

Họ và tên: Nguyễn Khánh Quy

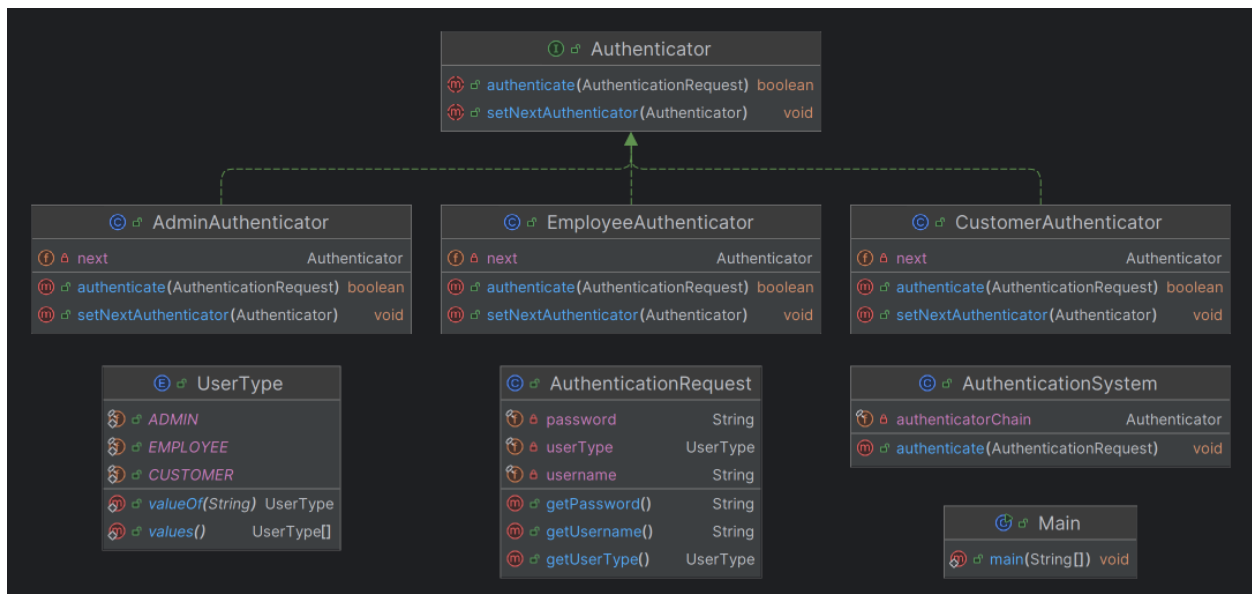
MSSV: 21110282

## Chain of Responsibility Pattern

Mẫu **Chain of Responsibility** trong lập trình hướng đối tượng là một mẫu thiết kế hành vi được sử dụng để xử lý một yêu cầu qua một chuỗi các đối tượng xử lý.

Mỗi đối tượng trong chuỗi có thể xác định xem liệu nó có thể xử lý yêu cầu hoặc chuyển tiếp yêu cầu đến đối tượng tiếp theo trong chuỗi. **Chain of Responsibility** giúp giảm sự phụ thuộc trực tiếp giữa các đối tượng xử lý yêu cầu và cho phép chúng được sắp xếp linh hoạt trong một chuỗi.

Dưới đây là một ví dụ về mẫu **Chain of Responsibility** bằng ngôn ngữ lập trình Java. Trong ví dụ này, chúng ta thiết kế một hệ thống xác thực nơi mỗi lớp xác thực có thể xác định xem liệu nó có thể xử lý yêu cầu xác thực từ một người dùng cụ thể hoặc chuyển tiếp yêu cầu đến lớp xác thực tiếp theo trong chuỗi. Mỗi lớp xác thực đóng vai trò là một mắt xích trong chuỗi xử lý yêu cầu xác thực.



## Bước 1: Tạo Enum định nghĩa loại người dùng (UserType)

Đây là một enum đơn giản để xác định loại người dùng, bao gồm **ADMIN**, **EMPLOYEE**, và **CUSTOMER**.

```
public enum UserType {  
  
    ADMIN,  
    EMPLOYEE,  
    CUSTOMER  
  
}
```

## Bước 2: Tạo lớp (AuthenticationRequest):

Đây là lớp đại diện cho yêu cầu xác thực, bao gồm các thông tin như tên người dùng, mật khẩu và loại người dùng.

```
public class AuthenticationRequest {  
  
    private final String username;  
    private final String password;  
    private final UserType userType;  
  
    public AuthenticationRequest(String username, String password, UserType  
userType) {  
        this.username = username;  
        this.password = password;  
        this.userType = userType;  
    }  
  
    public String getUsername() {  
        return username;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public UserType getUserType() {  
        return userType;  
    }  
  
}
```

### Bước 3: Tạo Interface (Authenticator):

Interface này định nghĩa hai phương thức cần thiết cho một đối tượng xác thực: **authenticate()** để xử lý yêu cầu xác thực và **setNextAuthenticator()** để thiết lập liên kết với đối tượng xác thực tiếp theo trong chuỗi.

```
public interface Authenticator {  
  
    boolean authenticate(AuthenticationRequest request);  
    void setNextAuthenticator(Authenticator next);  
  
}
```

### Bước 4: Tạo các lớp xác thực cụ thể (AdminAuthenticator, EmployeeAuthenticator, CustomerAuthenticator):

Đây là các lớp cụ thể triển khai interface Authenticator. Mỗi lớp xác thực xác định cách xử lý yêu cầu xác thực cho một loại người dùng cụ thể.

```
public class AdminAuthenticator implements Authenticator {  
  
    private Authenticator next;  
  
    @Override  
    public boolean authenticate(AuthenticationRequest request) {  
        if (request.getUserType() == UserType.ADMIN) {  
            // Xử lý xác thực cho quản trị viên  
            System.out.println("Authenticating admin: " +  
request.getUsername());  
            return true;  
        }  
        // Nếu không phải là quản trị viên, chuyển yêu cầu cho lớp xác thực  
        tiếp theo trong chuỗi  
        if (next != null) {  
            return next.authenticate(request);  
        }  
        return false;  
    }  
  
    @Override  
    public void setNextAuthenticator(Authenticator next) {  
        this.next = next;  
    }  
  
}
```

```

public class EmployeeAuthenticator implements Authenticator {

    private Authenticator next;

    @Override
    public boolean authenticate(AuthenticationRequest request) {
        if (request.getUserType() == UserType.EMPLOYEE) {
            // Xử lý xác thực cho nhân viên
            System.out.println("Authenticating employee: " +
request.getUsername());
            return true;
        }
        // Nếu không phải là nhân viên, chuyển yêu cầu cho lớp xác thực tiếp
theo trong chuỗi
        if (next != null) {
            return next.authenticate(request);
        }
        return false;
    }

    @Override
    public void setNextAuthenticator(Authenticator next) {
        this.next = next;
    }

}

```

```

public class CustomerAuthenticator implements Authenticator {

    private Authenticator next;

    @Override
    public boolean authenticate(AuthenticationRequest request) {
        if (request.getUserType() == UserType.CUSTOMER) {
            // Xử lý xác thực cho khách hàng
            System.out.println("Authenticating customer: " +
request.getUsername());
            return true;
        }
        // Nếu không phải là khách hàng, chuyển yêu cầu cho lớp xác thực tiếp
theo trong chuỗi
        if (next != null) {
            return next.authenticate(request);
        }
        return false;
    }

    @Override
    public void setNextAuthenticator(Authenticator next) {
        this.next = next;
    }

}

```

## Bước 5: Tạo lớp (AuthenticationSystem):

Lớp này là hệ thống xác thực chính. Trong **constructor** của nó, các đối tượng xác thực được tạo ra và kết nối thành một chuỗi. Ở đây, chuỗi xác thực bao gồm **AdminAuthenticator**, **CustomerAuthenticator** và **EmployeeAuthenticator**.

```
public class AuthenticationSystem {  
  
    private final Authenticator authenticatorChain;  
  
    public AuthenticationSystem() {  
        Authenticator customerAuthenticator = new CustomerAuthenticator();  
        Authenticator employeeAuthenticator = new EmployeeAuthenticator();  
        Authenticator adminAuthenticator = new AdminAuthenticator();  
  
        // Xác định chuỗi xác thực  
        customerAuthenticator.setNextAuthenticator(employeeAuthenticator);  
        employeeAuthenticator.setNextAuthenticator(adminAuthenticator);  
  
        this.authenticatorChain = customerAuthenticator;  
    }  
  
    public void authenticate(AuthenticationRequest request) {  
        authenticatorChain.authenticate(request);  
    }  
  
}
```

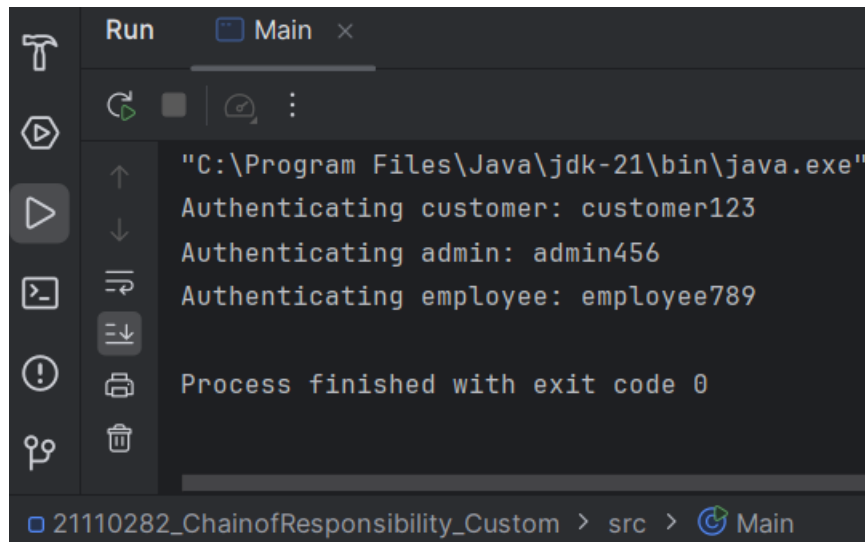
## Bước 6: Thử nghiệm:

Trong phương thức main của lớp **Main**, một hệ thống xác thực được tạo ra và một số yêu cầu xác thực được tạo ra và gửi qua hệ thống. Hệ thống xác thực sẽ xử lý mỗi yêu cầu bằng cách chuyển nó qua chuỗi các đối tượng xác thực cho đến khi nó được xử lý hoặc đến khi không còn đối tượng xác thực nào còn lại trong chuỗi.

```
public class Main {  
  
    public static void main(String[] args) {  
        AuthenticationSystem authSystem = new AuthenticationSystem();  
  
        // Yêu cầu xác thực  
        AuthenticationRequest request1 = new  
AuthenticationRequest("customer123", "customerPassword", UserType.CUSTOMER);  
        AuthenticationRequest request2 = new  
AuthenticationRequest("admin456", "adminPassword", UserType.ADMIN);  
        AuthenticationRequest request3 = new  
AuthenticationRequest("employee789", "employeePassword", UserType.EMPLOYEE);  
    }  
}
```

```
// Xác thực
authSystem.authenticate(request1);
authSystem.authenticate(request2);
authSystem.authenticate(request3);
}
}
```

## Kết quả:



```
Run Main x
"C:\Program Files\Java\jdk-21\bin\java.exe"
Authenticating customer: customer123
Authenticating admin: admin456
Authenticating employee: employee789
Process finished with exit code 0
21110282_ChainofResponsibility_Custom > src > Main
```

Trong ví dụ này, chúng ta triển khai mẫu thiết kế **Chain of Responsibility** để xử lý yêu cầu xác thực từ các loại người dùng khác nhau. Mỗi loại người dùng được xác thực bởi một lớp xác thực cụ thể, và nếu một lớp không thể xác thực yêu cầu, nó sẽ chuyển tiếp yêu cầu cho lớp xác thực tiếp theo trong chuỗi. Điều này tạo ra một cấu trúc linh hoạt cho việc xác thực và cho phép dễ dàng mở rộng hệ thống để xử lý các loại người dùng mới mà không làm thay đổi quá nhiều mã nguồn hiện có.