Assignment 1 - Part 2
Group 22a

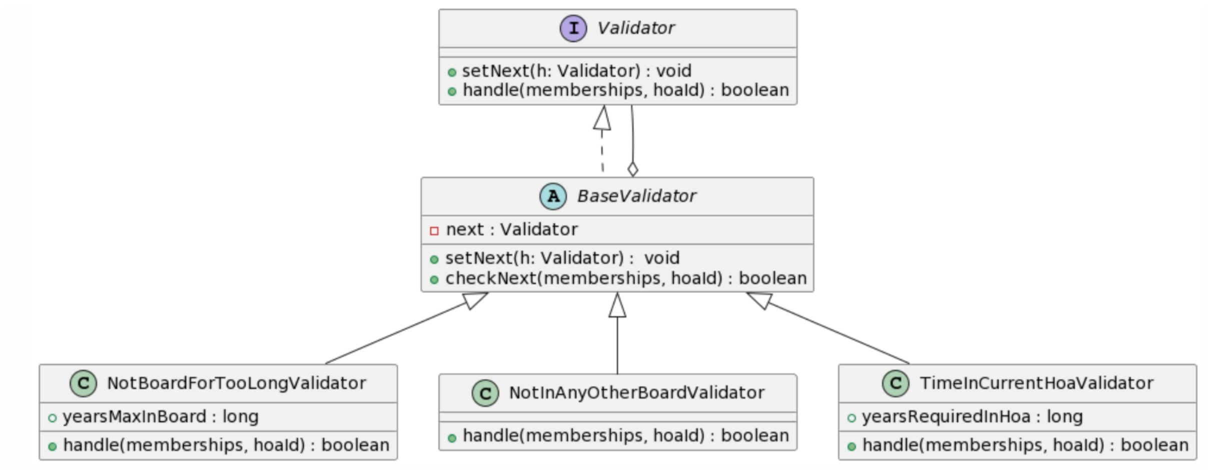
In this part of the assignment we will go over the design patterns we have implemented within the projects, how we implemented them, our reasons for choosing to do so and some code snippets highlighting them.

I. Chain of Responsibility

We decided to implement this behavioral design pattern as it suits the needs of our project related to elections. For a member to take part in any election (for example let's take BoardElection), they have to satisfy certain requirements, related to how long they stayed in the association and so on. Therefore we're given a set of constraints and we have to put a member through this "chain" of checks, if they fail one, they don't go on through the other checks, and if they pass all the handlers, they can participate in the Election.

The process works as follows. Each request to participate in an election is passed through the chain of handlers. Then each handler decides if the process is halted, or if the request can go on to the next handler in the chain. The class diagram for the implementation of this design pattern in our code is attached below along with code snippets for the handlers.

Class diagram:

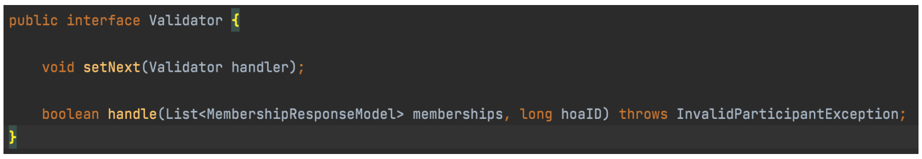

I Validator
setNext(h: Validator) : void
handle(memberships, hoaId) : boolean
A BaseValidator
next : Validator
setNext(h: Validator) : void
checkNext(memberships, hoaId) : boolean
C NotBoardForTooLongValidator
yearsMaxInBoard : long
handle(memberships, hoaId) : boolean
C NotInAnyOtherBoardValidator
handle(memberships, hoaId) : boolean
C TimeInCurrentHoaValidator
yearsRequiredInHoa : long
handle(memberships, hoaId) : boolean

Note that the bottom 3 classes are our concrete implementations.

Code snippets:

Shared API for all Validators

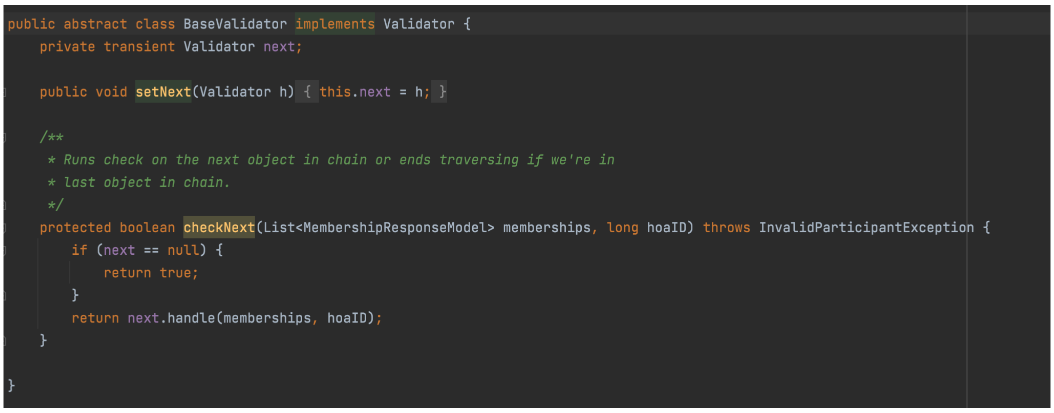
(***hoa/src/main/java/hoa/domain/electionchecks/Validator.java***):



```java
public interface Validator {

    void setNext(Validator handler);

    boolean handle(List<MembershipResponseModel> memberships, long hoaID) throws InvalidParticipantException;
}
```

Base Handler

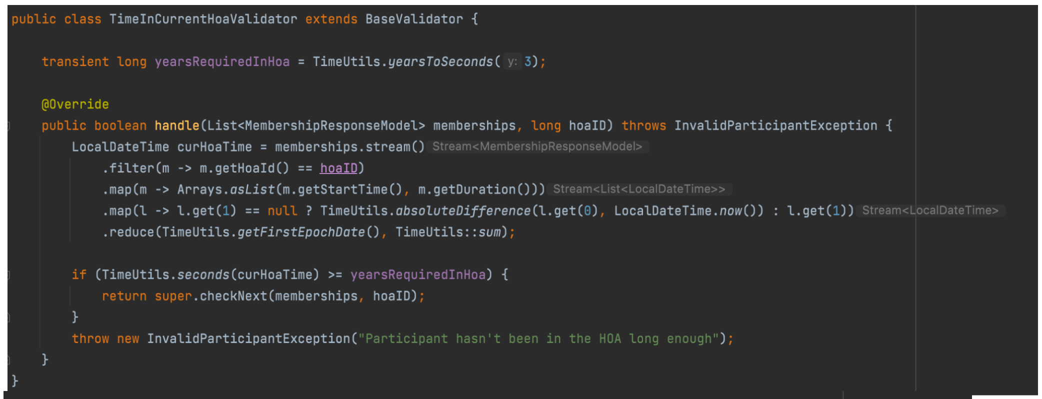
(***hoa/src/main/java/hoa/domain/electionchecks/BaseValidator.java***):



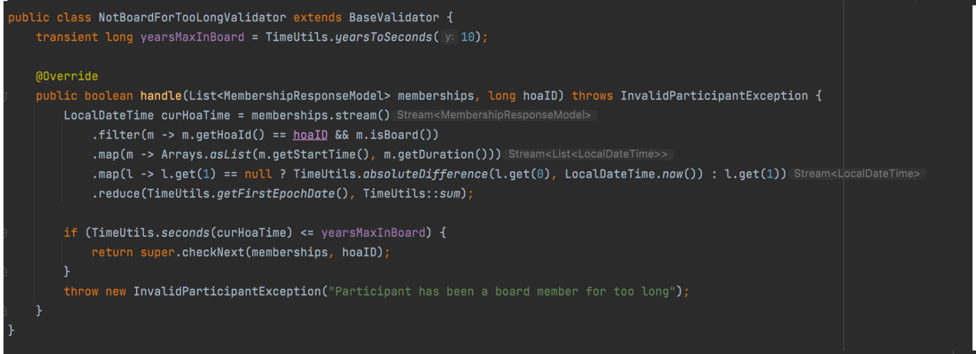
```java
public abstract class BaseValidator implements Validator {
    private transient Validator next;

    public void setNext(Validator h) { this.next = h; }

    /**
     * Runs check on the next object in chain or ends traversing if we're in
     * last object in chain.
     */
    protected boolean checkNext(List<MembershipResponseModel> memberships, long hoaID) throws InvalidParticipantException {
        if (next == null) {
            return true;
        }
        return next.handle(memberships, hoaID);
    }

}
```

Concrete Handlers

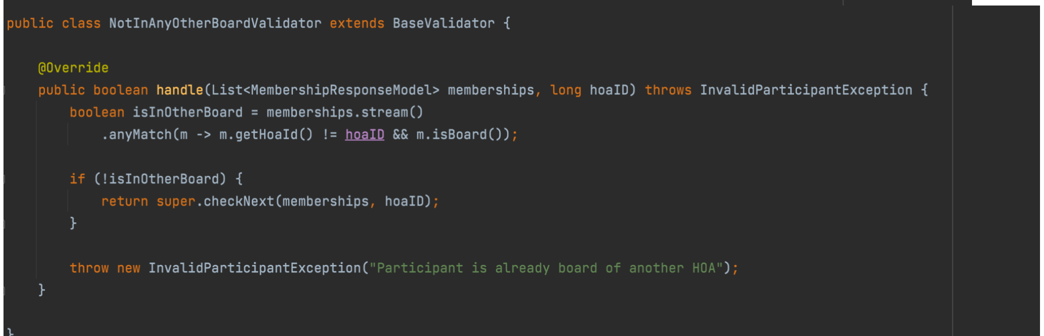
(***hoa/src/main/java/hoa/domain/electionchecks/NotBoardForTooLongValidator.java***):
(***hoa/src/main/java/hoa/domain/electionchecks/NotInAnyOtherBoardValidator.java***):
(***hoa/src/main/java/hoa/domain/electionchecks/TimeInCurrentHoaValidator.java***):



```java
public class TimeInCurrentHoaValidator extends BaseValidator {

    transient long yearsRequiredInHoa = TimeUtils.yearsToSeconds( y: 3);

    @Override
    public boolean handle(List<MembershipResponseModel> memberships, long hoaID) throws InvalidParticipantException {
        LocalDateTime curHoaTime = memberships.stream() Stream<MembershipResponseModel>
            .filter(m -> m.getHoaId() == hoaID)
            .map(m -> Arrays.asList(m.getStartTime(), m.getDuration())) Stream<List<LocalDateTime>>
            .map(l -> l.get(1) == null ? TimeUtils.absoluteDifference(l.get(0), LocalDateTime.now()) : l.get(1)) Stream<LocalDateTime>
            .reduce(TimeUtils.getFirstEpochDate(), TimeUtils::sum);

        if (TimeUtils.seconds(curHoaTime) >= yearsRequiredInHoa) {
            return super.checkNext(memberships, hoaID);
        }
        throw new InvalidParticipantException("Participant hasn't been in the HOA long enough");
    }
}
```



```java
public class NotBoardForTooLongValidator extends BaseValidator {
    transient long yearsMaxInBoard = TimeUtils.yearsToSeconds( y: 10);

    @Override
    public boolean handle(List<MembershipResponseModel> memberships, long hoaID) throws InvalidParticipantException {
        LocalDateTime curHoaTime = memberships.stream() Stream<MembershipResponseModel>
            .filter(m -> m.getHoaId() == hoaID && m.isBoard())
            .map(m -> Arrays.asList(m.getStartTime(), m.getDuration())) Stream<List<LocalDateTime>>
            .map(l -> l.get(1) == null ? TimeUtils.absoluteDifference(l.get(0), LocalDateTime.now()) : l.get(1)) Stream<LocalDateTime>
            .reduce(TimeUtils.getFirstEpochDate(), TimeUtils::sum);

        if (TimeUtils.seconds(curHoaTime) <= yearsMaxInBoard) {
            return super.checkNext(memberships, hoaID);
        }
        throw new InvalidParticipantException("Participant has been a board member for too long");
    }
}
```

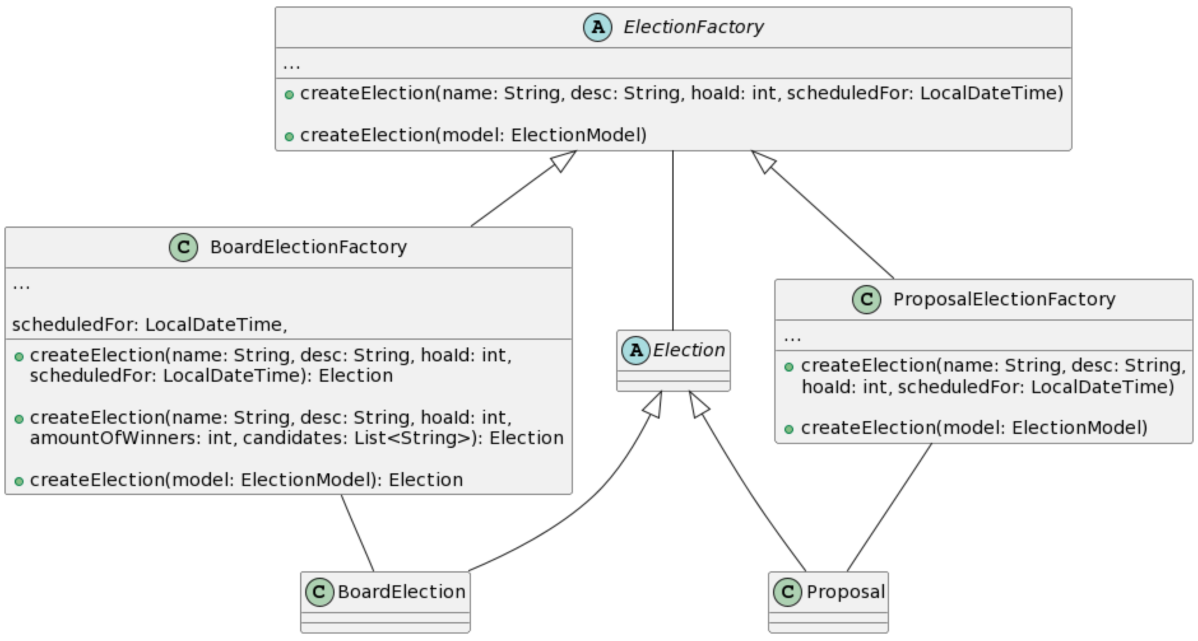


```java
public class NotInAnyOtherBoardValidator extends BaseValidator {

    @Override
    public boolean handle(List<MembershipResponseModel> memberships, long hoaID) throws InvalidParticipantException {
        boolean isInOtherBoard = memberships.stream()
            .anyMatch(m -> m.getHoaId() != hoaID && m.isBoard());

        if (!isInOtherBoard) {
            return super.checkNext(memberships, hoaID);
        }

        throw new InvalidParticipantException("Participant is already board of another HOA");
    }

}
```

II. Factory

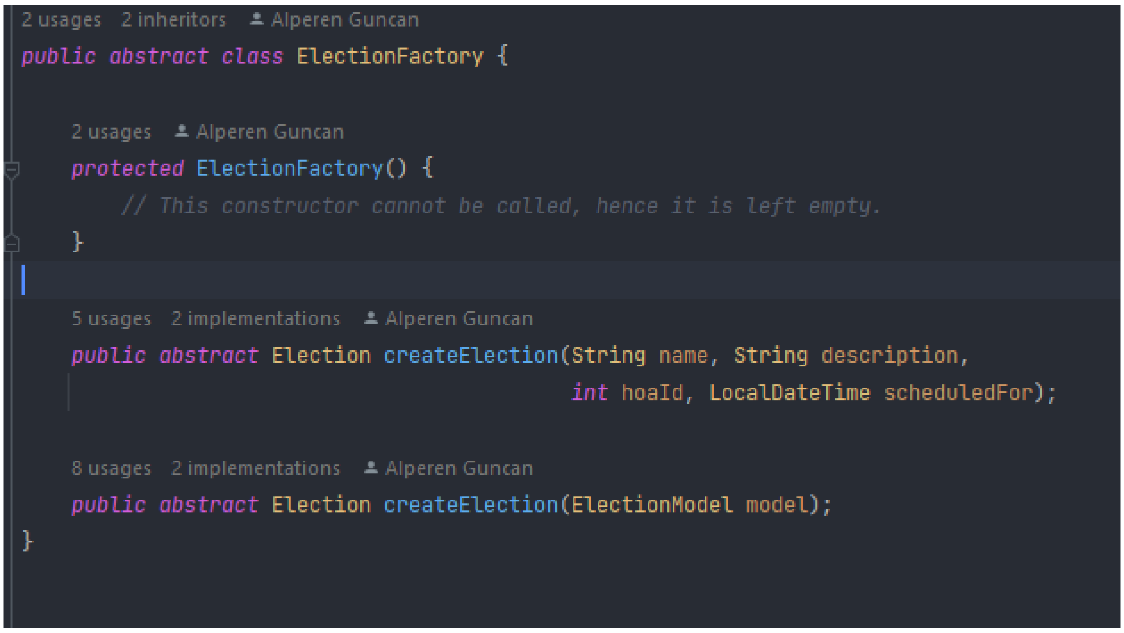
We chose to implement this creational design pattern because we have two types of elections, one where we choose board members and one where we can change the set of current requirements. The Factory Method provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The main reason a factory is preferred here is for future-proofing the election generation procedure - this way one can easily add new types on top of the original two. It also nicely combines some of the functionality that some elections share, like basic attributes and operations.

Factory is chosen over other creational patterns for quite simple reasons: *Abstract Factory* offers a higher level of abstraction that is deemed unnecessary for the current design. If elections differed even further (in several other characteristics other than vote choices), an abstract factory might have been useful, but that is not anticipated by the developers, nor required by the client. The proposals and board elections serve as "types" of elections, differing enough by design and construction, and being quite simple enough, that a complex *Builder* to build up the election class as different "representations" of the same object was also not considered appropriate. Hence why we stuck with a regular factory design pattern.



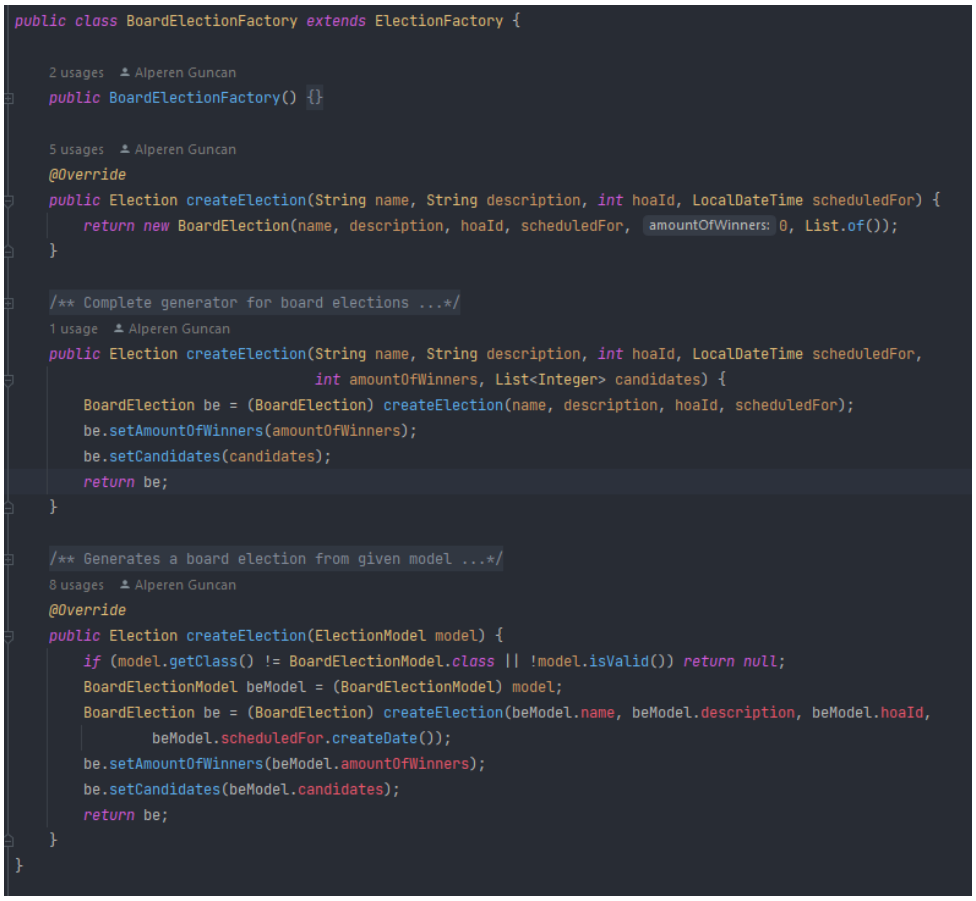
ElectionFactory
...
createElection(name: String, desc: String, hoaId: int, scheduledFor: LocalDateTime)
createElection(model: ElectionModel)
BoardElectionFactory
...
scheduledFor: LocalDateTime,
createElection(name: String, desc: String, hoaId: int, scheduledFor: LocalDateTime): Election
createElection(name: String, desc: String, hoaId: int, amountOfWinners: int, candidates: List<String>): Election
createElection(model: ElectionModel): Election
Election
ProposalElectionFactory
...
createElection(name: String, desc: String, hoaId: int, scheduledFor: LocalDateTime)
createElection(model: ElectionModel)
BoardElection
Proposal

CREATOR
(voting-microservice/src/main/java/voting/domain/factories/ElectionFactory.java):

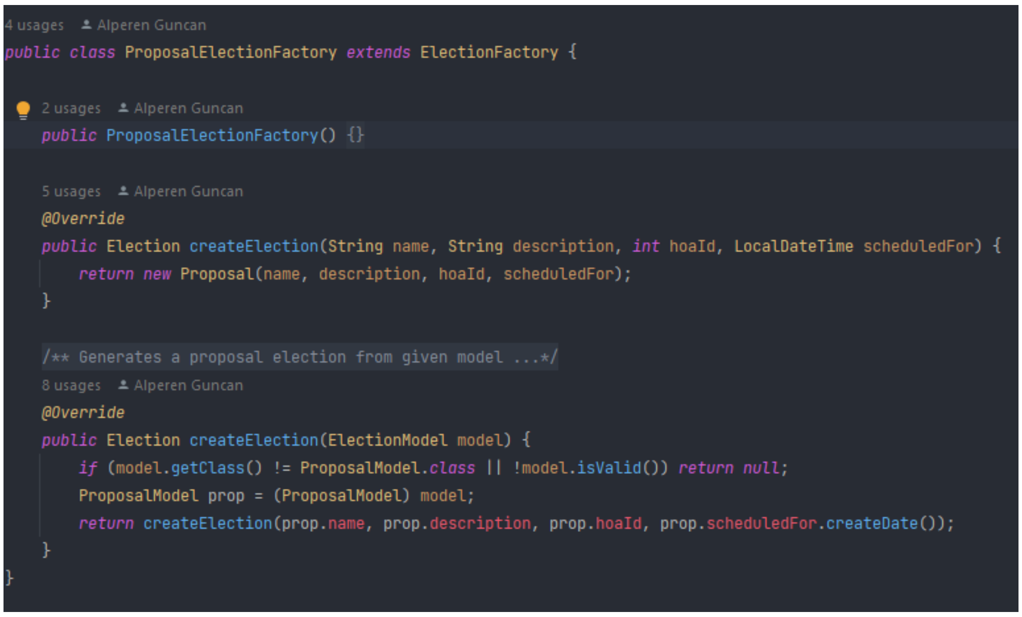


```java
2 usages  2 inheritors  Alperen Guncan
public abstract class ElectionFactory {

    2 usages  Alperen Guncan
    protected ElectionFactory() {
        // This constructor cannot be called, hence it is left empty.
    }

    5 usages  2 implementations  Alperen Guncan
    public abstract Election createElection(String name, String description,
                                            int hoaId, LocalDateTime scheduledFor);

    8 usages  2 implementations  Alperen Guncan
    public abstract Election createElection(ElectionModel model);
}
```

CONCRETE CREATORS
(voting-microservice/src/main/java/voting/domain/factories/BoardElectionFactory.java)
(voting-microservice/src/main/java/voting/domain/factories/ProposalElectionFactory.java):



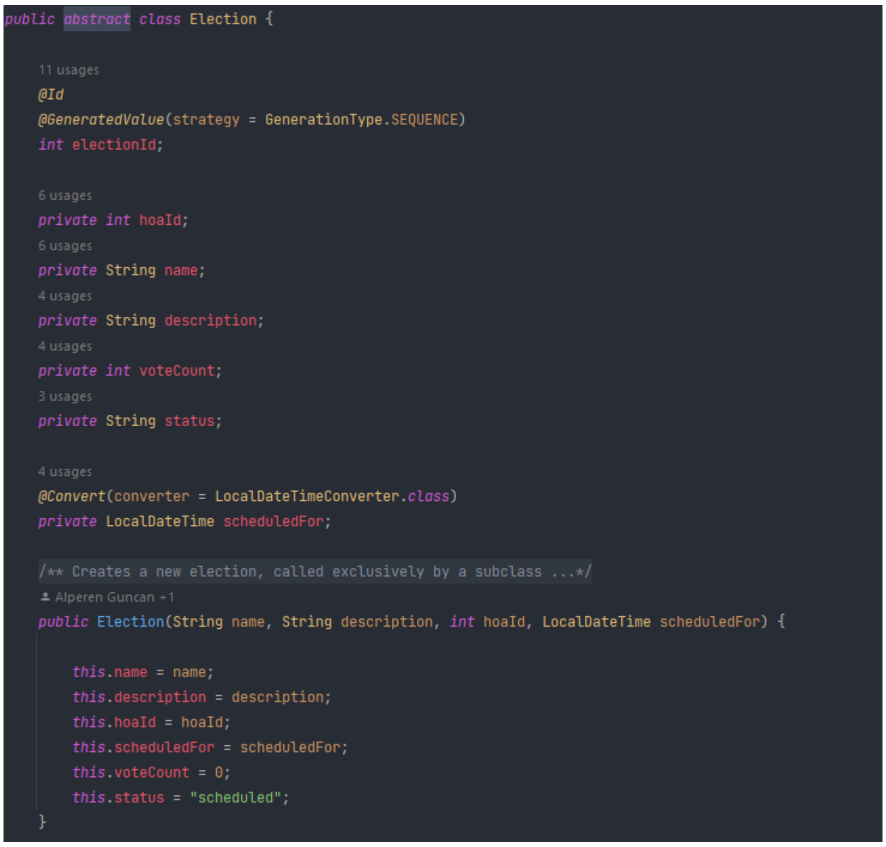
```java
public class BoardElectionFactory extends ElectionFactory {

    2 usages  Alperen Guncan
    public BoardElectionFactory() {}

    5 usages  Alperen Guncan
    @Override
    public Election createElection(String name, String description, int hoaId, LocalDateTime scheduledFor) {
        return new BoardElection(name, description, hoaId, scheduledFor,  amountOfWinners: 0, List.of());
    }

    /** Complete generator for board elections ...*/
    1 usage  Alperen Guncan
    public Election createElection(String name, String description, int hoaId, LocalDateTime scheduledFor,
                                   int amountOfWinners, List<Integer> candidates) {
        BoardElection be = (BoardElection) createElection(name, description, hoaId, scheduledFor);
        be.setAmountOfWinners(amountOfWinners);
        be.setCandidates(candidates);
        return be;
    }

    /** Generates a board election from given model ...*/
    8 usages  Alperen Guncan
    @Override
    public Election createElection(ElectionModel model) {
        if (model.getClass() != BoardElectionModel.class || !model.isValid()) return null;
        BoardElectionModel beModel = (BoardElectionModel) model;
        BoardElection be = (BoardElection) createElection(beModel.name, beModel.description, beModel.hoaId,
                beModel.scheduledFor.createDate());
        be.setAmountOfWinners(beModel.amountOfWinners);
        be.setCandidates(beModel.candidates);
        return be;
    }
}
```



```
4 usages   Alperen Guncan
public class ProposalElectionFactory extends ElectionFactory {

    2 usages   Alperen Guncan
    public ProposalElectionFactory() {}

    5 usages   Alperen Guncan
    @Override
    public Election createElection(String name, String description, int hoaId, LocalDateTime scheduledFor) {
        return new Proposal(name, description, hoaId, scheduledFor);
    }

    /** Generates a proposal election from given model ...*/
    8 usages   Alperen Guncan
    @Override
    public Election createElection(ElectionModel model) {
        if (model.getClass() != ProposalModel.class || !model.isValid()) return null;
        ProposalModel prop = (ProposalModel) model;
        return createElection(prop.name, prop.description, prop.hoaId, prop.scheduledFor.createDate());
    }
}
```

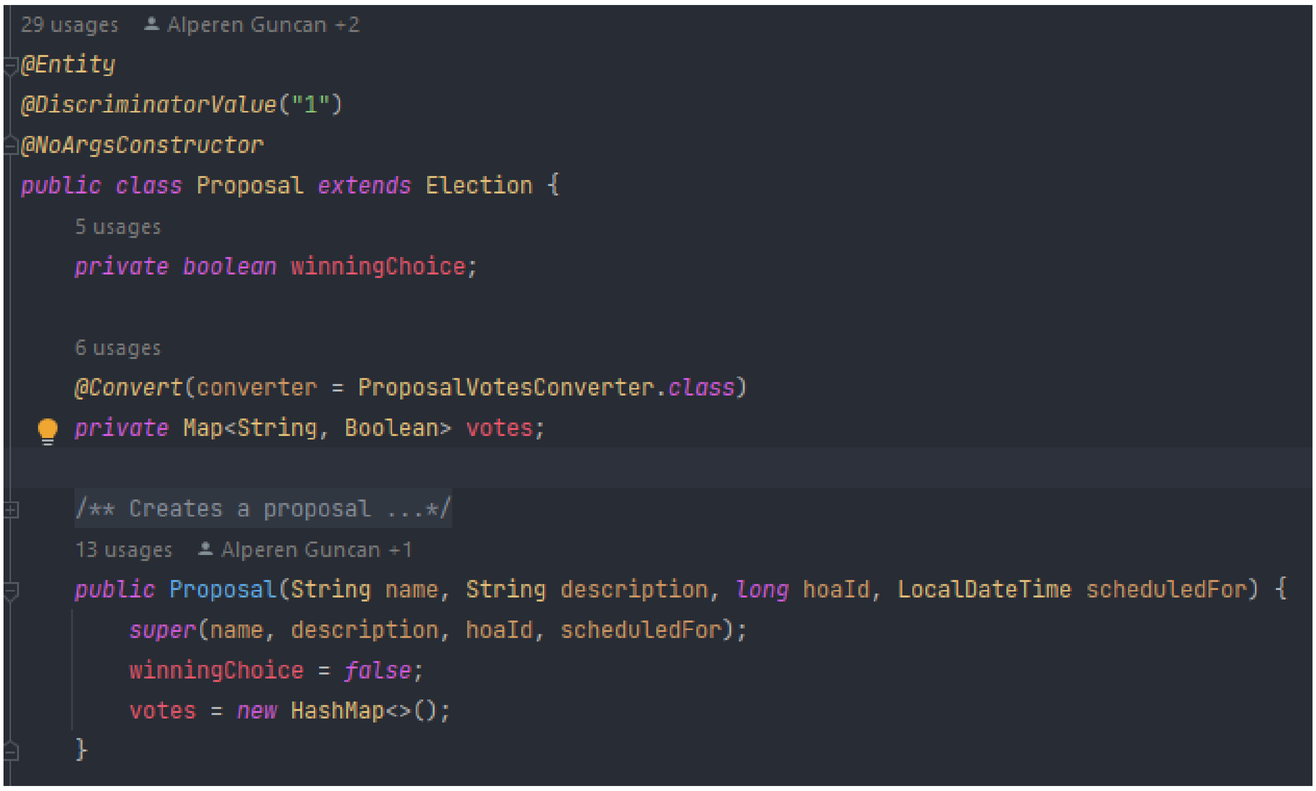
PRODUCT (voting-microservice/src/main/java/voting/domain/Election.java):

Design Choice to mention: Here an abstract class is used over the traditional interface for a general product. The main reasons include the fact that some methods (mostly the boilerplates, like equals and hashCode) are shared between the different election subclasses, the default JPA CRUD repositories have a tough time storing interfaces as entity entries, and the features we lose on are deemed as redundant (Elections will not extend anything else, so inheritance will not be a limitation, and the general product cannot be instantiated like an interface, so handling will be just as fine and the purpose of a factory is not lost).

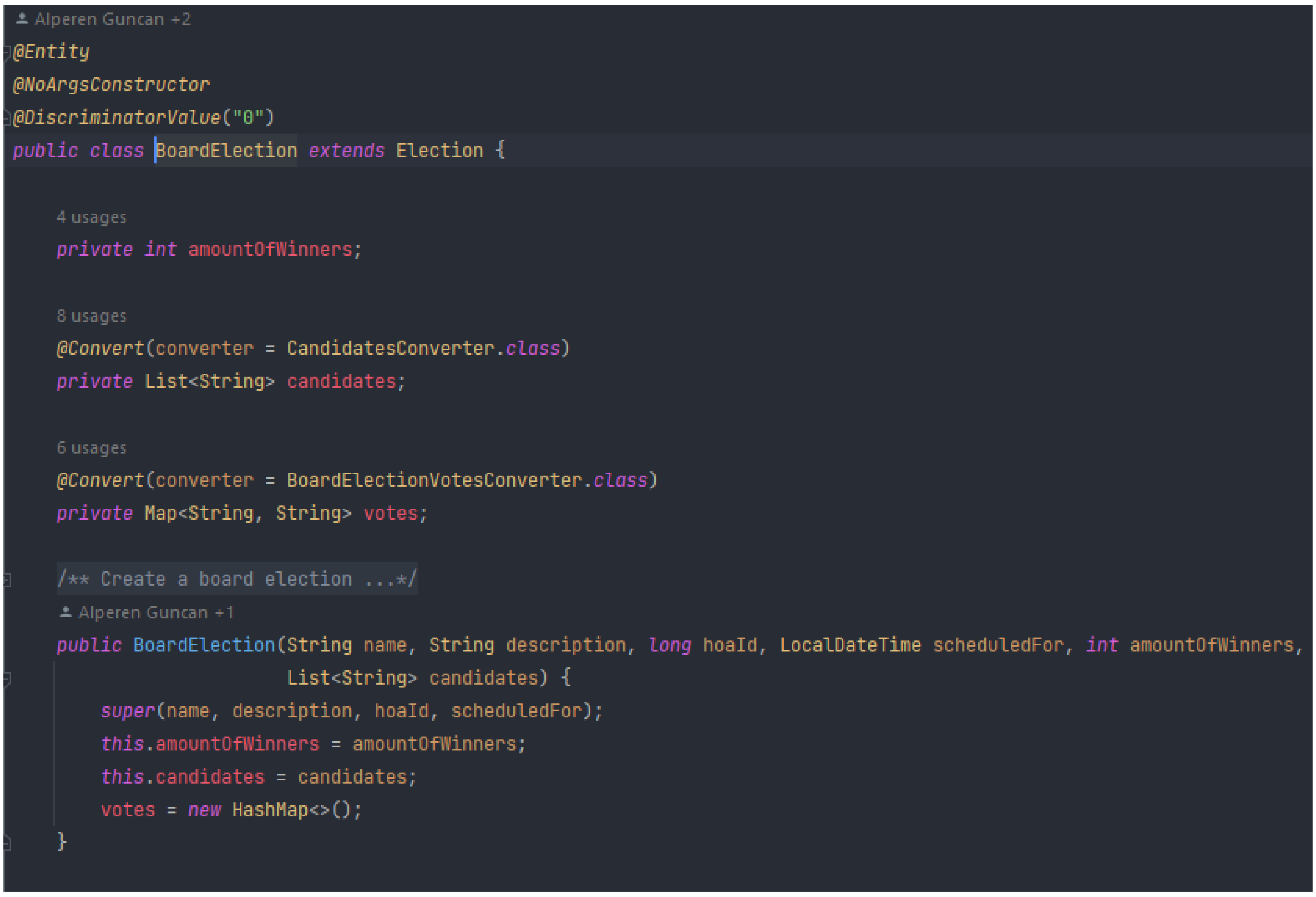


```
public abstract class Election {

    11 usages
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    int electionId;

    6 usages
    private int hoaId;
    6 usages
    private String name;
    4 usages
    private String description;
    4 usages
    private int voteCount;
    3 usages
    private String status;

    4 usages
    @Convert(converter = LocalDateTimeConverter.class)
    private LocalDateTime scheduledFor;

    /** Creates a new election, called exclusively by a subclass ...*/
    Alperen Guncan +1
    public Election(String name, String description, int hoaId, LocalDateTime scheduledFor) {

        this.name = name;
        this.description = description;
        this.hoaId = hoaId;
        this.scheduledFor = scheduledFor;
        this.voteCount = 0;
        this.status = "scheduled";
    }
```

CONCRETE PRODUCTS
(voting-microservice/src/main/java/voting/domain/Proposal.java)
(voting-microservice/src/main/java/voting/domain/BoardElection.java):

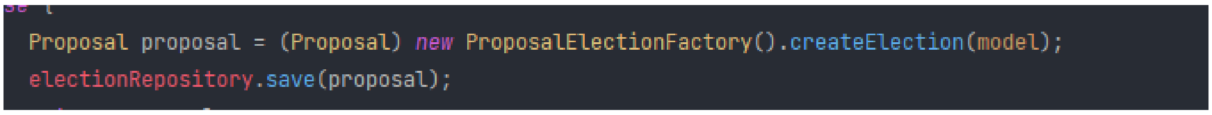


```java
29 usages  Alperen Guncan +2
@Entity
@DiscriminatorValue("1")
@NoArgsConstructor
public class Proposal extends Election {
    5 usages
    private boolean winningChoice;

    6 usages
    @Convert(converter = ProposalVotesConverter.class)
    private Map<String, Boolean> votes;

    /** Creates a proposal ...*/
    13 usages  Alperen Guncan +1
    public Proposal(String name, String description, long hoaId, LocalDateTime scheduledFor) {
        super(name, description, hoaId, scheduledFor);
        winningChoice = false;
        votes = new HashMap<>();
    }
```

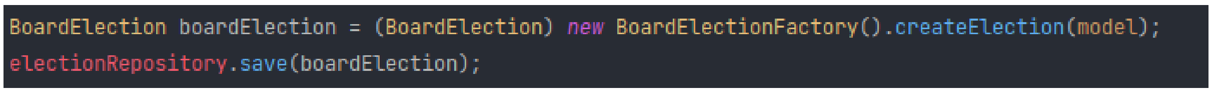


```java
Alperen Guncan +2
@Entity
@NoArgsConstructor
@DiscriminatorValue("0")
public class BoardElection extends Election {

    4 usages
    private int amountOfWinners;

    8 usages
    @Convert(converter = CandidatesConverter.class)
    private List<String> candidates;

    6 usages
    @Convert(converter = BoardElectionVotesConverter.class)
    private Map<String, String> votes;

    /** Create a board election ...*/
    Alperen Guncan +1
    public BoardElection(String name, String description, long hoaId, LocalDateTime scheduledFor, int amountOfWinners,
                         List<String> candidates) {
        super(name, description, hoaId, scheduledFor);
        this.amountOfWinners = amountOfWinners;
        this.candidates = candidates;
        votes = new HashMap<>();
    }
```

USAGE (voting-microservice/src/main/java/voting/services/ElectionService.java):



```java
  Proposal proposal = (Proposal) new ProposalElectionFactory().createElection(model);
  electionRepository.save(proposal);
```



```java
BoardElection boardElection = (BoardElection) new BoardElectionFactory().createElection(model);
electionRepository.save(boardElection);
```