.

5 Annexes - Sources

Présentation de l'entreprise

https://l3i.univ-larochelle.fr/Presentation-317

https://l3i.univ-larochelle.fr/Equipes

https://www.univ-larochelle.fr/wp-content/uploads/pdf/ULR-FICHES-LABO-L3i-2020-vf-web.pdf

Présentation de la Plateforme GALACTIC

https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-20min-complete.pdf

https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-20min-with-notes.pdf

https://galactic.univ-lr.fr/slides/fca/Galactic-FCA-slides-with-notes.pdf

Autres

En savoir plus sur le Code Smell: https://martinfowler.com/bliki/CodeSmell.html