Assignment 1: Task 2 - 27a

Design Patterns

2.1 Implemented Patterns:

2.1.1 Builder Pattern:

The aim of the builder design pattern is to separate the constructor of a complex object which has a lot of fields, from its representation so that the construction representation can create different representations.
The process of constructing an object is generic. It can be used to create different representations for the same object.

In general the Builder Design pattern helps with the minimization of the number of parameters in an object's constructor. As a result there is no need to pass null for optional parameters through the constructor. Furthermore, this creational design pattern helps writing the code faster by providing well-tested, proven development/design paradigms.
In our given setting, we have the basic Dto interface which is implemented by the RoleDto, the UserDto, the UuidDto, and the UserModify. Since the UserDto has a lot of fields, we have the builders for each attribute, since we do not have to modify all attributes at once. UserModify takes care of these modifications provided we have builders for each individual attribute.

As seen in the following code snippets the use of the builder for Dto helped us create Dto objects for usage as well as for testing in a much more elegant way. More than that in scenarios where not all parameters where required (i.e. For testing) there was no need to initialize them as null since the builder design pattern takes care of that.

```java
@JsonView(Views.Public.class)
public class UserDto implements Dto {
    private UUID id;
    private String netId;
    private Role role;
    private String address;
    private String description;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;

    public static UserDtoBuilder builder() {
        return new UserDtoBuilder();
    }

    public static class UserDtoBuilder {
        private transient UUID uuid1;
        private transient String netId1;
        private transient Role role1;
        private transient String address1;
        private transient String description1;
        private transient String firstName1;
        private transient String lastName1;
        private transient String email1;
        private transient String phoneNumber1;
```

Fig. 1 The UserDto which has the static builder() method which calls the UserDtoBuilder

```java
public class UserModify implements Dto {
    private String netId;
    private Role role;
    private String address;
    private String description;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;

    public static UserModifyBuilder builder() { return new UserModifyBuilder(); }

    public static class UserModifyBuilder {
        private transient String netId1;
        private transient Role role1;
        private transient String address1;
        private transient String description1;
        private transient String firstName1;
        private transient String lastName1;
        private transient String email1;
        private transient String phoneNumber1;

        UserModifyBuilder() {
        }

        public UserModifyBuilder netId(String netId) {
            this.netId1 = netId;
            return this;
```

Fig. 2 The UserModify which has builders for each individual attribute

```
UserModify userDto = UserModify.builder()
                .address("newAddress")
                .description("newDescription")
                .role(Role.FIRED)
                .netId("SomeNetId")
                .email("newEmail")
                .firstName("newFirstName")
                .lastName("newLastName")
                .phoneNumber("newPhoneNumber")
                .build();
```

Fig. 3: Builder Design pattern simplifies the construction of DTOs

```
UserDto userDto = UserDto.builder()
        .netId("netId")
        .uuid(getUuid( i: 1))
        .build();
```

Fig. 4: Builder Design pattern when not all parameters are required in testing

2.1.2 Facade Pattern:

The facade design pattern is a structural design pattern that provides a simplified interface to a more complex underlying system. It serves a role as a front-end which simplifies the usage of the underlying system(s) and provides an abstraction layer over the underlying functionality.

One of the main benefits of using the facade design pattern in general is that it can help to reduce the complexity of the system for the client. By providing a simplified interface, the client can interact with the system more easily and with less knowledge of the underlying details of the system. This can make it easier for the client to use the system and can also make the

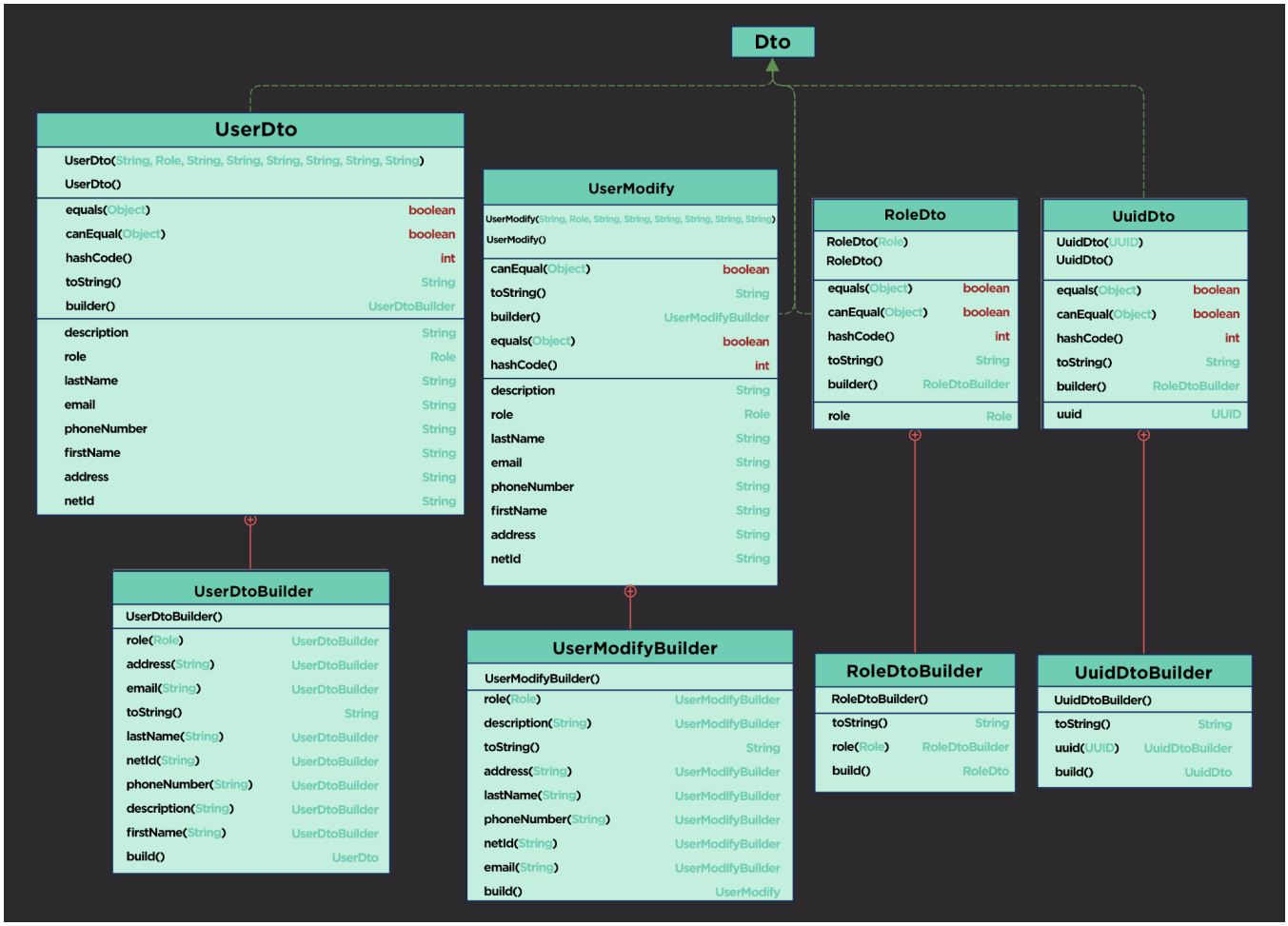
system more maintainable, as the client does not need to understand the details of the system in order to use it. This becomes very useful in the context of a microservice structure. Different "users" of a microservice (e.g. other microservices, as well as any other potential clients) are more "shielded" from the low level implementation details, which allows us to expose more ergonomic functions to them, as well as provide them with a more stable API (e.g. if the underlying implementation of the microservice changes, we only need to update the internal functions of the facade, not the clients themselves).

However, there are also some potential downsides to using the facade design pattern. One potential downside is that the facade may not provide access to all of the functionality of the underlying system, which can make it more difficult for the client to use the system in certain situations. In addition, the facade may add an additional layer of abstraction to the system, which can make it more difficult to understand and maintain. However, in our use-case this doesn't really play a role, as any client who for whatever reason wishes to interact with the microservice directly can do so by sending raw HTTP requests.

Another manifestation of the facade design pattern is the API Gateway microservice layout. In this scenario, the user interacts only with a single microservice, namely the API gateway. The gateway then routes the requests to appropriate microservice backends, based on the request it received from the user. This greatly simplifies the client-side interaction logic, as the client only has to interact with a single endpoint, and doesn’t have to be aware of the network infrastructure layout behind the gateway. Additionally, the API gateway can double as a load balancer, distributing requests to different instances of the same microservice, alleviating the load on them, and making the whole application more performant.

2.2 Class Diagrams:

Builder Class Diagram:



Facade Class Diagram: