Virtual Library Management System

Elisabet Arelly Sulú Vela
Data Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: 2309212@upy.edu.mx

Karen Cardiel Olea
Data Engineering
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: 2209039@upy.edu.mx

Ernesto Manuel Ihuit Dzib
Universidad Politécnica de Yucatán
Km. 4.5. Carretera Mérida — Tetiz
Tablaje Catastral 4448. CP 97357
Ucú, Yucatán. México
Email: ernesto.ihuit@upy.edu.mx

Abstract

This document presents the development of a Virtual Library Management System in Python. The system facilitates user interaction with publications, managing loans, inventory, and notifications. Utilizing object-oriented programming principles, it incorporates classes, inheritance, polymorphism, abstraction, and encapsulation. With a class hierarchy featuring at least six attribute inheritances and four abstract classes, the system ensures efficient management of library operations, providing users with a seamless experience.

Index Terms

Virtual, Library, Python, Encapsulation, Inheritence, Polymorphism, Abstraction, Usuario, Publicacion, Prestamo, Notificacion, Evento

Virtual Library Management System

I. Introduction

THE digitalization of libraries makes it easier to access information and manage resources. The project of this unit II aims to create a Virtual Library Management System that improves user interaction with various publications. The system allows users to manage loans, access inventory, and receive notifications.

The system is built using object-oriented programming, which helps organize the code. It has a class hierarchy with shared attributes, making it modular. By using polymorphism and defining abstract classes, the system provides a strong framework for managing data. This document explains the system's structure and functions, showing its ability to adapt in a changing digital environment.

II. Object-Oriented Programming

In Object-Oriented Programming (OOP) we have two important concepts to be defined: *class* and *object*. A class is a set of properties that is common to certain types of objects. For example, we can define a class that will cover all the dogs: animals with well-developed noses, four legs, a tail, and bark. The objects of this class are the dogs, which have in common all these characteristics but can have specific characteristics themselves, such as the color of the furr, the size, etc [1].

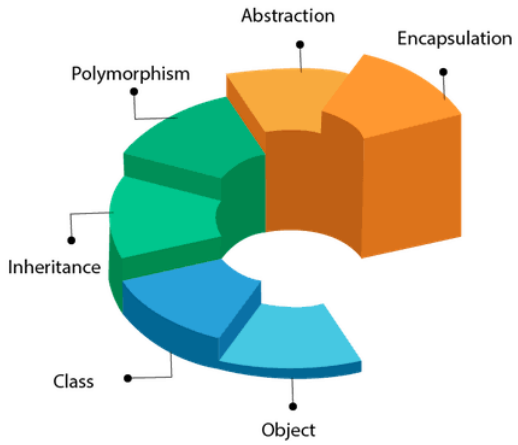

Fig. 1. OOPs (Object-Oriented Programming System). [2]

III. OOP Principles In The Code Of Our Virtual Library

The following is a brief definition of each fundamental principle of object-oriented programming (encapsulation, inheritance, polymorphism and abstraction). We will show how each of them was used in the code of our virtual library, as well as an explanation of how they were implemented in this library.

A. Encapsulation:

Encapsulation is the practice of restricting access to an object's data and methods, keeping them together in a unit to prevent external interference. In Python, this is achieved by using a double underscore prefix (__) to make variables private.

In the code, encapsulation has been applied in 5 different ways:

- `Usuario`: The attributes `_nombre` and `_id_usuario` are encapsulated within the class and can only be accessed through the methods `__init__` and `solicitar_prestamo`.
- `Publicacion`: The attributes `_titulo`, `_autor`, and `_disponible` are encapsulated within the class and can only be accessed through the methods `__init__`, `prestar`, `devolver`, and `obtener_informacion`.
- `Prestamo`: The attributes `_usuario` and `_publicacion` are encapsulated within the class and can only be accessed through the methods `__init__` and `calcular_tarifa`.
- `Evento`: The attributes `_nombre` and `_detalles` are encapsulated within the class and can only be accessed through the methods `__init__` and `registrar_asistencia`.

```
class Usuario(ABC):
    def __init__(self, nombre, id_usuario):
        self._nombre = nombre  # Atributo
            encapsulado
        self._id_usuario = id_usuario  #
            Atributo encapsulado

    def solicitar_prestamo(self, prestamo):  #
         Metodo que utiliza los atributos
        encapsulados
        return f"{self._nombre} ha solicitado
            el prestamo {prestamo._publicacion
            .obtener_informacion()}"

class Publicacion(ABC):
    def __init__(self, titulo, autor):
        self._titulo = titulo  # Atributo
            encapsulado
        self._autor = autor  # Atributo
            encapsulado
        self._disponible = True  # Atributo
            encapsulado

    def obtener_informacion(self): # Metodo
        que utiliza los atributos encapsulados
        return f"Titulo: {self._titulo}, Autor
            : {self._autor}"
```

In this example, the attributes `_nombre`, `_id_usuario`, `_titulo`, `_autor`, and `_disponible` are encapsulated

within the `Usuario` and `Publicacion` classes, respectively. These attributes can only be accessed through the methods defined in the classes.

B. Inheritance:

Inheritance allows an object to inherit the methods and properties of another object. This creates a relationship between a "father class" and a "child class," promoting code reuse. In the code, inheritance is applied in the following classes:

- `EstudianteDeSecundaria`, `EstudianteDePreparatoria`, `EstudianteDeUniversidad`, `EstudianteDePosgrado`, `EstudianteDeIntercambio`, `Investigador`, `Administrativo`, `Bibliotecario`, `AdultoINAPAM`, and `AdultoComun` inherit from the `Usuario` class.
- `Libro`, `Revista`, `Novela`, `Comic`, `Diccionario`, `Manual`, `Tesis`, `Enciclopedia`, `Articulo`, and `Boletin` inherit from the `Publicacion` class.
- `PrestamoDiario`, `PrestamoSemanal`, `PrestamoQuincenal`, `PrestamoMensual`, `PrestamoFinSemana`, `PrestamoVacaciones`, `PrestamoEstacional`, `PrestamoExtraordinario`, `PrestamoEmergencias`, and `PrestamoLargoPlazo` inherit from the `Prestamo` class.
- `NotificacionCorreo`, `NotificacionSMS`, `NotificacionWhatsApp`, `NotificacionRedesSociales`, `NotificacionLlamada`, `NotificacionCarta`, `NotificacionTablonAnuncios`, `NotificacionMensajeria`, `NotificacionSistemaPrestamos`, and `NotificacionMensajeVoz` inherit from the `Notificacion` class.
- `EventoDeLanzamientoDeLibro`, `EventoDeFirmaDeLibros`, `EventoDeLectura`, `EventoDeClubDeLectura`, `EventoDeDescuentos`, `EventoDeTallerDeEscritura`, `EventoDePresentacionDeLibro`, `EventoDeFeriaDelLibro`, `EventoDeLecturaDeCuentosInfantiles`, and `EventoDeEncuentroConElAutor` inherit from the `Evento` class.

This is an example of how inheritance is reflected in our code:

```
    #---------- PUBLICACION ------------

from abc import ABC, abstractmethod

class Publicacion(ABC):
    def __init__(self, titulo, autor):
        self._titulo = titulo
        self._autor = autor
        self._disponible = True

    @abstractmethod
    def consultar_disponibilidad(self):
        pass
    def prestar(self):
        self._disponible = False
    def devolver(self):
        self._disponible = True
        return f"La publicacion '{self._titulo}' de {self._autor} ha sido devuelta"
    def obtener_informacion(self):
        return f"Titulo: {self._titulo}, Autor: {self._autor}"

class Libro(Publicacion):
    def __init__(self, titulo, autor):
        super().__init__(titulo, autor)
    def consultar_disponibilidad(self):
        if self._disponible:
          return f"El libro '{self._titulo}' de {self._autor} esta disponible"
        else:
          return f"El libro '{self._titulo}' de {self._autor} no est  disponible"

class Revista(Publicacion):
    def __init__(self, titulo, autor):
        super().__init__(titulo, autor)
    def consultar_disponibilidad(self):
        return f"La revista '{self._titulo}' de {self._autor} esta disponible"

class Novela(Publicacion):
    def __init__(self, titulo, autor):
        super().__init__(titulo, autor)
    def consultar_disponibilidad(self):
        return f"La novela '{self._titulo}' de {self._autor} esta disponible"
```

In this code, we have a base class `Publicacion` that has several methods and attributes. The classes `Libro`, `Revista`, `Novela`, are subclasses of `Publicacion`. They inherit the methods and attributes of `Publicacion` and can also add new methods or override the ones inherited from `Publicacion`.
For example, the `Libro` class overrides the `consultar_disponibilidad` method to provide a more specific message for books. The `Revista` class also overrides this method, but with a different message. This is an example of inheritance, where a subclass inherits the properties and behavior of a parent class and can also add new properties or behavior or override the ones inherited from the parent class.

C. Polymorphism:

The principle allows different classes to use the same method name, but implement it in unique ways.

Polymorphism is present in the following way in the code:

- The `Usuario` class has a method `solicitar_prestamo` that is overridden in the child classes.
- The `Publicacion` class has a method `consultar_disponibilidad` that is overridden in the child classes.
- The `Prestamo` class has a method `calcular_tarifa` that is overridden in the child classes.
- The `Notificacion` class has a method `enviar_notificacion` that is overridden in the child classes.
- The `Evento` class has a method `registrar_asistencia` that is overridden in the child classes.

In the `Notificacion` class, you have an abstract method `enviar_notificacion` that is implemented by different subclasses, such as `NotificacionCorreo`, `NotificacionSMS`, `NotificacionWhatsApp`, etc.

```
    class Notificacion(ABC):
    def __init__(self, mensaje, usuario):
        self._mensaje = mensaje
        self._usuario = usuario

    @abstractmethod
    def enviar_notificacion(self):
        pass
```

In the `Biblioteca` class, we have a method `enviar_notificacion` that takes a `Notificacion` object as an argument and calls the `enviar_notificacion` method on it.

```
    class Biblioteca:
    # ...

    def enviar_notificacion(self, notificacion
        ):
        return notificacion.
            enviar_notificacion()
```

This is an example of polymorphism because the `enviar_notificacion` method in the `Biblioteca` class can work with objects of different classes that implement the `Notificacion` interface.

```
    notificacion_correo = NotificacionCorreo(
        mensaje, usuario)
notificacion_sms = NotificacionSMS(mensaje,
    usuario)

print(biblioteca.enviar_notificacion(
    notificacion_correo))  # Output: Enviando
    notificaci n por correo electr nico a
    usuario: mensaje
print(biblioteca.enviar_notificacion(
    notificacion_sms))  # Output: Enviando
    notificaci n por SMS a usuario: mensaje
```

D. Abstraction:

Abstraction helps create simple models of complex systems by focusing on common features, like wheels or engines, and ignoring details like the number of doors.

- The `Usuario` class is abstract and defines an abstract method `solicitar_prestamo`.
- The `Publicacion` class is abstract and defines an abstract method `consultar_disponibilidad`.
- The `Prestamo` class is abstract and defines an abstract method `calcular_tarifa`.
- The `Notificacion` class is abstract and defines an abstract method `enviar_notificacion`.
- The `Evento` class is abstract and defines an abstract method `registrar_asistencia` that is overridden in the child classes.

```
class Notificacion(ABC):
    def __init__(self, mensaje, usuario):
        self._mensaje = mensaje
        self._usuario = usuario

    @abstractmethod
    def enviar_notificacion(self):
        pass
```

In this example, we defined an abstract class `Notificacion` with an abstract method `enviar_notificacion`. This method is implemented by the subclasses `NotificacionCorreo`, `NotificacionSMS`, `NotificacionWhatsApp`, etc.

Another example:

```
    class Evento(ABC):
    def __init__(self, nombre, detalles):
        self._nombre = nombre
        self._detalles = detalles

    @abstractmethod
    def registrar_asistencia(self, usuario):
        pass
```

In this example, we defined an abstract class `Evento` with an abstract method `registrar_asistencia`. This method is implemented by the subclasses `EventoDeLanzamientoDeLibro`, `EventoDeClubDeLectura`, `EventoDeFeriaDelLibro`, etc.

IV. Results

The virtual library management system was successfully developed, implementing all the required functionalities. Five main classes were built (`Usuarios`, `Publicaciones`, `Prestamo`, `Notificacion` y `Evento`) with a total of 50 subclasses, and 5 abstraction classes, allowing a wide variety of interactions between components.

V. Conclusion

Although this project became more extensive as we added classes and subclasses, it helped us a lot to better understand the pillars of object-oriented programming. Practically applying concepts such as inheritance, polymorphism, abstraction and encapsulation allowed us to see how they are used in real situations. It also motivated us to do more research on how to improve the code and look for better solutions to get the results we wanted.

Although we had some stumbles, especially when it came to getting the main classes and their subclasses to interact correctly, teamwork and constant communication were key to solving problems and, thus, allowing us to find solutions and move forward.

REFERENCES

[1] N.D. Gomes, "Object-oriented programming with Python: core principles and examples," Medium, Sep. 11, 2022. [Online]. Available: https://naomy-gomes.medium.com/object-oriented-programming-with-python-core-principles-and-examples-ca64473735a8. [Accessed: Oct. 10, 2024].

[2] F. Lelli, "Object Oriented Programming: A curated set of resources," Aug. 2, 2019. [Online]. Available: https://francescolelli.info/tutorial/object-oriented-programming-a-curated-set-of-resources/. [Accessed: Oct. 10, 2024].

PROMPTS USED

In this section we add prompts used for the improvement or correction of our code.

"Me gustaría que después de que un libro se preste con la opción 3 del menu principal de la biblioteca, al consultar la opción 6 del menu principal de la biblioteca, el mismo libro aparezca como no disponible"
"Si agrego un usuario con la opción 1 del menú, agrego un libro con la opción 2 del menú, solicito un prestamo con la opción 3, y luego consulto la disponibilidad del mismo libro con la opción 6 del menú, sigue apareciendo disponible. Por qué no cambia la disponibilidad del libro que ya ha sido prestado?"
"Cada que utilizo la opción 3 del menú, imprime que el usuario o la publicación no se encuentra, a pesar de que acaban de ser agregadas. Cuál es el error que hace que siempre imprima la misma leyenda?"