Task 2: Design Patterns

Chain of Responsibility

1.1

The first design pattern we have chosen is “Chain of Responsibility” in the Matcher microservice because the activities have to go through 4 types of filtering before reaching the user.

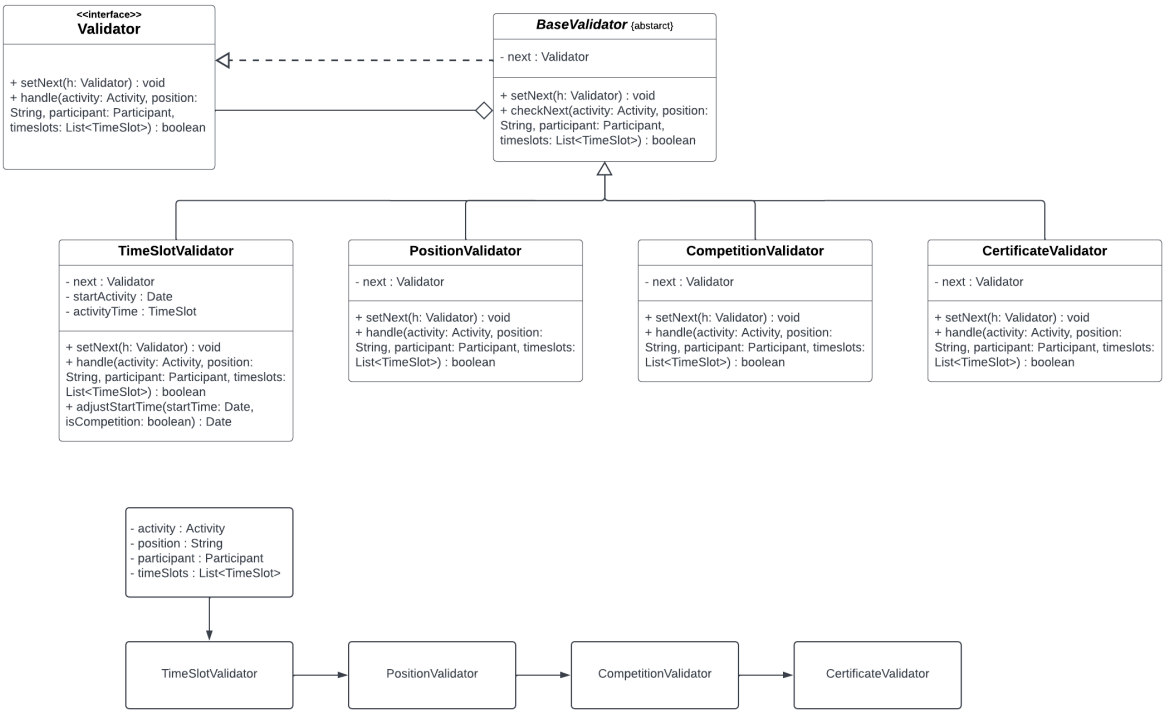
Firstly, we filter by TimeSlot with the TimeSlotValidator. The startDate of the activity is changed in order to meet the scenario requirements(-0.5 h for Training and –1 day for Competition) because the Participant needs time to reach the Activity location and staff has to be notified. If the startTime of the Participant is before or equal to the startTime of the activity and the endTime of the Participant is after or equal to the endTime of the activity the next validator will be called. Otherwise, false will be returned.

Secondly, we filter by position with the PositionValidator. If the Participant is able to fulfill the position passed to the method, the next validator will be checked. Otherwise, the chain stops.

Next, we filter by the Competitions's details with the CompetitionValidator. In case the Activity is a Training the next validator will be called. If not, Participant's gender, organization and competitiveness will be checked against Competition's specifications. In the circumstance that all of them match, the next validator will be analysed. Otherwise, false will be returned.

Finally, we filter by certificate with the CertificateValidator. If the current position is cox, this Validator will verify if the Participant's certificate is better than the boat used in the activity. If this is the case or the current position is not cox, then true will be returned because this is the last validator in the chain, else false will be returned.

1.2


<<interface>>
Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
BaseValidator {abstarct}
- next : Validator
+ setNext(h: Validator) : void
+ checkNext(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
TimeSlotValidator
- next : Validator
- startActivity : Date
- activityTime : TimeSlot
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
+ adjustStartTime(startTime: Date, isCompetition: boolean) : Date
PositionValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
CompetitionValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean
CertificateValidator
- next : Validator
+ setNext(h: Validator) : void
+ handle(activity: Activity, position: String, participant: Participant, timeslots: List<TimeSlot>) : boolean

- activity : Activity
- position : String
- participant : Participant
- timeSlots : List<TimeSlot>
TimeSlotValidator
PositionValidator
CompetitionValidator
CertificateValidator

Factory Method

2.1

The second design pattern we have chosen is “Factory Method”, its usage being evident in the Notification microservice, more specifically in NotificationController. This choice came to us naturally due to the implementation of TransferMatch, our DTO which is used to communicate between microservices.

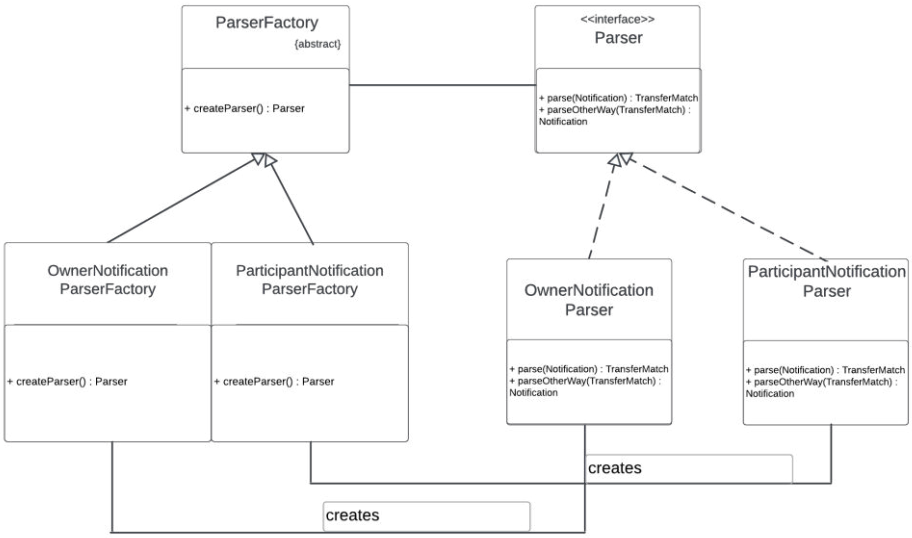
Firstly, when an owner accepts a list of users to partake in their activity, the matcher microservice sends a list of TransferMatch to the Notification microservice, which need to be parsed (since all attributes of TransferMatch are Strings), stored in the database as a Notification and sent to the accepted users. This action is being done by the “parseOtherWay” method in the ParticipantNotificationParser.

Secondly, when a user confirms that they would like to partake in an activity, the matcher microservice sends a TransferMatch to the Notification microservice, which will notify the owner of said activity about the participant. The transformation to a Notification object is done by the “parseOtherWay” method in the OwnerNotificationParser.

When an owner needs to be notified about the users that agreed to participate in their activity, the Notification microservice retrieves these Notifications from the database and they are converted to TransferMatches by the “parse” method in the OwnerNotificationParser.

Finally, when a user needs to be notified about the activities he has been accepted into, the Notification microservice retrieves these Notifications from the database and turns them into TransferMatches using the “parse” method in the ParticipantNotificationParser, DTOs which are being sent to the Participant microservice.

2.2


ParserFactory
{abstract}
+ createParser() : Parser
<<interface>>
Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
OwnerNotification ParserFactory
+ createParser() : Parser
ParticipantNotification ParserFactory
+ createParser() : Parser
OwnerNotification Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
ParticipantNotification Parser
+ parse(Notification) : TransferMatch
+ parseOtherWay(TransferMatch) : Notification
creates
creates