ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẶC TẢ ĐỒ ÁN

DATABASE ACCESS MANAGEMENT (DAM) FRAMEWORK

Khoa Công nghệ Thông tin
Bộ môn Công Nghệ Phần Mềm

Tp. Hồ Chí Minh, tháng 01 năm 2024

Mục lục

I. Thông tin nhóm

a. Danh sách thành viên

MSSV	Họ và tên	Email
20120032	Phan Trường An	20120032@student.hcmus.edu.vn
20120068	Phan Duy	20120068@student.hcmus.edu.vn
20120075	Lê Thị Minh Hiền	20120075@student.hcmus.edu.vn
20120251	Trần Đức Anh	20120251@student.hcmus.edu.vn

b. Phân chia công việc

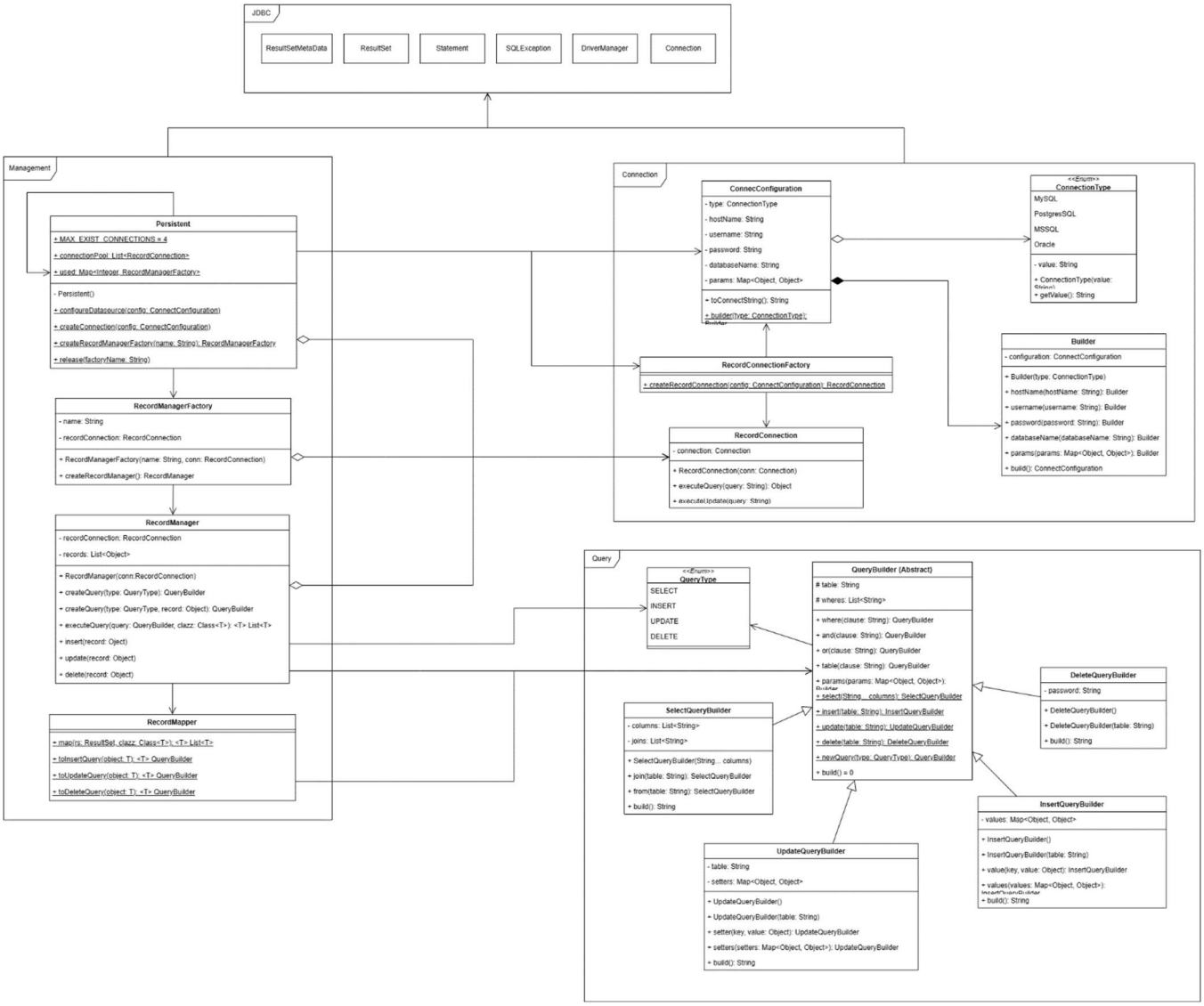
Task	Người thực hiện	Mức độ hoàn thành
Thống nhất các mẫu thiết kế có trong framework.	Trần Đức Anh, Lê Thị Minh Hiền, Phan Trường An, Phan Duy	100%
Vẽ class diagram, hỗ trợ viết tài liệu và hỗ trợ cài đặt framework.	Trần Đức Anh	100%
Cài đặt hầu hết Framework, hỗ trợ implement package.	Lê Thị Minh Hiền	100%
Implement package vào project, hỗ trợ cài đặt framework, quay video demo.	Phan Trường An	100%
Tổng hợp tài liệu, viết báo cáo, hỗ trợ cài đặt framework, kiểm thử implement.	Phan Duy	100%

c. Tỉ lệ điểm từng thành viên

Họ và tên	Tỉ lệ
Phan Trường An	25%
Phan Duy	25%
Lê Thị Minh Hiền	25%
Trần Đức Anh	25%
Tổng:	**100%**

II. Sơ đồ lớp của toàn bộ framework

a. Sơ đồ lớp



Hình 1 Class Diagam

b. Ý nghĩa của từng lớp

- **Package Connection:**
  - **Lớp ConnectConfiguration:** có nhiệm vụ định nghĩa cấu trúc thông tin kết nối đến cơ sở dữ liệu và cung cấp cách tiện lợi để xây dựng và quản lý thông tin này thông qua phương thức builder.

Các thành phần chính	Ý nghĩa
ConnectionType	- Là một đối tượng thể hiện kiểu kết nối (JDBC) - Được sử dụng trong việc xây dựng chuỗi kết nối
HostName	Lưu trữ thông tin về địa chỉ máy chủ của cơ sở dữ liệu
Username	Lưu trữ tên người dùng sử dụng để đăng nhập vào cơ sở dữ liệu
Password	Lưu trữ mật khẩu sử dụng để đăng nhập vào cơ sở dữ liệu
DatabaseName	Lưu trữ tên cơ sở dữ liệu mà kết nối đang định kết nối đến
Params	- Là một bảng ánh xạ các tham số khác nhau cần thiết cho việc kết nối (vd: timeout, ssl...) - Được sử dụng để thêm các tham số tùy chọn vào chuỗi kết nối
toConnectString()	- Phương thức này tạo và trả về chuỗi kết nối dựa trên thông tin đã được thiết lập trong các thuộc tính của đối tượng ConnectConfiguration - Sử dụng StringBuilder để hiệu quả hóa quá trình nối chuỗi
Builder	- Là một lớp nội tĩnh trong ConnectConfiguration - Cung cấp các phương thức để thiết lập thông tin kết nối bước từ bước, giúp tạo ra một đối tượng ConnectConfiguration hoàn chỉnh

  - **Lớp ConnectionType:** là một enum định nghĩa các kiểu kết nối. Có nhiệm vụ giúp quản lý và định nghĩa một cách cụ thể các kiểu kết nối khác nhau mà framework hỗ trợ. Việc sử dụng enum giúp giảm khả năng nhập sai giá trị khi khai báo kiểu kết nối, đồng thời giúp mã nguồn trở nên rõ ràng và dễ bảo trì.

Các thành phần chính	Ý nghĩa
Enum Constants	- MySQL, PostgresSQL, MSSQL, Oracle là các hằng số enum đại diện cho các kiểu kết nối cụ thể - Mỗi hằng số được khởi tạo với một giá trị chuỗi tương ứng với tên của kiểu kết nối
Value	- Là một trường private lưu trữ giá trị chuỗi của mỗi hằng số enum - Được khởi tạo thông qua constructor khi enum constants được khai báo
Constructor	- Hàm dựng của enum được gọi khi mỗi hằng số enum được khởi tạo - Hàm này thiết lập giá trị cho trường value tương ứng với giá trị chuỗi của kiểu kết nối
getValue()	- Là một phương thức công khai cung cấp giá trị chuỗi của kiểu kết nối - Được sử dụng trong quá trình xây dựng chuỗi kết nối trong lớp ConnectConfiguration

- **Lớp RecordConnection:** đơn giản hóa quá trình thực hiện truy vấn và cập nhật đối với cơ sở dữ liệu thông qua một đối tượng kết nối.

Các thành phần chính	Ý nghĩa
Connection	- Là trường final (private final Connection connection) lưu trữ đối tượng kết nối đến cơ sở dữ liệu - Được khởi tạo thông qua constructor khi đối tượng RecordConnection được tạo
Constructor	- Hàm dựng nhận một đối tượng Connection làm tham số và thiết lập trường connection. Giúp khởi tạo một đối tượng RecordConnection với kết nối đã được thiết lập.
executeQuery(String query)	- Phương thức này thực hiện truy vấn đến cơ sở dữ liệu và trả về kết quả dưới dạng đối tượng java.sql.ResultSet. - Sử dụng Statement để tạo và thực thi truy vấn.

	- Nếu có lỗi trong quá trình thực thi truy vấn, nó sẽ được bắt và ném ra một ngoại lệ RuntimeException.
executeUpdate(String query)	- Phương thức này thực hiện truy vấn cập nhật (insert, update, delete) đến cơ sở dữ liệu. - Sử dụng Statement để tạo và thực thi truy vấn. - Nếu có lỗi trong quá trình thực thi truy vấn, nó sẽ được bắt và ném ra một ngoại lệ RuntimeException.
Lombok Annotations (@Getter và @Setter):	- Sử dụng để tự động tạo các getter và setter cho trường connection. - Giúp giảm bớt công đoạn viết mã boilerplate và làm cho mã nguồn ngắn gọn hơn.

- **Lớp RecordConnectionFactory:** có nhiệm vụ tạo ra đối tượng RecordConnection dựa trên thông tin cấu hình kết nối (ConnectConfiguration)

Các thành phần chính	Ý nghĩa
createRecordConnection (ConnectConfiguration configuration)	- Phương thức này là một phương thức tĩnh, trả về một đối tượng RecordConnection -Nhận một đối tượng ConnectConfiguration làm tham số, chứa thông tin cấu hình kết nối đến cơ sở dữ liệu. - Xây dựng chuỗi kết nối từ ConnectConfiguration bằng cách gọi phương thức toConnectString() của nó - Dựa vào kiểu cơ sở dữ liệu trong ConnectConfiguration, chọn driver JDBC tương ứng và đăng ký nó bằng cách gọi Class.forName(driver) - Sử dụng DriverManager để tạo và trả về một đối tượng Connection dựa trên chuỗi kết nối và thông tin đăng nhập - Tạo và trả về một đối tượng RecordConnection sử dụng đối tượng Connection đã tạo

- **Package Query:**

- **Lớp QueryType:** là một enum, định nghĩa các loại truy vấn mà framework hỗ trợ.

Các thành phần chính	Ý nghĩa
Enum Constants	- SELECT, INSERT, UPDATE, DELETE là bốn hằng số enum đại diện cho các loại truy vấn cơ bản - Được sử dụng để định danh và phân biệt giữa các loại truy vấn khi xây dựng câu lệnh SQL tương ứng

- **Lớp QueryBuilder:** là một lớp trừu tượng chung cho việc xây dựng các câu truy vấn SQL. Nó định nghĩa các phương thức và thuộc tính chung mà tất cả các lớp con cần thực hiện.

Các thành phần chính	Ý nghĩa
table	- Là một trường dùng để lưu trữ tên bảng mà câu truy vấn sẽ thực hiện lên - Được thiết lập bằng phương thức table(String table)
wheres	- Là một danh sách các điều kiện (WHERE) trong câu truy vấn - Các điều kiện này được thêm vào danh sách thông qua các phương thức where(), and(), và or()
where(String clause)	Phương thức để thêm điều kiện (WHERE) vào danh sách
and(String clause)	Phương thức để thêm điều kiện (AND) vào danh sách
or(String clause)	Phương thức để thêm điều kiện (OR) vào danh sách
table(String table)	Phương thức để thiết lập tên bảng mà câu truy vấn sẽ thực hiện lên
select(String... columns), insert(String table), update(String table), delete(String table)	- Phương thức tạo và trả về một đối tượng của các lớp con cụ thể (SelectQueryBuilder, InsertQueryBuilder, UpdateQueryBuilder, DeleteQueryBuilder) tương ứng với loại truy vấn - Các phương thức này giúp bắt đầu quá trình xây dựng câu truy vấn của loại cụ thể

newQuery(QueryType type)	- Phương thức tạo và trả về một đối tượng của các lớp con cụ thể dựa trên loại truy vấn được chỉ định (SELECT, INSERT, UPDATE, DELETE) - Sử dụng switch-case để quyết định loại truy vấn và trả về đối tượng tương ứng
build()	Phương thức trừu tượng chịu trách nhiệm xây dựng câu truy vấn cuối cùng và trả về chuỗi kết quả

- **Lớp SelectQueryBuilder:** là một lớp con của QueryBuilder và chịu trách nhiệm xây dựng các câu truy vấn SELECT cụ thể. Nó hỗ trợ việc chọn cột, thiết lập bảng, thêm các điều kiện WHERE, và xử lý các câu truy vấn JOIN. Đặc biệt, khi cần nhóm dữ liệu, nó sử dụng GroupByBuilder để xây dựng phần GROUP BY và HAVING của câu truy vấn SELECT.

Các thành phần chính	Ý nghĩa
columns	- Là một danh sách lưu trữ tên các cột mà câu truy vấn SELECT sẽ chọn - Được khởi tạo thông qua constructor
joins	- Là một danh sách lưu trữ các bảng được kết nối trong câu truy vấn SELECT. - Được sử dụng khi có các câu truy vấn JOIN.
groupBy	- Là một đối tượng thuộc kiểu GroupByBuilder chịu trách nhiệm xây dựng phần GROUP BY và HAVING của câu truy vấn SELECT - Được khởi tạo khi gọi phương thức groupBy
join(String table)	Phương thức để thêm bảng vào danh sách kết nối
from(String table)	Phương thức để thiết lập tên bảng mà câu truy vấn SELECT sẽ thực hiện lên
groupBy(String...columns)	Phương thức để bắt đầu xây dựng phần GROUP BY của câu truy vấn SELECT
GroupByBuilder	Là một lớp con tĩnh trong SelectQueryBuilder, chịu trách nhiệm xây

	dựng phần GROUP BY và HAVING của câu truy vấn SELECT
build()	Phương thức triển khai từ lớp cha QueryBuilder, chịu trách nhiệm xây dựng câu truy vấn SELECT cuối cùng

- **Lớp InsertQueryBuilder**: là một lớp con của QueryBuilder và chịu trách nhiệm xây dựng các câu truy vấn INSERT cụ thể. Nó sử dụng tên bảng, danh sách các cột, và giá trị tương ứng để xây dựng câu lệnh SQL INSERT INTO. Các giá trị chuỗi sẽ được đặt trong dấu nháy đơn để đảm bảo tính nhất quán của câu truy vấn. Nếu có điều kiện WHERE, câu truy vấn cũng sẽ bao gồm một phần WHERE với các điều kiện đã được thiết lập từ lớp cha QueryBuilder.

Các thành phần chính	Ý nghĩa
values	- Là một Map lưu trữ các cặp key-value, biểu diễn giá trị của các cột và dữ liệu sẽ được chèn vào cơ sở dữ liệu - Được khởi tạo trong constructor
InsertQueryBuilder()	Constructor không tham số, khởi tạo một đối tượng InsertQueryBuilder với một HashMap trống cho các giá trị
InsertQueryBuilder(String table)	Constructor với tham số là tên bảng, khởi tạo đối tượng InsertQueryBuilder và thiết lập tên bảng
value(Object key, Object value	Phương thức để thêm một cặp key-value vào Map giá trị
values(Map<Object, Object> values)	Phương thức để thiết lập giá trị từ một Map mới
build()	Phương thức triển khai từ lớp cha QueryBuilder, chịu trách nhiệm xây dựng câu truy vấn INSERT cuối cùng

- **Lớp UpdateQueryBuilder:** là một lớp con của QueryBuilder và chịu trách nhiệm xây dựng các câu truy vấn UPDATE cụ thể. Nó sử dụng tên bảng, danh sách các cột cần cập nhật và giá trị tương ứng để xây dựng câu lệnh SQL UPDATE SET. Các giá trị chuỗi sẽ được đặt trong dấu nháy đơn để đảm bảo tính nhất quán của câu truy vấn. Nếu

có điều kiện WHERE, câu truy vấn cũng sẽ bao gồm một phần WHERE với các điều kiện đã được thiết lập từ lớp cha QueryBuilder.

Các thành phần chính	Ý nghĩa
table	- Là một trường lưu trữ tên bảng mà câu truy vấn UPDATE sẽ thực hiện lên - Được thiết lập thông qua constructor hoặc thông qua setter table()
setters	- Là một Map lưu trữ các cặp key-value, biểu diễn giá trị của các cột và dữ liệu sẽ được cập nhật trong câu truy vấn UPDATE. - Được khởi tạo trong constructor.
UpdateQueryBuilder()	Constructor không tham số, khởi tạo một đối tượng UpdateQueryBuilder với một HashMap trống cho các giá trị
UpdateQueryBuilder(String table)	Constructor với tham số là tên bảng, khởi tạo đối tượng UpdateQueryBuilder và thiết lập tên bảng
setter(Object key, Object value)	Phương thức để thêm một cặp key-value vào Map giá trị cần cập nhật
setters(Map<Object, Object> setters)	Phương thức để thiết lập giá trị cần cập nhật từ một Map mới
build()	Phương thức triển khai từ lớp cha QueryBuilder, chịu trách nhiệm xây dựng câu truy vấn UPDATE cuối cùng

- **Lớp DeleteQueryBuilder:** là một lớp con của QueryBuilder và chịu trách nhiệm xây dựng các câu truy vấn DELETE cụ thể. Nó sử dụng tên bảng và danh sách các điều kiện WHERE để xây dựng câu lệnh SQL DELETE FROM. Nếu có điều kiện WHERE, câu truy vấn sẽ bao gồm một phần WHERE với các điều kiện đã được thiết lập từ lớp cha QueryBuilder.

Các thành phần chính	Ý nghĩa
table	- Là một trường lưu trữ tên bảng mà câu truy vấn DELETE sẽ thực hiện lên. - Được thiết lập thông qua constructor hoặc thông qua setter table()
DeleteQueryBuilder()	Constructor không tham số, khởi tạo một đối tượng DeleteQueryBuilder
DeleteQueryBuilder(String table)	Constructor với tham số là tên bảng, khởi tạo đối tượng DeleteQueryBuilder và thiết lập tên bảng
build()	Phương thức triển khai từ lớp cha QueryBuilder, chịu trách nhiệm xây dựng câu truy vấn DELETE cuối cùng

- **Package Management:**
  - **Lớp Persistence:** là một lớp tiện ích được thiết kế để quản lý kết nối đến cơ sở dữ liệu và tạo các đối tượng RecordManagerFactory để quản lý các đối tượng RecordManager.

Các thành phần chính	Ý nghĩa
MAX_EXISTS_CONNECTIONS	Hằng số xác định số lượng kết nối tối đa trong connection pool
connectionPool	List chứa các đối tượng RecordConnection, là các kết nối đến cơ sở dữ liệu
used	Map lưu trữ thông tin về các RecordManagerFactory đã được sử dụng và kết nối cụ thể của chúng
configureDatasource (ConnectConfiguration configuration)	- Phương thức cấu hình nguồn dữ liệu cơ sở dữ liệu dựa trên cấu hình kết nối (ConnectConfiguration).

	- Gọi createConnection() để tạo các kết nối và thêm chúng vào connection pool
createConnection (ConnectConfiguration configuration)	Phương thức tạo kết nối đến cơ sở dữ liệu và thêm chúng vào connection pool
createRecordManagerFactory (String factoryName)	- Phương thức tạo và quản lý một RecordManagerFactory mới - Kiểm tra xem có sẵn RecordManagerFactory với factoryName đã được sử dụng trước đó không. - Nếu có, trả về RecordManagerFactory đã được sử dụng trước đó. - Nếu không, tìm một kết nối trống trong connection pool và tạo mới RecordManagerFactory sử dụng kết nối đó. - Nếu không có kết nối trống, ném ngoại lệ OutOfConnectionException
release(String factoryName)	Phương thức giải phóng RecordManagerFactory với factoryName khỏi danh sách used
Private Constructor	Là một constructor private để đảm bảo rằng lớp này không thể được khởi tạo bởi bên ngoài

- **Lớp RecordManagerFactory:** là một lớp đơn giản được thiết kế để tạo và quản lý các đối tượng RecordManager.

Các thành phần chính	Ý nghĩa
name	- Trường dữ liệu (String) lưu trữ tên của RecordManagerFactory. - Được đặt giá trị thông qua constructor và chỉ đọc (read-only)
recordConnection	- Trường dữ liệu (RecordConnection) lưu trữ kết nối đến cơ sở dữ liệu của RecordManagerFactory

	- Được đặt giá trị thông qua constructor và chỉ đọc (read-only
Constructor (public RecordManagerFactory(String name, RecordConnection recordConnection)	- Constructor để khởi tạo một đối tượng RecordManagerFactory với tên và kết nối đến cơ sở dữ liệu được cung cấp - Đảm bảo rằng thông tin cần thiết về tên và kết nối được thiết lập khi tạo đối tượng
createRecordManager()	- Phương thức tạo và trả về một đối tượng RecordManager mới, sử dụng kết nối đến cơ sở dữ liệu của RecordManagerFactory - RecordManager là một lớp chịu trách nhiệm thực hiện các thao tác với cơ sở dữ liệu
Getter cho name và recordConnection	Cung cấp phương thức getter để đọc giá trị của các trường dữ liệu name và recordConnection

- **Lớp RecordManager:** đóng vai trò quan trọng trong quản lý và thực hiện các thao tác tương tác với cơ sở dữ liệu.

Các thành phần chính	Ý nghĩa
recordConnection	- Trường dữ liệu (RecordConnection) lưu trữ kết nối đến cơ sở dữ liệu, được sử dụng để thực hiện các thao tác truy vấn và cập nhật - Được đặt giá trị thông qua constructor và có getter và setter để truy cập
records	- Trường dữ liệu (List<Object>) lưu trữ danh sách các đối tượng được quản lý bởi RecordManager - Có thể được sử dụng để theo dõi các bản ghi đã được truy vấn hoặc thực hiện thao tác
Constructor (public RecordManager(RecordConnection connection)	Constructor để khởi tạo một đối tượng RecordManager với một kết nối đến cơ sở dữ liệu đã cho
createQuery(QueryType type)	- Phương thức tạo và trả về một QueryBuilder mới dựa trên loại thao tác (QueryType) được cung cấp - Được sử dụng khi muốn tạo một câu truy vấn cụ thể (INSERT, UPDATE, DELETE) hoặc một câu truy vấn chung
createQuery(QueryType type, Object record)	- Phương thức tạo và trả về một QueryBuilder dựa trên loại thao tác và đối tượng ghi (record) được cung cấp - Sử dụng RecordMapper để chuyển đổi đối tượng ghi thành câu truy vấn tương ứng
executeQuery(QueryBuilder query, Class<T> clazz)	- Thực hiện một câu truy vấn SELECT và trả về danh sách các đối tượng của lớp clazz

	- Sử dụng RecordMapper để ánh xạ kết quả từ ResultSet thành các đối tượng
executeUpdate(QueryBuilder query)	Thực hiện câu truy vấn UPDATE, INSERT hoặc DELETE
insert(Object record), update(Object record), delete(Object record)	Gọi các phương thức tương ứng để thực hiện các thao tác INSERT, UPDATE, DELETE dựa trên đối tượng ghi cung cấp.

- **Lớp RecordMapper:** chịu trách nhiệm chuyển đổi giữa các đối tượng Java và các câu truy vấn SQL, đồng thời thực hiện ánh xạ dữ liệu từ ResultSet thành danh sách đối tượng Java**.** Giúp cung cấp một cách thuận tiện để ánh xạ giữa đối tượng Java và cơ sở dữ liệu SQL mà không cần phải viết mã SQL cụ thể.

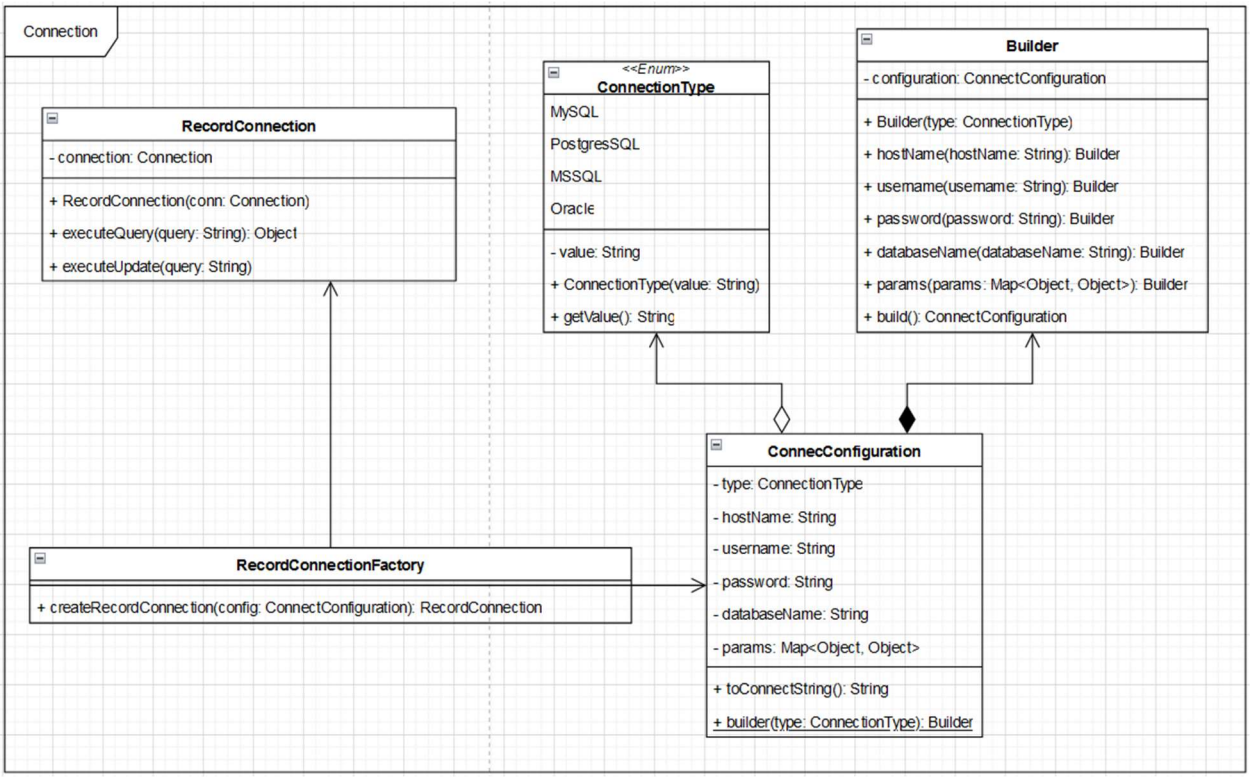
Các thành phần chính	Ý nghĩa
map(ResultSet rs, Class<T> clazz)	- Phương thức này chuyển đổi ResultSet thành danh sách đối tượng của kiểu clazz. - Duyệt qua các bản ghi của ResultSet và ánh xạ từng cột vào thuộc tính tương ứng của đối tượng. - Sử dụng các annotation (@Record, @Column, @Id) để xác định thông tin ánh xạ.
toInsertQuery(T object)	- Phương thức này tạo và trả về một QueryBuilder để thực hiện thao tác INSERT dựa trên đối tượng Java object. - Sử dụng reflection để đọc thông tin về cột (@Column) và ID (@Id). - Tạo câu truy vấn INSERT dựa trên thông tin đọc được
toUpdateQuery(T object)	- Phương thức này tạo và trả về một QueryBuilder để thực hiện thao tác UPDATE dựa trên đối tượng Java object.

	- Sử dụng reflection để đọc thông tin về cột và ID. - Tạo câu truy vấn UPDATE dựa trên thông tin đọc được và điều kiện WHERE.
toDeleteQuery(T object)	- Phương thức này tạo và trả về một QueryBuilder để thực hiện thao tác DELETE dựa trên đối tượng Java object. - Sử dụng reflection để đọc thông tin về cột ID. - Tạo câu truy vấn DELETE dựa trên thông tin đọc được và điều kiện WHERE.

III. Các mẫu thiết kế được sử dụng trong framework.

a. Package Connection

Ở Package Connection, nhóm sử dụng 2 mẫu thiết kế: ***Builder*** và ***Factory***.



Hình 2 Sơ đồ lớp của Package Connection sử dụng Builder và Factory

- **Builder:**
  - **Ý nghĩa:**
    - Sử dụng Builder để tạo kết nối URL.
    - Để tạo ra một đối tượng phức tạp step-by-step.
    - Cho phép xây dựng đối tượng với các thuộc tính tùy chọn mà không làm cho constructor trở nên quá phức tạp hoặc chứa quá nhiều tham số.

- **Đoạn code sử dụng mẫu:** Được ứng dụng trong lớp ConnectConfiguration.

```
public class ConnectConfiguration {
    private ConnectionType type;
    private String hostName;
    private String username;
    private String password;
    private String databaseName;
    private Map<Object, Object> params;
    public String toConnectString() {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append("jdbc:")
                .append(type.getValue())
                .append("://")
                .append(hostName)
                .append("/")
                .append(databaseName);
        if (!params.isEmpty()) {
            stringBuilder.append("?");
            List<Map.Entry<Object, Object>> entries = new
ArrayList<>(params.entrySet());
            for (int i = 0; i < entries.size(); ++i) {
                Map.Entry<Object, Object> entry =
entries.get(i);
                stringBuilder.append(entry.getKey());
                stringBuilder.append("=");
                stringBuilder.append(entry.getValue());
                if (i < entries.size() - 1) {
                    stringBuilder.append("&");
                }
            }
        }
        return stringBuilder.toString();
    }
    public static Builder builder(ConnectionType type) {
        return new Builder(type);
    }
    public static class Builder {
```

```
        private final ConnectConfiguration configuration = new
ConnectConfiguration();
        public Builder(ConnectionType type) {
            configuration.setType(type);
        }
        public Builder hostName(String hostName) {
            configuration.setHostName(hostName);
            return this;
        }
        public Builder username(String username) {
            configuration.setUsername(username);
            return this;
        }
        public Builder password(String password) {
            configuration.setPassword(password);
            return this;
        }
        public Builder databaseName(String databaseName) {
            configuration.setDatabaseName(databaseName);
            return this;
        }
        public Builder params(Map<Object, Object> params) {
            configuration.setParams(params);
            return this;
        }
        public ConnectConfiguration build() {
            return configuration;
        }
    }
}
```

- **Factory**
  - **Ý nghĩa:**
    - Mẫu thiết kế Factory được sử dụng để tạo ra các đối tượng mà không cần biết chi tiết cụ thể về cách chúng được tạo ra.
    - Tạo ra một giao diện chung để tạo đối tượng và để các lớp con quyết định loại đối tượng nào được tạo. Ở package này, nhằm

mục đích tạo kết nối đa dạng với các loại cơ sở dữ liệu khác nhau.

- **Đoạn code sử dụng mẫu:** Được ứng dụng trong lớp RecordConnectionFactory.
  - Lớp RecordConnectionFactory đóng vai trò như một Factory để tạo ra đối tượng RecordConnection.
  - Tùy thuộc vào loại cơ sở dữ liệu được chỉ định trong ConnectConfiguration, lớp này sử dụng một Factory Method để tạo ra đối tượng RecordConnection phù hợp.
  - Cung cấp một cách tiếp cận linh hoạt khi muốn thêm hỗ trợ cho các loại cơ sở dữ liệu mới.

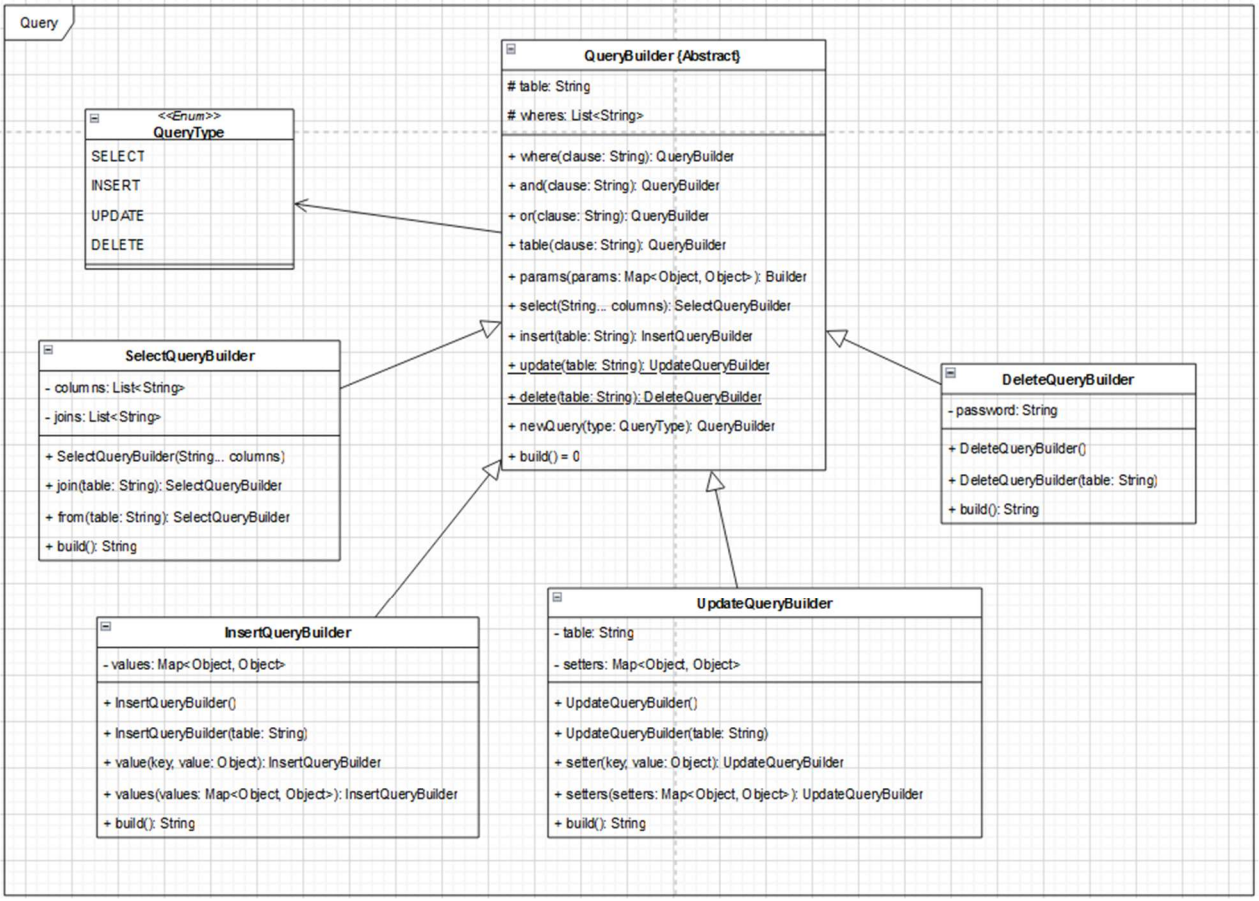
```
public class RecordConnectionFactory {

    public static RecordConnection
createRecordConnection(ConnectConfiguration configuration)
            throws ConnectionException {
        String connectionString =
configuration.toConnectString();
        String driver;
        try {
            switch (configuration.getType()) {
                case MySQL:
                    driver= "com.mysql.cj.jdbc.Driver";
                    break;
                case PostgresSQL:
                    driver = "org.postgresql.Driver";
                    break;
                case MSSQL:
                    driver =
"com.microsoft.sqlserver.jdbc.SQLServerDriver";
                    break;
                case Oracle:
                    driver = "oracle.jdbc.driver.OracleDriver";
                    break;
                default:
```

```
                    throw new
UnsupportedDatabaseException("Currently no support for this
database type: " + configuration.getType());
            }
            Class.forName(driver);
            Connection connection =
DriverManager.getConnection(connectionString,
                    configuration.getUsername(),
configuration.getPassword());
            return new RecordConnection(connection);
        } catch (Exception ex) {
            System.err.println(ex);
        }
        throw new ConnectionException("Can't connect to
database");
    }
}
```

b. Package Query

Ở Package Query, nhóm đã sử dụng 2 mẫu thiết kế ***Builder*** và **Factory.**



Hình 3 Sơ đồ lớp của Package Query sử dụng Builder và Factory

- **Builder:**
  - **Ý nghĩa:**
    - Tương tự như package Connection, Builder Design Pattern được sử dụng để tạo ra một đối tượng phức tạp step-by-step mà không làm cho constructor trở nên quá phức tạp.
    - Cho phép xây dựng các câu truy vấn SQL một cách linh hoạt và dễ dàng mở rộng
  - **Đoạn code sử dụng mẫu:** Được ứng dụng các lớp QueryBuilder, SelectQueryBuilder, InsertQueryBuilder, UpdateQueryBuilder, DeleteQueryBuilder.

- Phương thức where, and, or, table,... của lớp QueryBuilder cho phép xây dựng các câu truy vấn SQL theo các điều kiện và thông tin khác nhau.
- Các lớp con như SelectQueryBuilder, InsertQueryBuilder,... chủ động triển khai cách chúng xây dựng các phần cụ thể của câu truy vấn.

```
// QueryBuilder
public abstract class QueryBuilder {
    protected String table;
    protected List<String> wheres = new ArrayList<>();
    public QueryBuilder where(String clause) {
        wheres.add(clause);
        return this;
    }
    public QueryBuilder and(String clause) {
        wheres.add(" AND " + clause);
        return this;
    }
    public QueryBuilder or(String clause) {
        wheres.add(" OR " + clause);
        return this;
    }
    public QueryBuilder table(String table) {
        this.table = table;
        return this;
    }
    public static SelectQueryBuilder select(String...columns) {
        return new SelectQueryBuilder(columns);
    }
    public static InsertQueryBuilder insert(String table) {
        return new InsertQueryBuilder(table);
    }
    public static UpdateQueryBuilder update(String table) {
        return new UpdateQueryBuilder(table);
    }
    public static DeleteQueryBuilder delete(String table) {
        return new DeleteQueryBuilder(table);
    }
```

```
    public static QueryBuilder newQuery(QueryType type) {
        switch (type) {
            case SELECT:
                return select();
            case INSERT:
                return new InsertQueryBuilder();
            case UPDATE:
                return new UpdateQueryBuilder();
            case DELETE:
                return new DeleteQueryBuilder();
        }
        throw new RuntimeException("Please use valid query
type.");
    }
    public abstract String build();
}
```

```
// SelectQueryBuilder
public class SelectQueryBuilder extends QueryBuilder {
    private List<String> columns;
    private List<String> joins;
    private GroupByBuilder groupBy;
    public SelectQueryBuilder(String... columns) {
        this.columns = Arrays.asList(columns);
        this.joins = new ArrayList<>();
    }
    public SelectQueryBuilder join(String table) {
        if (joins == null) {
            joins = new ArrayList<>();
        }
        joins.add(table);
        return this;
    }
    public SelectQueryBuilder from(String table) {
        this.table = table;
        return this;
    }
    public GroupByBuilder groupBy(String...columns) {
```

```
        this.groupBy = new GroupByBuilder(this, columns);
        return this.groupBy;
    }
    public static class GroupByBuilder extends QueryBuilder {
        private final SelectQueryBuilder selectQueryBuilder;
        private List<String> columns;
        private List<String> having;
        public GroupByBuilder(SelectQueryBuilder
selectQueryBuilder) {
            this.selectQueryBuilder = selectQueryBuilder;
            this.having = new ArrayList<>();
        }
        public GroupByBuilder(SelectQueryBuilder
selectQueryBuilder, String...columns) {
            this(selectQueryBuilder);
            this.columns = Arrays.asList(columns);
        }
        public QueryBuilder having(String clause) {
            having.add(clause);
            return this;
        }
        public QueryBuilder and(String clause) {
            having.add(" AND " + clause);
            return this;
        }
        public QueryBuilder or(String clause) {
            having.add(" OR " + clause);
            return this;
        }
        @Override
        public String build() {
            if (columns.size() == 0) {
                return this.selectQueryBuilder.build();
            }
            StringBuilder builder = new StringBuilder();
            builder.append(this.selectQueryBuilder.build())
                    .append(" GROUP BY ");
            for (int i = 0; i < columns.size(); ++i) {
                String column = columns.get(i);
```

```
                builder.append(column);
                if (i < columns.size() - 1) {
                    builder.append(",");
                };
            }
            if (!this.having.isEmpty()) {
                builder.append(" HAVING ");
                for (String having : this.having) {
                    builder.append(having);
                }
            };
            return builder.toString();
        }
    }
    @Override
    public String build() {
        StringBuilder builder = new StringBuilder();
        builder.append("SELECT");
        if (!columns.isEmpty()) {
            builder.append(columns.get(0));
            for (String column : columns) {
                builder.append(",").append(column);
            }
        } else {
            builder.append(" * ");
        }
        builder.append(" FROM ");
        builder.append(this.table);
        if (!joins.isEmpty()) {
            for (int i = 1; i < joins.size(); ++i) {
                builder.append(" JOIN ").append(joins.get(i));
            }
        }
        if (!this.wheres.isEmpty()) {
            builder.append(" WHERE ");
            for (String where : this.wheres) {
                builder.append(where);
            }
        }
```

```
        return builder.toString();
    }
}
```

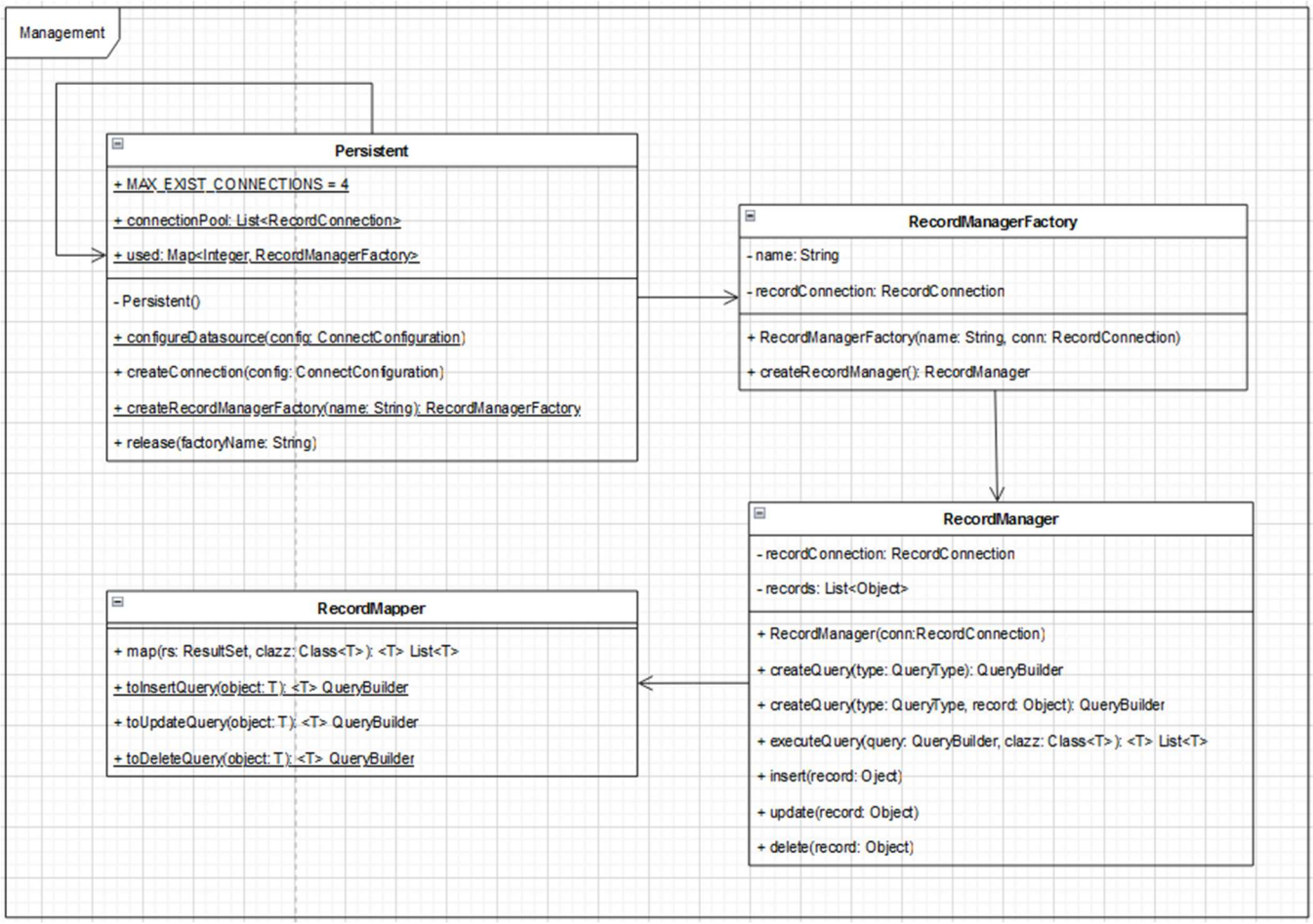
- Các lớp khác tương tự.

- **Factory**
  - **Ý nghĩa:**Tùy thuộc vào loại câu truy vấn được chỉ định, nó trả về một đối tượng con cụ thể để xử lý câu truy vấn.
  - **Đoạn code sử dụng mẫu:** Được ứng dụng trong phương thức newQuery của lớp QueryBuilder.

```
public static QueryBuilder newQuery(QueryType type) {
        switch (type) {
            case SELECT:
                return select();
            case INSERT:
                return new InsertQueryBuilder();
            case UPDATE:
                return new UpdateQueryBuilder();
            case DELETE:
                return new DeleteQueryBuilder();
        }
        throw new RuntimeException("Please use valid query
type.");
    }
```

c. Package Management

Ở Package Management, nhóm sử dụng 2 mẫu thiết kế ***Singleton*** và ***Factory***



Hình 4 Sơ đồ lớp Package Management sử dụng Singleton và Factory

- **Singleton**
  - **Ý nghĩa:**
    - Được sử dụng để đảm bảo rằng một lớp chỉ có duy nhất một thể hiện và cung cấp một cách để truy cập nó từ bất kỳ điểm nào trong hệ thống.
    - Cụ thể là đảm bảo rằng chỉ có một thể hiện của Persistence tồn tại, giúp tránh việc tạo ra nhiều kết nối cơ sở dữ liệu không cần thiết và duy trì trạng thái của connectionPool và used một cách hiệu quả.

- **Đoạn code sử dụng mẫu:** Ứng dụng trong lớp Persistence.
  - Lớp Persistence có một số trạng thái và phương thức cần được chia sẻ và duy trì xuyên suốt toàn bộ ứng dụng.

```
public class Persistence {
    public final static Integer MAX_EXISTS_CONNECTIONS = 4;
    public static List<RecordConnection> connectionPool = new
ArrayList<>();
    public final static Map<Integer, RecordManagerFactory> used
= new HashMap<>();
    private Persistence() {
    }
    public static void configureDatasource(ConnectConfiguration
configuration)
            throws ConnectionException {
        createConnection(configuration);
    }
    public static void createConnection(ConnectConfiguration
configuration)
            throws ConnectionException {
        for (int i = 0; i < MAX_EXISTS_CONNECTIONS; ++i) {
connectionPool.add(RecordConnectionFactory.createRecordConnectio
n(configuration));
        }
    }

    public static RecordManagerFactory
createRecordManagerFactory(String factoryName)
            throws OutOfConnectionException {
        int connectionId = used.entrySet().stream()
                .filter(entry ->
factoryName.equals(entry.getValue().getName()))
                .mapToInt(Map.Entry::getKey)
                .findFirst()
                .orElse(-1);
        if (connectionId != -1) {
            return used.get(connectionId);
        }
```

```
        int freeConnectionId = IntStream.range(0,
MAX_EXISTS_CONNECTIONS)
                .filter(id -> !used.containsKey(id))
                .findFirst()
                .orElse(-1);
        if (freeConnectionId == -1) {
            throw new OutOfConnectionException("Out of database
connections.");
        }
        RecordManagerFactory factory = new
RecordManagerFactory(factoryName,
                connectionPool.get(freeConnectionId));
        used.put(freeConnectionId, factory);
        return factory;
    }

    public static void release(String factoryName) {
        int connectionId = used.entrySet().stream()
                .filter(entry ->
factoryName.equals(entry.getValue().getName()))
                .mapToInt(Map.Entry::getKey)
                .findFirst()
                .orElse(-1);
        if (connectionId == -1) {
            return;
        }
        used.remove(connectionId);
    }
}
```

- **Factory:**
  - **Ý nghĩa:**
    - Phương thức createRecordManagerFactory sử dụng Factory Design Pattern để tạo ra các đối tượng RecordManagerFactory
    - Tùy thuộc vào factoryName được chỉ định, nó trả về một đối tượng RecordManagerFactory cụ thể để quản lý việc tạo và sử dụng các RecordManager.

- **Đoạn code sử dụng mẫu:**

```
public static RecordManagerFactory
createRecordManagerFactory(String factoryName)
            throws OutOfConnectionException {
        int connectionId = used.entrySet().stream()
                .filter(entry ->
factoryName.equals(entry.getValue().getName()))
                .mapToInt(Map.Entry::getKey)
                .findFirst()
                .orElse(-1);
        if (connectionId != -1) {
            return used.get(connectionId);
        }
        int freeConnectionId = IntStream.range(0,
MAX_EXISTS_CONNECTIONS)
                .filter(id -> !used.containsKey(id))
                .findFirst()
                .orElse(-1);
        if (freeConnectionId == -1) {
            throw new OutOfConnectionException("Out of database
connections.");
        }
        RecordManagerFactory factory = new
RecordManagerFactory(factoryName,
                connectionPool.get(freeConnectionId));
        used.put(freeConnectionId, factory);
        return factory;
    }
```

- Ngoài ra, nhóm cũng sử dụng mẫu Object pool để giới hạn số lượng Connection được tạo thông qua RecordConnection được tạo.

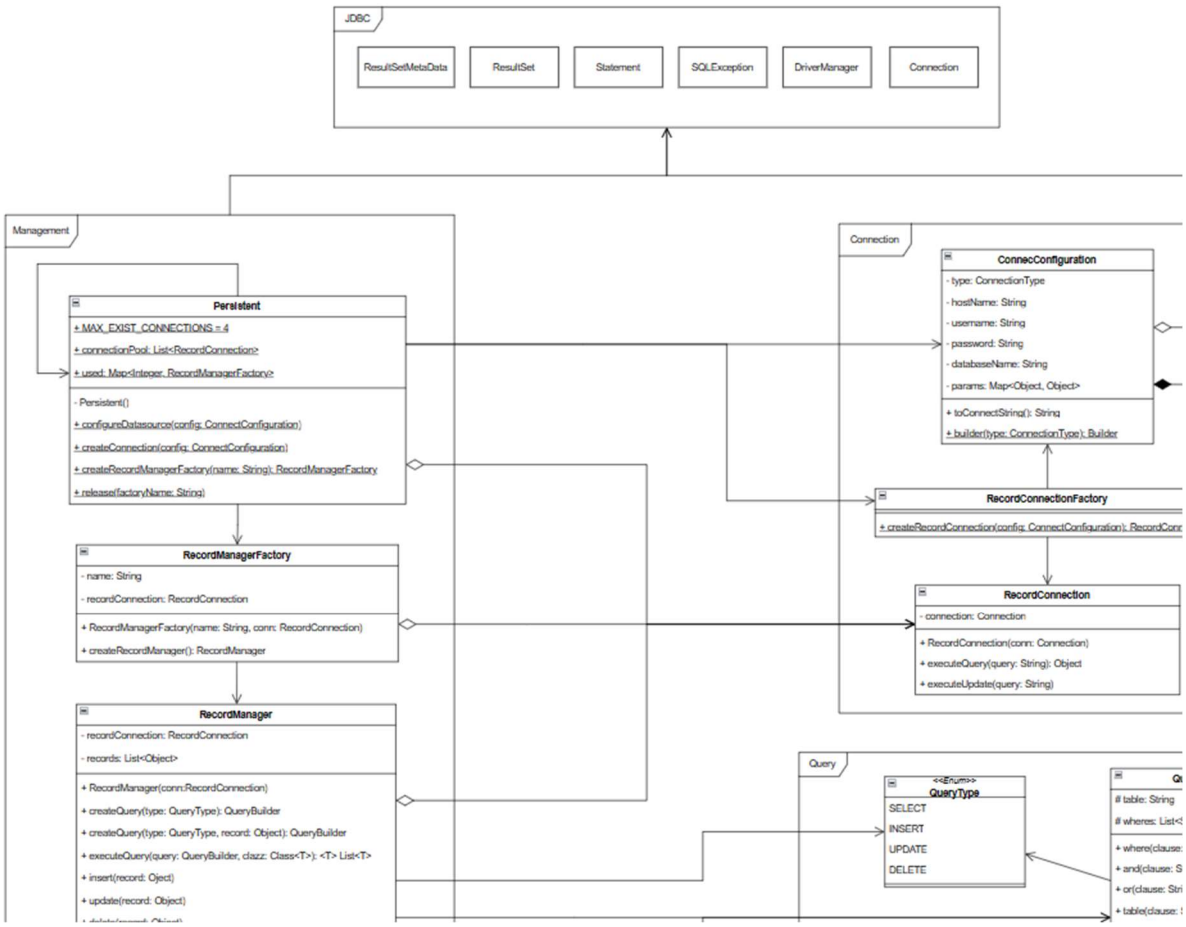
d. Mẫu Adapter trong framework

Ở framework này, nhóm tổ chức 2 lớp Adapter nối tiếp nhau là RecordManager và RecordConnection.

- **Ý nghĩa:**

- **RecordConnection là Adapter cho RecordManager (tương tác JDBC)**: RecordConnection có nhiệm vụ tương tác JDBC thông qua các phương thức executeQuery và executeUpdate. RecordManager sử dụng RecordConnection để thực hiện các thao tác cơ sở dữ liệu mà nó định nghĩa thông qua các phương thức như insert, update, delete.

- **RecordManager là Adapter cho Client (sử dụng RecordConnection và QueryBuilder):**
  - RecordManager chịu trách nhiệm chuyển đổi giao diện cụ thể của RecordConnection thành một giao diện mà Client dễ sử dụng hơn.
  - Các phương thức như insert, update, delete của RecordManager là các phương thức tiện ích cho Client, giúp Client tương tác với cơ sở dữ liệu một cách thuận tiện và trừu tượng hóa việc sử dụng RecordConnection và QueryBuilder.
- **Dễ mở rộng và bảo trì:** Nếu ta muốn thay đổi hoặc mở rộng cách RecordManager tương tác với cơ sở dữ liệu, bạn có thể thực hiện các thay đổi trong lớp RecordConnection mà không ảnh hưởng đến các thành phần khác trong hệ thống
- **Phân chia trách nhiệm:** RecordConnection đảm nhận trách nhiệm về việc tương tác cụ thể với cơ sở dữ liệu, trong khi RecordManager đảm nhận trách nhiệm quản lý và thực hiện các thao tác cơ sở dữ liệu trong một giao diện phù hợp với nhu cầu của ứng dụng.



Hình 5 Sơ đồ lớp sử dụng Adapter

- **Đoạn code sử dụng mẫu**

```
// RecordManagerFactory class
public class RecordManagerFactory {

    private final String name;
    private final RecordConnection recordConnection;

    public RecordManagerFactory(String name, RecordConnection
recordConnection) {
        this.name = name;
        this.recordConnection = recordConnection;
    }

    public RecordManager createRecordManager() {
        return new RecordManager(recordConnection);
```

```
    }
}

// RecordManager class
public class RecordManager {

    private RecordConnection recordConnection;
    private final List<Object> records = new ArrayList<>();

    public RecordManager(RecordConnection connection) {
        this.recordConnection = connection;
    }

    // ... (các phương thức khác)

    public <T> List<T> executeQuery(QueryBuilder query, Class<T>
clazz)
            throws SQLException, InstantiationException,
IllegalAccessException, UnsupportedActionException,
InvocationTargetException, NoSuchMethodException {
        ResultSet resultSet = (ResultSet)
recordConnection.executeQuery(query.build());
        return RecordMapper.map(resultSet, clazz);
    }

    public void executeUpdate(QueryBuilder query) {
        recordConnection.executeUpdate(query.build());
    }

    // ... (các phương thức khác)
}

// RecordConnection class
public class RecordConnection {

    private final Connection connection;

    public RecordConnection(Connection connection) {
        this.connection = connection;
```

```
    }

    public Object executeQuery(String query) {
        try {
            Statement statement = connection.createStatement();
            return statement.executeQuery(query);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    public void executeUpdate(String query) {
        try {
            Statement statement = connection.createStatement();
            statement.executeUpdate(query);
        } catch (SQLException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```