.**GENERARE TESTE FOLOSIND AI**

**You**

Comp Applic In Engineering - 2023 - Climent - Automatic assessment of object oriented programming assignments with unit.pdf

PDF

deci am de facut proiect pentru facultate la testarea sistemelor software si am de reprodus un studiu legat de asta pe cae l am gasit eu pe net, iti voi lasa documentul pdf ca sa l studiezi, eu am incercat sa fac ceva similar cu ce au facut oamenii respectivi in studiul lor. am construit urmatoarele clase: class Character: def \_\_init\_\_(self, name, strength): if not isinstance(name, str): raise TypeError("Name should be a string") if not isinstance(strength, float) and not isinstance(strength, int): raise TypeError("Strength should be a int") self.\_name = name self.\_strength = max(0, min(strength, 5)) @property def name(self): return self.\_name @name.setter def name(self, value): if not isinstance(value, str): raise ValueError("Name must be a string.") self.\_name = value @property def strength(self): return self.\_strength @strength.setter def strength(self, value): if not isinstance(value, (int, float)): raise ValueError("Strength must be a number.") if value < 0 or value > 5: raise ValueError("Strength must be between 0 and 5.") self.\_strength = value def \_\_str\_\_(self): return f"{self.\_name} {self.\_strength}" def \_\_gt\_\_(self, other): return self.\_strength > other.strength def fight(self, other): if self > other: self.strength = min(self.strength + 1, 5) print(f"{self.name} wins and now has strength {self.strength}") else: print(f"{other.name} wins and now has strength {other.strength}") from models.character import Character class Alien(Character): def \_\_init\_\_(self, name, strength, advancedTech): if not isinstance(advancedTech, bool): raise ValueError("Test that providing a non-boolean type for advancedTech raises an error") super().\_\_init\_\_(name, strength) self.\_advancedTech = advancedTech @property def advancedTech(self): return self.\_advancedTech @advancedTech.setter def advancedTech(self, value): if not isinstance(value, bool): raise ValueError("advancedTech must be a boolean value.") self.\_advancedTech = value def \_\_str\_\_(self): return f"{self.name} {self.strength} {'with' if self.advancedTech else 'without'} advanced technology" def \_\_gt\_\_(self, other): if isinstance(other, Alien) and self.advancedTech and not other.advancedTech: return True elif isinstance(other, Alien) and self.advancedTech == other.advancedTech: return self.strength > other.strength return self.strength > other.strength from models.character import Character class Human(Character): def \_\_init\_\_(self, name, strength, planet): if not isinstance(planet, str): raise TypeError("Planet should be a string") super().\_\_init\_\_(name, strength) self.\_planet = planet @property def planet(self): return self.\_planet @planet.setter def planet(self, value): if not isinstance(value, str): raise ValueError("Planet must be a string.") self.\_planet = value def \_\_str\_\_(self): return f"{self.name} from {self.planet} with strength {self.strength}" def fight(self, other): if isinstance(other, Human): print("Humans cannot fight among each other.") raise ValueError("Crew members must be from the same planet.") else: super().fight(other) from models.human import Human class Astronaut(Human): def \_\_init\_\_(self, name, strength, planet): super().\_\_init\_\_(name, strength, planet) def \_\_str\_\_(self): return f"Astronaut {self.name} from {self.planet} with strength {self.strength}" from models.astronaut import Astronaut from models.human import Human class Commander(Human): def \_\_init\_\_(self, name, strength, planet): super().\_\_init\_\_(name, strength, planet) self.\_crew = [] @property def crew(self): return self.\_crew @crew.setter def crew(self, new\_crew): for member in new\_crew: if not isinstance(member, Astronaut) or member.planet != self.planet: raise ValueError("All crew members must be Astronauts from the same planet as the commander.") self.\_crew = new\_crew def add\_crew\_member(self, astronaut): if not isinstance(astronaut, Astronaut): raise ValueError("Only astronauts can be added to the crew.") if astronaut.planet != self.planet: raise ValueError("Crew members must be from the same planet.") self.\_crew.append(astronaut) def \_\_str\_\_(self): crew\_names = ', '.join([astronaut.name for astronaut in self.\_crew]) return f"Commander {self.name} from {self.planet} with strength {self.strength} and with crew [{crew\_names}]" eu ce vreau sa faci tu este sa studiezi pdf ul pe care ti l voi atasa si sa mi generezi teste unitare dupa modelul celor din studiu dar pentru clasele pe care ti le am dat eu

**ChatGPT**

The study you provided discusses using unit testing for automatic assessment of OOP principles in student programming assignments. It specifically uses Python and its **unittest** library to create and execute tests. Here are examples of unit tests based on the study that you can adapt to your classes:

**1. Testing Character class constructors and properties**

This includes tests for correct initialization, type checking, and range enforcement for the attributes of the **Character** class.

import unittest

from models.character import Character

class TestCharacter(unittest.TestCase):

def test\_character\_initialization(self):

char = Character("Hero", 4)

self.assertEqual(char.name, "Hero")

self.assertEqual(char.strength, 4)

def test\_character\_strength\_boundaries(self):

char = Character("Hero", 6)

self.assertEqual(char.strength, 5)

char.strength = -1

self.assertEqual(char.strength, 0)

def test\_invalid\_type\_name(self):

with self.assertRaises(TypeError):

Character(123, 3)

def test\_invalid\_type\_strength(self):

with self.assertRaises(TypeError):

Character("Hero", "strong")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**2. Testing Alien and Human subclasses**

Here, you can include tests to validate specific behaviors of these subclasses, such as the handling of additional attributes or overridden methods.

from models.alien import Alien

from models.human import Human

class TestAlienHuman(unittest.TestCase):

def test\_alien\_technology(self):

alien = Alien("Zorg", 3, True)

self.assertTrue(alien.advancedTech)

def test\_invalid\_alien\_technology\_type(self):

with self.assertRaises(ValueError):

Alien("Zorg", 3, "yes")

def test\_human\_planet(self):

human = Human("John Doe", 2, "Earth")

self.assertEqual(human.planet, "Earth")

def test\_invalid\_human\_planet\_type(self):

with self.assertRaises(TypeError):

Human("John Doe", 2, 404)

**3. Testing interactions and behaviors**

You can create unit tests for the interaction methods like **fight()** to ensure that characters behave correctly in simulated battles.

class TestCharacterInteraction(unittest.TestCase):

def test\_fight\_winner(self):

char1 = Character("Hero", 5)

char2 = Character("Villain", 3)

char1.fight(char2)

self.assertEqual(char1.strength, 5)

self.assertEqual(char2.strength, 3) # Assuming fight logic in description

def test\_fight\_strength\_increase(self):

char1 = Character("Hero", 4)

char2 = Character("Villain", 3)

char1.fight(char2)

self.assertEqual(char1.strength, 5) # Hero wins and gains strength

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

These tests should be a good starting point for building your test suite according to the principles discussed in the study you provided. They check for correct initialization, enforcement of attribute constraints, and proper method functioning, aligning with the study's methodology of using unit tests to validate OOP principles and class behaviors. Adjust the specific test logic based on the actual implementations and requirements of your **fight()** method or other class methods.

**Teste Generate de AI (ChatGPT)**

1. **Verificarea Atentă a Tipurilor de Date**: AI-ul generează teste care pun accent pe verificarea tipurilor de date și pe asigurarea corectitudinii acestora, cum ar fi testele pentru validarea că `**advancedTech**` este de tip boolean și că `**planet**` este un șir de caractere. Aceste teste sunt cruciale pentru a evita erorile de tip și pentru a asigura integritatea datelor în sistem.
2. **Simplu și Direct**: Testele AI sunt, în general, mai directe și focalizate pe cazuri specifice de eroare. Ele nu se aventurează în complexitatea interacțiunilor dintre metode și obiecte, concentrându-se mai degrabă pe validarea datelor de intrare și pe verificările de bază.

**Teste Generate Manual**

1. **Complexitate și Diversitate**: Testele create manual au o complexitate mai mare și explorează în profunzime comportamentul și interacțiunile dintre obiecte. De exemplu, testele de luptă verifică nu doar rezultatul final, ci și logica și consecințele intermediare, cum ar fi modificările de forță ale caracterelor în urma luptelor.
2. **Capturarea și Verificarea Output-ului**: Prin redirecționarea output-ului, testele manuale capturează și verifică mesajele afișate, oferind o perspectivă asupra experienței utilizatorului și a feedback-ului vizual.

**Concluzii și Diferențe**

* **Focus**: AI se concentrează pe corectitudinea datelor și pe validarea tipurilor, în timp ce abordarea manuală explorează comportamentul detaliat și interacțiunile complexe.
* **Complexitate**: AI păstrează teste simple și orientate spre cazuri specifice, în timp ce metoda manuală dezvoltă teste care pot aborda scenarii de utilizare variate și complexe.
* **Feedback și Interacțiune**: Testele AI sunt mai puțin preocupate de feedback-ul în timp real sau de interacțiunea vizuală, pe când testele manuale pot verifica și aceste aspecte prin capturarea output-ului.

**Utilizarea Complementară**

Ideal, utilizarea unei combinații de teste generate de AI și manuale poate oferi un echilibru între verificarea rapidă a corectitudinii datelor și explorarea profundă a logicii și interacțiunilor în software. AI poate accel

era procesul inițial de testare și identificarea rapidă a erorilor comune de tip sau valoare, în timp ce testele manuale pot extinde acoperirea pentru a include scenarii de utilizare complexe și feedback-ul utilizatorului. Acest echilibru maximizează eficiența în detectarea și corectarea erorilor, contribuind la dezvoltarea unui software robust și bine testat.

Pentru a valorifica la maximum potențialul acestor unelte, este important să se înțeleagă și să se ajusteze teste în funcție de nevoile specifice ale proiectului, îmbinând automatizarea AI cu insight-urile detaliate oferite de abordarea manuală. Aceasta combină eficiența cu profunzimea, asigurând o bază solidă pentru testarea și îmbunătățirea continuă a software-ului.