Report: Dynamic Typing & Duck Typing

Task 2

Advantage of Dynamic Typing

Example: The `speak` method in `Player` and `Enemy` classes

In the code, both the `Player` and `Enemy` classes have a `speak` method. This method outputs a random dialogue. The advantage of dynamic typing here is that the same method could theoretically be used for objects of different types, as long as they have a similar structure or attributes.

```
class Player(Plane):
    # ...
    def speak(self):
            # ...
class Enemy(Plane):
    # ...
    def speak(self):
            # ...
# Example usage:
if random.randint(0, 1) == 0:
    self._speaker = global_player
else:
    self._speaker = global_enemy
self._speaker.speak()
```

Explanation:

Dynamic typing here allows functions or methods to be applied on objects of different types without specific type declarations. It makes the code more flexible and reusable, as the `speak` method that can be used by different entities in the game.

Disadvantage of Dynamic Typing

Example: Bullet Movement and Collision Checking

In this program, the movement and collision checking of bullets can introduce issues due to dynamic typing. The `Bullet` class does not explicitly declare its type or the type of objects it interacts with. This can lead to errors. For example, if you accidentally pass a non-Plane object to the `is_collision` method, or if you modify the object attributes without realizing their impact on methods like `move`.

```
class Bullet:
    # ...
    def move(self):
        # ...
        # Dynamic typing can be a disadvantage here
        if self._bullet_type == BulletType.BUL_FROM_ENEMY:
            if global_player.is_collision(self._location):
                global_player.hit()
                # What if global_player is not a Plane obje
ct?
        else:
            if global_enemy.is_collision(self._location):
                global_enemy.hit()
                # What if global_enemy is not a Plane objec
t?
```

Explanation:

With dynamic typing, type checking is carried out at runtime. If an object is not of the expected type, errors will only be discovered during execution, leading to potential reliability issues. This lack of type safety can result in runtime errors that are hard to debug, especially in more complex systems where the data flow isn't straightforward.

Task 4

Scenario 1: Handling of `move()` Method for Different Elements

Python Implementation:

In Python, each object in the `global_elements` list can have its unique `move` method. The interpreter doesn't check the type at compile-time but instead

looks for the existence of the method at runtime.

```
# Each class defines its own move() method
class Player(Plane):
    def move(self): # Player movement logic
class Enemy(Plane):
    def move(self): # Enemy movement logic
class Gift:
    def move(self): # Gift movement logic
# Calling move() on each object
def move_all(self):
    for element in global_elements:
        self.move_one(element)
```

C++ Implementation:

In C++, you need to ensure that each class derived from `Element` implements the `move` method. This enforces a stricter structure, which can be beneficial for maintaining correctness but reduces flexibility.

```
// Derived classes must implement the move method
class Player : public Plane { void move() {// Player moveme
nt logic} };
class Enemy : public Plane { void move() {// Enemy movement
logic} };
class Gift : public Element { void move() {// Gift movement
logic} };
// Calling move() on each object
void move_all(){
    for (int i = 0; i < global_elements.size(); i++) { move
_one(global_elements[i]); }
}
```

Scenario 2: `Bullet` and `Plane` Interaction

Python Implementation:

In Python, duck typing allows us to interact with objects based on their behavior rather than their explicit type. This means that the `Bullet` class can call the `is_collision()` method of a plane object without explicitly checking the

type of the object. As long as the object has an `is_collision()` method that behaves correctly for the bullet, the interaction will work seamlessly.

```
class Bullet: # ...
    def move(self): # ...
        if self._bullet_type == BulletType.BUL_FROM_ENEMY:
            if global_player.is_collision(self._location): #
        else:
            if global_enemy.is_collision(self._location): # .
```

C++ Implementation:

In C++, we can achieve similar behavior using inheritance and virtual functions. We define a base class `Element` with a virtual function `isCollision()`, which is overridden in derived classes `Player` and `Enemy`. Then, the `Bullet` class can interact with any object of type `Element` without needing to know the specific derived type.

```
void Bullet::move() { // ...
    if (bullet_type == BUL_FROM_ENEMY) {
        if (global_player->is_collision(location)) {
            global_player->hit();
            set_validity(false); }
    } else {
        if (global_enemy->is_collision(location)) {
            global_enemy->hit();
            set_validity(false); }
    } // ...
}
```

Duck Typing Advantage: Python's approach provides flexibility and convenience by focusing on object behavior rather than explicit types. This approach simplifies interfaces, promotes rapid prototyping, reduces boilerplate code, and encourages composition over inheritance. It's particularly valuable in dynamically typed languages like Python for quick iteration and handling evolving codebases efficiently.