Unidad didáctica 06

Programación de Componentes de Acceso a Datos

Índice

UD 06: Programación de Componentes de Acceso a Datos

1. Introducción

En esta unidad, nos centraremos en desarrollar componentes de acceso a datos en Python que permitan gestionar información almacenada en distintos sistemas de almacenamiento, incluyendo ficheros en formatos JSON y CSV y bases de datos. La programación orientada a componentes en el contexto del acceso a datos permite crear módulos reutilizables e independientes que encapsulan la lógica de manipulación de datos. Esto facilita su integración en aplicaciones más amplias y mejora la modularidad y mantenibilidad del código.

1.1. Objetivos

- Desarrollar componentes en Python que gestionen información en formatos JSON y CSV y en bases de datos, con control de transacciones y versiones.

- Implementar registro de actividad mediante la biblioteca `logging` para el seguimiento de las operaciones del componente.

- Proporcionar una interfaz de acceso a datos robusta y estandarizada.

- Documentar y probar los componentes desarrollados, asegurando su capacidad para integrarse en aplicaciones mayores.

2. Conceptos Fundamentales de Programación Orientada a Componentes

La **programación orientada a componentes (COP)** es un paradigma de desarrollo de software que se centra en la creación de módulos independientes, conocidos como **componentes**. Cada componente encapsula una funcionalidad específica y se puede reutilizar, combinar y reemplazar fácilmente. Este enfoque permite dividir una aplicación en varias partes modulares, lo que facilita el desarrollo, mantenimiento y escalabilidad del sistema.

En el contexto de acceso a datos, la programación orientada a componentes tiene un papel esencial. Los componentes de acceso a datos permiten crear módulos de software que manejan operaciones

de lectura, escritura, actualización y eliminación de datos de manera eficiente y estandarizada. Estos módulos pueden gestionar datos en diferentes formatos y sistemas, como bases de datos relacionales, bases de datos documentales, archivos planos y estructuras JSON o CSV.

2.1. Definición y Objetivo de la Programación Orientada a Componentes

La programación orientada a componentes busca **modularizar** y **abstraer** funcionalidades específicas en unidades de software que son autónomas y pueden ser reutilizadas en múltiples proyectos. Este enfoque es especialmente útil en aplicaciones grandes y complejas, ya que los componentes permiten dividir la funcionalidad en piezas más pequeñas y manejables.

Un componente en COP suele tener las siguientes características:

- **Encapsulamiento**: Cada componente encapsula su funcionalidad y es autónomo en su operación. Esto significa que tiene su propia lógica y datos, y no depende directamente de otros componentes para funcionar.
- **Interfaz bien definida**: Los componentes exponen una interfaz que permite a otros módulos o aplicaciones interactuar con ellos sin necesidad de conocer su implementación interna.
- **Independencia**: Un componente bien diseñado se puede desarrollar, probar, y mantener de manera independiente, sin afectar a otros módulos del sistema.
- **Reutilización**: Los componentes se pueden reutilizar en diferentes proyectos o módulos, lo cual reduce la duplicación de código y acelera el desarrollo.

En el contexto de acceso a datos, un componente de acceso a datos encapsula la lógica de interacción con el almacenamiento de datos y ofrece una interfaz sencilla para gestionar la información. Estos componentes permiten a los desarrolladores manipular datos sin preocuparse por la implementación de bajo nivel, lo que mejora la productividad y reduce el margen de error.

2.2. Objetivos de la Programación Orientada a Componentes en Acceso a Datos

1. **Simplificar el desarrollo y mantenimiento**:
    - Los componentes de acceso a datos proporcionan un punto centralizado para gestionar todas las interacciones con las fuentes de datos. Esto simplifica la lógica del código y facilita el mantenimiento a largo plazo.
2. **Mejorar la modularidad y escalabilidad**:
    - La modularidad es una ventaja clave, ya que cada componente puede ser mejorado, reemplazado o ampliado sin afectar al resto del sistema. Por ejemplo, si se cambia el formato de almacenamiento de datos, solo se necesita actualizar el componente de acceso a datos.
3. **Fomentar la reutilización y consistencia**:

- Al encapsular la lógica de acceso a datos en un componente reutilizable, es posible garantizar que las mismas operaciones de acceso y manipulación se utilicen en diferentes partes de una aplicación o incluso en diferentes proyectos, lo que aumenta la consistencia y reduce la duplicación de código.

2.3. Ventajas e Inconvenientes de la Programación Orientada a Componentes

La programación orientada a componentes tiene una serie de ventajas e inconvenientes que es importante tener en cuenta al decidir si este enfoque es adecuado para una aplicación.

Ventajas de la Programación Orientada a Componentes

1. **Reutilización de Código**:
    - Una de las mayores ventajas de la COP es la **reutilización de código**. Al diseñar componentes que encapsulan funcionalidades específicas, es posible reutilizarlos en diferentes aplicaciones. Esto reduce el tiempo de desarrollo y permite que los desarrolladores se enfoquen en agregar nuevas funcionalidades en lugar de rehacer código.
2. **Facilidad de Mantenimiento**:
    - La COP facilita el **mantenimiento** de aplicaciones grandes, ya que cada componente encapsula una función o conjunto de funciones. Si se necesita actualizar o corregir alguna funcionalidad, es posible trabajar en el componente específico sin afectar al resto del sistema.
3. **Modularidad y Escalabilidad**:
    - La **modularidad** es una ventaja clave en la COP. Cada componente puede ser tratado como una unidad independiente, lo que permite añadir o eliminar funcionalidades de manera flexible. Además, este enfoque facilita la **escalabilidad** de las aplicaciones, ya que permite integrar nuevos componentes según sea necesario.
4. **Consistencia y Control de Errores**:
    - Al utilizar componentes para gestionar el acceso a datos, se garantiza que todas las operaciones de acceso sigan un estándar de consistencia. Los componentes también ayudan a **controlar errores** de manera centralizada, lo cual es útil en sistemas donde la integridad de los datos es crítica.
5. **Abstracción y Flexibilidad**:
    - Los componentes proporcionan una capa de **abstracción** que permite cambiar o mejorar la implementación sin afectar a la lógica de negocio que los utiliza. Esto es especialmente útil en aplicaciones que necesitan interoperar con varios sistemas de datos, ya que permite modificar el componente sin cambiar el resto de la aplicación.

Inconvenientes de la Programación Orientada a Componentes

1. **Complejidad Inicial**:
    - El diseño y desarrollo de componentes puede requerir **más planificación y esfuerzo inicial** en comparación con la creación de clases simples. Diseñar componentes que sean verdaderamente independientes, modulares y reutilizables implica una inversión significativa en tiempo y recursos al inicio del proyecto.
2. **Sobrecarga en el Rendimiento**:
    - La integración de múltiples componentes puede causar una **sobrecarga en el rendimiento** del sistema, especialmente si los componentes se comunican entre sí o si requieren operaciones de entrada y salida frecuentes. Aunque la COP ayuda en la modularidad y escalabilidad, su uso en sistemas de alta carga puede requerir optimización adicional.
3. **Dificultad en la Integración**:
    - Integrar componentes de diferentes fuentes, lenguajes o frameworks puede ser un desafío, ya que cada componente puede tener sus propias dependencias, convenciones de interfaz o restricciones. Esto es especialmente relevante cuando se intenta combinar componentes desarrollados en diferentes tecnologías.
4. **Dependencia de Interfaces**:
    - La COP requiere interfaces bien definidas y estandarizadas para que los componentes se puedan comunicar y funcionar correctamente en conjunto. Diseñar y mantener estas interfaces es fundamental, pero puede ser difícil de coordinar, especialmente en proyectos grandes o en equipos distribuidos.
5. **Coste en el Mantenimiento y Actualización**:
    - Aunque la COP facilita el mantenimiento, también introduce el desafío de actualizar y mantener componentes de manera independiente, lo que puede implicar que se deban gestionar múltiples versiones de componentes compatibles en un mismo proyecto.

2.4. Ejemplos en el Contexto de Acceso a Datos

La programación orientada a componentes se utiliza ampliamente en el desarrollo de aplicaciones que necesitan gestionar datos de múltiples fuentes de almacenamiento. A continuación, se muestran algunos ejemplos de cómo los componentes pueden facilitar el acceso a datos en aplicaciones reales:

- **Componente de Conexión a Bases de Datos Relacionales**:
    - Un componente diseñado para gestionar la conexión y manipulación de datos en bases de datos relacionales como MySQL o PostgreSQL. Este componente puede incluir funcionalidades como manejo de transacciones, control de errores y optimización de consultas.

- **Componente de Lectura y Escritura en Archivos**:
    - Un componente diseñado para gestionar datos en archivos, como CSV o JSON. Este tipo de componente es especialmente útil para aplicaciones que trabajan con datos estáticos o para realizar operaciones de exportación e importación de datos.
- **Componente de Mapeo Objeto Relacional (ORM)**:
    - Un ORM es un tipo de componente que permite a los desarrolladores trabajar con bases de datos relacionales utilizando objetos en el lenguaje de programación. En Python, un ejemplo sería el uso de SQLAlchemy para gestionar entidades como “Usuarios” o “Pedidos” a través de una interfaz orientada a objetos.
- **Componente de Acceso a Bases de Datos Documentales**:
    - Un componente que proporciona acceso a bases de datos documentales, como MongoDB, en las cuales los datos se almacenan en documentos JSON. Este componente puede incluir funcionalidades específicas para consultas, manipulación y gestión de colecciones.
- **Componente para Almacenamiento de Datos en Bases de Datos Orientadas a Objetos**:
    - Este tipo de componente permite la persistencia de objetos de Python directamente en bases de datos orientadas a objetos, como ZODB. Esto es útil en aplicaciones donde los datos están fuertemente ligados a objetos y relaciones complejas.

3. Herramientas de Desarrollo de Componentes en Python

En Python, existen múltiples bibliotecas que facilitan el desarrollo de componentes de acceso a datos. Algunas de estas herramientas se pueden utilizar como referencia para el diseño de componentes propios.

3.1. Herramientas Relevantes

- SQLAlchemy: ORM que facilita la creación y manipulación de objetos mapeados a tablas de bases de datos relacionales.
- Pandas: Biblioteca para manipulación de datos tabulares en formatos como CSV y Excel.
- PyMongo: Biblioteca para conexión y manipulación de datos en bases de datos MongoDB.
- ZODB: Base de datos orientada a objetos que permite almacenar objetos de Python de manera persistente.

Estas herramientas pueden servir como referencia en el desarrollo de componentes propios, ayudando a identificar buenas prácticas y capacidades clave para un componente de acceso a datos.

3.2. Evaluación de las Herramientas

Las herramientas se deben evaluar según criterios como facilidad de uso, flexibilidad, rendimiento, y soporte para diversos sistemas de almacenamiento. La elección dependerá del tipo de datos y de las necesidades específicas del sistema.

4. Programación del Componente de Acceso a Datos: `DataManager`, que gestionan información almacenada en ficheros.

En esta sección, se desarrolla un componente llamado `DataManager`, diseñado para gestionar datos en formatos JSON y CSV, con funcionalidades avanzadas como control de versiones, transacciones y logging.

4.1. Descripción del Componente `DataManager`

El componente `DataManager` proporciona métodos para:

- Leer y escribir datos en archivos JSON y CSV.

- Controlar versiones: Cada vez que se realiza una confirmación de cambios, el componente incrementa la versión, permitiendo un seguimiento de las modificaciones.

- Control de transacciones: Soporta operaciones de inicio, confirmación y reversión de transacciones, lo que ayuda a mantener la integridad de los datos.

- Logging: Implementa la biblioteca `logging` para registrar todas las operaciones realizadas en el archivo.

4.2. Código del Componente

```
import json
import csv
import logging
import os
from datetime import datetime
from copy import deepcopy

# Configuración de logging para guardar en un archivo
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
```

```
    handlers=[
        logging.FileHandler("log_datos.log"),  # Guardar logs en un archivo
    ]
)

class DataManager:
    def __init__(self, ruta_archivo, tipo_archivo='json'):
        self.ruta_archivo = ruta_archivo
        self.tipo_archivo = tipo_archivo
        self.version = 1
        self.transaccion_activa = False
        self.copia_datos = None

        # Cargar datos si el archivo existe, o inicializar con un diccionario vacío
        if os.path.exists(ruta_archivo):
            self.datos = self._leer_archivo()
                 logging.info(f"Archivo {tipo_archivo.upper()} cargado con éxito. Versión actual:
{self.version}")
        else:
            self.datos = []
            self._guardar_archivo()

    def _leer_archivo(self):
        if self.tipo_archivo == 'json':
            with open(self.ruta_archivo, 'r') as archivo:
                return json.load(archivo)
        elif self.tipo_archivo == 'csv':
            datos = []
            with open(self.ruta_archivo, mode='r') as archivo:
                lector = csv.DictReader(archivo)
                for fila in lector:
                    datos.append(fila)
            return datos
        else:
            raise ValueError("Tipo de archivo no soportado. Use 'json' o 'csv'.")

    def _guardar_archivo(self):
        if self.tipo_archivo == 'json':
```

```
            with open(self.ruta_archivo, 'w') as archivo:
                json.dump(self.datos, archivo, indent=4)
        elif self.tipo_archivo == 'csv':
            if self.datos:
                with open(self.ruta_archivo, mode='w', newline='') as archivo:
                    escritor = csv.DictWriter(archivo, fieldnames=self.datos[0].keys())
                    escritor.writeheader()
                    escritor.writerows(self.datos)
                logging.info(f"Archivo {self.tipo_archivo.upper()} guardado. Versión actual:
{self.version}")

    def iniciar_transaccion(self):
        if self.transaccion_activa:
            raise Exception("Ya hay una transacción activa.")
        self.transaccion_activa = True
        self.copia_datos = deepcopy(self.datos)  # Crear una copia de seguridad de los datos
        logging.info("Transacción iniciada.")

    def confirmar_transaccion(self):
        if not self.transaccion_activa:
            raise Exception("No hay una transacción activa para confirmar.")
        self.version += 1  # Incrementa la versión al confirmar la transacción
        self.transaccion_activa = False
        self.copia_datos = None  # Limpiar la copia de seguridad
        self._guardar_archivo()  # Guardar cambios en el archivo
        logging.info("Transacción confirmada y cambios guardados.")

    def revertir_transaccion(self):
        if not self.transaccion_activa:
            raise Exception("No hay una transacción activa para revertir.")
        self.datos = self.copia_datos  # Revertir a la copia de seguridad
        self.transaccion_activa = False
        self.copia_datos = None
        logging.warning("Transacción revertida. Los cambios no se guardaron.")

    def leer_dato(self, clave, valor):
        return [dato for dato in self.datos if dato.get(clave) == valor]
```

```
def escribir_dato(self, nuevo_dato):
    if not self.transaccion_activa:
        raise Exception("Debe iniciar una transacción antes de realizar cambios.")
    self.datos.append(nuevo_dato)
    logging.info(f"Dato agregado: {nuevo_dato}")

def eliminar_dato(self, clave, valor):
    if not self.transaccion_activa:
        raise Exception("Debe iniciar una transacción antes de realizar cambios.")
    self.datos = [dato for dato in self.datos if dato.get(clave) != valor]
    logging.info(f"Datos con {clave}={valor} eliminados.")

def obtener_configuracion(self):
    return {
        "ruta_archivo": self.ruta_archivo,
        "tipo_archivo": self.tipo_archivo,
        "version": self.version,
        "transaccion_activa": self.transaccion_activa
    }

def actualizar_configuracion(self, nueva_ruta, nuevo_tipo=None):
    if self.transaccion_activa:
        raise Exception("No se puede cambiar la configuración durante una transacción.")
    self.ruta_archivo = nueva_ruta
    if nuevo_tipo:
        self.tipo_archivo = nuevo_tipo
    logging.info(f"Configuración actualizada. Nueva ruta del archivo: {self.ruta_archivo}")
```

4.3. Explicación del Componente

El componente `DataManager` se ha diseñado para gestionar datos almacenados en archivos JSON y CSV, proporcionando una serie de funcionalidades avanzadas que lo convierten en una herramienta robusta para la manipulación de datos. Su estructura modular permite realizar operaciones como lectura, escritura, eliminación y búsqueda de datos, así como operaciones de control de versiones y transacciones, y un sistema de registro de actividad (logging) que registra cada operación relevante.

A continuación, se presenta un análisis detallado de cada parte del componente:

4.3.1. Inicialización del Componente (`__init__`)

La función `__init__` es el constructor del componente. Al inicializarse, configura el archivo de datos y el tipo de archivo (JSON o CSV) que se va a gestionar. Además, verifica si el archivo ya existe, en cuyo caso carga los datos; si no existe, inicializa una estructura de datos vacía y guarda el archivo:

```
def __init__(self, ruta_archivo, tipo_archivo='json'):
    self.ruta_archivo = ruta_archivo
    self.tipo_archivo = tipo_archivo
    self.version = 1
    self.transaccion_activa = False
    self.copia_datos = None
```

Atributos:

- `ruta_archivo`: Ruta donde se almacenará o se cargará el archivo.
- `tipo_archivo`: Indica el tipo de archivo que se manipulará (`json` o `csv`).
- `version`: Controla el número de versión del archivo, que se incrementa con cada confirmación de transacción.
- `transaccion_activa`: Bandera booleana que indica si una transacción está en curso.
- `copia_datos`: Almacena una copia de los datos antes de iniciar una transacción, lo que permite revertir los cambios si es necesario.

- **Lógica de inicialización**:
  - Si el archivo existe en la ruta especificada, el componente carga los datos en `self.datos` usando el método `_leer_archivo()`.
  - Si el archivo no existe, se inicializa `self.datos` como una lista vacía, y se guarda inmediatamente para crear el archivo vacío en el sistema.

4.3.2. Lectura y Escritura de Archivos (`_leer_archivo` y `_guardar_archivo`)

Los métodos `_leer_archivo` y `_guardar_archivo` son responsables de la carga y almacenamiento de datos en el archivo especificado, respetando el tipo de archivo (JSON o CSV).

Lectura del archivo:

```
def _leer_archivo(self):
    if self.tipo_archivo == 'json':
        with open(self.ruta_archivo, 'r') as archivo:
            return json.load(archivo)
    elif self.tipo_archivo == 'csv':
        datos = []
        with open(self.ruta_archivo, mode='r') as archivo:
            lector = csv.DictReader(archivo)
            for fila in lector:
```

```
                datos.append(fila)
        return datos
```

- **Caso JSON**: Utiliza `json.load` para leer el archivo JSON y cargarlo como una lista de diccionarios en Python.
- **Caso CSV**: Utiliza `csv.DictReader` para leer el archivo CSV y convierte cada fila en un diccionario. Cada diccionario representa una fila en el archivo, y se almacena en una lista `datos` que luego se devuelve.

Guardado del archivo:

```
def _guardar_archivo(self):
    if self.tipo_archivo == 'json':
        with open(self.ruta_archivo, 'w') as archivo:
            json.dump(self.datos, archivo, indent=4)
    elif self.tipo_archivo == 'csv' and self.datos:
        with open(self.ruta_archivo, mode='w', newline='') as archivo:
            escritor = csv.DictWriter(archivo, fieldnames=self.datos[0].keys())
            escritor.writeheader()
            escritor.writerows(self.datos)
    logging.info(f"Archivo {self.tipo_archivo.upper()} guardado. Versión actual: {self.version}")
```

- **Caso JSON**: Utiliza `json.dump` para escribir el contenido de `self.datos` en el archivo en formato JSON.
- **Caso CSV**: Usa `csv.DictWriter` para escribir cada elemento de `self.datos` (una lista de diccionarios) en el archivo CSV. Se asegura de escribir un encabezado de columnas, usando las claves del primer diccionario en la lista como encabezado.

4.3.3. Gestión de Transacciones

Las transacciones permiten encapsular una serie de operaciones en una unidad que puede confirmarse o revertirse en bloque. `DataManager` incluye métodos para iniciar, confirmar y revertir transacciones, lo cual garantiza que los datos se manipulan de manera segura.

Iniciar una transacción (`iniciar_transaccion`):

```
def iniciar_transaccion(self):
    if self.transaccion_activa:
        raise Exception("Ya hay una transacción activa.")
    self.transaccion_activa = True
    self.copia_datos = deepcopy(self.datos)  # Crear una copia de seguridad de los datos
    logging.info("Transacción iniciada.")
```

- Verifica si ya hay una transacción activa. Si es así, lanza una excepción, evitando conflictos de transacciones anidadas.
- Establece `self.transaccion_activa` en `True` e inicializa `self.copia_datos` con una copia profunda (`deepcopy`) de `self.datos`. Esto permite revertir cambios si la transacción falla.

Confirmar una transacción (`confirmar_transaccion`):

```
def confirmar_transaccion(self):
    if not self.transaccion_activa:
        raise Exception("No hay una transacción activa para confirmar.")
    self.version += 1  # Incrementa la versión al confirmar la transacción
    self.transaccion_activa = False
    self.copia_datos = None  # Limpiar la copia de seguridad
    self._guardar_archivo()  # Guardar cambios en el archivo
    logging.info("Transacción confirmada y cambios guardados.")
```

- Verifica que haya una transacción activa antes de confirmar.
- Incrementa `self.version` para reflejar una nueva versión de los datos.
- Establece `self.transaccion_activa` en `False` y elimina la copia de seguridad (`self.copia_datos`).
- Llama a `_guardar_archivo()` para persistir los cambios en el archivo y registra la confirmación de la transacción en `logging`.

Revertir una transacción (`revertir_transaccion`):

```
def revertir_transaccion(self):
    if not self.transaccion_activa:
        raise Exception("No hay una transacción activa para revertir.")
    self.datos = self.copia_datos  # Revertir a la copia de seguridad
    self.transaccion_activa = False
    self.copia_datos = None
    logging.warning("Transacción revertida. Los cambios no se guardaron.")
```

- Verifica que haya una transacción activa para revertir.
- Restaura `self.datos` a partir de `self.copia_datos`, revirtiendo todos los cambios realizados en la transacción.
- Establece `self.transaccion_activa` en `False` y elimina `self.copia_datos`.
- Registra la reversión de la transacción como una advertencia (`WARNING`).

4.3.4. Operaciones CRUD (Crear, Leer, Actualizar y Eliminar)

Estas operaciones permiten al usuario del componente manipular los datos en el archivo de manera controlada.

Leer datos (`leer_dato`):

```
def leer_dato(self, clave, valor):
    return [dato for dato in self.datos if dato.get(clave) == valor]
```

- `leer_dato` recibe una clave y un valor y devuelve una lista de diccionarios que coinciden con el valor en la clave especificada. Esto permite realizar búsquedas dentro de los datos.

Escribir datos (`escribir_dato`):

```
def escribir_dato(self, nuevo_dato):
    if not self.transaccion_activa:
```

```
            raise Exception("Debe iniciar una transacción antes de realizar cambios.")
        self.datos.append(nuevo_dato)
        logging.info(f"Dato agregado: {nuevo_dato}")
```

- `escribir_dato` permite añadir un nuevo dato (diccionario) a `self.datos`. Requiere que haya una transacción activa para asegurar la coherencia de los datos y registrar la operación.

Eliminar datos (`eliminar_dato`):

```
def eliminar_dato(self, clave, valor):
    if not self.transaccion_activa:
        raise Exception("Debe iniciar una transacción antes de realizar cambios.")
    self.datos = [dato for dato in self.datos if dato.get(clave) != valor]
    logging.info(f"Datos con {clave}={valor} eliminados.")
```

- `eliminar_dato` elimina todas las entradas de `self.datos` donde el valor de `clave` coincide con el valor especificado, permitiendo eliminar múltiples datos a la vez.

4.3.5. Métodos de Configuración y Acceso

El componente `DataManager` incluye métodos para acceder y actualizar la configuración, permitiendo una fácil integración en otras aplicaciones.

Obtener configuración (`obtener_configuracion`):

```
def obtener_configuracion(self):
    return {
        "ruta_archivo": self.ruta_archivo,
        "tipo_archivo": self.tipo_archivo,
        "version": self.version,
        "transaccion_activa": self.transaccion_activa
    }
```

- `obtener_configuracion` devuelve un diccionario con la configuración actual del componente, incluyendo la ruta, tipo de archivo, versión y el estado de la transacción.

Actualizar configuración (`actualizar_configuracion`):

```
def actualizar_configuracion(self, nueva_ruta, nuevo_tipo=None):
    if self.transaccion_activa:
        raise Exception("No se puede cambiar la configuración durante una transacción.")
    self.ruta_archivo = nueva_ruta
    if nuevo_tipo:
        self.tipo_archivo = nuevo_tipo
    logging.info(f"Configuración actualizada. Nueva ruta del archivo: {self.ruta_archivo}")
```

- `actualizar_configuracion` permite cambiar la ruta del archivo o el tipo de archivo, siempre y cuando no haya una transacción activa.

Este apartado proporciona una visión exhaustiva del componente `DataManager`, detallando cada método y su propósito en la manipulación de datos JSON y CSV con control de transacciones y logging.

Actividad de clase 1

Usa el componente mostrado en este apartado para escribir en un JSON 3 instancias del objeto que te fue asignado a principios de curso. Luego, sin instanciar de nuevo la clase, cambia a CSV y escribe las 3 mismas instancias en un fichero csv. Adjunta dentro del pdf de esta unidad didáctica **3 capturas**:

1. Captura del código de tu actividad
2. Captura del resultado de la ejecución
3. Captura con los mensajes de log.

5. Programación de Componentes para Gestión de Información en Bases de Datos mediante Conectores

Esta sección presentará el desarrollo de un componente que utiliza MySQL como base de datos y mysql-connector-python como el conector para interactuar con ella. El componente será capaz de realizar operaciones CRUD (crear, leer, actualizar y eliminar) en la base de datos, así como gestionar transacciones.

5.1. Recordatorio sobre el uso de conectores en Python para acceso a bases de datos

Como vimos en la unidad didáctica 1, los conectores permiten a las aplicaciones comunicarse con bases de datos mediante un protocolo de conexión específico. En Python, existen varios conectores para diferentes bases de datos, como `psycopg2` para PostgreSQL, `sqlite3` para SQLite, y `mysql-connector-python` para MySQL.

En este apartado, usaremos el conector `mysql-connector-python` para interactuar con una base de datos MySQL, lo cual permite que nuestro componente ejecute consultas SQL y manipule datos de manera eficiente.

Ventajas de usar un Conector para la Gestión de Datos en Bases de Datos:

- Abstracción del acceso a datos: Los conectores proporcionan una API para realizar operaciones en la base de datos sin necesidad de preocuparse por los detalles de bajo nivel de la conexión.

- Portabilidad y escalabilidad: Permiten que el mismo código funcione en diferentes bases de datos con mínimas modificaciones, siempre y cuando exista un conector compatible.

- Gestión de transacciones: Facilitan el manejo de transacciones para asegurar que las operaciones se realicen de manera segura y que los datos mantengan su integridad.

5.2. Descripción del Componente `DatabaseManager`

El componente `DatabaseManager` es un módulo de Python que encapsula las operaciones de acceso a una base de datos MySQL utilizando el conector `mysql-connector-python`. Este componente se encargará de realizar operaciones CRUD sobre una tabla de ejemplo llamada `Herramientas`.

Funcionalidades Principales del Componente:

1. Conectar y desconectar: Establece una conexión con la base de datos y la cierra cuando no es necesaria.
2. Operaciones CRUD: Permite crear, leer, actualizar y eliminar registros en la tabla `Herramientas`.
3. Gestión de transacciones: Proporciona métodos para iniciar, confirmar y revertir transacciones, asegurando que las operaciones de modificación de datos sean seguras y consistentes.
4. Manejo de errores: Utiliza excepciones para gestionar errores en la conexión y en las consultas, lo cual permite un manejo robusto de fallos.

5.3. Código del Componente `DatabaseManager`

A continuación, se muestra el código detallado del componente `DatabaseManager`, el cual incluye métodos para realizar las operaciones CRUD y gestionar transacciones.

```
import logging
import mysql.connector
from mysql.connector import Error

# Configuración de logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("databasemanager.log"),  # Logs guardados en un archivo
        logging.StreamHandler(),  # Logs también en consola
    ]
)

class DatabaseManager:
    def __init__(self, host, user, password, database):
        self.host = host
        self.user = user
        self.password = password
```

```
        self.database = database
        self.connection = None
        self.transaccion_activa = False

    def conectar(self):
        """Conectar a la base de datos MySQL"""
        try:
            self.connection = mysql.connector.connect(
                host=self.host,
                user=self.user,
                password=self.password,
                database=self.database
            )
            if self.connection.is_connected():
                self.connection.autocommit = False  # Desactiva el autocommit
                logging.info("Conexión exitosa a la base de datos.")
        except Error as e:
            logging.error(f"Error al conectar a la base de datos: {e}")

    def desconectar(self):
        """Cerrar la conexión a la base de datos"""
        if self.connection and self.connection.is_connected():
            if self.transaccion_activa:
                logging.warning("Cerrando conexión con una transacción activa. Revirtiendo...")
                self.revertir_transaccion()
            self.connection.close()
            logging.info("Conexión cerrada.")

    def crear_herramienta(self, nombre, tipo, material, uso, marca):
        """Insertar una nueva herramienta en la base de datos"""
        try:
            cursor = self.connection.cursor()
            query = """
                INSERT INTO Herramientas (nombre, tipo, material, uso, marca)
                VALUES (%s, %s, %s, %s, %s)
            """
            cursor.execute(query, (nombre, tipo, material, uso, marca))
            logging.info(f"Herramienta '{nombre}' insertada exitosamente.")
```

```
        except Error as e:
            logging.error(f"Error al insertar la herramienta '{nombre}': {e}")

    def leer_herramientas(self):
        """Leer todas las herramientas de la base de datos"""
        try:
            cursor = self.connection.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Herramientas")
            herramientas = cursor.fetchall()
            logging.info("Herramientas recuperadas:")
            for herramienta in herramientas:
                logging.info(herramienta)
            return herramientas
        except Error as e:
            logging.error(f"Error al leer las herramientas: {e}")
            return None

    def actualizar_herramienta(self, id, nombre, tipo, material, uso, marca):
        """Actualizar una herramienta en la base de datos"""
        try:
            cursor = self.connection.cursor()
            query = """
                UPDATE Herramientas
                SET nombre = %s, tipo = %s, material = %s, uso = %s, marca = %s
                WHERE id = %s
            """
            cursor.execute(query, (nombre, tipo, material, uso, marca, id))
            logging.info(f"Herramienta con ID {id} actualizada exitosamente.")
        except Error as e:
            logging.error(f"Error al actualizar la herramienta con ID {id}: {e}")

    def eliminar_herramienta(self, id):
        """Eliminar una herramienta de la base de datos"""
        try:
            cursor = self.connection.cursor()
            query = "DELETE FROM Herramientas WHERE id = %s"
            cursor.execute(query, (id,))
            logging.info(f"Herramienta con ID {id} eliminada exitosamente.")
```

```
        except Error as e:
            logging.error(f"Error al eliminar la herramienta con ID {id}: {e}")


    def iniciar_transaccion(self):
        """Iniciar una transacción"""
        try:
            if not self.transaccion_activa:
                #self.connection.rollback()
                #self.connection.start_transaction()
                self.transaccion_activa = True
                logging.info("Transacción iniciada.")
            else:
                logging.warning("Transacción ya en curso. No se puede iniciar otra.")
        except Error as e:
            logging.error(f"Error al iniciar la transacción: {e}")
            raise

    def confirmar_transaccion(self):
        """Confirmar (commit) una transacción"""
        try:
            if self.transaccion_activa:
                self.connection.commit()
                self.transaccion_activa = False
                logging.info("Transacción confirmada.")
            else:
                logging.warning("No hay ninguna transacción activa para confirmar.")
        except Error as e:
            logging.error(f"Error al confirmar la transacción: {e}")
            raise

    def revertir_transaccion(self):
        """Revertir (rollback) una transacción"""
        try:
            if self.transaccion_activa:
                self.connection.rollback()
                self.transaccion_activa = False
                logging.info("Transacción revertida.")
```

```
            else:
                logging.warning("No hay ninguna transacción activa para revertir.")
        except Error as e:
            logging.error(f"Error al revertir la transacción: {e}")
            raise
```

5.4. Uso del componente

```
# Ejemplo de uso del componente DatabaseManager

if __name__ == "__main__":
    db_manager = DatabaseManager("localhost", "usuario", "usuario", "1dam")
    db_manager.conectar()

    try:
        # Bloque de inserción
        db_manager.iniciar_transaccion()
        db_manager.crear_herramienta("Martillo", "Manual", "Acero", "Percusión", "Truper")
        db_manager.confirmar_transaccion()

        # Leer datos (no requiere transacción)
        herramientas = db_manager.leer_herramientas()

        # Bloque de actualización
        db_manager.iniciar_transaccion()
         db_manager.actualizar_herramienta(1, "Martillo", "Manual", "Acero reforzado", "Percusión",
"Stanley")
        db_manager.confirmar_transaccion()

        # Bloque de eliminación
        db_manager.iniciar_transaccion()
        db_manager.eliminar_herramienta(1)
        db_manager.confirmar_transaccion()

        # Bloque con transacción revertida
        db_manager.iniciar_transaccion()
        db_manager.crear_herramienta("Taladro", "Eléctrico", "Plástico", "Perforación", "Bosch")
        db_manager.revertir_transaccion()
```

```
except Exception as e:
    logging.error(f"Se produjo un error: {e}")
    if db_manager.transaccion_activa:
        db_manager.revertir_transaccion()

finally:
    db_manager.desconectar()
```

Explicación del Código

1. Conectar y Desconectar:

- El método `conectar` establece una conexión con la base de datos.
- El método `desconectar` cierra la conexión, liberando los recursos utilizados.

2. Operaciones CRUD:

- Crear: `crear_herramienta` inserta una nueva herramienta en la tabla `Herramientas` de la base de datos.
- Leer: `leer_herramientas` lee todas las herramientas y las imprime.
- Actualizar: `actualizar_herramienta` permite modificar los datos de una herramienta específica.
- Eliminar: `eliminar_herramienta` elimina una herramienta de la base de datos según su ID.

3. Gestión de Transacciones:

- Iniciar: `iniciar_transaccion` inicia una nueva transacción.
- Confirmar: `confirmar_transaccion` guarda permanentemente los cambios realizados durante la transacción.
- Revertir: `revertir_transaccion` deshace los cambios realizados en la transacción.

Este componente es reutilizable y encapsula las operaciones comunes de acceso a datos en una base de datos MySQL, lo que facilita su integración en diferentes aplicaciones.

Actividad de clase 2

Escribe el mismo componente pero para el objeto que te fue asignado a principios de curso. Luego, haz uso de su funcionalidad para hacer las 4 operaciones CRUD, haciendo uso de las funciones de transacciones ofrecidas por el componente.

Adjunta dentro del pdf de esta unidad didáctica **3 capturas**:

4. Captura del código de tu actividad
5. Captura del resultado de la ejecución
6. Captura con los mensajes de log.

6. Programación de Componentes que Usan Mapeo Objeto-Relacional

En este apartado vamos a ver el componente `DatabaseManagerORM`, que es una implementación diseñada para gestionar la interacción entre una aplicación y una base de datos MySQL utilizando el ORM **Peewee**. Este componente encapsula toda la lógica necesaria para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre dos tablas: **Proveedores** y **Herramientas**.

El diseño del componente sigue principios de programación orientada a objetos y se centra en:

1. Proveer una interfaz clara para interactuar con las tablas.
2. Registrar todas las acciones realizadas mediante un sistema de **logs**.
3. Facilitar la conexión y desconexión de la base de datos, manteniendo las mejores prácticas.

6.1. Estructura del Componente

1. **Clase `DatabaseManagerORM`**:
    - Actúa como intermediaria entre la base de datos y la aplicación.
    - Encapsula la configuración de conexión a la base de datos y define los métodos para operar sobre las tablas.
2. **Modelos Peewee**:
    - Los modelos `Proveedor` y `Herramienta` representan las tablas en la base de datos.
    - Utilizan relaciones **uno a muchos**: un proveedor puede tener muchas herramientas, pero cada herramienta pertenece a un único proveedor.
3. **Logging**:
    - Cada operación (éxito o error) se registra en un archivo llamado `databasemanager.log`.
    - Los logs permiten monitorear las acciones realizadas en tiempo real o analizarlas posteriormente.

6.2. Operaciones Disponibles

1. **Conexión a la Base de Datos**:
    - El componente establece la conexión a una base de datos MySQL usando Peewee.
    - Si la conexión falla, registra un error en el archivo de logs.
2. **Creación de Tablas**:

- Las tablas `proveedores` y `herramientas` se crean automáticamente si no existen en la base de datos.
- El método `crear_tablas` asegura que la estructura esté lista antes de realizar operaciones.

3. **Gestión de Proveedores**:
    - Operaciones disponibles:
        - **Crear** un proveedor con nombre y contacto.
        - **Leer** un proveedor específico por su nombre.
        - **Actualizar** el contacto de un proveedor existente.
        - **Eliminar** un proveedor.
4. **Gestión de Herramientas**:
    - Operaciones disponibles:
        - **Crear** una herramienta asociada a un proveedor específico.
        - **Leer** todas las herramientas asociadas a un proveedor.
        - **Actualizar** el tipo de una herramienta.
        - **Eliminar** una herramienta.
5. **Transacciones**:
    - Las operaciones críticas, como la creación o eliminación de registros, pueden agruparse en transacciones.
    - Esto garantiza la atomicidad: todas las operaciones se ejecutan correctamente o ninguna lo hace.

6.3. Código del componente:

```
import logging
from peewee import Model, CharField, ForeignKeyField, MySQLDatabase

# Configuración de logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("databasemanager_orm.log"),
        logging.StreamHandler()
    ]
)

# Configuración de la base de datos MySQL
```

```
db = MySQLDatabase(
    "1dam",  # Nombre de la base de datos
    user="usuario",  # Usuario de MySQL
    password="usuario",  # Contraseña de MySQL
    host="localhost",  # Host
    port=3306  # Puerto por defecto de MySQL
)

# Modelos de la base de datos
class Proveedor(Model):
    nombre = CharField()
    direccion = CharField()
    class Meta:
        database = db

class Herramienta(Model):
    nombre = CharField()
    tipo = CharField()
    marca = CharField()
    uso = CharField()
    material = CharField()
    proveedor = ForeignKeyField(Proveedor, backref='herramientas')
    class Meta:
        database = db

# Componente DatabaseManagerORM
class DatabaseManagerORM:
    def __init__(self):
        self.db = db

    def conectar(self):
        """Conecta la base de datos y crea las tablas."""
        self.db.connect()
        self.db.create_tables([Proveedor, Herramienta])
        logging.info("Conexión establecida y tablas creadas.")

    def desconectar(self):
        """Cierra la conexión a la base de datos."""
```

```
        if not self.db.is_closed():
            self.db.close()
            logging.info("Conexión cerrada.")

    def iniciar_transaccion(self):
        """Inicia una transacción."""
        self.db.begin()
        logging.info("Transacción iniciada.")

    def confirmar_transaccion(self):
        """Confirma (commit) una transacción."""
        self.db.commit()
        logging.info("Transacción confirmada.")

    def revertir_transaccion(self):
        """Revierte (rollback) una transacción."""
        self.db.rollback()
        logging.info("Transacción revertida.")

    def crear_proveedor(self, nombre, direccion):
        """Inserta un nuevo proveedor."""
        proveedor = Proveedor.create(nombre=nombre, direccion=direccion)
        logging.info(f"Proveedor creado: {proveedor.nombre} - {proveedor.direccion}")
        return proveedor

    def crear_herramienta(self, nombre, tipo, marca, uso, material, proveedor):
        """Inserta una nueva herramienta."""
        herramienta = Herramienta.create(
            nombre=nombre, tipo=tipo, marca=marca, uso=uso, material=material, proveedor=proveedor
        )
        logging.info(f"Herramienta creada: {herramienta.nombre} - {herramienta.tipo}")
        return herramienta

    def leer_herramientas(self):
        """Lee todas las herramientas."""
        herramientas = Herramienta.select()
        logging.info("Leyendo herramientas:")
        for herramienta in herramientas:
```

```
                                    logging.info(f"{herramienta.nombre}    -    {herramienta.tipo}
({herramienta.proveedor.nombre})")
        return herramientas
```

Actividad de clase 3:

Esta actividad se llevará a cabo con los objetos herramientas y proveedores, no con los objetos que te fueron asignados a principios de curso. Además en un apartado se solicita que uses tu DNI. Consiste en usar el componente proporcionado, de la siguiente forma:

1. Configuración Inicial. Conexión a la Base de Datos:

   Configura el componente para conectarse a la base de datos MySQL llamada 1dam con las credenciales proporcionadas.

   Asegúrate de que el componente cree las tablas necesarias: proveedores y herramientas.

2. Gestión de Proveedores. Crear Proveedores:
   - Crea dos proveedores con los siguientes datos:
     - Proveedor A: Contacto 123-456-789.
     - Proveedor B: Contacto 987-654-321.
   - Actualizar Información de un Proveedor. Actualiza el contacto del proveedor Proveedor A . Ponle tu DNI.
   - Eliminar un Proveedor: Elimina al proveedor Proveedor B de la base de datos.

- 3. Gestión de Herramientas. Crear Herramientas:
  - Agrega las siguientes herramientas asociadas al proveedor Proveedor A:
    - Martillo: Tipo Manual.
    - Taladro: Tipo Eléctrico.
  - Consultar Herramientas por Proveedor: Recupera todas las herramientas asociadas al proveedor Proveedor A y muestra su información.
  - Actualizar una Herramienta: Actualiza el tipo de la herramienta Martillo a Reforzado.
  - Eliminar una Herramienta: Elimina la herramienta Taladro de la base de datos.
- 4. Logs. Verificar el Archivo de Logs. Asegúrate de que las acciones realizadas (creaciones, actualizaciones, eliminaciones) estén registradas en el archivo databasemanager.log **y adjunta una captura de pantalla que muestre el contenido de dicho fichero.**

Adjunta dentro del pdf de esta unidad didáctica **3 capturas**:

1. Captura del código de tu actividad
2. Captura del resultado de la ejecución
3. Captura con los mensajes de log.

Tu programa debe generar la siguiente salida en consola:

```
2024-11-22 19:53:37,874 - INFO - Conexión establecida y tablas creadas.

Gestión de Proveedores:
2024-11-22 19:53:37,877 - INFO - Transacción iniciada.
2024-11-22 19:53:37,879 - INFO - Proveedor creado: Proveedor A - Contacto 123-456-789
2024-11-22 19:53:37,881 - INFO - Proveedor creado: Proveedor B - Contacto 987-654-321
2024-11-22 19:53:37,886 - INFO - Transacción confirmada.

Cambio del dato por mi DNI 12345678A al proveedor A
2024-11-22 19:53:37,889 - INFO - Transacción iniciada.
2024-11-22 19:53:37,897 - INFO - Transacción confirmada.
Eliminar al proveedor B
2024-11-22 19:53:37,899 - INFO - Transacción iniciada.
2024-11-22 19:53:37,906 - INFO - Transacción confirmada.

Gestión de Herramientas:
2024-11-22 19:53:37,907 - INFO - Transacción iniciada.
2024-11-22 19:53:37,911 - INFO - Herramienta creada: Martillo - Manual
2024-11-22 19:53:37,915 - INFO - Herramienta creada: Taladro - Eléctrico
2024-11-22 19:53:37,919 - INFO - Transacción confirmada.

Herramientas asociadas al proveedor Proveedor A:
Martillo - Manual
Taladro - Eléctrico

Actualizar herramienta Martillo a tipo reforzado
2024-11-22 19:53:37,926 - INFO - Transacción iniciada.
2024-11-22 19:53:37,932 - INFO - Transacción confirmada.

Eliminar herramienta Taladro
2024-11-22 19:53:37,932 - INFO - Transacción iniciada.
2024-11-22 19:53:37,938 - INFO - Transacción confirmada.
2024-11-22 19:53:37,939 - INFO - Conexión cerrada.
```

7. Programación de Componentes que Gestionan Información en Bases de Datos Documentales

7.1 Recordatorio sobre Bases de Datos Documentales

En la unidad didáctica 5 ya abordamos las bases de datos documentales como MongoDB. Recordemos que estas bases de datos almacenan datos en formato de documentos (como JSON o BSON), lo que permite flexibilidad en la estructura de los datos.

Ventajas clave de MongoDB:

- Almacenamiento de documentos con estructuras dinámicas.
- Facilidad para trabajar con datos jerárquicos y no estructurados.
- Escalabilidad horizontal, capacidad de distribuir datos y operaciones entre múltiples servidores (nodos), permitiendo que el sistema crezca de manera eficiente a medida que aumentan las demandas de almacenamiento y procesamiento.

Estos sistemas son ideales para aplicaciones modernas donde la estructura de los datos puede variar y donde se requiere alta disponibilidad.

7.2 Desarrollo del Componente: `DatabaseManagerDocumental`

El componente `DatabaseManagerDocumental` proporciona una interfaz para realizar operaciones CRUD (Create, Read, Update, Delete) en una colección de MongoDB. Además, como siempre hasta ahora:

1. **Logging** para registrar todas las operaciones realizadas.
2. **Soporte para transacciones**, útil en escenarios donde se requiere consistencia entre varias operaciones.

7.3. Código del Componente

```
import logging
from pymongo import MongoClient
from pymongo.errors import PyMongoError

# Configuración de logging
logging.basicConfig(
    level=logging.INFO,
```

```
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("databasemanager_documental.log"),  # Logs guardados en un archivo
        logging.StreamHandler(),  # Logs también en consola
    ]
)

class DatabaseManagerDocumental:
    def __init__(self, uri, database_name, collection_name):
        """Inicializa el componente DatabaseManagerDocumental."""
        self.uri = uri
        self.database_name = database_name
        self.collection_name = collection_name
        self.client = None
        self.db = None
        self.collection = None

    def conectar(self):
        """Conectar a la base de datos MongoDB"""
        try:
            self.client = MongoClient(self.uri)
            self.db = self.client[self.database_name]
            self.collection = self.db[self.collection_name]
            logging.info(f"Conectado a MongoDB: {self.database_name}.{self.collection_name}")
        except PyMongoError as e:
            logging.error(f"Error al conectar a MongoDB: {e}")

    def desconectar(self):
        """Cerrar la conexión a MongoDB"""
        if self.client:
            self.client.close()
            logging.info("Conexión a MongoDB cerrada.")

    def crear_documento(self, documento):
        """Insertar un nuevo documento en la colección"""
        try:
            result = self.collection.insert_one(documento)
            logging.info(f"Documento insertado con ID: {result.inserted_id}")
```

```
            return result.inserted_id
        except PyMongoError as e:
            logging.error(f"Error al insertar el documento: {e}")

    def leer_documentos(self, filtro={}):
        """Leer documentos de la colección según un filtro"""
        try:
            documentos = list(self.collection.find(filtro))
            logging.info(f"Documentos recuperados: {len(documentos)}")
            for doc in documentos:
                logging.info(doc)
            return documentos
        except PyMongoError as e:
            logging.error(f"Error al leer los documentos: {e}")
            return []

    def actualizar_documento(self, filtro, actualizacion):
        """Actualizar un documento en la colección"""
        try:
            result = self.collection.update_one(filtro, {"$set": actualizacion})
            if result.modified_count > 0:
                logging.info(f"Documento actualizado: {filtro}")
            else:
                logging.warning(f"No se encontró documento para actualizar: {filtro}")
        except PyMongoError as e:
            logging.error(f"Error al actualizar el documento: {e}")

    def eliminar_documento(self, filtro):
        """Eliminar un documento de la colección"""
        try:
            result = self.collection.delete_one(filtro)
            if result.deleted_count > 0:
                logging.info(f"Documento eliminado: {filtro}")
            else:
                logging.warning(f"No se encontró documento para eliminar: {filtro}")
        except PyMongoError as e:
            logging.error(f"Error al eliminar el documento: {e}")
```

```
    def iniciar_transaccion(self):
        """Iniciar una transacción"""
        try:
            self.session = self.client.start_session()
            self.session.start_transaction()
            logging.info("Transacción iniciada.")
        except PyMongoError as e:
            logging.error(f"Error al iniciar la transacción: {e}")

    def confirmar_transaccion(self):
        """Confirmar (commit) una transacción"""
        try:
            if self.session:
                self.session.commit_transaction()
                logging.info("Transacción confirmada.")
        except PyMongoError as e:
            logging.error(f"Error al confirmar la transacción: {e}")

    def revertir_transaccion(self):
        """Revertir (rollback) una transacción"""
        try:
            if self.session:
                self.session.abort_transaction()
                logging.info("Transacción revertida.")
        except PyMongoError as e:
            logging.error(f"Error al revertir la transacción: {e}")
```

7.4 Ejemplo de uso

```
# Ejemplo de uso del componente DatabaseManagerDocumental
if __name__ == "__main__":
    # Configurar el componente
    db_manager = DatabaseManagerDocumental(
        uri="mongodb://localhost:27017",
        database_name="1dam",
        collection_name="herramientas"
    )
```

```
    db_manager.conectar()


    try:
        # Crear documentos dentro de una transacción
        db_manager.iniciar_transaccion()
          db_manager.crear_documento({"nombre": "Martillo", "tipo": "Manual", "material": "Acero",
"marca": "Truper"})
               db_manager.crear_documento({"nombre":  "Taladro",  "tipo":  "Eléctrico",  "material":
"Plástico", "marca": "Bosch"})
        db_manager.confirmar_transaccion()


        # Leer todos los documentos
        db_manager.leer_documentos()


        # Actualizar un documento
        db_manager.iniciar_transaccion()
        db_manager.actualizar_documento({"nombre": "Martillo"}, {"material": "Acero reforzado"})
        db_manager.confirmar_transaccion()


        # Eliminar un documento
        db_manager.iniciar_transaccion()
        db_manager.eliminar_documento({"nombre": "Taladro"})
        db_manager.confirmar_transaccion()


    except Exception as e:
        logging.error(f"Error general: {e}")
        db_manager.revertir_transaccion()


    finally:
        db_manager.desconectar()
```

7.5. Log

```
2024-11-22 19:56:35,044 - INFO - Conectado a MongoDB: 1dam.herramientas
2024-11-22 19:56:35,044 - INFO - Transacción iniciada.
2024-11-22 19:56:35,108 - INFO - Documento insertado con ID: 6740d3e3a237cab24ba2c8d6
2024-11-22 19:56:35,109 - INFO - Documento insertado con ID: 6740d3e3a237cab24ba2c8d7
2024-11-22 19:56:35,109 - INFO - Transacción confirmada.
2024-11-22 19:56:35,120 - INFO - Documentos recuperados: 2
```

```
2024-11-22 19:56:35,120 - INFO - {'_id': ObjectId('6740d3e3a237cab24ba2c8d6'), 'nombre': 'Martillo', 'tipo': 'Manual', 'material': 'Acero', 'marca': 'Truper'}

2024-11-22 19:56:35,120 - INFO - {'_id': ObjectId('6740d3e3a237cab24ba2c8d7'), 'nombre': 'Taladro', 'tipo': 'Eléctrico', 'material': 'Plástico', 'marca': 'Bosch'}

2024-11-22 19:56:35,120 - INFO - Transacción iniciada.

2024-11-22 19:56:35,131 - INFO - Documento actualizado: {'nombre': 'Martillo'}

2024-11-22 19:56:35,131 - INFO - Transacción confirmada.

2024-11-22 19:56:35,131 - INFO - Transacción iniciada.

2024-11-22 19:56:35,133 - INFO - Documento eliminado: {'nombre': 'Taladro'}

2024-11-22 19:56:35,133 - INFO - Transacción confirmada.

2024-11-22 19:56:35,135 - INFO - Conexión a MongoDB cerrada.
```

Actividad de clase 4

Realiza el componente para el objeto que te fue asignado a principios de curso.

Adjunta dentro del pdf de esta unidad didáctica **3 capturas**:

1. Captura del código de tu actividad
2. Captura del resultado de la ejecución
3. Captura con los mensajes de log.

8. Programación de Componentes que Gestionan Información en Bases de Datos Orientadas a Objetos

8.1. Intoducción y recordatorio sobre bases de datos orientadas a objetos

Las bases de datos orientadas a objetos ofrecen una solución eficiente y natural para almacenar datos en estructuras complejas, permitiendo la persistencia de objetos directamente desde el código. En Python, **ZODB** (Zope Object Database) es una base de datos orientada a objetos altamente integrada, que permite trabajar con datos como si fueran objetos Python sin necesidad de un modelo relacional.

En este apartado, desarrollaremos un componente denominado **`DatabaseManagerObject`** que:

1. Proporcionará una interfaz CRUD para gestionar objetos persistentes.
2. Registrará todas las operaciones en un sistema de logging (con logs en consola y archivo).

3. Implementará soporte para transacciones para garantizar la consistencia en las operaciones críticas.

El objeto de trabajo será **`Herramienta`**, que representa una herramienta con atributos básicos como `nombre`, `tipo`, `material`, `uso` y `marca`.

8.2. Código del componente

```
import logging
import transaction
from ZODB import DB, FileStorage
from persistent import Persistent

# Configuración de logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("databasemanager_object.log"),  # Logs guardados en un archivo
        logging.StreamHandler(),  # Logs también en consola
    ]
)

class Herramienta(Persistent):
    """Clase que representa una herramienta."""
    def __init__(self, nombre, tipo, material, uso, marca):
        self.nombre = nombre
        self.tipo = tipo
        self.material = material
        self.uso = uso
        self.marca = marca



class DatabaseManagerObject:
    """Componente para gestionar bases de datos orientadas a objetos con ZODB."""
    def __init__(self, filepath="1dam.fs"):
        self.filepath = filepath
        self.db = None
```

```
        self.connection = None
        self.root = None
        self.transaccion_iniciada = False

    def conectar(self):
        """Conecta a la base de datos ZODB."""
        try:
            storage = FileStorage.FileStorage(self.filepath)
            self.db = DB(storage)
            self.connection = self.db.open()
            self.root = self.connection.root()
            if "herramientas" not in self.root:
                self.root["herramientas"] = {}
                transaction.commit()
            logging.info("Conexión establecida con ZODB.")
        except Exception as e:
            logging.error(f"Error al conectar a ZODB: {e}")

    def desconectar(self):
        """Cierra la conexión a la base de datos."""
        try:
            if self.connection:
                self.connection.close()
                self.db.close()
                logging.info("Conexión a ZODB cerrada.")
        except Exception as e:
            logging.error(f"Error al cerrar la conexión a ZODB: {e}")

    def iniciar_transaccion(self):
        """Inicia una transacción."""
        try:
            transaction.begin()
            self.transaccion_iniciada = True
            logging.info("Transacción iniciada.")
        except Exception as e:
            logging.error(f"Error al iniciar la transacción: {e}")

    def confirmar_transaccion(self):
```

```
        """Confirma la transacción."""
        if self.transaccion_iniciada:
            try:
                transaction.commit()
                self.transaccion_iniciada = False
                logging.info("Transacción confirmada.")
            except Exception as e:
                logging.error(f"Error al confirmar la transacción: {e}")

    def revertir_transaccion(self):
        """Revierte la transacción."""
        if self.transaccion_iniciada:
            try:
                transaction.abort()
                self.transaccion_iniciada = False
                logging.info("Transacción revertida.")
            except Exception as e:
                logging.error(f"Error al revertir la transacción: {e}")

    def crear_herramienta(self, id, nombre, tipo, material, uso, marca):
        """Crea y almacena una nueva herramienta."""
        try:
            if id in self.root["herramientas"]:
                raise ValueError(f"Ya existe una herramienta con ID {id}.")
            self.root["herramientas"][id] = Herramienta(nombre, tipo, material, uso, marca)
            logging.info(f"Herramienta con ID {id} creada exitosamente.")
        except Exception as e:
            logging.error(f"Error al crear la herramienta con ID {id}: {e}")

    def leer_herramientas(self):
        """Lee y muestra todas las herramientas almacenadas."""
        try:
            herramientas = self.root["herramientas"]
            for id, herramienta in herramientas.items():
                logging.info(
                    f"ID: {id}, Nombre: {herramienta.nombre}, Tipo: {herramienta.tipo}, "
                                f"Material: {herramienta.material}, Uso: {herramienta.uso}, Marca:
{herramienta.marca}"
```

```
                )
            return herramientas
        except Exception as e:
            logging.error(f"Error al leer las herramientas: {e}")

    def actualizar_herramienta(self, id, nombre, tipo, material, uso, marca):
        """Actualiza los atributos de una herramienta."""
        try:
            herramienta = self.root["herramientas"].get(id)
            if not herramienta:
                raise ValueError(f"No existe una herramienta con ID {id}.")
            herramienta.nombre = nombre
            herramienta.tipo = tipo
            herramienta.material = material
            herramienta.uso = uso
            herramienta.marca = marca
            logging.info(f"Herramienta con ID {id} actualizada exitosamente.")
        except Exception as e:
            logging.error(f"Error al actualizar la herramienta con ID {id}: {e}")

    def eliminar_herramienta(self, id):
        """Elimina una herramienta por su ID."""
        try:
            if id not in self.root["herramientas"]:
                raise ValueError(f"No existe una herramienta con ID {id}.")
            del self.root["herramientas"][id]
            logging.info(f"Herramienta con ID {id} eliminada exitosamente.")
        except Exception as e:
            logging.error(f"Error al eliminar la herramienta con ID {id}: {e}")
```

8.3. Ejemplo uso

```
if __name__ == "__main__":
    manager = DatabaseManagerObject()
    manager.conectar()

    try:
        # Crear herramientas con transacción
```

```
        manager.iniciar_transaccion()
        manager.crear_herramienta(1, "Martillo", "Manual", "Acero", "Percusión", "Truper")
        manager.crear_herramienta(2, "Taladro", "Eléctrico", "Plástico", "Perforación", "Bosch")
        manager.confirmar_transaccion()

        # Leer herramientas
        manager.leer_herramientas()

        # Actualizar una herramienta con transacción
        manager.iniciar_transaccion()
              manager.actualizar_herramienta(1, "Martillo Grande", "Manual", "Acero Reforzado",
"Percusión", "Stanley")
        manager.confirmar_transaccion()

        # Eliminar una herramienta con transacción
        manager.iniciar_transaccion()
        manager.eliminar_herramienta(2)
        manager.confirmar_transaccion()

        # Leer herramientas nuevamente
        manager.leer_herramientas()

    except Exception as e:
        logging.error(f"Error general: {e}")
        manager.revertir_transaccion()
    finally:
        manager.desconectar()
```

8.4. Actividad de clase 5

Construye este componente pero para el tipo de objeto que te fue asignado a principios de curso y además úsalo de la siguiente forma:

1) Inserta tres objetos (controlado con transacciones)

2) Muestra todos los objetos

3) Intenta insertar un objeto con un ID ya creado, controlado con transacciones

4) Muestra todos los objetos

5) Actualiza un objeto cambiando cualquier atributo, controlado con transacciones

6) Muestra todos los objetos

7) Elimina un objeto con id que no exista, controlado con transacciones

8) Muestra todos los objetos

Ejemplo del log que has de obtener:

```
2024-11-20 14:00:00 - INFO - Conexión establecida con ZODB.

2024-11-20 14:00:01 - INFO - Transacción iniciada.

2024-11-20 14:00:01 - INFO - Herramienta con ID 1 creada exitosamente.

2024-11-20 14:00:01 - INFO - Herramienta con ID 2 creada exitosamente.

2024-11-20 14:00:01 - INFO - Herramienta con ID 3 creada exitosamente.

2024-11-20 14:00:01 - INFO - Transacción confirmada.

2024-11-20 14:00:02 - INFO - ID: 1, Nombre: Martillo, Tipo: Manual, Material: Acero, Uso: Percusión, Marca: Truper

2024-11-20 14:00:02 - INFO - ID: 2, Nombre: Taladro, Tipo: Eléctrico, Material: Plástico, Uso: Perforación, Marca: Bosch

2024-11-20 14:00:02 - INFO - ID: 3, Nombre: Sierra, Tipo: Manual, Material: Acero, Uso: Corte, Marca: Stanley

2024-11-20 14:00:03 - INFO - Transacción iniciada.

2024-11-20 14:00:03 - ERROR - Error al crear la herramienta con ID 1: Ya existe una herramienta con ID 1.

2024-11-20 14:00:03 - INFO - Transacción revertida.

2024-11-20 14:00:04 - INFO - ID: 1, Nombre: Martillo, Tipo: Manual, Material: Acero, Uso: Percusión, Marca: Truper

2024-11-20 14:00:04 - INFO - ID: 2, Nombre: Taladro, Tipo: Eléctrico, Material: Plástico, Uso: Perforación, Marca: Bosch

2024-11-20 14:00:04 - INFO - ID: 3, Nombre: Sierra, Tipo: Manual, Material: Acero, Uso: Corte, Marca: Stanley

2024-11-20 14:00:05 - INFO - Transacción iniciada.

2024-11-20 14:00:05 - INFO - Herramienta con ID 2 actualizada exitosamente.

2024-11-20 14:00:05 - INFO - Transacción confirmada.

2024-11-20 14:00:06 - INFO - ID: 1, Nombre: Martillo, Tipo: Manual, Material: Acero, Uso: Percusión, Marca: Truper

2024-11-20 14:00:06 - INFO - ID: 2, Nombre: Taladro Industrial, Tipo: Eléctrico, Material: Metal, Uso: Perforación, Marca: Bosch

2024-11-20 14:00:06 - INFO - ID: 3, Nombre: Sierra, Tipo: Manual, Material: Acero, Uso: Corte, Marca: Stanley

2024-11-20 14:00:07 - INFO - Transacción iniciada.

2024-11-20 14:00:07 - ERROR - Error al eliminar la herramienta con ID 99: No existe una herramienta con ID 99.

2024-11-20 14:00:07 - INFO - Transacción revertida.
```

```
2024-11-20 14:00:08 - INFO - ID: 1, Nombre: Martillo, Tipo: Manual, Material: Acero, Uso: Percusión, Marca: Truper

2024-11-20 14:00:08 - INFO - ID: 2, Nombre: Taladro Industrial, Tipo: Eléctrico, Material: Metal, Uso: Perforación, Marca: Bosch

2024-11-20 14:00:08 - INFO - ID: 3, Nombre: Sierra, Tipo: Manual, Material: Acero, Uso: Corte, Marca: Stanley

2024-11-20 14:00:08 - INFO - Conexión a ZODB cerrada.
```

Anexo A. Uso de la Biblioteca `logging`

La biblioteca `logging` en Python es una herramienta que permite capturar información sobre el estado y el flujo de ejecución de un programa en tiempo real. Mediante el uso de `logging`, los desarrolladores pueden registrar eventos importantes, advertencias, errores y otros datos útiles para depuración y monitoreo. En aplicaciones de acceso a datos, como el componente `DataManager`, `logging` ayuda a llevar un registro de las transacciones, cambios en el archivo, y otros eventos importantes, lo que es fundamental para asegurar la trazabilidad y mantener la integridad de los datos.

A.1. Ventajas del uso de `logging` en Aplicaciones de Acceso a Datos

El registro de actividad aporta varios beneficios clave en el contexto del acceso a datos:

- **Trazabilidad**: Permite un seguimiento detallado de las operaciones que se realizan en los datos, como modificaciones, eliminaciones o consultas, lo cual es útil para auditar y depurar problemas.
- **Detección de Errores**: Facilita la identificación de errores en el código al registrar información sobre la ejecución, como advertencias y excepciones.
- **Mantenimiento y Diagnóstico**: Ayuda a los desarrolladores a entender el comportamiento del sistema en producción y a diagnosticar problemas sin interrumpir el servicio.
- **Notificaciones y Alertas**: Es posible configurar el sistema de `logging` para que envíe notificaciones o alertas ante eventos críticos, como fallos en la conexión o transacciones fallidas.

A.2. Conceptos Fundamentales de `logging`

Python proporciona cinco niveles de `logging` que ayudan a categorizar los eventos de la aplicación:

1. **DEBUG**: Proporciona información detallada y granular, generalmente útil solo durante la fase de desarrollo.
2. **INFO**: Registra información general sobre el estado de la aplicación, como operaciones exitosas y eventos de rutina.
3. **WARNING**: Indica un problema potencial o una situación inusual que no detiene la ejecución del programa pero merece atención.
4. **ERROR**: Registra problemas que impiden que una operación específica se complete con éxito.
5. **CRITICAL**: Señala un problema grave que afecta el funcionamiento de la aplicación, como una pérdida de conexión a la base de datos o un fallo en una transacción crítica.

A.3. Configuración de `logging`

En el componente `DataManager`, configuramos el `logging` para capturar mensajes a nivel `INFO` y superiores, con un formato estándar que incluye la fecha, la hora, el nivel del mensaje y el contenido del mensaje:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
```

En esta configuración:

- **level=logging.INFO** establece que solo se registrarán mensajes a partir del nivel `INFO`. Los mensajes `DEBUG` no se registrarán.
- **format** define el formato del mensaje de logging:
    - `%(asctime)s`: Muestra la fecha y hora del evento.
    - `%(levelname)s`: Indica el nivel de importancia del evento (`INFO`, `WARNING`, `ERROR`, etc.).
    - `%(message)s`: Contiene el mensaje de logging.

Este formato es adecuado para el registro en aplicaciones de acceso a datos, ya que permite visualizar rápidamente cuándo ocurrió un evento y el nivel de importancia asociado.

A.4. Uso de `logging` en el Componente `DataManager`

En el componente `DataManager`, `logging` se utiliza para registrar varios eventos importantes. Veamos los casos específicos de cómo se aplican los distintos niveles de `logging` en este contexto.

1. **INFO**: Registro de eventos rutinarios y exitosos, como la carga y guardado de datos, inicio y confirmación de transacciones, y la modificación de datos.

   ```
   logging.info(f"Archivo {self.tipo_archivo.upper()} guardado. Versión actual: {self.version}")
   ```

   Este mensaje se registra cuando el archivo de datos (JSON o CSV) se guarda correctamente, informando sobre la versión actual del archivo. Es útil para llevar un control de los cambios y la frecuencia con la que se modifica el archivo.

2. **WARNING**: Advertencias para situaciones inusuales o posibles problemas, como la reversión de una transacción.

   ```
   logging.warning("Transacción revertida. Los cambios no se guardaron.")
   ```

Este mensaje se registra si la transacción se revierte, alertando al usuario de que los cambios no se han confirmado. Esto permite a los desarrolladores o administradores identificar transacciones fallidas que podrían requerir atención.

3. **ERROR y CRITICAL**: Aunque en el ejemplo no incluimos mensajes `ERROR` o `CRITICAL`, podrían agregarse para eventos críticos, como fallos de conexión o errores inesperados en el sistema de archivos.

```
if not os.path.exists(self.ruta_archivo):
    logging.error("No se encontró el archivo especificado para lectura.")
```

Un mensaje `ERROR` como este sería útil si se intenta leer un archivo que no existe, alertando de un fallo en una operación de acceso a datos esencial.

A.5. Personalización y Extensión del `logging`

Python permite configurar `logging` de varias maneras avanzadas para adaptarlo a diferentes necesidades en producción:

- **Múltiples destinos (Handlers)**: Se puede configurar el `logging` para enviar mensajes a diferentes destinos, como un archivo de texto, una base de datos, o incluso un servidor remoto. Esto es útil para mantener un archivo de registro detallado y consultar el historial de eventos cuando sea necesario.

```
file_handler = logging.FileHandler('log_datos.log')
file_handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
logging.getLogger().addHandler(file_handler)
```

Este código añade un `FileHandler` para guardar todos los mensajes de `logging` en un archivo llamado `log_datos.log`. De esta manera, se puede mantener un historial persistente de los eventos en un archivo.

- **Configuración de Niveles de `logging` por Handler**: Si se necesitan mensajes de `DEBUG` solo para un archivo específico y mensajes `INFO` para la consola, se pueden aplicar configuraciones distintas a cada `Handler`.
- **Logging jerárquico**: Se pueden configurar loggers específicos para diferentes partes de una aplicación y asignarles configuraciones independientes, permitiendo un control detallado de qué eventos se registran y dónde.

A.6. Ejemplos de Logging en `DataManager`

A continuación, se muestra cómo se aplican las operaciones de logging en el componente `DataManager`:

```
# Ejemplo de inicio de una transacción y registro de eventos
def iniciar_transaccion(self):
    if self.transaccion_activa:
        logging.error("Ya hay una transacción activa.")
        raise Exception("Transacción ya en curso")
    self.transaccion_activa = True
    self.copia_datos = deepcopy(self.datos)
    logging.info("Transacción iniciada.")

# Ejemplo de confirmación de una transacción
def confirmar_transaccion(self):
    if not self.transaccion_activa:
        logging.warning("Intento de confirmación sin transacción activa.")
        raise Exception("No hay transacción para confirmar")
    self.version += 1
    self.transaccion_activa = False
    self.copia_datos = None
    self._guardar_archivo()
    logging.info("Transacción confirmada y cambios guardados.")
```

En estos ejemplos:

- El método `iniciar_transaccion` verifica si ya hay una transacción activa. Si es así, registra un mensaje de error y lanza una excepción. Si no hay transacción activa, inicia una y registra el evento.
- El método `confirmar_transaccion` registra un mensaje de advertencia si se intenta confirmar una transacción sin que exista una activa, lo cual evita errores potenciales.

Resumen

La biblioteca `logging` en Python proporciona una forma robusta y flexible de monitorear y depurar la actividad en aplicaciones de acceso a datos. En el componente `DataManager`, el `logging` permite:

- Mantener un registro de todas las transacciones y operaciones de modificación de datos.
- Generar advertencias cuando ocurren condiciones inusuales.
- Facilitar la detección de problemas mediante el registro de errores.
- Ofrecer una trazabilidad completa de eventos, útil para auditoría y mantenimiento.

Este sistema de registro es esencial para garantizar que las aplicaciones mantengan un historial detallado de sus operaciones, y es una práctica clave en el desarrollo de componentes profesionales y fiables en aplicaciones orientadas a datos.