solid_soloution

April 22, 2024

1 Tarea de Principios SOLID

Desarrolle un ejemplo suyo (no copiado de otro lado) en el cual presente la aplicación de todos y cada uno de los principios de diseño **SOLID**.

Puede ser un solo ejemplo aplicando todo los conceptos, o bien, un ejemplo distinto para cada principio.

Lo importante es que cada ejemplo presente lo siguiente: - Código SIN aplicar el respectivo principio - Código APLICANDO el respectivo principio - Justificación del problema sin aplicar el principio, y justificación de lo en la aplicación del principio.

Los principios a utilizar son:

Single Responsibility Principle **(SRP)** o Principio de Responsabilidad Única

Open-Closed Principle **(OCP)** o Principio de Abierto/Cerrado

Liskov Substitution Principle **(LSP)** o Principio de Substitución de Liskov

Interface Segregation Principle **(ISP)** o Principio de Segregación de Interfaz

Dependency Inversion Principle **(DIP)** o Principio de Inversión de Dependencias

1.1 Autor

Carlos Andrés Mata Calderón - 2019033834

1.2 Solución

1.2.1 Single Responsibility Principle (SRP)

Crear una clase que gestione usuarios en un sistema.

Código SIN aplicar SRP En este ejemplo, una única clase UserManager se encarga de manejar datos del usuario, autenticar al usuario y presentar la información del usuario.

[11]:
```python
class UserManager:
    def __init__(self, user_data):
        self.user_data = user_data

    def change_user_password(self, new_password):
        self.user_data["password"] = new_password
```

```
        print("Password changed.")

    def authenticate_user(self, username, password):
        if username == self.user_data["username"] and password == self.
 ↪user_data["password"]:
            print("User authenticated.")
            return True
        else:
            print("Authentication failed.")
            return False

    def display_user(self):
        print(f"User: {self.user_data['username']}")
        print(f"Email: {self.user_data['email']}")
```

Código APLICANDO SRP Se dividen las responsabilidades en tres clases distintas, para cada una de las responsabilidades.

[12]:
```
class UserDataManager:
    def __init__(self, user_data):
        self.user_data = user_data

    def change_user_password(self, new_password):
        self.user_data["password"] = new_password
        print("Password changed.")

class UserAuth:
    def __init__(self, user_data):
        self.user_data = user_data

    def authenticate_user(self, username, password):
        if username == self.user_data["username"] and password == self.
 ↪user_data["password"]:
            print("User authenticated.")
            return True
        else:
            print("Authentication failed.")
            return False

class UserDisplay:
    def __init__(self, user_data):
        self.user_data = user_data

    def display_user(self):
        print(f"User: {self.user_data['username']}")
        print(f"Email: {self.user_data['email']}")
```

Justificación En el primer caso, UserManager tiene múltiples razones para cambiar (datos, autenticación, presentación), lo que viola SRP. Separando estas responsabilidades, cada clase tiene una única razón para cambiar, mejorando la mantenibilidad y la escalabilidad del código.

1.2.2 Open-Closed Principle (OCP)

Sistema de procesamiento de documentos.

Código SIN aplicar La clase DocumentProcessor tiene métodos específicos para cada tipo de documento. Cada vez que se necesita añadir soporte para un nuevo tipo de documento, se modifica esta clase.

[13]:
```python
class DocumentProcessor:
    def processWord(self, content):
        print(f"Processing Word document with content: {content}")

    def processPDF(self, content):
        print(f"Processing PDF document with content: {content}")

    def processExcel(self, content):
        print(f"Processing Excel spreadsheet with content: {content}")
```

Código APLICANDO OCP Se introduce una interfaz DocumentHandler y se implementan clases específicas para cada tipo de documento. La clase DocumentProcessor no necesita saber detalles sobre cómo se procesa cada formato; solo necesita interactuar con la interfaz DocumentHandler:

[14]:
```python
from abc import ABC, abstractmethod

class DocumentHandler(ABC):
    @abstractmethod
    def processDocument(self, content):
        pass

class WordHandler(DocumentHandler):
    def processDocument(self, content):
        print(f"Processing Word document with content: {content}")

class PDFHandler(DocumentHandler):
    def processDocument(self, content):
        print(f"Processing PDF document with content: {content}")

class ExcelHandler(DocumentHandler):
    def processDocument(self, content):
        print(f"Processing Excel spreadsheet with content: {content}")

class DocumentProcessor:
    def __init__(self):
```

```
        self.handlers = {
            "word": WordHandler(),
            "pdf": PDFHandler(),
            "excel": ExcelHandler()
        }

    def processDocument(self, doc_type, content):
        handler = self.handlers.get(doc_type)
        if handler:
            handler.processDocument(content)
        else:
            print("Document type not supported.")
```

Justificación En el código sin OCP, cualquier adición de un nuevo formato de documento requiere cambios en la clase DocumentProcessor, lo que viola el principio de abierto/cerrado. Usando la interfaz DocumentHandler, el sistema puede extenderse fácilmente para nuevos formatos sin modificar el código existente, manteniendo la clase DocumentProcessor cerrada a modificaciones.

1.2.3 Liskov Substitution Principle (LSP)

Sistema de formas geométricas para calcular el área.

Código SIN aplicar LSP La clase base Shape proporciona un método calculateArea() que no se adapta adecuadamente para todas las formas derivadas, especialmente cuando los parámetros necesarios para el cálculo difieren entre formas

[15]:
```
class Shape:
    def calculateArea(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculateArea(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculateArea(self):
        return 3.14 * self.radius * self.radius
```

Código APLICANDO LSP La estructura para asegurar que Shape sea una interfaz adecuada para todas sus formas derivadas y que el método calculateArea sea implementado de manera que

las instancias de Shape puedan ser sustituidas sin necesidad de conocer el tipo específico de forma

[16]:
```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculateArea(self) -> float:
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculateArea(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculateArea(self):
        return 3.14 * self.radius * self.radius
```

Justificación En el enfoque inicial, no se puede usar Shape de manera intercambiable sin conocer su tipo específico, lo que viola LSP. Asegurando que calculateArea sea implementado apropiadamente en cada subclase, las instancias de la clase base pueden ser sustituidas por instancias de cualquier subclase.

1.2.4 Interface Segregation Principle (ISP)

Sistema de gestión de documentos.

Código SIN aplicar ISP Una interfaz DocumentManager grande incluye múltiples métodos que no todos los clientes necesitarán usar.

[17]:
```python
class DocumentManager:
    def open_document(self, file_path):
        pass

    def save_document(self, content, file_path):
        pass

    def print_document(self, content):
        pass

    def review_document(self, content):
        pass
```

```
class MyDocumentManager(DocumentManager):
    def open_document(self, file_path):
        print(f"Opening document from {file_path}")

    def save_document(self, content, file_path):
        print(f"Saving document to {file_path}")

    def print_document(self, content):
        print(f"Printing document content: {content}")

    def review_document(self, content):
        print(f"Reviewing document content: {content}")
```

Código APLICANDO ISP La interfaz DocumentManager se divide en varias interfaces más pequeñas y especializadas. Cada cliente implementa solo las interfaces que necesita para su funcionamiento específico

Justificación En el primer enfoque, los clientes que solo necesitan imprimir documentos aún dependen de la interfaz completa DocumentManager. Al separar la interfaz, los clientes solo implementan las interfaces que necesitan, lo cual es más eficiente y claro.

[18]:
```
from abc import ABC, abstractmethod

class DocumentOpener(ABC):
    @abstractmethod
    def open_document(self, file_path):
        pass

class DocumentSaver(ABC):
    @abstractmethod
    def save_document(self, content, file_path):
        pass

class DocumentPrinter(ABC):
    @abstractmethod
    def print_document(self, content):
        pass

class DocumentReviewer(ABC):
    @abstractmethod
    def review_document(self, content):
        pass

class MyPrinter(DocumentPrinter):
    def print_document(self, content):
        print(f"Printing document content: {content}")
```

```
class MyReviewer(DocumentReviewer):
    def review_document(self, content):
        print(f"Reviewing document content: {content}")

# Example use case where a client needs only printing and reviewing capabilities
class PrintReviewStation(MyPrinter, MyReviewer):
    pass
```

1.2.5 Dependency Inversion Principle (DIP)

Sistema de logística para el manejo de envíos.

Código SIN aplicar DIP La clase ShippingManager depende directamente de clases concretas como Truck y Ship para manejar diferentes tipos de transporte. Esto crea un acoplamiento fuerte y reduce la flexibilidad del sistema

[19]:
```
class Truck:
    def deliver(self, destination):
        print(f"Delivering to {destination} by road.")

class Ship:
    def deliver(self, destination):
        print(f"Delivering to {destination} by sea.")

class ShippingManager:
    def __init__(self):
        self.truck = Truck()
        self.ship = Ship()

    def send_delivery(self, type, destination):
        if type == "road":
            self.truck.deliver(destination)
        elif type == "sea":
            self.ship.deliver(destination)

# Usage
manager = ShippingManager()
manager.send_delivery("road", "123 Elm St")
manager.send_delivery("sea", "456 Maple Ave")
```

```
Delivering to 123 Elm St by road.
Delivering to 456 Maple Ave by sea.
```

Código APLICANDO DIP Creamos una interfaz TransportationMethod y hacemos que ShippingManager dependa de esta interfaz en lugar de clases concretas. Esto permite la adición de nuevos métodos de transporte sin modificar el ShippingManager

[20]:
```python
from abc import ABC, abstractmethod

class TransportationMethod(ABC):
    @abstractmethod
    def deliver(self, destination):
        pass

class Truck(TransportationMethod):
    def deliver(self, destination):
        print(f"Delivering to {destination} by road.")

class Ship(TransportationMethod):
    def deliver(self, destination):
        print(f"Delivering to {destination} by sea.")

class Airplane(TransportationMethod):
    def deliver(self, destination):
        print(f"Delivering to {destination} by air.")

class ShippingManager:
    def __init__(self, transportation_methods):
        self.transportation_methods = transportation_methods

    def send_delivery(self, type, destination):
        transport = self.transportation_methods.get(type)
        if transport:
            transport.deliver(destination)
        else:
            print("Transport method not supported.")

# Usage
transports = {"road": Truck(), "sea": Ship(), "air": Airplane()}
manager = ShippingManager(transports)
manager.send_delivery("road", "123 Elm St")
manager.send_delivery("sea", "456 Maple Ave")
manager.send_delivery("air", "789 Oak Blvd")
```

```
Delivering to 123 Elm St by road.
Delivering to 456 Maple Ave by sea.
Delivering to 789 Oak Blvd by air.
```

Justificación Sin DIP, ShippingManager está fuertemente acoplado a medios de transporte específicos, lo que dificulta cambiar o añadir nuevos medios sin modificar el gestor de envíos. Al depender de una interfaz, ShippingManager se vuelve más flexible y fácil de extender, facilitando la integración de nuevos tipos de transporte sin impactar el código existente.