Projet Agile

Qilian GU, Yufei LIU, Amine El Hassani, Aizé Baudin

Sommaire

- Diagramme des classes générale
- Partie I : La musique
- Partie II : L'atelier
- Partie III : Le merge.

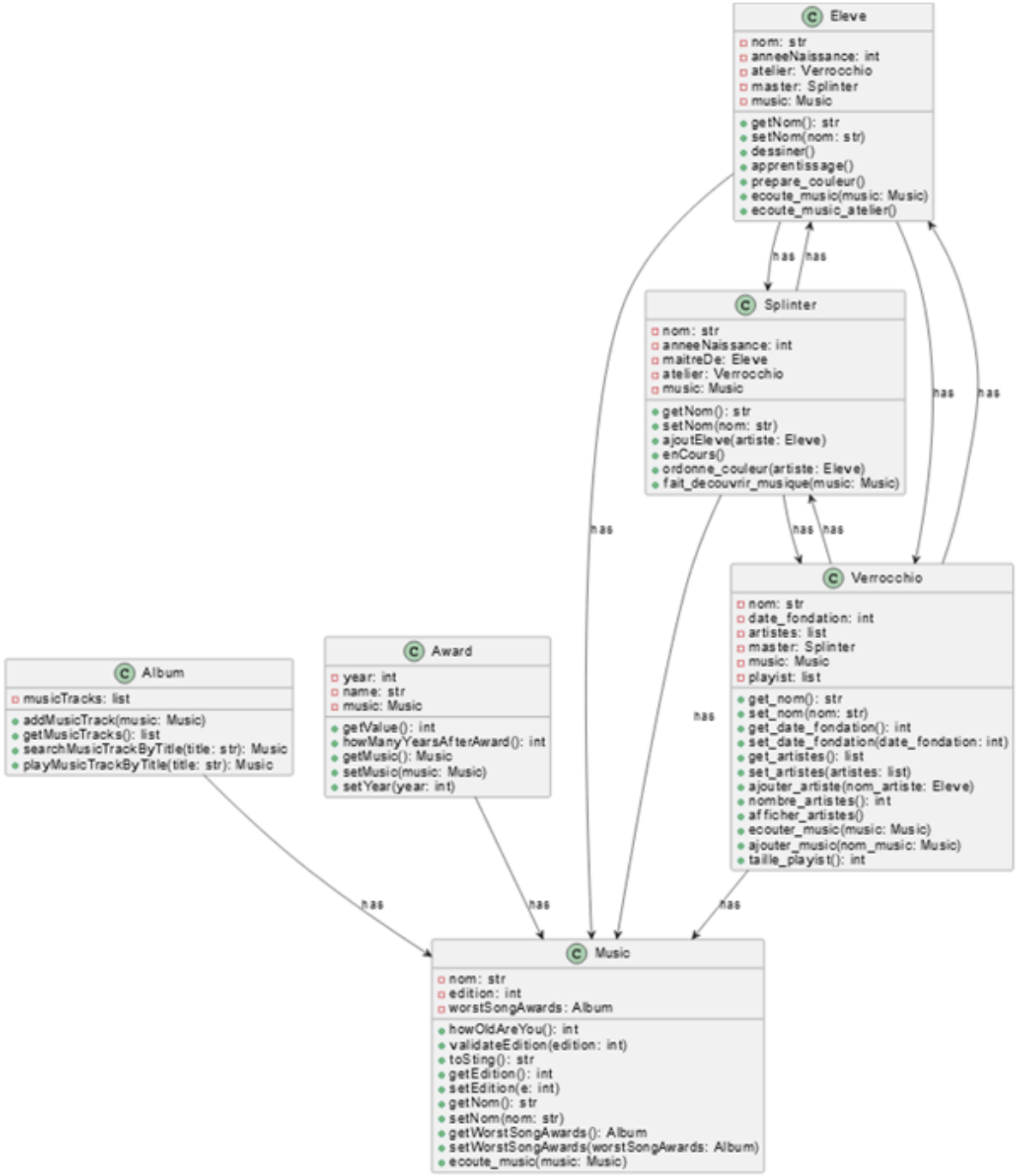


Diagramme des classes générale

Voici une présentation générale de l'organisation et de l'interaction entre les classes à l'aide du diagramme des classes fait à partir du site PlantUML (vous pouvez le retrouver dans le github du projet)

Partie I : musique

- Deep within the heart of the music industry, a group of rebels dared to challenge the conventional notion of musical excellence. They believed that even the worst of songs deserved recognition, if only for their ability to evoke laughter and spark a sense of camaraderie among listeners.

Class Album: generates albums with music titles.

Class Music: generates music with characteristics such as the title and edition and to which album they belong.

Class Award: the award for the worst music, including the music title and the year of award

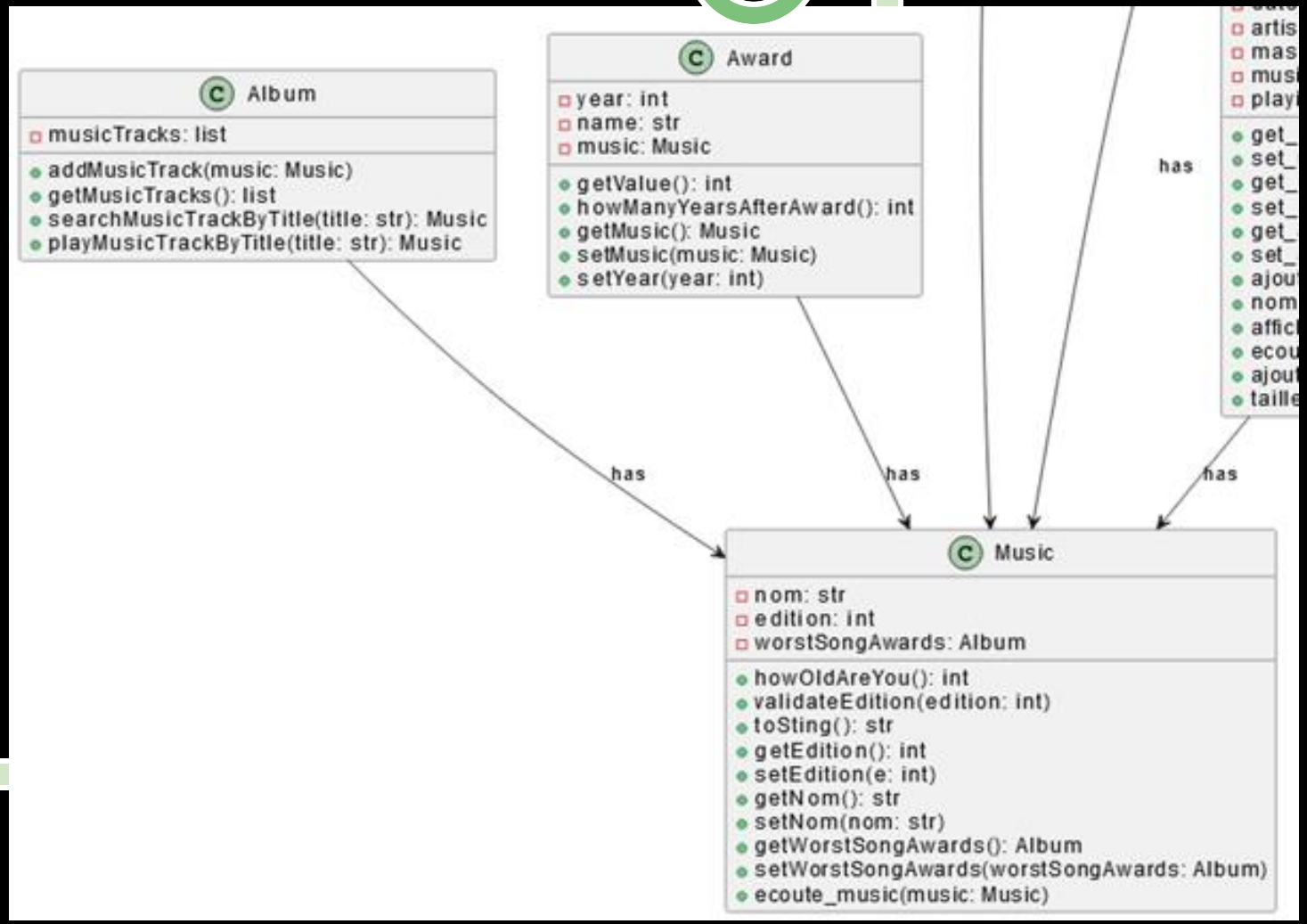
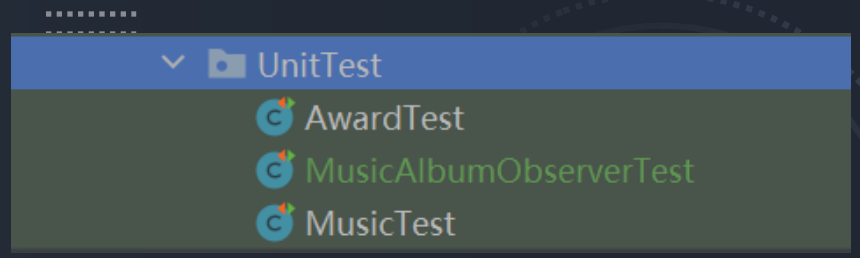


Diagramme des classes : Musique

Test Music



✓ Unit test of implemented method

```
@BeforeEach
public void setUp() {
    music = new Music("Love Story", 2021);
}

@Test
public void testAge(){
    Music SymphonyNo5 = new Music();
    SymphonyNo5.setEdition(1804);
    int result = SymphonyNo5.howOldAreYou();
    assertEquals(219, result);
}

@Test
public void testAgeFailed(){
    Music SymphonyNo5 = new Music();
    SymphonyNo5.setEdition(1804);
    int result = SymphonyNo5.howOldAreYou();
    assertEquals(219, result);
}
```

Test Music

Album test and music streaming test with Cucumber


```
Feature: US No.1 Create an album containing multiple music tracks
  As a music enthusiast
  I want to create an album containing multiple music tracks
  So that I can organize and enjoy my favorite songs

  Scenario: Create a new album and add multiple music tracks
    Given I have an empty album
    When I add a music track with title "Track 1" and edition 2022
    Then the album should contain 1 music tracks
    And the album should have a music track with title "Track 1" and edition 2022
```

```
public class AlbumStepDefinitions {
    private Album album;

    @Given("I have an empty album")
    public void i_have_an_empty_album() {
        album = new Album();
    }

    @When("I add a music track with title {string} and edition {int}")
    public void i_add_a_music_track_with_title_and_edition(String title, int edition) {
        Music music = new Music(title, edition);
        album.addMusicTrack(music);
    }

    @Then("the album should contain {int} music tracks")
    public void the_album_should_contain_music_tracks(int numberOfTracks) {
        assertEquals(numberOfTracks, album.getMusicTracks().size());
    }

    @Then("the album should have a music track with title {string} and edition {int}")
    public void the_album_should_have_a_music_track_with_title_and_edition(String title, int edition) {
        assertTrue(album.getMusicTracks().stream().anyMatch(m -> m.getNom().equals(title) && m.getEdition() == edition));
    }
}
```

```
✓ Test Results                                24 ms
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" ...
Testing started at 13:28 ...

1 Scenarios (1 passed)
4 Steps (4 passed)
0m0.747s
```

Test Music

Album test and music streaming test with Cucumber

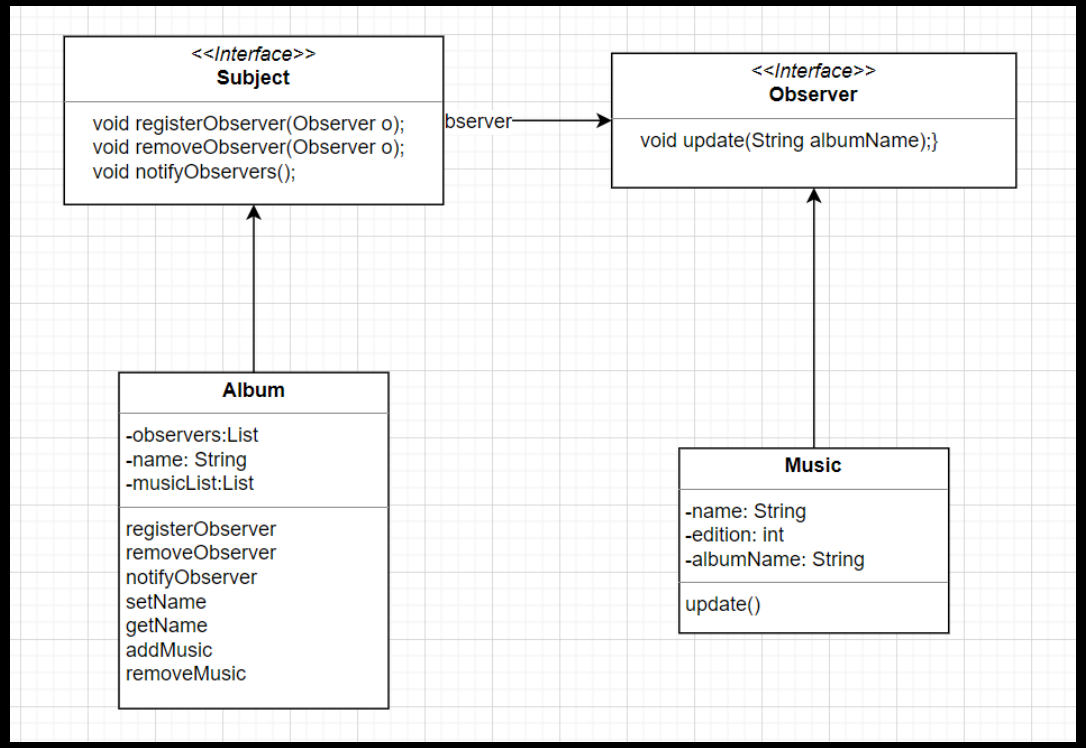
```
Feature: Search and play music tracks
  As a user of the music streaming service
  I want to search for music tracks by title
  And play the selected track
  So that I can listen to my favorite songs

  Scenario Outline: Search for a music track by title and play it
    Given the music streaming service has the following music tracks
      | title    | edition |
      | Track 1  | 2021    |
      | Track 2  | 2022    |
      | Track 3  | 2020    |
    When I search for the music track with title "<title>"
    Then I should see the music track with title "<title>" and edition <edition>
    When I select the music track with title "<title>"
    Then the music track "<title>" should start playing

    Examples:
      | title    | edition |
      | Track 1 | 2021    |
      | Track 3 | 2020    |
```

```
yufei-liu-source
@Given("the music streaming service has the following music tracks")
public void the_music_streaming_service_has_the_following_music_tracks(io.cucumber.datatable.Da
    List<List<String>> musicData = dataTable.asLists(String.class);
    musicStreaming = new Album();

    for (List<String> row : musicData.subList(1, musicData.size())) {
        String title = row.get(0);
        int edition = Integer.parseInt(row.get(1));
        Music music = new Music(title, edition);
        musicStreaming.addMusicTrack(music);
    }
}

yufei-liu-source
@When("I search for the music track with title {string}")
public void i_search_for_the_music_track_with_title(String title) {
    searchedMusic = musicStreaming.searchMusicTrackByTitle(title);
}

yufei-liu-source
@Then("I should see the music track with title {string} and edition {int}")
public void i_should_see_the_music_track_with_title_and_edition(String title, int edition) {
    assertEquals(title, searchedMusic.getNom());
    assertEquals(edition, searchedMusic.getEdition());
}
```

```
Test Results  23 ms
"C:\Program Files\Java\jdk-17.0.1\bin\java.exe" ...
Testing started at 13:31 ...

2 Scenarios (2 passed)
10 Steps (10 passed)
0m0.736s
```

Design Pattern

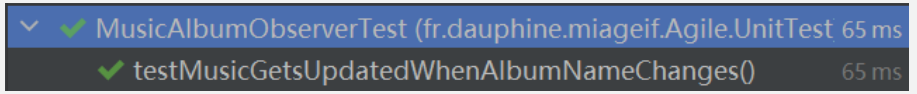
- Observer



Observer Test

- In this example, the class music is an observer and the album is a subject, once the name of subject changed, here the album named ' worst of France ' was changed as ' best of France' , so the music belonged to the album ' worst of France' now belongs to the album 'best of France' now.

```java
public class MusicAlbumObserverTest {
    4 usages
    private Album album;
    3 usages
    private Music music1;
    3 usages
    private Music music2;

    @BeforeEach
    public void setup() {
        album = new Album( name: "WorstOfFrance");
        music1 = new Music( nom: "la boheme",  edition: 2023);
        music2 = new Music( nom: "See you again",  edition: 2024);
        album.addMusicTrack(music1);
        album.addMusicTrack(music2);
    }

    @Test
    public void testMusicGetsUpdatedWhenAlbumNameChanges() {
        album.setName("BestOfFrance");
        assertEquals( expected: "BestOfFrance", music1.getAlbumName());
        assertEquals( expected: "BestOfFrance", music2.getAlbumName());
    }
}
```



Partie II : L'atelier d'artiste

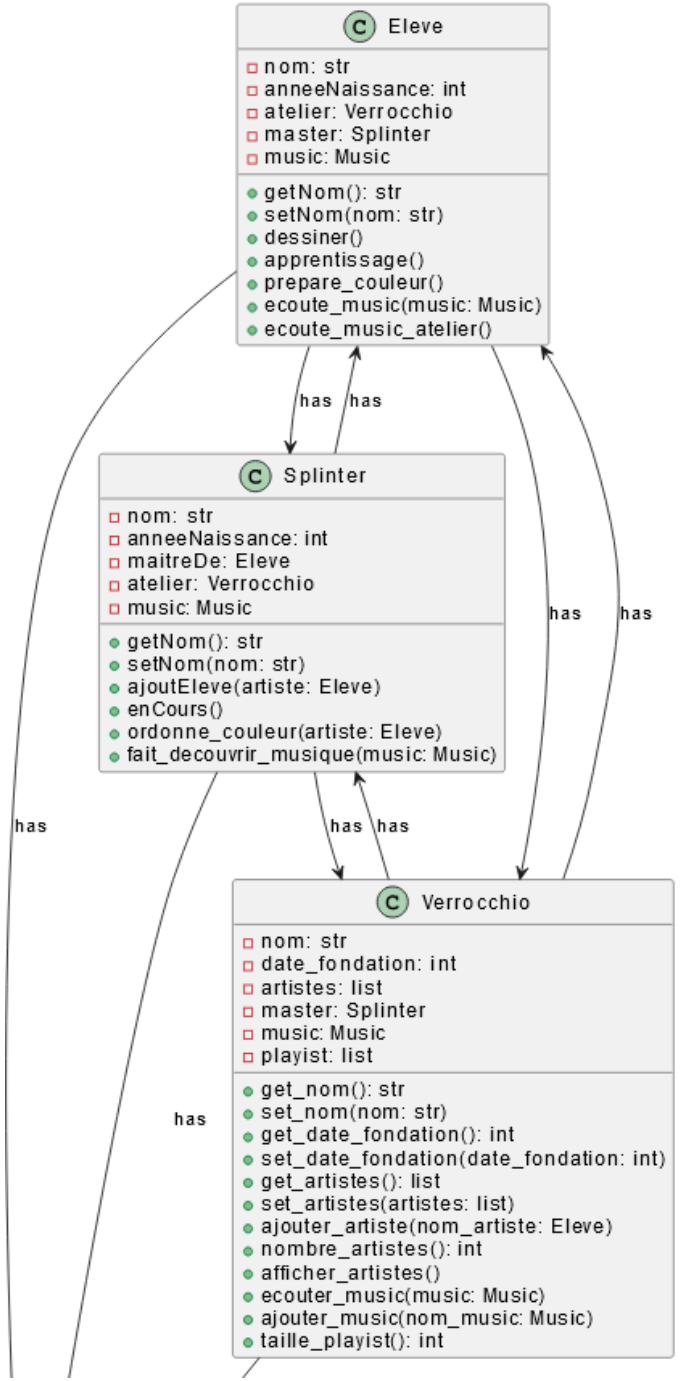
Voici ci-contre la boutique de l'atelier de Verrochio, du réelle atelier qui accueilla les plus illustres artiste de la Renaissance.

Nous allons ici présenter l'évolution imaginaire de cette atelier, en atelier de surhumain qui luttent contre les forces du mal de l'obscurentisme, toujours plus insistant de nos jours...

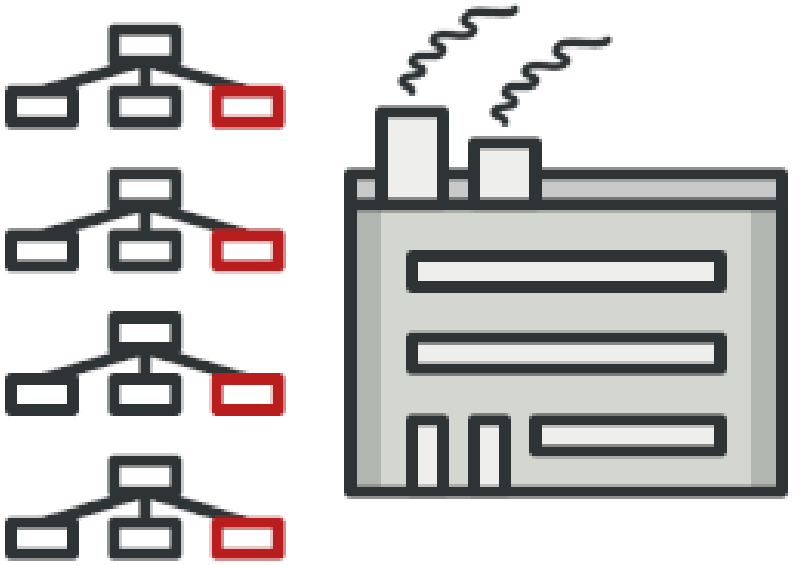
Diagrammes des classe : Atelier

Voici comment va s'organiser le développement de notre atelier de surhumain, prêt à apprendre tout les secret de l'Art, afin de lutter contre les forces obscurentistes !

- **Eleve** : est une classe qui va créer des élèves au fure et à mesure que le maître (Splinter) accepte l'élèves dans l'atelier
- **Splinter** : est une classe dite singleton, qui gère et créer son atelier selon ses ordres.
- **Verrocchio** : classe sui créer l'atelier (photo vu en introduction de la partie). Elle va réceptionner les élèves, et présenter l'action des élèves et des ordre du maître Splinter.



II.1 Design pattern utilisé



Nous n'avons pas explicitement utilisé un design pattern spécifique mais nous nous sommes inspirés du design pattern **abstract factory**.

Ce choix fut induit par l'utilisation courante de ce pattern lors de la création de class en Python, puisque cela définit une interface pour créer toutes les instances de classes distinctes, mais laisse la création des instances de classes réel à des classes d'utilisation concrètes. Chaque type d'utilisation correspond à une certaine instance.

A Modern-style sofa doesn't match Victorian-style chairs.

II.2 presentation des class - l'atelier

```python
from typing import List

import Splinter
from Eleve import *


10 usages
class Verrocchio:
    def __init__(self, nom: str, date_fondation: int, master: Splinter):
        self.nom = nom
        self.date_fondation = date_fondation
        self.artistes = []
        self.master = master
        self.music = Music
        self.playist = []
```

- Nous introduisons ici classiquement l'atelier Verrocchio qui accueillera nos artistes.

- Il est définit en particulier par le maître, ici Splinter.

II.2 Présentation des class - Le maître

```
3  import Eleve
4  import Music
5  import Verrocchio
6
7
   4 usages
8  class Splinter:
9      def __init__(self, nom, anneeNaissance):
10         self.nom = nom
11         self.anneeNaissance = anneeNaissance
12         self.maitreDe = Eleve
13         self.atelier = Verrocchio
14         self.music = Music
```

- De même la class maître est construite classiquement tout comme celle de Verrochio.
- Ici ce n'est pas l'atelier qui définit le maître, mais bien le choix des élèves qui sont déterminé par le maître.

II.2 Présentation des class - les élèves

- Et voici enfin la class des élèves qui définirons nos artistes en herbes. Ils devront suivre le maître Splinter dans ses pas pour poursuivre leur mission ultime.

- Ils n'auront pour seule liberté de choisir leur musique personnelle, et leur autre liberté sera la contrainte des ordres du maître Splinter et d'exercer leur talent d'artiste.

```python
import Music



11 usages
class Eleve:
    Eleve = None

    def __init__(self, nom, anneeNaissance, atelier, master):
        self.nom = nom
        self.anneeNaissance = anneeNaissance
        self.atelier = atelier
        self.master = master
        self.music = Music
```

II.3 Le test unitaire du concours d'art - feature

ajout_music_test.feature | ajout_artiste_test.feature | ajout_music_test.py | music_atelier.py

```
@tag
Feature : add les artistes pour devenir des surhumains qui combattent l'obscurentisme
    en tant que master Splinter,
    afin que les students deviennent des pointes en peinture, mais aussi en architecture, en sculture,
    en mathematics, etc...


Scenario : Ajout Leonardo
    Given un eleve Leonardo
    When Splinter accueil artiste
    Then Leonard est mon eleve

Scenario Outline : add artiste
    Given eleve <eleve>
    When Splinter accueil artiste
    Then artiste <ajouter_artiste>
    Examples:
      | eleve       | ajouter_artiste                |
      | "Leonardo"  | "Bienvenu Leonardo a Verrochio"  |
```

- Voici le fichier feature, qui permet d'établir le scénario de l'ajout d'un artiste dans l'atelier, selon le format python, à savoir, un scénatio adapté à la bibliothèque behave que nous allons voir dans le slide suivant.

ajout_artiste_test.feature | ajouter_artistes_test.py | Concours_art_test.py | Verrocchio.py

```python
from dataclasses import dataclass
from behave import *
from Verrocchio import *


@dataclass
class ManageArtisteInSteps:
    _artiste: Eleve.Eleve
    _master: Splinter
    _atelier: Verrocchio

    def __init__(self):
        self._master = Splinter.getNom(self._master)
        self._atelier = Verrocchio.ajouter_artiste(nom_artiste=self._artiste.getNom())

    @given("un eleve Leonardo")
    def un_eleve_leonard(self):
        self._artiste = Eleve.Eleve("Leonardo", 1452, "Verrocchio", self._master)

    @when("Splinter accueil artiste")
    def ajouter_artiste_atelier(self):
        _master = Splinter.ajoutEleve(self._artiste)
```

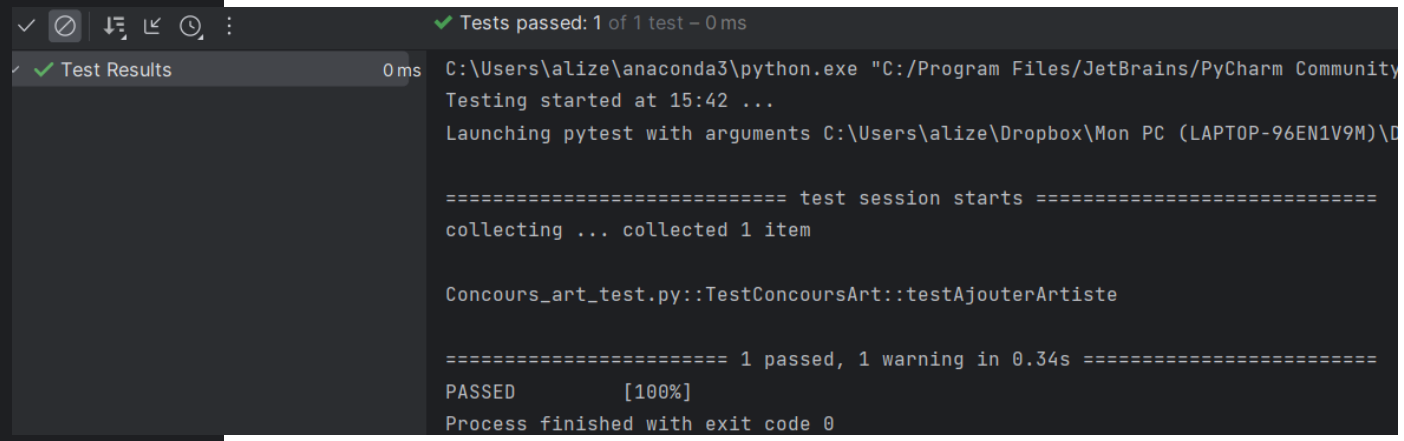
II.3 Le test unitaire du concours d'art – définition

Voici une partie du code de la définiton de la feature. Nouc utilison la bibiothèque beahave qui permet de "donner vie" au scénario du slie précédent à l'aide des anotations **@given("<initiation de l'action>")**, **@when(""<description de l'action>")** et **@then("<la conclusion de l'action>")**

II.3 Le test unitaire concours d'art - le concours

- Voici l'exécution du test qui utilise la bilbiothèque **unittest**, qui permet l'exécution du test via la bibliothèque behave.
- Ici la fonction **testAjouterArtist(self)** fait office de la fonction **main()** du code classique en python. C'est là où l'on va configurer nos paramètres qui permettent d'introduire les attributs et modules des classes associée à l'atelier Verrocchio nécessaire à l'ajout d'artiste dans l'atelier.
- Ici, Léonard de Vinci à passé avec succès le concours d'art (nous n'en attendons pas moins de lui)

```python
import unittest
import Verrocchio
from Eleve import Eleve
from Splinter import Splinter


class TestConcoursArt(unittest.TestCase):

    def setUp(self):
        ...

    def tearDown(self):
        pass

    def testAjouterArtiste(self):
        master = Splinter("Splinter", 1435)
        atelier = Verrocchio.Verrocchio("Verrochio", 1438, master)
        artiste1 = Eleve("Leonardo da Vinci", 1452, atelier, master)
        # artiste2 = Eleve("Donatello", 1452, atelier, master)
        atelier.ajouter_artiste(artiste1)
        # atelier.ajouter_artiste(artiste2)
        self.assertEqual(1, atelier.nombre_artistes())
```



III. Le merge - Python

Nous voici à la partie Merge des codes : faire venir de la music dans l'atelier.

Pour les besoins nous avons modifié les classes, en gardant, dans le cas Python, le même pattern, à savoir l'abstrat factory, qui permet de définir les fonctions des classes en objet abstrait, d'outils qui s'enboitent dans un format simple, sans de fonguration particulière

Verrocchio.py × Splinter.py Eleve.py

```
47
48  # ---------------------------------------------------------------------- Pa
        1 usage (1 dynamic)
49      def ecouter_music(self, music):
50          print("Dans l'atelier nous écoutons la musique " + music.getNom())
51
        2 usages (1 dynamic)
52      def ajouter_music(self, nom_music: Music):
53          self.playist.append(nom_music)
54
        1 usage (1 dynamic)
55      def taille_playist(self) -> int:
56          return len(self.playist)
```

Verrocchio.py Splinter.py × Eleve.py

```
27
28      def ordonne_couleur(self, artiste):
29          self.maitreDe.prepare_couleur(artiste)

# ------------------------- Partie music
        def fait_decouvrir_musique(self, music):
            self.atelier.ecouter_musique(music.getNom())
```

Verrocchio.py Splinter.py Eleve.py ×

```
29
30  # ----------------------------------------------------------------
        def ecoute_music(self, music):
32          print(self.nom + "écoute la music "+ self.music.getNom(music))
33
34      def ecoute_music_atelier(self):
35          self.atelier.ecouter_music(self.music)
```

III. Le merge – test Python

Verrocchio.py | Splinter.py | Eleve.py | ajout_music_test.feature | ajout_music_test.py

```
@tag
Feature : nous allons faire decouvrir different musicien aux artistes de l atelier Verrochio


Scenario : Ecouter Jimi Hendrix
    Given une musique Jimi Hendrix
    When Splinter fait ecouter musique
    Then Jimi Hendrix est ecouter

Scenario Outline : ecouter musique
    Given musique <music>
    When Splinter fait ecouter musique
    Then musique <ajout_music>
    Examples:
      | music           | ajout_music                |
      | "Jimi Hendrix"  | "All along the watchtower "|
```

ajout_music_test.feature | ajout_music_test.py | music_atelier.py

```
from dataclasses import dataclass
from behave import *

import Music
import Verrocchio
from Verrocchio import *


@dataclass
class ManageArtisteInSteps:
    _artiste: Eleve.Eleve
    _master: Splinter
    _atelier: Verrocchio
    _music: Music

    def __init__(self):
        self._master = Splinter.getNom(self._master)
        self._atelier = Verrocchio.ajouter_artiste(nom_artiste=self._artiste.getNom())
        self._music = Music.getNom()

    @given("une musique Jimi Hendrix")
    def une_musique_hendrix(self):
        self._music = Music.Music("Jimi Hendrix - All Along the Watchtower",1967)
```

Nous mettons en place ainsi un test unittest de la même manière que précédement pour ajouter une musique dans la playist de l'atelier.

III. Le merge – resultat test Python

ajout_music_test.feature | ajout_music_test.py | music_atelier.py

```
3   from Eleve import Eleve
4   from Music import Music
5   from Splinter import Splinter
6
7
8   class TestAjoutMusic(unittest.TestCase):
9
10      def setUp(self):
11          ...
12
13      def tearDown(self):
14          pass
15
16      def testAjouterMusic(self):
17          master = Splinter("Splinter", 1435)
18          atelier = Verrocchio.Verrocchio("Verrocchio", 2022, master)
19          artiste1 = Eleve("Leonardo da Vinci", 1452, atelier, master)
20          atelier.ajouter_artiste(artiste1)
21          music1 = Music("Jimi Hendrix - All Along the Watchtower", 1967)
22          #atelier.ecouter_music(music1)
23          atelier.ajouter_music(music1)
24          self.assertEqual(1, atelier.taille_playist())
```

Tests passed: 1 of 1 test – 0 ms

Test Results 0 ms

```
C:\Users\alize\anaconda3\python.exe "C:/Program Files/JetBrains/PyCharm Community
Testing started at 16:10 ...
Launching pytest with arguments C:\Users\alize\Dropbox\Mon PC (LAPTOP-96EN1V9M)\D

============================= test session starts =============================
collecting ... collected 1 item

music_atelier.py::TestAjoutMusic::testAjouterMusic PASSED                [100%]

======================== 1 passed, 1 warning in 0.77s =========================

Process finished with exit code 0
```

Ici nous avons donc :

- Le venu du maître (master), la venue de l'élève Léonard de Vinci (artiste1), dans l'atelier Verrochio (atelier), pour écouter la musique et décidé de l'ajouter ou non à la playiste.
- Etant donnée que c'est du Jimi Hendrix, la musique fut directement ajoutée à la playiste de l'atelier.


Merci pour votre lecture, de
votre attention, ainsi que de
votre suivi cette année !