SEM Assignment 1 - Task 2 - group 33A

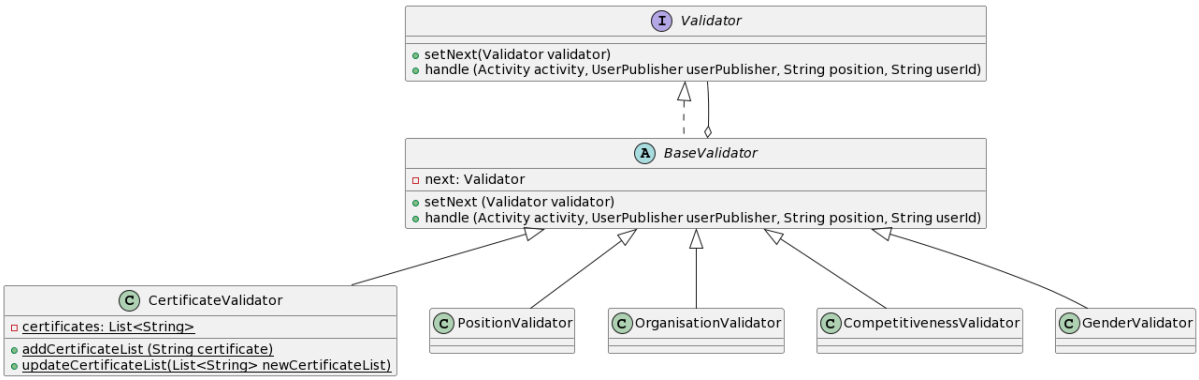
Laimonas Lipinskas (5559375)
Lotte Kremer (4861957)
Nicolae Radu (5527104)
Razvan Loghin (5480620)
Dawid Plona (5205018)
Vlad Nitu (5529557)

December 23, 2022

Chain of Responsibility pattern

The ***"Chain of Responsibility" design pattern*** was implemented as decided during the team meetings. It follows the lecture notes and is adapted to the given scenario. It uses the *Validator* interface that is being implemented by the *BaseValidator* abstract class, there are 5 different validator classes that extend from it (*Position*, *Certificate*, *Gender*, *Organization* and *Competitiveness*), each of which checks a specific task and represents a building block of the chain.

The following represents the Chain of Responsibility design pattern ***diagram*** implemented in the Activity microservice:



The following paragraph explains *how each sub-validator works.*

a. ***Position Validator*** - Checks if the position from the activity is included in the positions list that the user is capable of fulfilling. In other words, whether 'userPositions.contains(activityPosition)'.

b. ***Organisation Validator*** - Checks whether the user's organisation suits the activity's organisation requirement.

c. Gender Validation - Checks whether the user's gender suits the activity's gender requirement (Male or Female)

d. ***Competitiveness Validator*** - Ensures that an amateur user does not participate in a Competition that requires a high competitiveness level.

e. ***Certificate Validator*** - Checks whether the users' certificate supersedes the activity's certificate requirement. Superseding relation is defined as follows: C4 is superseded by 4+, which is superseded by 8+. These certificates are stored in a list in such a way: for two given certificates A and B, B supersedes A iff index(B) is larger than index(A), where the index represents the position of a certificate in a list. This is how the design supports adding new certificates later to the system by manipulating the list.

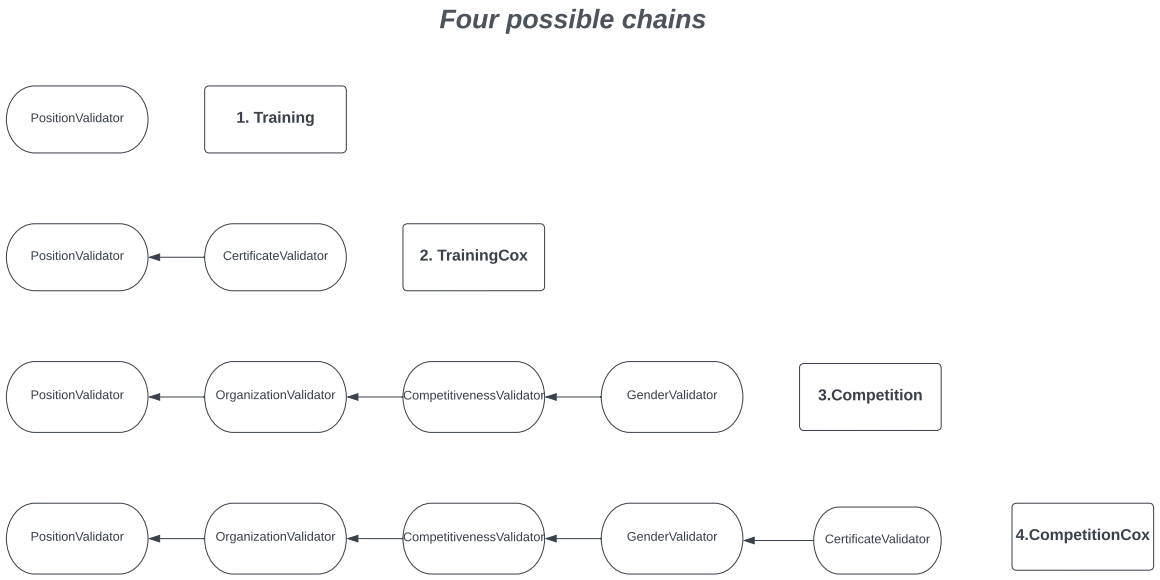
The ***logic of the chain*** is as following:

1. The *basic validator made for Trainings* - Training validator, which is the simplest one

2. The *TrainingCox validator*, which chains after the Training one and adds the CertificateCheck for the Cox position

3. The *simple Competition validator*, which chains after the Training one and has 3 additional checks: organisation, competitiveness and gender

4. The *CompetitionCox validator*, which chains after the simple Competition validator and has one additional check: the certificate



Strategy pattern

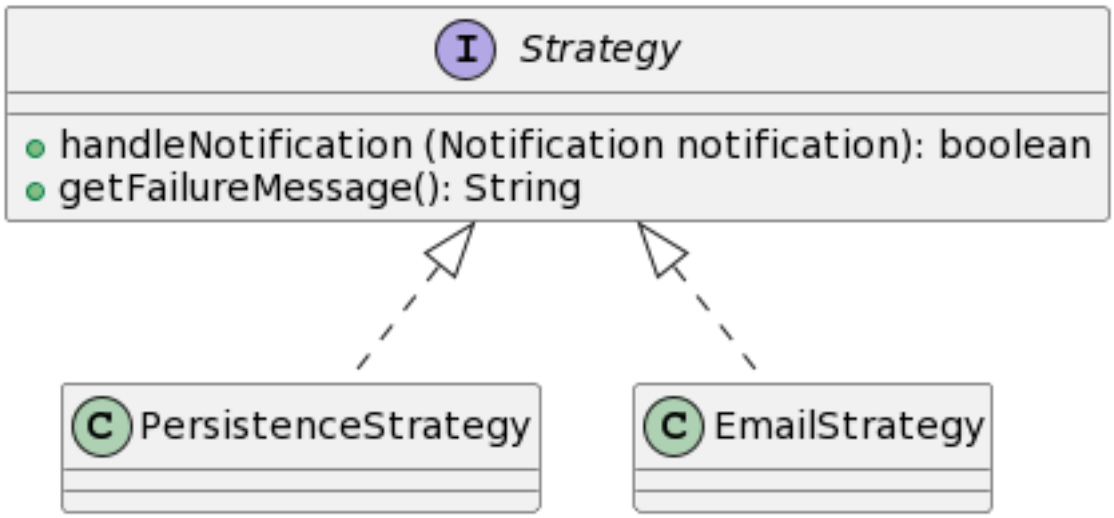
Strategy pattern was chosen as applicable to notification handling. This design was chosen to facilitate implementing new ways to notify the application's users in the future. The pattern is implemented by having a Strategy interface that contains a handler method called 'handleNotification', which processes the notification, and returns if the processing was a success, and 'getFailureMessage', which displays a different unsuccessful message, depending on the strategy used.

There are two strategies implemented, both of which implement the interface "NotificationStrategy":

```
8 usages  2 implementations  Laimonas Lipinskas
public interface NotificationStrategy {
    7 usages  2 implementations  Laimonas Lipinskas
    boolean handleNotification(Notification notification);

    2 usages  2 implementations  Laimonas Lipinskas
    String getFailureMessage();
}
```

The two strategies that our system currently supports are:



1. Persistence Strategy that saves the notification provided in the Notification database, so that the users can later request the piled-up notifications by means of a GET request to the '/getNotifications' API. Thus, it is ensured that no notifications are being lost when the notified user is offline, as they can retrieve them by communicating with the Notification subsystem. If persisting the given notification is not possible, 'getFailureMessage' is used to announce the client that "Notification failed to be saved to the database for later user requests". Persistance is done by using the "NotificationDatabaseService", which communicates with the JPA Repository specific to the Notification microservice, which is connected to a MySQL database hosted on Projects EWI.

```
@Component
public class PersistenceStrategy implements NotificationStrategy {
```

2. Email Strategy, which is meant to notify the user by sending them an e-mail to the address that they provided when creating their account, given that the user has a valid email address. It was decided to model the e-mail notifications by using the JavaMailSender class that is already implemented in 'springframework.mail.javamail' package. When sending a notification, a personalized e-mail message is created in the 'handleNotification' method, and sender is set as "sem33aservice@gmail.com", the developers' contact point. If the e-mail does not reach the targeted user, the 'getFailureMessage' method is used in order to announce to the client that "Failed to send email". There are two types of notifications being handled: "notifyUser" and "notifyOwner". If the e-mail targets an owner, the subject is: 'A new request', letting the user know that they needs to further decide whether they accept this user to take part in his activity, compared to targeting an owner, for which the subject is "Accepted for activity", confirming that they were allowed to take part in the activity they applied for.

```
@Primary
@Component
public class EmailStrategy implements NotificationStrategy {
```

As presented above, each strategy is annotated with the '@Component' annotation, so that Spring can wire this bean as a component. Moreover, there is only one strategy that can be used at a time. Thus, Spring needs and indication as to which one to use, otherwise it will not function properly because it won't be able to choose between these 2 strategies. As a design choice, it was decided to use the EmailStrategy as the primary strategy, so if an application user decides to choose another strategy type, they will have to annotate the other component as '@Primary'.