Persistence with Files in Python using the DAO Pattern

Roberto A. Bittencourt

DAO Pattern

Data Access Object (DAO)

- The Data Access Object (DAO) pattern is a structural design pattern that provides an abstract interface to some type of database or other persistence mechanism.
- By mapping application calls to the persistence layer, DAOs facilitate the separation of business logic from database operations.
- DAO is also known as:
  - Data Mapper
  - Repository Pattern (closely related)

DAO Goals

- The goals of the DAO pattern are to:
  - Separate database access logic from business logic.
  - Encapsulate the access to data sources.
  - Provide a uniform interface to access data from different sources.

DAO Pros and Cons

- **Key Features:**
  - **Encapsulation**: Hides the details of data access logic.
  - **Abstraction**: Provides an abstract interface to different types of data sources.
  - **Decoupling**: Decouples business logic from data access logic.
  - **Maintainability**: Makes the code easier to maintain and test
- **Trade-offs**
  - **Complexity**: Adds an extra layer of abstraction which could increase the complexity of the project.
  - **Performance**: Overhead of additional method calls and object creation.

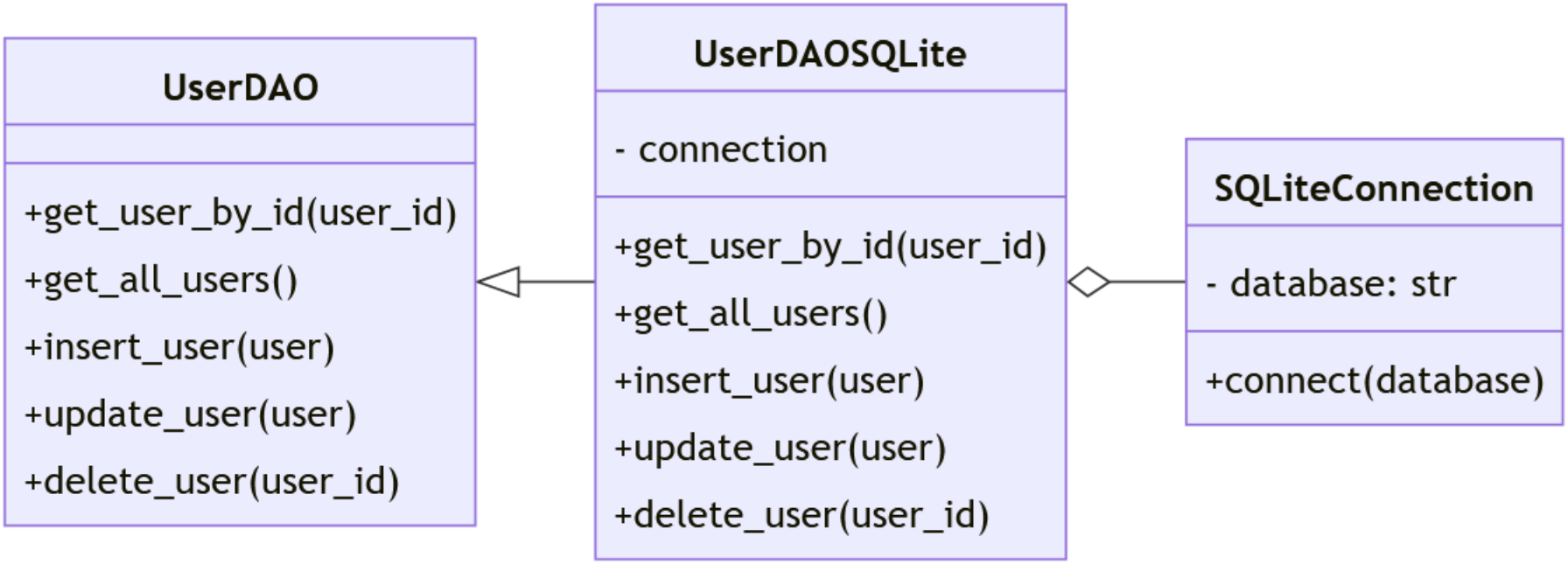
How it works

- The DAO pattern abstracts and encapsulates all access to the data source.
- The underlying data source could be a database, file system, or any other persistence mechanism.
- This abstraction allows for flexibility and makes the application easier to maintain and test.

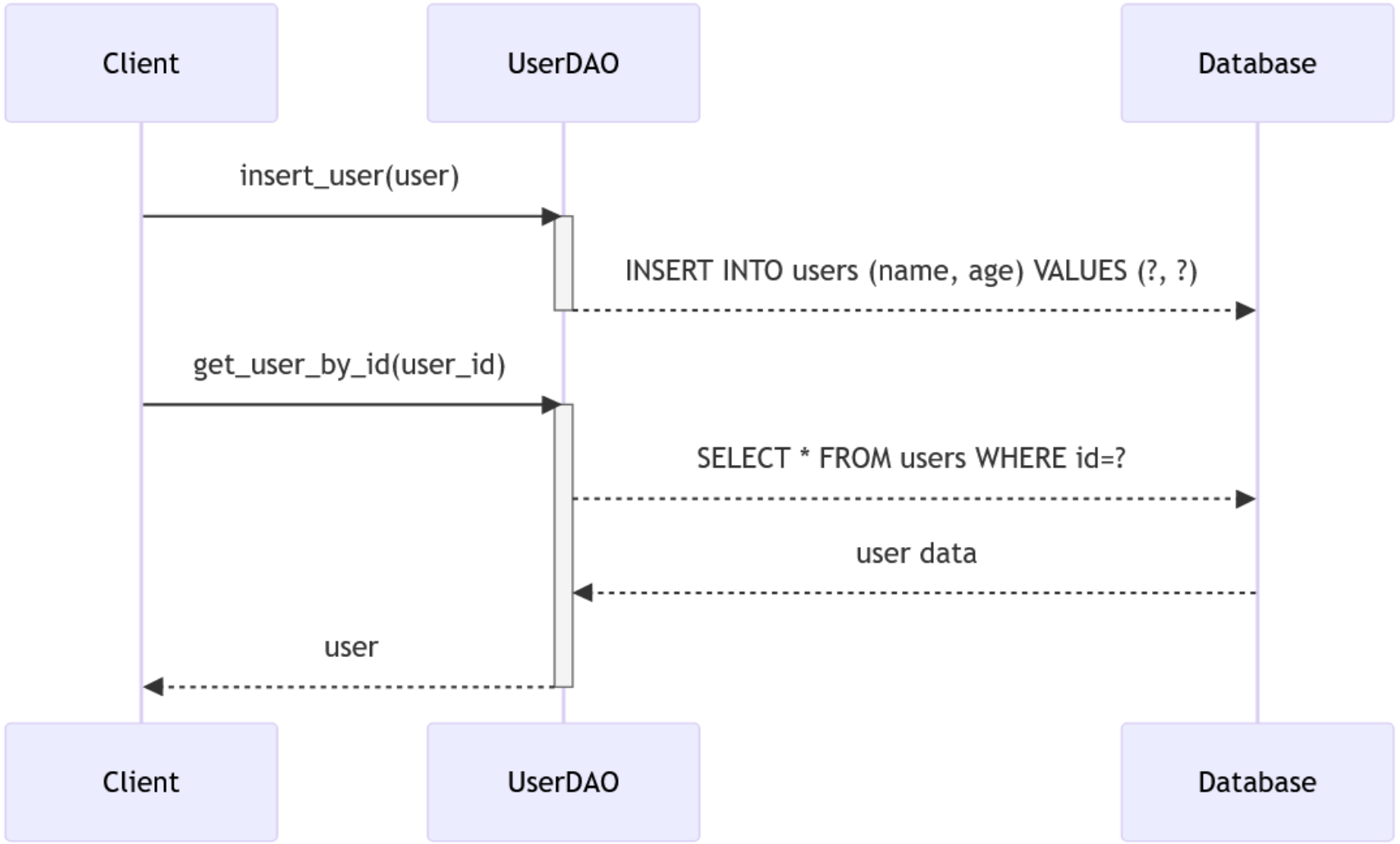
Components of a DAO PAttern

- The pattern typically involves the following components:
  - **DAO Interface**: Defines the standard operations to be performed on a model object(s).
  - **DAO Implementation**: Implements the concrete operations defined in the DAO interface.
  - **Model Object**: Represents the data being accessed.
  - **Client**: Utilizes the DAO interface for performing CRUD operations.

DAO Structure



DAO Behavior



Code: Model Object

```
class User:
    def __init__(self, user_id, name, age):
        self.id = user_id
        self.name = name
        self.age = age
```

Code: DAO Interface

```
from abc import ABC, abstractmethod
class UserDAO(ABC):
    @abstractmethod
    def get_user_by_id(self, key):
        pass

    @abstractmethod
    def get_all_users(self):
        pass

    @abstractmethod
    def insert_user(self, user):
        pass

    @abstractmethod
    def update_user(self, user):
        pass

    @abstractmethod
    def delete_user(self, user_id):
        pass
```

Code: DAO Implementation for databases

```
class UserDAOSQLite(UserDAO):
    def __init__(self, database):
        self.connection = sqlite3.connect(database)
    def get_user_by_id(self, user_id):
        cursor = self.connection.cursor()
        cursor.execute('SELECT * FROM users WHERE id=?', (user_id,))
        return cursor.fetchone()
    def get_all_users(self):
        cursor = self.connection.cursor()
        cursor.execute('SELECT * FROM users')
        return cursor.fetchall()
    def insert_user(self, user):
        cursor = self.connection.cursor()
        cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)',
         (user.name, user.age))
        self.connection.commit()
    def update_user(self, user):
        cursor = self.connection.cursor()
        cursor.execute('UPDATE users SET name=?, age=? WHERE id=?',
         (user.name, user.age, user.id))
        self.connection.commit()
    def delete_user(self, user_id):
        cursor = self.connection.cursor()
        cursor.execute('DELETE FROM users WHERE id=?', (user_id,))
        self.connection.commit()
```

Code: Client Code

```
def main():
    user_dao = UserDAOSQLite('users.db')

    new_user = User(None, 'John Doe', 30)
    user_dao.insert_user(new_user)

    user = user_dao.get_user_by_id(1)
    print(user)

    all_users = user_dao.get_all_users()
    print(all_users)


if __name__ == "__main__":
    main()
```

DAO Pattern Implementation with Python File Libraries

Some design decisions with our example model scenario…

- Reusing the example classes from our exemple model:
  - School and Student
- Recalling that School has a collections of students
  - We will need access to this collection in the DAO class
- Student will be our model class
  - We will then create StudentDAO
- Using a single file to store the collection of students
- Testing our system for persistence by changing the tests for our School class

Recalling the **Student** model class...

```
from datetime import date
class Student:
  def __init__(self,name,number,birth_year):
    self.name = name
    # number will be the key
    self.number = number
    self.birth_year = birth_year

  def age(self):
    currentYear = date.today().year
    return currentYear - self. birth_year

  def __eq__(self, other):
    return self.name == other.name and self.number  == \
     other.number and self.birth_year == other.birth_year

  def __str__(self):
    return "name: %s, number: %d, birth year: %d" % \
     (self.name, self.number, self.birth_year)
```

Recalling the **School** model class that works as a container of students...

```
class School:
    def __init__(self, name):
        self.name = name
        self.students = []

    def search(self, key):
        # code returns a Student
    def create(self, name, number, birth_year):
        # code returns a boolean
    def retrieve(self, name):
        # code returns a list of students
    def update(self, key, name, number, birth_year):
        # code returns a boolean
    def delete(self, key):
        # code returns a boolean
    def list_all(self):
        # code returns a list of students
```

StudentDAO Interface

```
from abc import ABC, abstractmethod
class StudentDAO(ABC):
    @abstractmethod
    def search_student(self, key):
        pass
    @abstractmethod
    def create_student(self, student):
        pass
    @abstractmethod
    def retrieve_students(self, name):
        pass
    @abstractmethod
    def update_student(self, key, student):
        pass
    @abstractmethod
    def delete_student(self, key):
        pass
    @abstractmethod
    def list_all_students(self):
        pass
```

How to handle collections in this new scenarios with persistence?

- A simple solution would be not using DAO. Keep the CRUD methods in School just the way they are and:
  - Inside the School constructor, load the students' file into the student collection.
  - For each operation that changes data (create, update and delete), save the full student collection into the students' file
- The above solution is simple but is also bad
  - If we decide to change persistence technologies (e.g., using relational databases or using cloud storage), changes would affect much of the School code
- Our solution using DAO is simple:
  - Put both the student collection and file access inside the DAO

How would the DAO use affect School? Let us refactor the constructor first

```
class School:

  def __init__(self, name):
    self.name = name
    # the DAO below will depend on the technology
    # replace ConcreteStudentDAO with the concrete
    # DAO you decided to use according to chosen tech
    # (e.g, StudentDAOSQLite, StudentDAOPickle,
    #  StudentDAOJSON, StudentDAOCloudStorage)
    self.student_dao = ConcreteStudentDAO()
```

Let us also refactor the School methods to delegate the work to the DAO...

```
# ... Continuing the School class

def search(self, key):
    return self.student_dao.search_student(key)
def create(self, name, number, birth_year):
    student = Student(name, number, birth_year)
    return self.student_dao.create_student(student)
def retrieve(self, name):
    return self.student_dao.retrieve_students(name)
def update(self, key, name, number, birth_year):
    student = Student(name, number, birth_year)
    return self.student_dao.update_student(key, student)
def delete(self, key):
    return self.student_dao.search_student(key)
def list_all(self):
    return self.student_dao.list_all_students()
```

Now, let us create the concrete DAO class

- Your concrete DAO class will have:
  - The student collection
  - Access to the persistence mechanism
- Your DAO class will operate on:
  - Accessing the student collection (reuse and change the existing CRUD code from School)
  - Accessing the persistence mechanism (e.g, loading, saving)
- In the next slides, we will create concrete DAO classes for the Python **`pickle`** library

StudentDAO with the pickle library

Will the also DAO affect testing?

- Now your testing code will handle two different issues:
  - Testing whether the collection is working correctly
  - Testing whether the persistence is also working correctly
- Do you want your tests to handle those issues together or separately?
- Let us deal with them separately by using a system property called **autosave**
  - By default, autosave is off: your tests check collections only
  - If you turn **autosave** on: tests check both collections and persistence
  - You will need to retrieve the **autosave** value along your system when you build objects that may persist
    - A simple way to recover autosave is by using class variables in a Configuration class
- Obs.: if you are dealing with a database or with cloud storage, you will only need to handle persistence
  - The other issues are part of those outside system's internals

Let us first create a Configuration class to hold information about persistence...

```
class Configuration:
    # Note the lack of a constructor
    # Also note below the class variables ("static")
    # autosave is initially false, but integration tests
    # may change that in the setUp()
    autosave = False
    filename = 'students.dat'
```

Constructor for StudentDAOPickle

```
from pickle import load, dump
from student_dao import StudentDAO

class StudentDAOPickle(StudentDAO):

    def __init__(self):
        conf = Configuration()
        self.autosave = conf.__class__.autosave
        self.filename = conf.__class__.filename
        if self.autosave:
            try:
                with open(self.filename, 'rb') as file:
                    self.students = load(file)
            except FileNotFoundError:
                self.students = []
        else:
            self.students = []
```

CRUD methods for StudentDAOPickle (1)

```
# continuing StudentDAOPickle (1)

    def search_student(self, key):
        # unordered list requests a linear search
        for element in self.students:
            if (element.number == key):
                return element
        return None
    def create_student(self, student):
        if not self.search_student(student.number):
            self.students.append(student)
            # save file after creating student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def retrieve_students(self, name):
        retrieved = []
        for element in self.students:
            if name in element.name:
                retrieved.append(element)
        return retrieved
```

CRUD methods for StudentDAOPickle (2)

```
# continuing StudentDAOPickle (2)
    def update_student(self, key, student):
        existing_student = self.search_student(key)
        if existing_student:
            existing_student = student
            # save file after updating student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def delete_student(self, key):
        element = self.search_student(key)
        if element:
            self.students.remove(element)
            # save file after deleting student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def list_all_students(self):
        students_list = []
        for student in self.students:
            students_list.append(student)
        return students_list
```

StudentDAO works! How to test it?

- So far, we have just created the structure to handle tests both without persistence and with persistence
- The way we designed our DAO class with default autosave as False makes the previous School tests work with no changes testing models in memory only
- Next class, we will see how to Test School with persistence by making some changes to the tests
  - Those will be our persistence/integration tests