**Evaluarea automată a principilor de OOP**

**Testare unitară în Python**

În cadrul acestei lucrări, ne concentrăm pe dezvoltarea unei metode de evaluare automată (AA) pentru un domeniu care prezintă unele provocări în evaluarea sa practică: programarea orientată pe obiecte (OOP). Pentru a evalua corectitudinea aplicării principiilor OOP într-o aplicație reală, ne folosim de testarea unitară.

Accentul este pus pe încurajarea elevilor să înțeleagă și să aplice corect principiile OOP complexe, precum și pe capacitatea lor de a proiecta în mod adecvat clasele și relațiile dintre acestea. În acest sens, prezentăm un studiu de caz bazat pe o sarcină practică, în care elevii sunt solicitați să creeze personaje pentru un joc video. Această sarcină este remarcabilă prin faptul că este incrementală și abordează toate cele patru principii OOP într-un singur context.

Cele patru principii de OOP sunt:

1. *Încapsularea* constă în gruparea datelor și metodelor conexe într-o singură entitate (clasa). Încapsularea permite, de asemenea, ca codul să fie legat liber. Încapsularea permite modificarea datelor numai prin intermediul metodelor, ceea ce face codul mai robust.
2. *Abstracția* permite ascunderea membrilor, metodelor și implementării în interiorul clasei. Există trei cuvinte cheie asociate cu cât de ascunși sunt membrii clasei:

* *public* - dacă membrul clasei este declarat public, atunci acesta poate fi accesat oriunde.
* *protected* - dacă membrul clasei este declarat ca protected, atunci acesta poate fi accesat numai în cadrul clasei în sine și prin moștenirea claselor copil.
* *privat* - dacă membrul clasei este declarat ca privat, atunci acesta poate fi accesat numai de clasa care definește membrul.

1. *Moștenirea* este un principiu foarte util al OOP care construiește relații între clase. Acest lucru le permite să împărtășească membrii clasei și metodele. Moștenirea surprinde o relație „este-o” între clase. Vă permite să luați o clasă existentă și să o specializați și/sau să o extindeți
2. *Polimorfismul* înseamnă că o metodă poate face față diferitelor tipuri de intrări. Apoi, același cod poate fi aplicat mai multor tipuri de date. Există două tipuri de polimorfism: supraîncărcare și suprascriere.

* *Supraîncărcarea* permite unei clase să aibă mai multe metode cu același nume, dar un set diferit de parametri și implementare. De exemplu, unul foarte popular este operatorul + (de exemplu, pentru concatenarea a două șiruri).
* *Suprascrierea* are loc atunci când se înlocuiește o metodă moștenită cu o altă metodă având aceeași semnătură.

In continuare prezentăm un studiu de caz real al unei aplicații OOP — „implementarea personalor unui joc video, Supraviețuirea în Cosmos” — care aplică toate cele patru principii OOP pentru a rezolva acest singur scenariu.

**Clasa Alien**

Atribute:

* **name**: nume alien (ex: “Zyglorx”)
* **strength**: nivel de forta [0-5]
* **advancedTech**: are tehnologie avansata [True/False]

Metode asociate:

1. Contructor pentru initializare:

Ex: alien1 = Alient(“Zyglorx”, 3, true)

In Python, constructorul este o metodă specială numită \_\_init\_\_.

1. Proprietățile de accesare și modificare a valorilor tuturor:

Ex: alien1.name = “Xing”;

1. Metoda \_\_str\_\_, care este o metodă specială din Python care este utilizată pentru afișarea informațiilor unui obiect într-o reprezentare șir frumos formatată.

Ex: nume ani experinta viza de munca

Zyglorx 3 True

1. Supraîncărcarea operatorul > (\_\_gt\_\_) determină daca un extraterestu are mai multa putere in ceea ce priveste lupta decat un alt extraterestru. Extraterestrul ce detine o tehnologie avansata este mai puternic decat unul ce nu dispune de una. Daca amandoi dispun de tehnologie avansata , atunci o sa fie mai puternic extraterestrul cu forta mai mare.
2. Extraterestrul are o metoda numita, “fight” care se realizeaza cu un alt extraterestru. Daca un extraterestru este mai puternic decat altul, atunci castiga si puterea lui creste cu 1. Extraterestrul castigator este afisat pe ecran

Scopul nostru este sa implementan atat teste pentru gestionarea erorilor de tipuri si valori pentru atributele clasei, dar si pentru verifcarea tuturor functionalitatilor clasei Alien.

In aplicatia nostra nu avem doar clasa Alien, exista si alte case precum, clasele “**Astronaut**” si “**Commander**” ce au atat atributele nume si forta (cs si obiectele din clasa Alien), dar au si anumite atribute suplimentare:

* **planet**: planeta de origine
* **crew**: echipajul (comandantii au un atriput suplimentar ce este reprezentat de o lista de tip Astronaut).

Aceasta lista reprezinta astronoutii ce se afla sub comanda unui comandant. Nu exista o lista maxima de astronauti ce pot sa fie sub conducerea unui comandant, insa ei trebuie sa fie de pe aceeasi planeta.

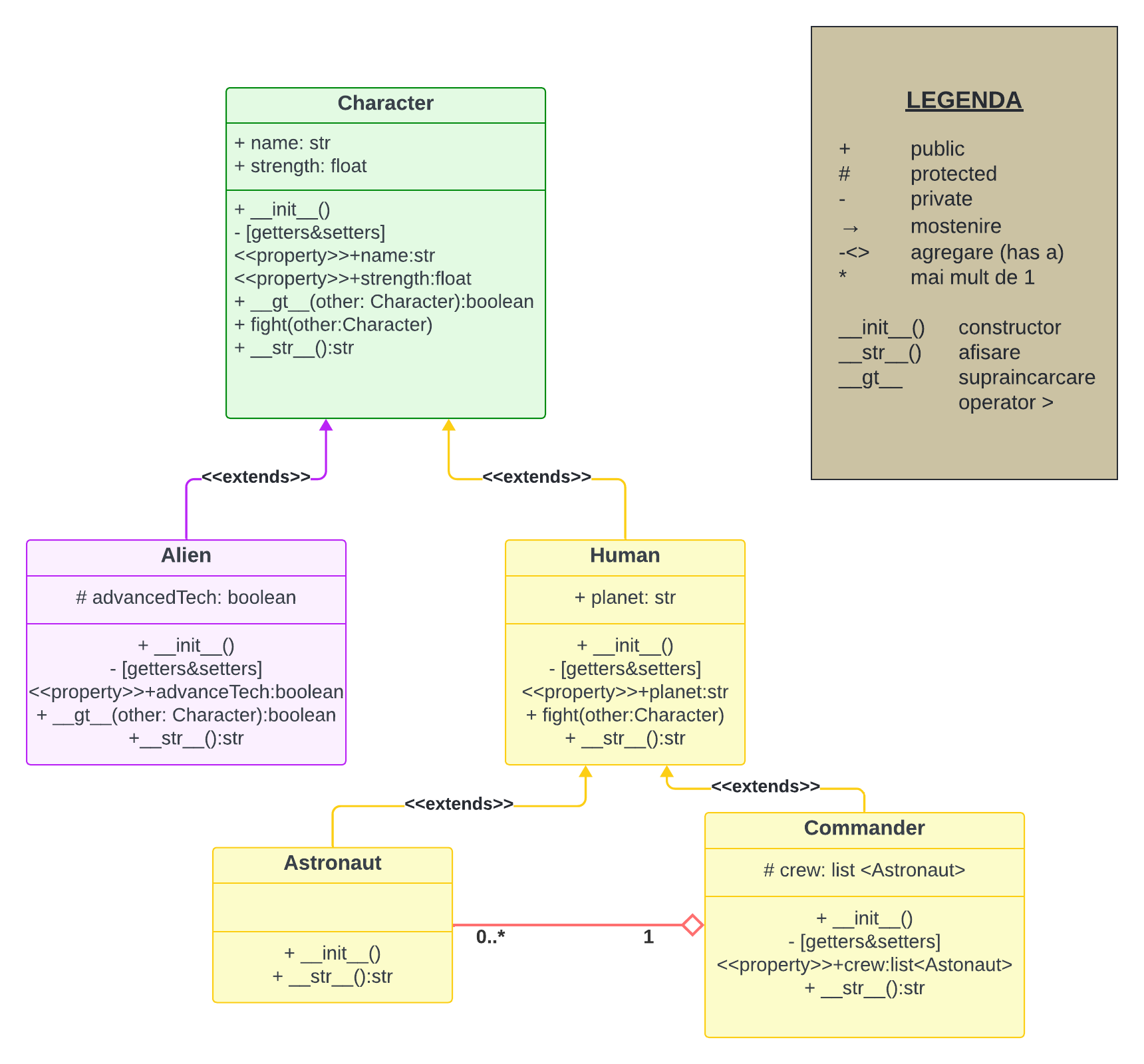
Astronautii si comandantii au aceleasi metode ca si Extragerestrii, insa exista si cateva diferente.

* Toate personajele se pot folosi de operatorul > cu orice alt tip de caracter, Nu trebuie sa uitam, insa, ca doar extraterestrii dispun de tehnologie avansata.
* Comandantii si astronautii se pot confrunta doar cu extraterestrii. Daca se incearca o lupta intre acestia se va afisa un mesaj de eroare.
* Mesajul \_str\_ returneaza reprezentarea in sir a instantelor in acest format. In cazul comandantilor, lista astronautilor se va afisa intre paranteze drepte.

Observam ca dispunem de trei clase Alien, Astronaut si Commander. Clasele Astrounaut si Commander au in comun planeta de provenienta si alte metode specializate(operatorul >). Pentru aceste doua clase putem sa formam o super clasa, aceaasta poate sa fie numita clasa **Human.**

Ceea ce am omis, este faptul ca toate clasele au doua propietati comune(nume si putere), dar si trei metode comune(>, lupta, \_str\_) ceea ce inseamna ca ar trebui sa existe o super clasa a tuturor claselor. Stabilim sa numim aceasta clasa **Character.**

**Diagrama UML**



Testare(descriere teste+bucati de cod)