Software Engineering Methods 2021

Assignment 1 Part 2

SEM group 34c

Design patterns

Facade

The implementation can be found in `GatewayController.java`.

We want to design a gateway microservice that takes incoming requests from the user and routes them to the correct microservice. In order to do this, we have several `****MicroserviceAdapter` classes that communicate with their respective microservice. The `GatewayController` class abstracts these microservice adapters away into one simple package: thus creating a facade design pattern.

The facade design pattern tries to simplify complex behaviour and interactions between multiple classes into a very straightforward small black box that “just does what it needs to do”. This comes with the benefit of having a good bird’s eye overview of complex functionality since the facade class exposes only what the application really needs.

As you can see in the following code snippet, the complex behaviour is completely abstracted away into some simple endpoint handlers. The user of this class (in our case this is Spring’s endpoint router) can use this simple interface to achieve complex calls to external microservices that internally perform a bunch of other actions. The user of the facade class does not really care what happens inside the black box. All it needs is the interface exposed by the facade.

```java
/** Gateway endpoint for authentication. ...*/
no usages  Iannis de Zwart
@PostMapping("/authenticate")
public ResponseEntity<AuthenticationResponseModel> authenticate(@RequestBody AuthenticationRequestModel request) {
    logger.info("Received authentication request from user: " + request.getNetId());
    return authenticationMicroserviceAdapter.authenticate(request);
}

/** Endpoint for registration. ...*/
no usages  Iannis de Zwart
@PostMapping("/register")
public ResponseEntity<Void> register(@RequestBody RegistrationRequestModel request) {
    logger.info("Received register request for user: " + request.getNetId());
    return authenticationMicroserviceAdapter.register(request);
}

/** Endpoint for creating a new offer. ...*/
no usages  Iannis de Zwart
@PostMapping("/create/match")
public ResponseEntity<Void> createActivityMatch(@RequestBody MatchCreationRequestModel request,
                                                @RequestHeader(HttpHeaders.AUTHORIZATION) String authToken) {
    logger.info(String.format("Received createActivityMatch request from user: %s, for activity: %s",
            request.getUserId(), request.getActivityId()));
    return activityMatchMicroserviceAdapter.createActivityMatch(request, authToken);
}
```

Pros:

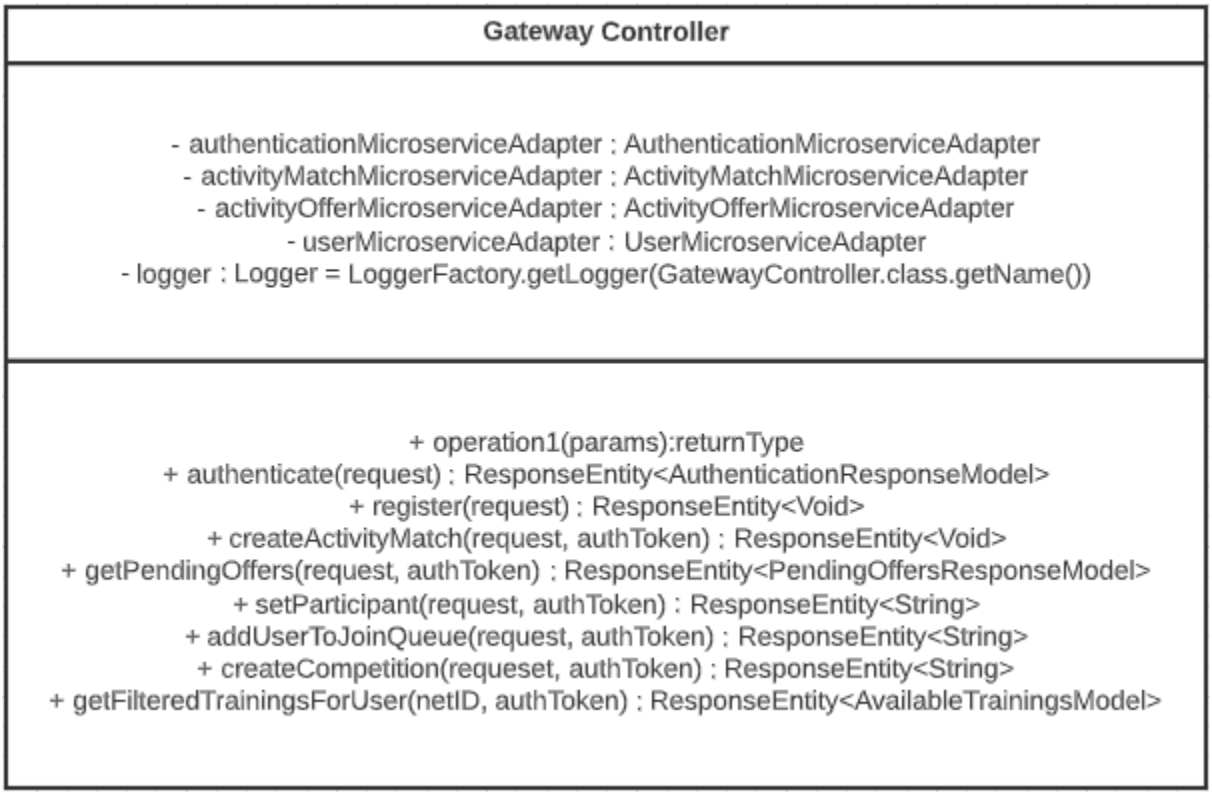
- Isolates complex behaviour into a simple and understandable interface
- Organises the behaviour of tightly coupled classes

Cons:

- Can become a god object that is used excessively by other classes throughout the application

We believe that using a facade design pattern for the gateway is a great investment since it makes code much more readable, and abstracts away complex logic in smaller bite-sized components that are used together under a facade.

Class diagram:



Builder Pattern

In User-Microservice

In our application, we also used the Builder Pattern for two different objects.
We used the builder pattern for creating a User in the context of User-Microservice. In this specific context, a User can have multiple attributes: username, password, gender, the positions he is able to fill on the boat, his/her availabilities regarding taking part in activities, the organization in which he/she is taking part of, the certificates for the boats and also whether the user is a professional or an amateur. Also, in our application, we created one abstract class User that has two children, one for a Professional User and one for an Amateur User. Having all of these attributes makes the User a complex Object which would require a big constructor with an abundance of parameters. This is where the Builder Pattern comes in handy and allows us to extract the object construction code out of its own classes and move it to separate classes called builders. In our specific context, there are two different builder classes that implement the same set of building steps, but in different manners. The important part is that you don't need to call all of the steps. One can call only those steps that are necessary for producing a particular configuration of an object. For example, we have an interface called UserBuilder that declares the user's construction steps that are common to all types of user builders. Our application also contains two different concrete builders that provide different implementations of the construction steps. There is the AmateurBuilder and the ProfessionalBuilder (although just the AmateurBuilder is implemented at the moment since the ProfessionalBuilder comes as a should-have).

```
public interface UserBuilder<T extends User> {

    UserBuilder<T> setNetId(NetId netId);

    UserBuilder<T> setPassword(HashedPassword password);

    UserBuilder<T> setGender(Gender gender);

    UserBuilder<T> setCertificates(List<String> certificates);
```

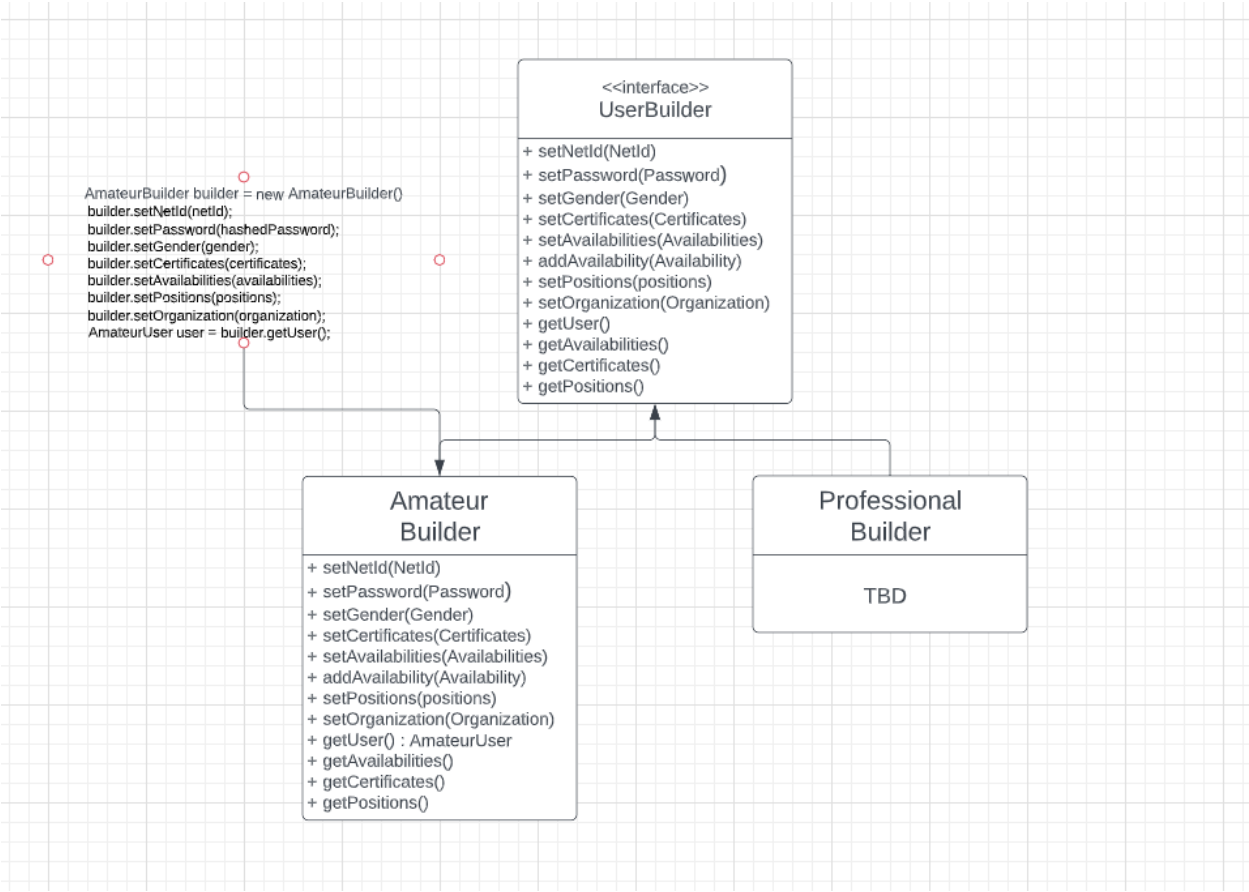
```java
public class AmateurBuilder implements UserBuilder<AmateurUser> {

    private transient NetId netId;
    private transient HashedPassword password;
    private transient Gender gender;
    private transient List<String> certificates;
    private transient TreeMap<LocalDateTime, LocalDateTime> availabilities;
    private transient List<TypesOfPositions> positions;
    private transient String organization;

    @Override
    public UserBuilder<AmateurUser> setNetId(NetId netId) {
        this.netId = netId;
        return this;
    }

    @Override
    public UserBuilder<AmateurUser> setPassword(HashedPassword password) {
        this.password = password;
        return this;
    }

    @Override
    public UserBuilder<AmateurUser> setGender(Gender gender) {
        this.gender = gender;
        return this;
    }
```

```java
@Override
public List<Availability> getAvailabilities() {
    List<Availability> resultedAvailabilities = new ArrayList<>();
    for (Map.Entry<LocalDateTime, LocalDateTime> entry : availabilities.entrySet()) {
        LocalDateTime start = entry.getKey();
        LocalDateTime end = entry.getValue();
        resultedAvailabilities.add(new Availability(netId, start, end));
    }
    return resultedAvailabilities;
}

@Override
public Set<UserCertificate> getCertificates() {
    Set<UserCertificate> noDuplicateCertificates = new HashSet<>();
    for (String certificate : certificates) {
        noDuplicateCertificates.add(new UserCertificate(netId, certificate));
    }
    return noDuplicateCertificates;
}

@Override
public List<PositionEntity> getPositions() {
    return positions.stream()
            .distinct()
            .map(x -> new PositionEntity(netId, x))
            .collect(Collectors.toList());
}
```

```java
// create new account
AmateurBuilder builder = new AmateurBuilder();
builder.setNetId(netId)
        .setPassword(hashedPassword)
        .setGender(gender)
        .setCertificates(certificates)
        .setAvailabilities(availabilities)
        .setPositions(positions)
        .setOrganization(organization);
AmateurUser user = builder.getUser();
```

As you can clearly see in the last picture, after setting all the parameters needed for creating an Amateur User, the last part is fetching all the information in order to create a new object.

Class Diagram



In Activity-Offer-Microservice

We also use the Builder Pattern in **Activity Offer** class in the context of Activity Offer Microservice. We would like to be able to construct our Activity Offer step-by-step because it is a complex object with a lot of fields. These attributes are *ID, position* for which a certain offer is for, *isActive* value to determine if the offer is still valid, *start and end times* of an offer, the *ID of an owner, and type* (whether training or a competition). We also store the *name of an offer*, as well as its *description*. Some of these parameters needn't be used every time, for example, the name and description of a certain offer are not required in every use case. **Activity Offer** is an abstract class, from which **Training Offer** and **Competition Offer** inherit.
To avoid having many monstrous constructors with different parameters, we implemented a builder pattern here as well to separate the construction of this complex object from its representation. This choice will improve the flexibility and maintainability of our application. Below we can see the implementation of an **ActivityOfferBuilder** generic interface(to deal with an abstract class).

```
38 usages  2 implementations
public interface ActivityOfferBuilder<T extends ActivityOffer> {
    2 implementations
    ActivityOfferBuilder<T> setPosition(TypesOfPositions position);
    5 usages  2 implementations
    ActivityOfferBuilder<T> setActive(boolean isActive);
    2 implementations
    ActivityOfferBuilder<T> setStartTime(LocalDateTime startTime);
    2 implementations
    ActivityOfferBuilder<T> setEndTime(LocalDateTime endTime);
    2 implementations
    ActivityOfferBuilder<T> setOwnerId(String ownerId);
    2 implementations
    ActivityOfferBuilder<T> setBoatCertificate(String boatCertificate);
    2 implementations
    ActivityOfferBuilder<T> setType(TypesOfActivities type);
    2 implementations
    ActivityOfferBuilder<T> setName(String name);
    2 implementations
    ActivityOfferBuilder<T> setDescription(String description);
    2 implementations
    ActivityOfferBuilder<T> setOrganisation(String organisation);
    3 usages  2 implementations
    ActivityOfferBuilder<T> setFemale(boolean isFemale);
    3 usages  2 implementations
    ActivityOfferBuilder<T> setPro(boolean isPro);
    9 usages  2 implementations
    T build();
}
```

Each of the methods declared here set one of the many parameters. **build()** method is going to create an object with all of the declared attributes in the builder.

Below we can see the implementation of a builder class for **TrainingOffer**.

```
3 usages
public class TrainingOfferBuilder implements ActivityOfferBuilder<TrainingOffer> {

    2 usages
    private transient TypesOfPositions position;
    2 usages
    private transient boolean isActive;
    2 usages
    private transient LocalDateTime startTime;
    2 usages
    private transient LocalDateTime endTime;
    2 usages
    private transient String ownerId;
    2 usages
    private transient String boatCertificate;
    2 usages
    private transient TypesOfActivities type;
    2 usages
    private transient String name;
    2 usages
    private transient String description;

    1 usage
    @Override
    public ActivityOfferBuilder<TrainingOffer> setPosition(TypesOfPositions position) {
        this.position = position;
        return this;
    }

    1 usage
    @Override
    public ActivityOfferBuilder<TrainingOffer> setActive(boolean isActive) {
        this.isActive = isActive;
        return this;
    }
```

And here is the implementation of the methods. We set certain parameters and return **this** object, which is a point for the builder pattern.

```java
@Override
public ActivityOfferBuilder<TrainingOffer> setStartTime(LocalDateTime startTime) {
    this.startTime = startTime;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setEndTime(LocalDateTime endTime) {
    this.endTime = endTime;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setOwnerId(String ownerId) {
    this.ownerId = ownerId;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setBoatCertificate(String boatCertificate) {
    this.boatCertificate = boatCertificate;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setType(TypesOfActivities type) {
    this.type = type;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setName(String name) {
    this.name = name;
    return this;
}

@Override
public ActivityOfferBuilder<TrainingOffer> setDescription(String description) {
    this.description = description;
    return this;
}
```

Finally, we can create an object for which parameters have been set by using the build method. It simply creates a new **TrainingOffer** with the previously set parameters.

```
    1 usage
    @Override
    public TrainingOffer build() {
        return new TrainingOffer(position, isActive,
                                 startTime, endTime,
                                 ownerId, boatCertificate,
                                 type, name, description);
    }
}
```

We use that implementation of the builder pattern for instance in **ActivityOfferService:**

```
    1 usage
    public TrainingOffer setTrainingParameters(TypesOfPositions position, boolean isActive,
                                               LocalDateTime startTime, LocalDateTime endTime,
                                               String ownerId, String boatCertificate,
                                               TypesOfActivities type, String name, String description) {
        TrainingOfferBuilder builder = new TrainingOfferBuilder();
        builder.setPosition(position)
                .setActive(isActive)
                .setStartTime(startTime)
                .setEndTime(endTime)
                .setOwnerId(ownerId)
                .setBoatCertificate(boatCertificate)
                .setType(type)
                .setName(name)
                .setDescription(description);
        return builder.build();
    }

```

This allows us to create every training, without using different constructor methods.
The builder pattern implementation causes a lot of overhead code and increases its complexity, however, at the same time, it allows us to reuse the same construction code when building various representations of **ActivityOffer**.

Class diagram