SEM Assignment 1 - Design patterns

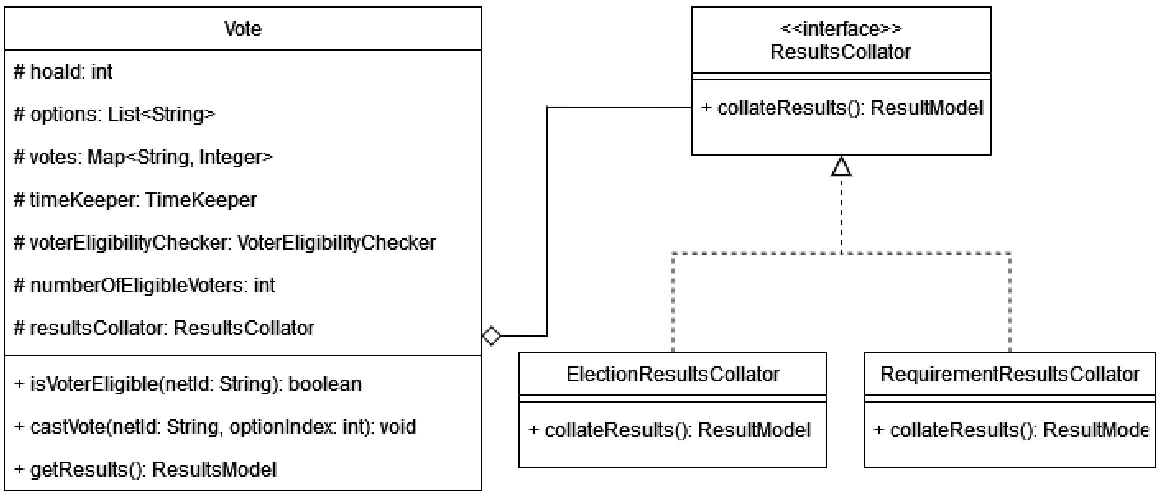
Group 18B

1 Strategy Design Pattern

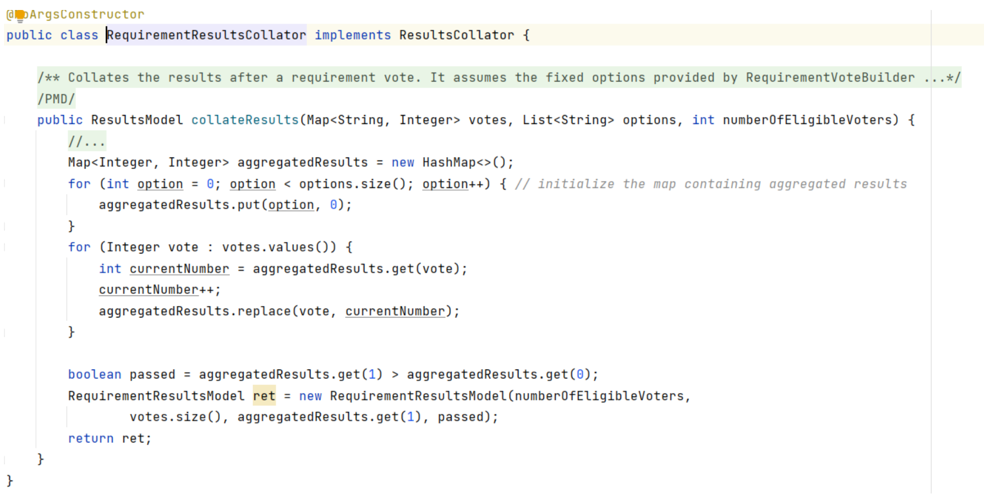
This pattern is applied within `voting-microservice`. It is used for collating the results of each type of voting with different response models. The types of voting are very similar, and the only difference between them are the methods for checking a voter's eligibility and collating results. In order to allow testing, inversion of control was used with regard to checking a voter's eligibility (since this action requires making a request to another microservice, and the interface can be easily mocked for testing). Because IoC is already used for the first differentiating factor (i.e., whether one can vote in those elections), we considered the two different voting types not dissimilar enough to make an inheritance structure for them, just so that the different vote collation methods can be accommodated. Therefore, a strategy was chosen for collating the votes for the different types of voting, and the strategy chosen is later determined based on the used builder (see next section).

The design choice of using composition rather than inheritance allows the independent usage of either combination of `VoterEligibilityChecker` and `ResultsCollator` should there be the need for that. The existence of separate builders (see Builder Design Pattern) allows easy management of those differences.

Additionally, a minor use of the strategy design pattern is done in the testing of the `hoa-microservice`. There, argument matches are used, all of which employ the `ArgumentMatcher` interface, with the purpose of making sure repository items are correctly saved. Despite the `ArgumentMatcher` being a part of Mockito, the team determined it is worth mentioning as a small part of the project.

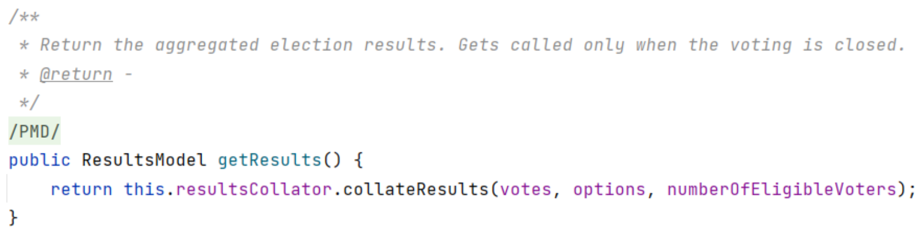

Vote
# hoaId: int
# options: List<String>
# votes: Map<String, Integer>
# timeKeeper: TimeKeeper
# voterEligibilityChecker: VoterEligibilityChecker
# numberOfEligibleVoters: int
# resultsCollator: ResultsCollator
+ isVoterEligible(netId: String): boolean
+ castVote(netId: String, optionIndex: int): void
+ getResults(): ResultsModel
<<interface>>
ResultsCollator
+ collateResults(): ResultModel
ElectionResultsCollator
+ collateResults(): ResultModel
RequirementResultsCollator
+ collateResults(): ResultMode

Class diagram of the deployed Strategy design pattern



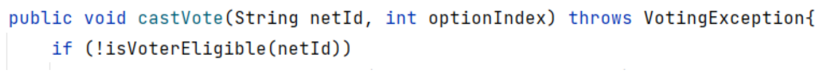
```
@oArgsConstructor
public class RequirementResultsCollator implements ResultsCollator {

    /** Collates the results after a requirement vote. It assumes the fixed options provided by RequirementVoteBuilder ...*/
    /PMD/
    public ResultsModel collateResults(Map<String, Integer> votes, List<String> options, int numberOfEligibleVoters) {
        //...
        Map<Integer, Integer> aggregatedResults = new HashMap<>();
        for (int option = 0; option < options.size(); option++) { // initialize the map containing aggregated results
            aggregatedResults.put(option, 0);
        }
        for (Integer vote : votes.values()) {
            int currentNumber = aggregatedResults.get(vote);
            currentNumber++;
            aggregatedResults.replace(vote, currentNumber);
        }

        boolean passed = aggregatedResults.get(1) > aggregatedResults.get(0);
        RequirementResultsModel ret = new RequirementResultsModel(numberOfEligibleVoters,
                votes.size(), aggregatedResults.get(1), passed);
        return ret;
    }
}
```

Results Collator Example



```
/**
 * Return the aggregated election results. Gets called only when the voting is closed.
 * @return -
 */
/PMD/
public ResultsModel getResults() {
    return this.resultsCollator.collateResults(votes, options, numberOfEligibleVoters);
}
```

Results Collator Use



```
public void castVote(String netId, int optionIndex) throws VotingException{
    if (!isVoterEligible(netId))
```

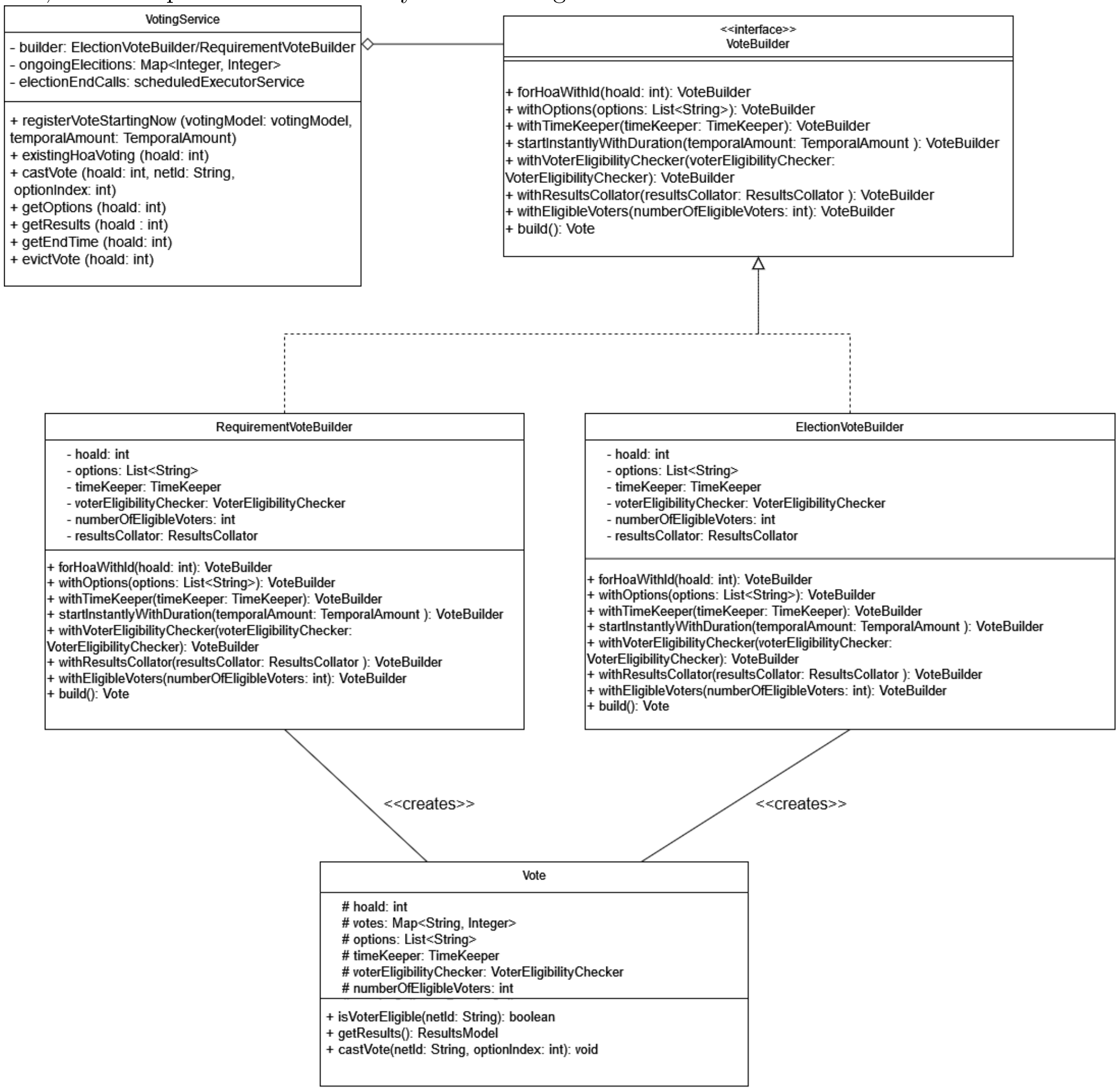
Casting a vote employing VotingEligibilityChecker

2 Builder Design Pattern

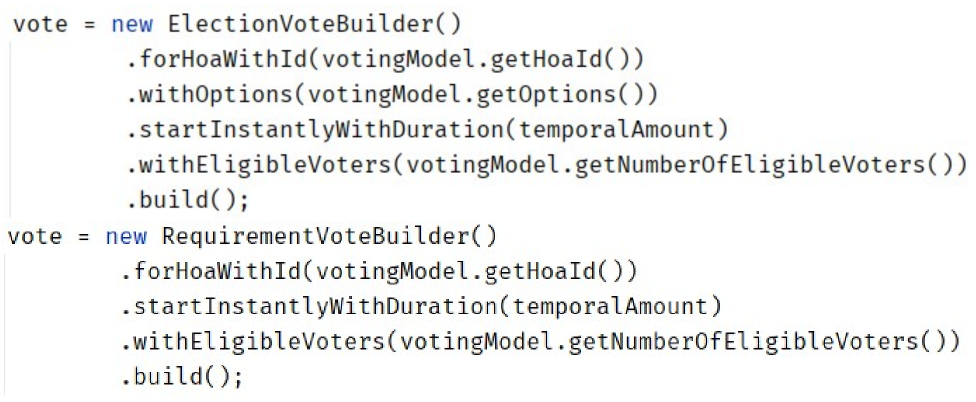
This pattern is applied within `voting-microservice`, and more specifically, when creating the objects that manage the votes. It was chosen is because it allows easy step-by-step creation of those objects that require multiple parameters. The microservice currently two implemented types of votes inside of it (Requirement Voting and Election Voting), and the reason it was extracted into its own microservices is to allow for the future expansion of more voting types for more types of clients (not just home owner associations). A builder is best for this type of extension. This can be seen with the `VoteBuilder` interface, which allows the existence of multiple builders. Therefore, with the appearance of more vote types, this design pattern can easily accommodate them. Moreover, the step-wise creation of an instance mandates the use of methods, whose names facilitate the usage of ubiquitous language within the code base.

An abstract factory was also considered for this task. In the end, it was decided against it since the factory focuses on creating the instance, with the task of populating it offloaded to the methods that need the created objects. It was found that votes require a considerable number of parameters, especially since the use of interfaces such as `TimeKeeper` was needed to allow testing.

Not necessarily all methods of the builder have to be used, in order to construct an object of type Vote. Some of them are added for testing purposes (more specifically, the `TimeKeeper` and VoterEligibilityChecker), and for the RequiremenVoteBuilder, the options are most often assumed to be three - "for", "against", and "abstain". In case they are not, they can be specified with the method withOptions in conjunction with providing an implementation of VoterEligibilityChecker so as to properly count the results after the vote's end. The builder can easily be extended with more methods in order for it to be able to create more types of objects, which helps with the scalability of the voting microservice.

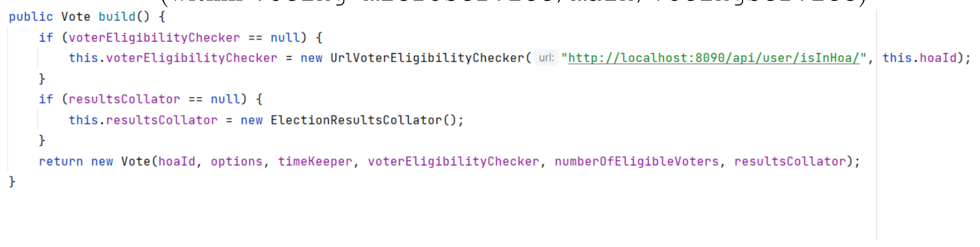

VotingService
- builder: ElectionVoteBuilder/RequirementVoteBuilder
- ongoingElecitions: Map<Integer, Integer>
- electionEndCalls: scheduledExecutorService
+ registerVoteStartingNow (votingModel: votingModel, temporalAmount: TemporalAmount)
+ existingHoaVoting (hoaId: int)
+ castVote (hoaId: int, netId: String, optionIndex: int)
+ getOptions (hoaId: int)
+ getResults (hoaId : int)
+ getEndTime (hoaId: int)
+ evictVote (hoaId: int)
<<interface>>
VoteBuilder
+ forHoaWithId(hoaId: int): VoteBuilder
+ withOptions(options: List<String>): VoteBuilder
+ withTimeKeeper(timeKeeper: TimeKeeper): VoteBuilder
+ startInstantlyWithDuration(temporalAmount: TemporalAmount): VoteBuilder
+ withVoterEligibilityChecker(voterEligibilityChecker: VoterEligibilityChecker): VoteBuilder
+ withResultsCollator(resultsCollator: ResultsCollator): VoteBuilder
+ withEligibleVoters(numberOfEligibleVoters: int): VoteBuilder
+ build(): Vote
RequirementVoteBuilder
- hoaId: int
- options: List<String>
- timeKeeper: TimeKeeper
- voterEligibilityChecker: VoterEligibilityChecker
- numberOfEligibleVoters: int
- resultsCollator: ResultsCollator
+ forHoaWithId(hoaId: int): VoteBuilder
+ withOptions(options: List<String>): VoteBuilder
+ withTimeKeeper(timeKeeper: TimeKeeper): VoteBuilder
+ startInstantlyWithDuration(temporalAmount: TemporalAmount): VoteBuilder
+ withVoterEligibilityChecker(voterEligibilityChecker: VoterEligibilityChecker): VoteBuilder
+ withResultsCollator(resultsCollator: ResultsCollator): VoteBuilder
+ withEligibleVoters(numberOfEligibleVoters: int): VoteBuilder
+ build(): Vote
ElectionVoteBuilder
- hoaId: int
- options: List<String>
- timeKeeper: TimeKeeper
- voterEligibilityChecker: VoterEligibilityChecker
- numberOfEligibleVoters: int
- resultsCollator: ResultsCollator
+ forHoaWithId(hoaId: int): VoteBuilder
+ withOptions(options: List<String>): VoteBuilder
+ withTimeKeeper(timeKeeper: TimeKeeper): VoteBuilder
+ startInstantlyWithDuration(temporalAmount: TemporalAmount): VoteBuilder
+ withVoterEligibilityChecker(voterEligibilityChecker: VoterEligibilityChecker): VoteBuilder
+ withResultsCollator(resultsCollator: ResultsCollator): VoteBuilder
+ withEligibleVoters(numberOfEligibleVoters: int): VoteBuilder
+ build(): Vote
<<creates>>
<<creates>>
Vote
# hoaId: int
# votes: Map<String, Integer>
# options: List<String>
# timeKeeper: TimeKeeper
# voterEligibilityChecker: VoterEligibilityChecker
# numberOfEligibleVoters: int
+ isVoterEligible(netId: String): boolean
+ getResults(): ResultsModel
+ castVote(netId: String, optionIndex: int): void

Class diagram of the deployed Builder design pattern



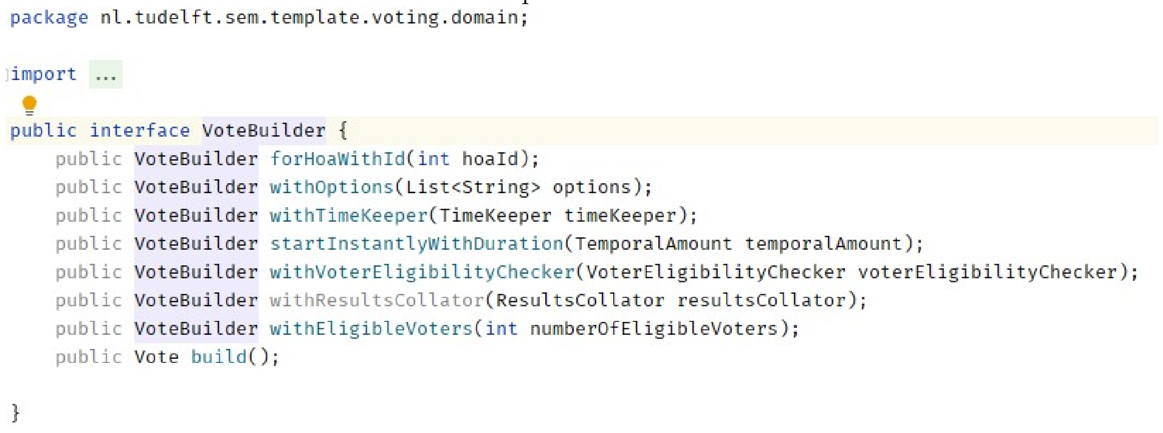
```java
vote = new ElectionVoteBuilder()
        .forHoaWithId(votingModel.getHoaId())
        .withOptions(votingModel.getOptions())
        .startInstantlyWithDuration(temporalAmount)
        .withEligibleVoters(votingModel.getNumberOfEligibleVoters())
        .build();
vote = new RequirementVoteBuilder()
        .forHoaWithId(votingModel.getHoaId())
        .startInstantlyWithDuration(temporalAmount)
        .withEligibleVoters(votingModel.getNumberOfEligibleVoters())
        .build();
```

Examples of an object created with the builder
(within `voting-microservice/main/VotingService`)



```java
public Vote build() {
    if (voterEligibilityChecker == null) {
        this.voterEligibilityChecker = new UrlVoterEligibilityChecker( url: "http://localhost:8090/api/user/isInHoa/", this.hoaId);
    }
    if (resultsCollator == null) {
        this.resultsCollator = new ElectionResultsCollator();
    }
    return new Vote(hoaId, options, timeKeeper, voterEligibilityChecker, numberOfEligibleVoters, resultsCollator);
}
```

Build Method Example within ElectionVoteBuilder



```java
package nl.tudelft.sem.template.voting.domain;

import ...

public interface VoteBuilder {
    public VoteBuilder forHoaWithId(int hoaId);
    public VoteBuilder withOptions(List<String> options);
    public VoteBuilder withTimeKeeper(TimeKeeper timeKeeper);
    public VoteBuilder startInstantlyWithDuration(TemporalAmount temporalAmount);
    public VoteBuilder withVoterEligibilityChecker(VoterEligibilityChecker voterEligibilityChecker);
    public VoteBuilder withResultsCollator(ResultsCollator resultsCollator);
    public VoteBuilder withEligibleVoters(int numberOfEligibleVoters);
    public Vote build();

}
```

The interface `VoteBuilder`