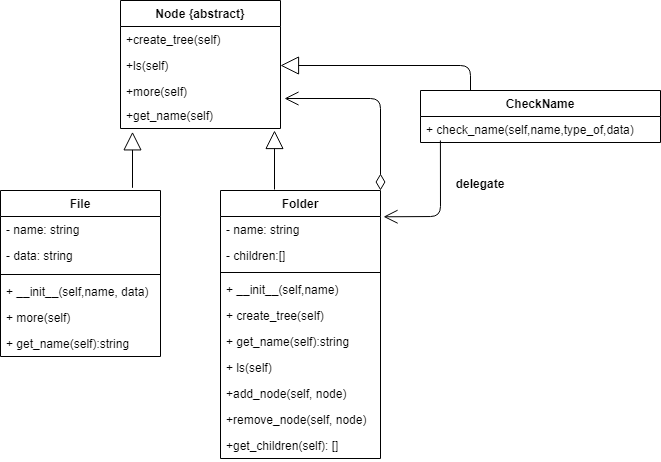
|  |  |  |
| --- | --- | --- |
| Aleksandra Duda | Gr. 31 lab 02 | Techniki Obiektowe |
| Nr indeksu: 130510 | Laboratorium nr 3 | 27.10.20r. |

**Sprawozdanie z labolatorium nr 3**

1. **Cel ćwiczenia**

Implementacja wirtualnego systemu plików.

1. **Model UML**

****

1. **Przebieg ćwiczenia**

* Z racji tego, że cały program napisałam w Pythonie to zamiast interfejsu pozwoliłam sobie zaimplementować klasę abstrakcyjną:



Node jest „węzłem” – z tej klasy będzie korzystał zarówno folder, jak i plik.

* Następnie zaimplementowałam klasę, która będzie odpowiedzialna za tworzenie pliku wraz danymi , gdzie będzie możliwe wyświetlanie jego zawartości. Każdy plik musi mieć więc swoją nazwę i może zawierać treść – za to odpowiada w moim programie konstruktor. Funkcja **more()**  służy do wyświetlania zawartości pliku, natomiast **get\_name()**  jest funkcją, która zwraca nazwę pliku (staram się pilnować hermetyzacji programów). Bez tej funkcji wszystkie zmienne klasy musiałyby być publiczne. Klasa **File**  poprzez dziedziczenie implementuje klasę abstrakcyjną **Node.**

class File(Node):

    def \_\_init\_\_(self,name,data):

        self.\_\_name = name

        self.\_\_data = data

    def more(self):

        print(self.\_\_data)

    def get\_name(self):

        return self.\_\_name

* Kolejną klasą jest **Folder** , który również implementuje klasę abstrakcyjną **Node.** W folderach mogą się znajdować pliki i inne foldery, dlatwgo też tutaj zaimplementowałam fukcję **create\_tree()** , która tworzy strukturę utworzonych plików. Dodatkowo należało zaimplementować metodę **ls(),** która wyświetla foldery oraz pliki w bieżącym folderze.

Aby utworzyć strukturę plików musimy mieć także funkcje dzięki którym będziemy mogli to zrobić, jak np. **add\_node()**, która odpowiada za dodawanie węzłóworaz **remove\_node()**,która odpowiada za usuwanie węzłów.

W przypadku tworzenia struktur plików i folderów ważne jest uporządkowanie – czyli, żeby każdy folder bądź plik znajdował się tam gdzie go stworzyliśmy (w odpowiednim folderze). Dlatego mamy zmienną taką jak **self\_\_children**  jest to tablica „dzieci” folderu – czyli tablica plików i folderów znajdujących się w danym folderze. Dzięki funkcji **get\_chilren()**  mamy stały dostęp do potomków folderu.

class Folder(Node):

    def \_\_init\_\_(self,name):

        self.\_\_name = name

        self.\_\_children = []

    def create\_tree(self):

        print("", self.\_\_name)

        children = self.get\_children()

        for child in children:

            print(" |\_\_",child.get\_name())

            next\_node\_children =  child.get\_children()

            if next\_node\_children !=[]:

                for child in next\_node\_children:

                    print("     |\_",child.get\_name())

    def get\_name(self):

        return self.\_\_name

    def ls(self):

        children = self.get\_children()

        for child in children:

            print(child.get\_name())

    def add\_node(self,node):

        self.\_\_children.append(node)

    def remove\_node(self,node):

        node\_children = node.get\_children()

        if node\_children == []:

            self.\_\_children.remove(node)

        else:

            node\_children.clear()

            self.\_\_children.remove(node)

    def get\_children(self):

        return self.\_\_children

Klasy, które zaprezentowałam powyżej były projektowane przy pomocy wzorca Kompozytu.

* Ostatnia klasa to **CheckName** dzięki której weryfikowałam, czy podany plik bądź folder ma poprawną nazwę. Została wykorzystana w celu stworzenia wzorca **Proxy.**

class CheckName(Node):

    def check\_name(self,name,type\_of, data = None):

        count = name.count(".")

        invalidcharacters= set(string.punctuation)

        invalidcharacters.remove(".")

        if any(char  in invalidcharacters  for char in name ) or count>=2:

            print ("Name of " ,name, " is invalid! Can't create object.")

            return 0

        else:

            print ("Name of " ,name, " is valid!")

            if type\_of == 'file':

                return File(name,data)

            else:

                return Folder(name)

Przy tworzeniu nowego obiektu musimy określić czy chcemy stworzyć plik, czy folder.

* Folder musi zawierać nazwę
* Plik musi zawierać nazwę i może zawierać treść

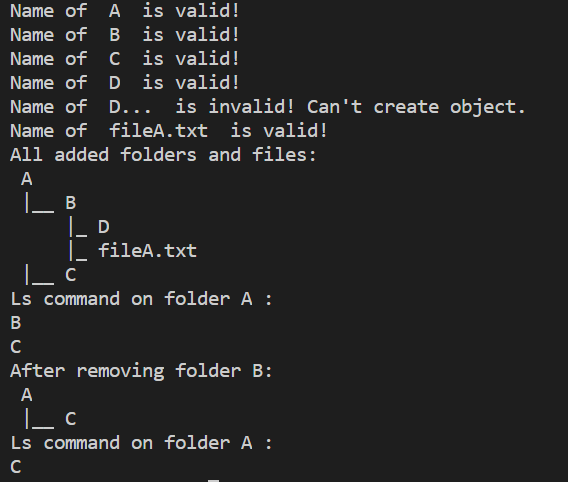
Funkcja **check\_name()** sprawdza, czy nazwa pliku bądź folderu jest prawidłowa, czyli czy na przykład nie zawiera znaków specjalnych albo więcej niż jedną kropkę (biorę pod uwagę pliki z rozszerzeniami). W zależności od tego co chcieliśmy stworzyć, jeśli nazwa jest poprawna, funkcja w klasie **ChceckName** oddelegowuje do innej klasy, aby stworzyć obiekt.

1. **Wnioski**

Stworzyłam killka folderów, w tym także podfoldery z plikiem.

Funkcja sprawdzająca poprawność nazwy wypisuje, czy nazwa jest zgodna z przyjętymi zasadami. Następnie jest rysowana struktura drzewa powstałych plików i folderów. Pozowliłam sobie przetestować również funkcję usuwania węzłów, która sprawdza, czy folder „miał dzieci” i jeśli tak to najpierw je usuwa, a potem dopoiero folder macierzysty. Drzewo rysuje się poprawnie, komenda ls również prawidłowo przedstawia stan folderów i plików.

Rezultaty w moim programie przedstawiają się następująco:



Największy problem miałam z implementacją drzewa i dodawaniem węzłów tak, żeby strutktura była prawidłowa. Mimo to myślę, że wykorzystanie drzewa w tym projekcie jest najlepszym rozwiązaniem.

Wykorzystanie wzorca Proxy pomogło mi w kontroli tworzenia obiektów plików i folderów. Gdy podana nazwa była zła to obiekt nie został stworzony.

Kompozycja zaś pozwoliła mi na korzystanie z pewnych operacji dla danego obiektu w taki sam sposób jaknbym to robiła dla grupy obiektów.