ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

Pruebas Para Detección de Antipatrones

Nombres: Kevin Castillo, Josué Merino, Josué Moreno

Patrón

Un patrón en el desarrollo de software es una solución probada y recomendada para un problema común que se encuentra en el diseño o la implementación de un sistema. Los patrones proporcionan un enfoque estructurado y reutilizable para resolver problemas recurrentes, ayudando a mejorar la calidad, la flexibilidad y el mantenimiento del software. Los patrones pueden ser específicos del lenguaje de programación o independientes del lenguaje.

Antipatrón

Un antipatrón, también conocido como patrón de diseño negativo, es una solución común pero ineficiente, incorrecta o inapropiada para un problema en el desarrollo de software. Los antipatrones son situaciones o prácticas que se consideran contraproducentes o problemáticas y que pueden conducir a problemas de rendimiento, mantenibilidad y escalabilidad. Identificar y evitar los antipatrones ayuda a mantener la calidad y la eficiencia del software.

- Feature Envy es un antipatrón en el que un objeto o clase accede demasiado a los datos o métodos de otro objeto. Esto puede conducir a un diseño deficiente, acoplamiento excesivo y código difícil de mantener. Se resuelve reorganizando la lógica para que cada objeto acceda y utilice sus propios datos y métodos de manera más coherente y cohesiva.

Caso 1:

Es un código de Python , donde existen 2 clases, la clase “Customer” con los atributos: name, address y la clase “Address” con los atributos: street, city, country. En la clases Customer existe la función __init__ y en la clase “Address” las funciones: __init__ y get_full_address, que devuelve la calle la ciudad y el país.

Caso 2:

A diferencia del caso 1, la función “get_full_address” se encuentra en la clase “Customer” en vez de en la clase “Address”. El código de uso crea una instancia de la clase Address con la

calle "123 Main Street", ciudad "Cityville" y país "Countryland". Luego, crea una instancia de la clase Customer con el nombre "John Doe" y la dirección previamente creada. Finalmente, llama al método get_full_address en la instancia de customer y guarda el resultado en la variable full_address. El resultado se imprime en la consola.

Caso3:

La clase ShoppingCart representa un carrito de compras y tiene una lista llamada ítems para almacenar los artículos agregados al carrito. La clase Ítem representa un artículo y tiene tres atributos: name, price and quantity (.

El método __init__ en la clase ShoppingCart se utiliza para inicializar la lista de ítems, que comienza como una lista vacía.

El método add_item en la clase ShoppingCart recibe un objeto y lo agrega a la lista ítems del carrito.

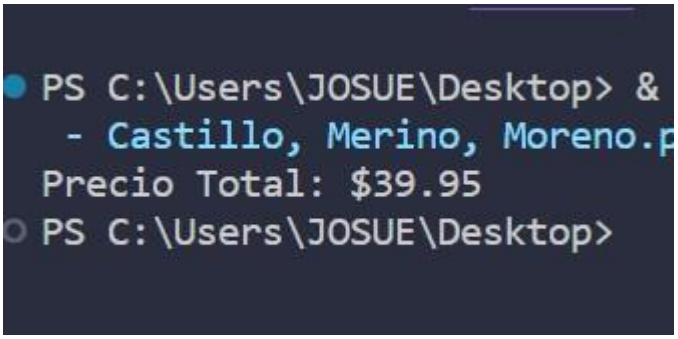
El método calculate_total_price en la clase ShoppingCart calcula el precio total de todos los artículos en el carrito. Multiplica el precio (price) del artículo por la cantidad (quantity) y suma el resultado a total. Al final, devuelve el valor total.

El código de uso crea una instancia de la clase ShoppingCart llamada cart. Luego, se llaman los métodos add_item en cart dos veces, pasando objetos Item con diferentes nombres, precios y cantidades. Después, se llama al método calculate_total_price en cart y el resultado se guarda en la variable total_price. Finalmente, se imprime en la consola el mensaje "Total Price:" seguido del valor de total_price.

Código Realizado:

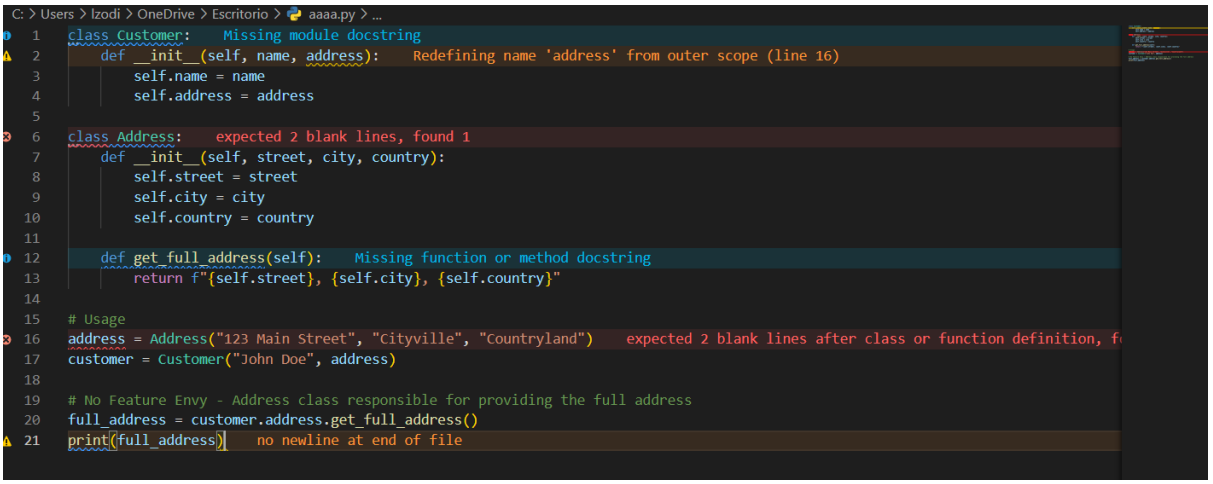
```python
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, item):
        self.items.append(item)

    def calculate_total_price(self):
        total = 0
        for item in self.items:
            total += item.calcular_precio()
        return total

class Item:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def calcular_precio(self):
        return self.price*self.quantity


cart = ShoppingCart()
cart.add_item(Item("Product 1", 10.99, 2))
cart.add_item(Item("Product 2", 5.99, 3))


total_price = cart.calculate_total_price()
print(f"Precio Total: ${total_price}")
```

Ejecución

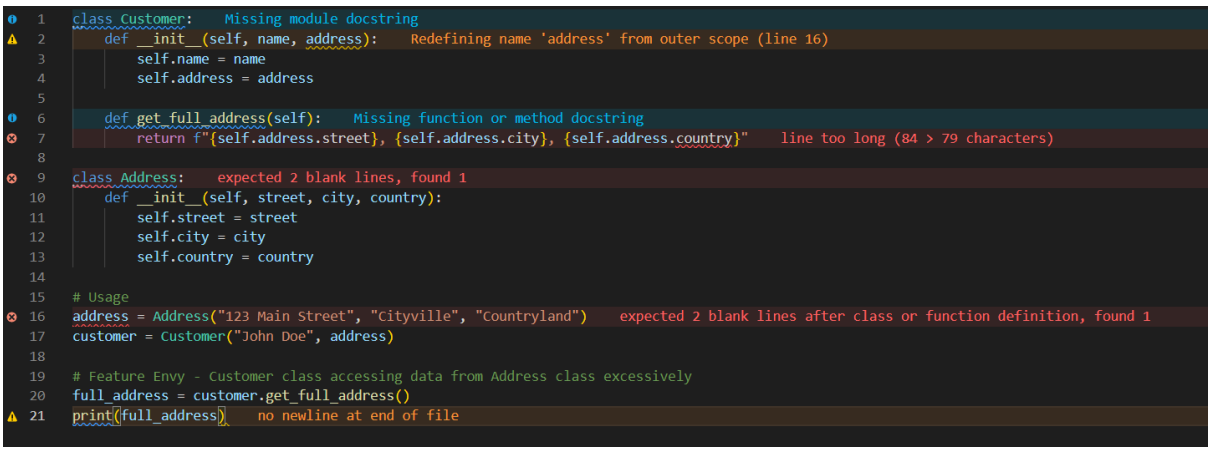


Resultados de uso de la herramienta Python Linting:

Código 1:



Código 2:



Código 3 (Solución Propia):

C: > Users > lzodi > OneDrive > Escritorio > aaaa.py > ...

```python
class ShoppingCart:    Missing module docstring
    def _init_(self):
        self.items = []    Attribute 'items' defined outside __init__

    def add_item(self, item):    Missing function or method docstring
        self.items.append(item)

    def calculate_total_price(self):    Missing function or method docstring
        total = 0
        for item in self.items:
            total += item.calcular_precio()
        return total

class Item:    expected 2 blank lines, found 1
    def _init_(self, name, price, quantity):
        self.name = name    Attribute 'name' defined outside __init__
        self.price = price    Attribute 'price' defined outside __init__
        self.quantity = quantity    Attribute 'quantity' defined outside __init__

    def calcular_precio(self):    Missing function or method docstring
        return self.price*self.quantity

# Usage
cart = ShoppingCart()    expected 2 blank lines after class or function definition, found 1
cart.add_item(Item("Product 1", 10.99, 2))
cart.add_item(Item("Product 2", 5.99, 3))

# Feature Envy - ShoppingCart class accessing data from Item class excessively
total_price = cart.calculate_total_price()
print(f"Precio Total: ${total_price}")    no newline at end of file
```

Código 4 (Original):

```
1  public class ShoppingCart {    SyntaxError: Unexpected token 'class'
2      private List<Item> items;
3
4      public ShoppingCart() {
5          items = new ArrayList<>();
6      }
7
8      public void addItem(Item item) {
9          items.add(item);
10     }
11
12     public double calculateTotalPrice() {
13         double total = 0;
14         for (Item item : items) {    Statements must be separated by newlines or semicolons
15             total += item.getPrice() * item.getQuantity();
16         }
17         return total;
18     }
19 }
20
21 public class Item {    Statements must be separated by newlines or semicolons
22     private String name;
23     private double price;
24     private int quantity;
25
26     public Item(String name, double price, int quantity) {
27         this.name = name;
28         this.price = price;
29         this.quantity = quantity;
30     }
31
32     public String getName() {
33         return name;
34     }
35
36     public double getPrice() {
37         return price;
38     }
```

```
39
40     public int getQuantity() {
41         return quantity;
42     }
43 }
44
45 // Usage    Expected expression
46 ShoppingCart cart = new ShoppingCart();    Statements must be separated by newlines or semicolons
47 cart.addItem(new Item("Product 1", 10.99, 2));    "(" was not closed
48 cart.addItem(new Item("Product 2", 5.99, 3));    "(" was not closed
49
50 // Feature Envy - ShoppingCart class accessing data from Item class excessively    Expected expression
51 double total = cart.calculateTotalPrice();    Statements must be separated by newlines or semicolons
52 System.out.println("Total Price: $" + total);    Statement ends with an unnecessary semicolon
```

Codigo4 (Solución Propia)

```java
import java.util.ArrayList;
import java.util.List;
import caso.pkg4.Item;    The import caso.pkg4.Item is never used

public class ShoppingCart {
    private List<Item> items;

    public ShoppingCart() {
        items = new ArrayList<>();
    }

    public void addItem(Item item) {
        items.add(item);
    }

    public double calculateTotalPrice() {
        double total = 0;
        for (Item item : items) {
            total += item.getTotalPrice();
        }
        return total;
    }

    Run | Debug
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem(new Item(name:"Product 1", price:10.99, quantity:2));
        cart.addItem(new Item(name:"Product 2", price:5.99, quantity:3));

        double total = cart.calculateTotalPrice();
        System.out.println("Total Price: $" + total);
    }
}
```

CONCLUSIONES

1. Organización y reutilización del código: Los códigos de ejemplo muestran cómo organizar la funcionalidad en clases y cómo utilizar objetos para representar entidades del dominio, como carritos de compras y elementos. Este enfoque permite una mayor modularidad y reutilización del código, ya que puedes crear múltiples instancias de las clases y utilizar métodos para realizar acciones específicas. Entender cómo las clases y los objetos interactúan entre sí es fundamental en la POO.

2. Abstracción y encapsulación: Los códigos utilizan conceptos de abstracción y encapsulación para ocultar los detalles internos de las clases y exponer solo la interfaz

necesaria. Por ejemplo, en la clase `Item`, los atributos `name`, `price` y `quantity` se mantienen privados y solo se acceden a través de métodos getters. Esto ayuda a mantener la integridad de los datos y permite cambios internos en la implementación sin afectar el código que utilizan las clases.

3. Análisis de código y depuración: Al revisar los códigos, es importante comprender cómo utilizar herramientas de análisis de código y depuración para identificar fallas o inconsistencias. Las herramientas como compiladores, linters y depuradores son fundamentales para detectar y corregir errores en el código. Al utilizar estas herramientas, se puede mejorar la calidad y la confiabilidad del código, asegurándose de que funcione correctamente y siga las mejores prácticas.