Laboratory Activity Part A: Pattern Identification & Justification

Pattern	Application in Project	Justification
Singleton	**DatabaseConnection class in database.py.**	**Prevents opening multiple redundant connections to the SQLite file, reducing overhead and potential locking issues.**
Repository	**CourseRepository and StudentRepository.**	**Decouples the models from the SQL queries. If you ever switch from SQLite to MySQL, you only change the Repository, not the rest of the app.**
Factory	**RepositoryFactory.**	**Simplifies object creation. The Service layer doesn't need to know "how" to build a repository; it just asks the Factory for one.**

PART B - IMPLEMENTATION

1. Singleton Pattern: Database Connection

Purpose: Ensures only one instance of the database connection manager exists to save resources.

Refactored config/database.py

Python

```
import sqlite3


class DatabaseConnection:
    _instance = None


    def __new__(cls):
        if cls._instance is None:
```

```
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance._conn = sqlite3.connect('online_courses.db', check_same_thread=False)
            cls._instance._conn.row_factory = sqlite3.Row
        return cls._instance

    def get_connection(self):
        return self._instance._conn

def init_db():
    db = DatabaseConnection()
    conn = db.get_connection()
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS courses (id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL, description TEXT)''')
    cursor.execute('''CREATE TABLE IF NOT EXISTS students (id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL, email TEXT UNIQUE NOT NULL)''')
    conn.commit()

init_db()
```

plantUml DIAGRAM :

```
@startuml
skinparam style strictuml
skinparam monochrome true

class DatabaseConnection <<Singleton>> {
    - static _instance: DatabaseConnection
    - _conn: sqlite3.Connection
    --
    + __new__(): DatabaseConnection
```

+ get_connection(): Connection

}

note right of DatabaseConnection::__new__

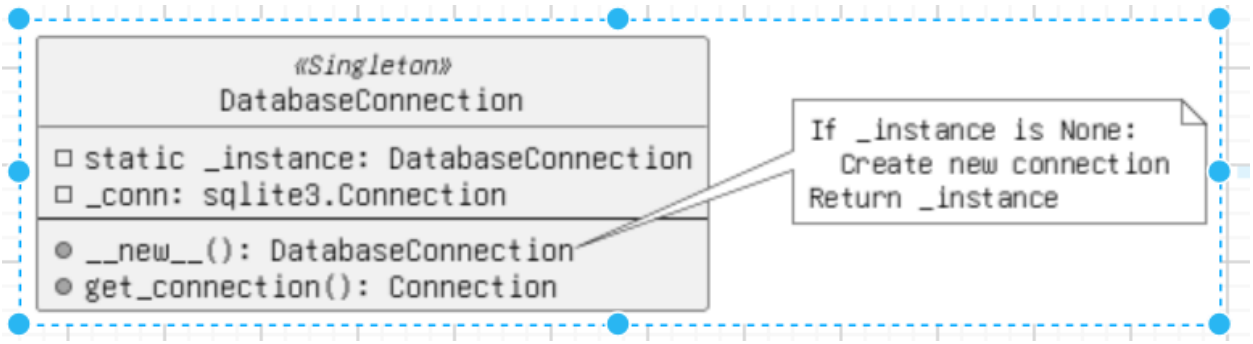
If _instance is None:

Create new connection

Return _instance

end note

@enduml



2. Repository Pattern: Data Access

Purpose: Separates the business logic from the data access logic. We move the SQL queries out of the Models and into dedicated Repository classes.

New repositories/courseRepository.py

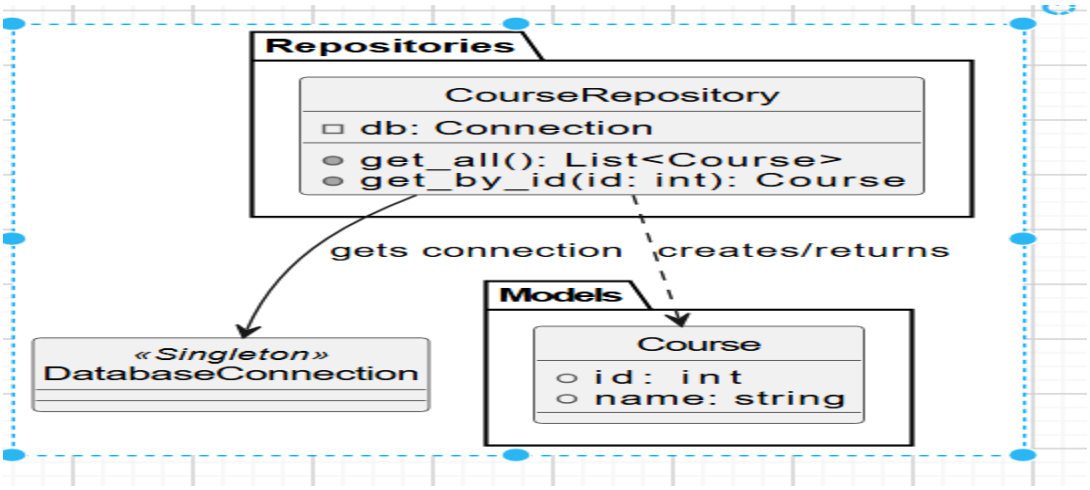
Python

```
from config.database import DatabaseConnection
from models.courseModel import Course


class CourseRepository:
    def __init__(self):
        self.db = DatabaseConnection().get_connection()


    def get_all(self):
        cursor = self.db.cursor()
        cursor.execute("SELECT * FROM courses")
        return [Course(row['id'], row['name'], row['description']) for row in cursor.fetchall()]


    def get_by_id(self, course_id):
        cursor = self.db.cursor()
        cursor.execute("SELECT * FROM courses WHERE id=?", (course_id,))
        row = cursor.fetchone()
        return Course(row['id'], row['name'], row['description']) if row else None
```

plantUml DIAGRAM :

```
@startuml
skinparam style strictuml
skinparam monochrome true


package "Models" {
    class Course {
        + id: int
        + name: string
    }
}
```

```
package "Repositories" {
    class CourseRepository {
        - db: Connection
        + get_all(): List<Course>
        + get_by_id(id: int): Course
    }
}

class DatabaseConnection <<Singleton>>

CourseRepository --> DatabaseConnection : gets connection
CourseRepository ..> Course : creates/returns
@enduml
```



3. Factory Pattern: Object Creation

Purpose: Handles the instantiation of repositories or models, providing a centralized way to create objects without specifying the exact class.

New factories/repoFactory.py

Python

```
from repositories.courseRepository import CourseRepository
```

```
# Assume a similar StudentRepository exists
from repositories.studentRepository import StudentRepository

class RepositoryFactory:
    @staticmethod
    def get_repository(repo_type):
        if repo_type == 'course':
            return CourseRepository()
        elif repo_type == 'student':
            return StudentRepository()
        return None
```

Refactored Service Layer

The Services now use the **Factory** to get a **Repository**, which uses the **Singleton** database connection.

Refactored services/courseService.py

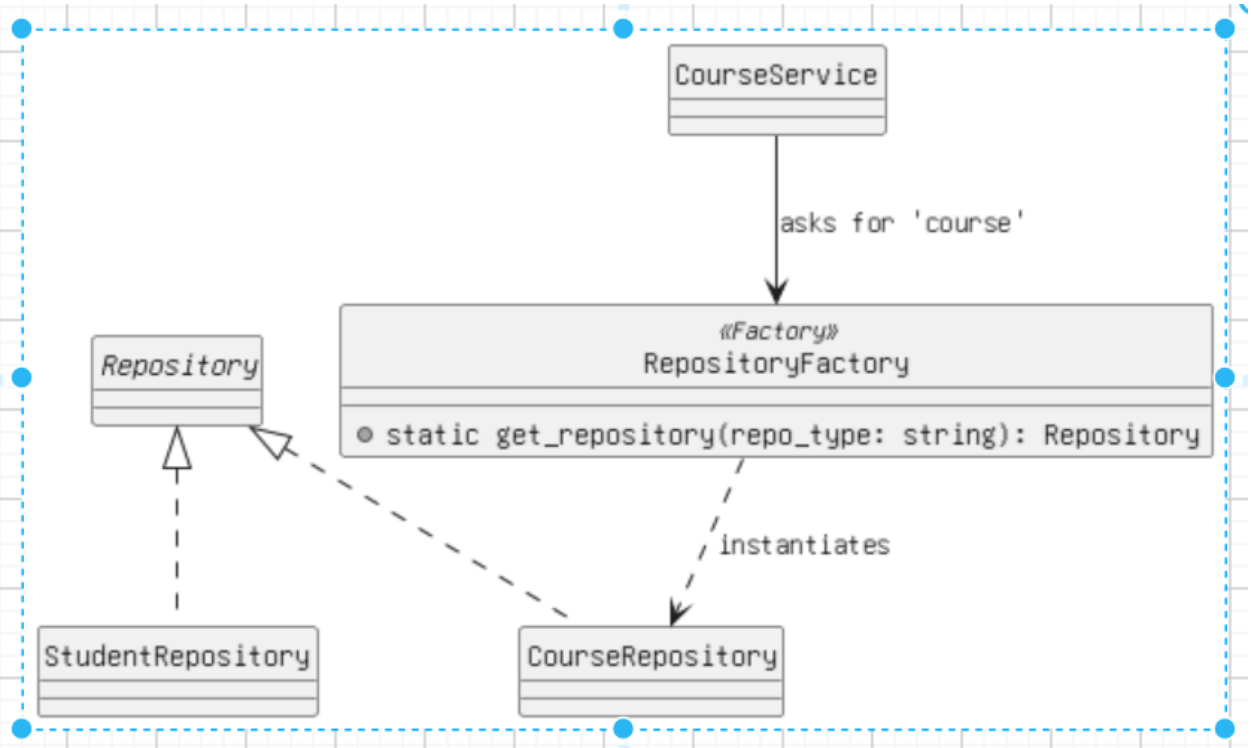
Python

```
from factories.repoFactory import RepositoryFactory

# Use the factory to get the repository
course_repo = RepositoryFactory.get_repository('course')

def get_all_courses():
    return course_repo.get_all()

def get_course(course_id):
    return course_repo.get_by_id(course_id)
```

plantUml DIAGRAM:

```
@startuml
```

```
skinparam style strictuml
skinparam monochrome true

interface Repository

class CourseRepository implements Repository
class StudentRepository implements Repository

class RepositoryFactory <<Factory>> {
   + static get_repository(repo_type: string): Repository
}

class CourseService

CourseService --> RepositoryFactory : asks for 'course'
RepositoryFactory ..> CourseRepository : instantiates
@enduml
```



Laboratory Activity Part C: Documentation (UML Description)

To complete your **draw.io** task:

1. **Singleton:** Draw a class DatabaseConnection with a private static attribute _instance and a public get_connection() method.
2. **Repository:** Draw an Interface/Class CourseRepository with methods like get_all(). Show it interacting with the DatabaseConnection.
3. **Factory:** Draw a RepositoryFactory class with a method get_repository(type). Draw arrows showing it "creates" the Repository classes.

Short design explanation document :

1. Structural Overview: MVC

The application is divided into three main layers to separate concerns:

- **Models:** Define the data structure (Course and Student).
- **Views:** HTML templates (Jinja2) for the user interface.
- **Controllers:** Handle user requests and coordinate between Services and Views.

---

2. Design Pattern Implementations

A. Singleton Pattern (Resource Management)

Location: config/database.py The **Singleton** pattern is applied to the database connection. By using the __new__ method, we ensure that only one instance of the connection exists throughout the application's lifecycle.

- **Benefit:** Prevents multiple connections from slowing down the system and avoids "Database is locked" errors in SQLite.

B. Repository Pattern (Data Abstraction)

Location: repositories/ folder We decoupled the data access logic from the Models. The **Repositories** contain all the raw SQL queries.

- **Benefit:** If the database type changes (e.g., from SQLite to PostgreSQL), only the Repository files need to be modified; the Models and Services remain untouched.

C. Factory Pattern (Object Creation)

Location: factories/repoFactory.py The **Factory** pattern provides a centralized interface for creating objects. Instead of the Service layer manually importing and instantiating different repositories, it calls RepositoryFactory.get_repository('type').

- **Benefit:** Simplifies the Service layer and makes it easier to add new entities (like "Instructors" or "Grades") in the future.

---

3. System Flow

1. **Request:** User accesses /courses.
2. **Controller:** courseController receives the request.
3. **Service:** Calls courseService, which asks the **Factory** for a **Repository**.
4. **Repository:** Uses the **Singleton** connection to fetch data and maps it to a **Model**.
5. **Response:** The Controller sends the data to the **View** for rendering.