SEM Assignment 2 - Refactoring

Group 18b

January 18, 2023

1 Software Metrics Used and Discoveries

The goal of the team with the assignment is a more understandable and sustainable code base. To this goal, all microservices and their composing parts were examined in an attempt to find code not adhering to standards. Tools used in the assistance of discovering such code were the 'MetricsTree' plugin for IntelliJ and the command-line tool `ck` (Aniche, 2015), which gave feedback on various code metrics. Along with them, the team tried to employ as many of the metrics discussed in the 'Software Engineering Methods' course at TU Delft. The metrics' numbers were given whenever applicable. Examining the prototype codebase, the team was pleased at the lack of serious flaws in their system, which is why most of the refactors did not turn out to be expensive. The most common issues encountered were classes that could be split into multiple parts , lack of cohesion in some methods, cyclomatic complexity and poor grouping of parameters. How they were identified and fixed on both a class- and method-level is discussed further in the report with code snippet examples (the code snippets do not show every refactoring step and are intended to illustrate the difference in the previous and current state of the given excerpts). The full overview of all of the code metrics has been exported on a class and method level into CSV files for before and after the refactoring. Those files are stored in the GitLab repository under `docs/metrics/` and are preceded with `before-refactor-` and `after-refactor-` accordingly.

2 Class-Level Refactorings

2.1 Extracting Classes - Requirements Microservice - `RequirementsController` - Extract Report Functionalities into its own controller

Inside of the `RequirementsController` class, there existed the functionality to send reports, send notifications about reports and getting a list of all reports for a specific home owner association. This was on top of the already existing functionality in the controller that managed all of the requirements between microservices. Effectively, the `RequirementsController` was turned

into a blob class, both too long and coupling two separate functionalities together. This was solved by extracting the report functionalities into its own new `ReportController` class. The metric used for evaluating this class was **LCOM**, which had a value of 2 before the refactor. The `RequirementsController` and `ReportController` both now have an LCOM value of 1, a substantial improvement. Additionally, the **LOC** of `RequirementsController` decreased from 358 to 235, a substantial improvement making it easier to work with.

```java
@GetMapping("/getNotifications")
public ResponseEntity<NotificationModel> getNotification() {
    try {
        if (parseUserFromToken() != null) {
            List<Notification> notificationList = notificationService.getAll();
            List<Event> ret = new ArrayList<>();
            for (Notification n: notificationList) {
                n.getEvent().setId(n.getId());
                Map<String, Boolean> users = n.getUsers(); // map of users and a boolean for markedRead
                for (String user: users.keySet()) {
                    if (user.equals(parseUserFromToken().getUsername().toString()) && !users.get(user)) {
                        // check our user and markedRead
                        ret.add(n.getEvent());
                        break;
                    }
                }
            }
            NotificationModel fin = new NotificationModel(ret);
            return ResponseEntity.ok(fin);
        } else {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Invalid user");
        }
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }
}
```

(a) Before Refactoring Example Method

```java
@RestController
@RequestMapping("/report")
public class ReportController {
    private final transient ReportService reportService;
```

(b) After Refactoring: `ReportController` Signature

```java
@GetMapping("/getNotifications")
public ResponseEntity<NotificationModel> getNotification() {
    try {
        if (parseUserFromToken() == null) throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Invalid user");
        List<Notification> notificationList = notificationService.getAll();
        List<Event> ret = new ArrayList<>();
        for (Notification n: notificationList) {
            n.getEvent().setId(n.getId());
            Map<String, Boolean> users = n.getUsers(); // map of users and a boolean for markedRead
            for (String user: users.keySet()) {
                if (user.equals(parseUserFromToken().getUsername().toString()) && !users.get(user)) {
                    // check our user and markedRead
                    ret.add(n.getEvent());
                    break;
                }
            }
        }
        return ResponseEntity.ok(new NotificationModel(ret));
```

(c) After Refactoring: `ReportController` Example Method

2.2 Extracting Classes - Gateway Microservice - `GatewayController` - Split Functionalities into two classes

The `GatewayController` is the class where the users of the application will be sending all of their requests from and other microservices will send their requests to. As a result, the class grew to be quite large, with its **LOC** clocking it at the massive 373. The solution the team came up with is splitting some core functionality of the class into a separate service `SendRequestService`, containing one single method for sending requests instead of having all separate methods send their own requests. This will additionally increase the class's cohesion since it is not focusing on all network functionalities, and instead client and server separately (which can be entirely different at times). **LOC** goes down to 297. This type of splitting also improves the scalability of the class once more requests start getting made from the Gateway Microservice in the case the application is expanded.

```java
/**
 * Routing method used to retrieve activities for HOAs that the user is member of.
 *
 * @return The responseEntity passed back from the method in the HOA microservice.
 */
@GetMapping("/pnb/allActivities")
public ResponseEntity allActivities() {
    //Get bearer token
    String token = ((ServletRequestAttributes) RequestContextHolder.getRequestAttributes())
            .getRequest().getHeader(AUTHORIZATION_LITERAL);
    String username = getClaimFromToken(token.split( regex: " ")[1], Claims::getSubject);

    RestTemplate restTemplate = new RestTemplate();
    HttpEntity entity = buildEntity( body: null);
    String url = "http://localhost:8090/pnb/allActivitiesForUser/" + username;
    return restTemplate.exchange(url, HttpMethod.GET, entity, Object.class);
}
```

(a) Sample `GatewayController` Method Before Refactoring

```java
@Service
public class SendRequestService {

    private static final RestTemplate restTemplate = new RestTemplate();

    /** Builds a standardized HTTP entity to pass along with HTTP requests ...*/
    public ResponseEntity buildAndSend(String url, Object body, HttpMethod method) {
        //Extract Bearer Token from incoming request
        String token = ((ServletRequestAttributes) Objects.requireNonNull(RequestContextHolder.getRequestAttributes()))
                .getRequest().getHeader( name: "Authorization");

        //Remove the "bearer prefix" from the beginning of the token.
        token = token.split( regex: " ")[1];
        HttpHeaders headers = new HttpHeaders();
        headers.setBearerAuth(token);
        HttpEntity entity = new HttpEntity(body, headers);

        return restTemplate.exchange(url, method, entity, Object.class);
    }
}
```

(b) The `SendRequestService` class

```java
/**
 * Routing method used to retrieve activities for HOAs that the user is member of.
 *
 * @return The responseEntity passed back from the method in the HOA microservice.
 */
@GetMapping("/pnb/allActivities")
public ResponseEntity allActivities() {
    String token = ((ServletRequestAttributes) Objects.requireNonNull(RequestContextHolder.getRequestAttributes()))
            .getRequest().getHeader( name: "Authorization").split( regex: " ")[1];
    String username = getClaimFromToken(token, Claims::getSubject);
    String url = "http://localhost:8090/pnb/allActivitiesForUser/" + username;

    return sendRequestService.buildAndSend(url,  body: null, HttpMethod.GET);
}
```

(c) Sample `GatewayController` method after refactoring

2.3 Increasing Cohesion - Hoa Microservice - Removing Dependency on Gregorian Calendar

During the creation of the project, the Public Notice Board was created before the creation of a `commons` module. The creation of the aforementioned module was a move to increase the cohesion of classes and ease data transfer between microservices. However, before `DateModel` was implemented, the `GregorianCalendar` object was used within the Public Notice Board bounded context. It was used to keep track of posted activities. Examining the code, it was discovered that this dependency on an external library, while functional, was entirely unnecessary and decreased cohesion (as it had to be converted

to other objects). This refactor replaces all uses of `GregorianCalendar` with `DateModel`, resulting in far simpler methods that do not deal with conversions, and thus increasing the cohesion between all of the classes in the Public Notice Board bounded context.

```java
@Data
@NoArgsConstructor
public class DateModel {
    private int year;
    private int month;
    private int day;

    public DateModel(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}
```

(a) The `DateModel` Object

```java
public Activity createActivity(int hoaId, String name, DateModel dateModel, String description) throws Exception {

    int year = dateModel.getYear();
    int month = dateModel.getMonth();
    int day = dateModel.getDay();
    GregorianCalendar time = new GregorianCalendar(year, month, day);

    if (existsByNameAndTime(name, time)) throw new ActivityNameAlreadyInUseException(name);

    Hoa hoa = hoaService.getHoaById(hoaId);


    Activity activity = new Activity(hoa, name, time, description);
    activityRepository.save(activity);
    return activity;
}
```

(b) Sample PNB Method Before Refactoring

```java
public Activity createActivity(int hoaId, String name, DateModel time, String description) throws Exception {

    if (existsByNameAndTime(name, time)) throw new ActivityNameAlreadyInUseException(name);

    Hoa hoa = hoaService.getHoaById(hoaId);


    Activity activity = new Activity(hoa, name, time, description);
    activityRepository.save(activity);
    return activity;
}
```

(c) Sample PNB Method After Refactoring

2.4 Decreasing size of inheritance tree - HOA Microservice/Requirements Microservice - Removing unnecessary inheritance from `HasEvents`

At the start of the project, in the HOA and Requirements microservices, all of the entities were made to extend the template's `HasEvents` class. It was planned for most of our classes to have a list of events that could be released. Eventually, `HasEvents` was moved to the commons folder, but its parenthood over many classes stayed, particularly `User`, `HasAddress`, `Report` and `Requirements`. Some classes do still employ it (PNB and Gateway classes),

but it is not nearly as widespread as initially expected. The old implementation of `HasAddress` left with with a **NOC** (Number of Children) of 7, with it now being decreased to 2. Additionally, the depth of the inheritance tree is shortened by 1 for all of the classes that were refactored (thus decreasing their number of attributes and methods). This refactoring change should increase the cohesion between the aforementioned classes, reduce coupling and increase code readability.

```java
/**
 * A base class for adding domain event support to an entity.
 */
public abstract class HasEvents {
    private final transient List<Object> domainEvents = new ArrayList<>();

    protected void recordThat(Object event) { domainEvents.add(Objects.requireNonNull(event)); }

    @DomainEvents
    protected Collection<Object> releaseEvents() { return Collections.unmodifiableList(domainEvents); }

    @AfterDomainEventPublication
    protected void clearEvents() { this.domainEvents.clear(); }
}
```

(a) The `HasEvents` Class

```java
public class User extends HasEvents {
```

(b) The `User` Class Before Refactoring

```java
public class User {
```

(c) The `User` Class After Refactoring

2.5 Increasing Cohesion + Decreasing Coupling - HOA Microservice/Requirements Microservice - Creating a Parameter Object for different services

As the largest microservice with the most business logic, the HOA has the most services and repositories. To save on the amount of logic being written, those services can help each other. However, for some services like the `VoteService`, it turned out to have as many as four services as parameters. It is not wise to give a single service access to so many repositories/services. Additionally, there was no coherent format on whether a service would contain other services or repositories. In the end, there was a mix between the two that did not make sense. Services should not have access to other services' repositories in the first place since that breaks data separation principles and is a sign of the Feature Envy code smell. To remedy this, a container object (initialized with all null values and several constructors) `ServiceParameterClass` was created, which would be then filled with the necessary services on startup. This increases the cohesion of all of the services (since they do not need to be explicitly concerned with other services) and decreases the coupling between the different classes. Since all of the previous accessing is done through a single

object, this should increase code readability and maintainability in case more service classes are added in the future.

```java
@Service
public class VoteService {

    private transient ResultsRepository resultsRepository;
    private transient HoaRepository hoaRepository;
    private transient UserRepository userRepository;
    private transient BoardMemberRepository boardMemberRepository;

    public VoteService(ResultsRepository resultsRepository, HoaRepository hoaRepository,
                       UserRepository userRepository, BoardMemberRepository boardMemberRepository) {
        this.resultsRepository = resultsRepository;
        this.hoaRepository = hoaRepository;
        this.userRepository = userRepository;
        this.boardMemberRepository = boardMemberRepository;
    }
```

(a) `VoteService` Initialization Before Refactring

```java
@Service
public class HoaService {

    private transient HoaRepository hoaRepository;

    private transient VoteService voteService;

    @Autowired
    public HoaService(HoaRepository hoaRepository, VoteService voteService) {
        this.hoaRepository = hoaRepository;
        this.voteService = voteService;
    }
```

(b) Showcase of the declaration inconsistency (`VoteService` is initialized with repositories and `HoaService` with services)

```java
@Getter
public class ServiceParameterClass {
    private transient HoaService hoaService = null;
    private transient UserService userService = null;
    private transient BoardMemberService boardMemberService = null;
    private transient ConnectionService connectionService = null;

    public ServiceParameterClass() {}

    public ServiceParameterClass(HoaService hoaService) { this.hoaService = hoaService; }

    public ServiceParameterClass(HoaService hoaService, UserService userService, BoardMemberService boardMemberService) {
        this.hoaService = hoaService;
        this.userService = userService;
        this.boardMemberService = boardMemberService;
    }

    public ServiceParameterClass(HoaService hoaService, ConnectionService connectionService,
            BoardMemberService boardMemberService) {
        this.hoaService = hoaService;
        this.connectionService = connectionService;
        this.boardMemberService = boardMemberService;
    }

}
```

(c) The new `ServiceParameterClass` class

```java
@Service
public class VoteService {

    private transient ResultsRepository resultsRepository;

    private transient ServiceParameterClass services;

    public VoteService(ResultsRepository resultsRepository, HoaService hoaService,
                       UserService userService, BoardMemberService boardMemberService) {
        this.resultsRepository = resultsRepository;
        this.services = new ServiceParameterClass(hoaService, userService, boardMemberService);
    }
```

(d) `VoteService` Initialization After Refactoring

3 Method-Level Refactorings

3.1 Cyclomatic Complexity - Gateway - `NotificationController.getNotification(...)`

There was a refactor to remove the unnecessary nesting in the notifications controller class. The group managed to reduce the method **CC** (Cyclomatic Complexity) from 3 to 2. Before, the try-catch block in the method combined with the final conditional created a third independent path through the method. This is done by having the initial conditional statement to filter out non-null values and moving the return inside of the try-block.

```java
@GetMapping("/getNotifications")
public ResponseEntity<NotificationModel> getNotification() {
    try {
        if (parseUserFromToken() != null) {
            List<Notification> notificationList = notificationService.getAll();
            List<Event> ret = new ArrayList<>();
            for (Notification n: notificationList) {
                n.getEvent().setId(n.getId());
                Map<String, Boolean> users = n.getUsers(); // map of users and a boolean for markedRead
                for (String user: users.keySet()) {
                    if (user.equals(parseUserFromToken().getUsername().toString()) && !users.get(user)) {
                        // check our user and markedRead
                        ret.add(n.getEvent());
                        break;
                    }
                }
            }
            NotificationModel fin = new NotificationModel(ret);
            return ResponseEntity.ok(fin);
        } else {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Invalid user");
        }
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }
}
```

(a) Before Refactoring

```java
@GetMapping("/getNotifications")
public ResponseEntity<NotificationModel> getNotification() {
    try {
        if (parseUserFromToken() == null) throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Invalid user");
        List<Notification> notificationList = notificationService.getAll();
        List<Event> ret = new ArrayList<>();
        for (Notification n: notificationList) {
            n.getEvent().setId(n.getId());
            Map<String, Boolean> users = n.getUsers(); // map of users and a boolean for markedRead
            for (String user: users.keySet()) {
                if (user.equals(parseUserFromToken().getUsername().toString()) && !users.get(user)) {
                    // check our user and markedRead
                    ret.add(n.getEvent());
                    break;
                }
            }
        }
        return ResponseEntity.ok(new NotificationModel(ret));
```

(b) After Refactoring

3.2 Lines of Code - HOA Microservice - `PnbController`

A lot of the methods in the class threw unnecessary exceptions, and contained both excessive whitespace and redundant checks for things done by the PnB service. As a result, it was deemed necessary to refactor it, thus reducing the **LOC** (but not the **eLOC**) and increasing the readability of the class. Total **LOC** has been reduced by 21 (137 → 126). A total of 4 methods have been refactored that way.

```java
/**
 * Responds with a list of all activities belonging to the given HOA.
 *
 * @param hoaId the ID of the HOA
 * @return a response entity containing the list of relevant activities
 * @throws Exception
 */
@GetMapping("/activitiesForHoa/{hoaId}")
public ResponseEntity<List<ActivityModel>> getActivitiesForHoa(@PathVariable int hoaId) throws Exception {

    try {
        return ResponseEntity.ok(
                activitiesToModels(
                        activityService.getActivitiesByHoaId(hoaId)
                )
        );
    } catch (Exception e) {
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }

}
```

(a) Sample Method Before Refactoring

```java
/**
 * Responds with a list of all activities belonging to the given HOA.
 *
 * @param hoaId the ID of the HOA
 * @return a response entity containing the list of relevant activities
 */
@GetMapping("/activitiesForHoa/{hoaId}")
public ResponseEntity<List<ActivityModel>> getActivitiesForHoa(@PathVariable int hoaId) {
    try {
        return ResponseEntity.ok(activitiesToModels(activityService.getActivitiesByHoaId(hoaId)));
    } catch (Exception e) {
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }
}
```

(b) Sample Method After Refactoring

3.3 Lines of Code/Code Complexity/Code Readibility - Hoa Microservice - `UserController.joinHoa()` - Usage of Lambda Statements to Replace Conditional Statements

In the creation of the `UserController` class, there occurred a very long and redundant conditional statement that could be replaced with a simple lambda expression. This both reduced the lines of code and makes it clearer to the person reading the code that it is a simple pre-condition for sanitizing input. As is visible from by the code snippet, this is another **LOC** reduction refactor. Lambda simplifications of the same level were done in `HoaController`

with the `getAllHoas()` method and in `UserController` once again with the `getAllUsers()` method. The **CC** (Cyclomatic Complexity) of this method specifically was reduced from 6 to 4, and the RFC dropped from 8 to 6. It is worth mentioning that these lambda statements are different from the ones refactored in `ReportController` later in the report since they are not implementing service functionality and are instead sanitizing the input data in the actual request body.

```java
@PostMapping("/joinHoa")
public ResponseEntity<FullUserHoaModel> joinHoa(@RequestBody JoinRequestModel joinRequest)
        throws HoaDoesNotExistException, UserDoesNotExistException {

    if (joinRequest.anyNull()) throw new UserDoesNotExistException("User does not exist");

    if (joinRequest.getUserDisplayName() == null || joinRequest.getHoaName() == null
            || joinRequest.getCountry() == null || joinRequest.getCity() == null || joinRequest.getStreet() == null
            || joinRequest.getHoaName() == null || joinRequest.getPostalCode() == null) {
        return ResponseEntity.badRequest().build();
    }

    FullAddressModel address = new FullAddressModel(
            joinRequest.getCountry(), joinRequest.getCity(),
            joinRequest.getStreet(), joinRequest.getHouseNumber(),
            joinRequest.getPostalCode()
    );

    UserHoa connection = this.userService.joinAssociation(
            joinRequest.getHoaName(), joinRequest.getUserDisplayName(), address
    );

    return ResponseEntity.ok().body(connection.toFullModel());
}
```

(a) `joinHoa()` Before Refactoring

```java
@PostMapping("/joinHoa")
public ResponseEntity<FullUserHoaModel> joinHoa(@RequestBody JoinRequestModel joinRequest)
        throws HoaDoesNotExistException, UserDoesNotExistException {

    if (joinRequest.anyNull()) throw new UserDoesNotExistException("User does not exist");

    if (Stream.of(joinRequest.getClass().getDeclaredFields()).anyMatch(Objects::isNull)){
        return ResponseEntity.badRequest().build();
    }

    FullAddressModel address = new FullAddressModel(
            joinRequest.getCountry(), joinRequest.getCity(),
            joinRequest.getStreet(), joinRequest.getHouseNumber(),
            joinRequest.getPostalCode()
    );

    UserHoa connection = this.userService.joinAssociation(
            joinRequest.getHoaName(), joinRequest.getUserDisplayName(), address
    );

    return ResponseEntity.ok().body(connection.toFullModel());
}
```

(b) `joinHoa()` After Refactoring

3.4 Cyclomatic complexity / Object-oriented abuser `Voting microservice - VoteController - castVote`

The method calls `Vote.castVote(...)`, which threw a `VotingException`, and the error message was used to define what behavior needs to be taken depending on the error message. Therefore, `VoteController.castVote(...)` contained a big if-else if statement that looked into the exception messages. What was done was to create several other exceptions (that extend `VotingException`), and

then enlarge the `try...catch` block to handle the different responses depending on the exception type. This reduced the **CC** (Cyclomatic Complexity) of the current method and removed an object-oriented abuser (i.e., the response to the exception is not determined with a `switch` statement or equivalent sequence of if statements, but rather the different instances of exceptions do so).

```java
@PostMapping("/vote/{hoaId}/castVote/{userName}")
public ResponseEntity<Void> castVote(@PathVariable int hoaId,
                                     @RequestBody int optionIndex,
                                    @PathVariable String userName) {

    if (!votingService.existingHoaVoting(hoaId)) {
        return ResponseEntity.notFound().build();
    }
    try {
        String netId = userName;
        votingService.castVote(hoaId, netId, optionIndex);
        return ResponseEntity.ok().build();
    } catch (VotingException e) {
        if (e.getMessage().equals("Voter is not eligible")) {
            return ResponseEntity.status(HttpStatus.FORBIDDEN).build();
        } else if (e.getMessage().equals("Chosen option index is invalid")
                || e.getMessage().equals("Vote is still ongoing")) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        }
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

(a) `castVote(...)` Before Refactoring

```java
@PostMapping("/vote/{hoaId}/castVote/{userName}")
public ResponseEntity<Void> castVote(@PathVariable int hoaId,
                                     @RequestBody int optionIndex,
                                    @PathVariable String userName) {

    if (!votingService.existingHoaVoting(hoaId)) {
        return ResponseEntity.notFound().build();
    }
    try {
        String netId = userName;
        votingService.castVote(hoaId, netId, optionIndex);
        return ResponseEntity.ok().build();
    } catch (IneligibleVoterException e) {
        return ResponseEntity.status(HttpStatus.FORBIDDEN).build();
    } catch (InvalidOptionException | VoteClosedException e) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
    }
}
```

(b) `castVote(...)` After Refactoring

3.5 Code repetition `Voting microservice - VotingService - registerVoteStartingNow(...)`

This is the method that gets called whenever an election is called. For that purpose it uses instances of classes that implement the `VoteBuilder` interface.

Though the constructed objects will be different, they still share parameters. Additionally, after each `Vote` object is instantiated, it has to be stored within the structure that manages the ongoing votes, and the code to be executed when the vote is over, has to be scheduled. It's clear that much of the functionality is common for the two possible vote types.

```
public void registerVoteStartingNow(VotingModel votingModel,
                                    TemporalAmount temporalAmount) {
    Vote vote;
    if (votingModel.getVotingType().equals(VotingType.ELECTIONS_VOTE)) {
        vote = new ElectionVoteBuilder()
                .forHoaWithId(votingModel.getHoaId())
                .withOptions(votingModel.getOptions())
                .startInstantlyWithDuration(temporalAmount)
                .withEligibleVoters(votingModel.getNumberOfEligibleVoters())
                .build();
        ongoingElections.put(votingModel.getHoaId(), vote);
        electionEndCalls.schedule(new VoteEndCallable(vote, votingService: this),
                vote.getTimeKeeper().getDurationInSeconds(),
                TimeUnit.SECONDS);
    } else if (votingModel.getVotingType().equals(VotingType.REQUIREMENTS_VOTE)) {
        vote = new RequirementVoteBuilder()
                .forHoaWithId(votingModel.getHoaId())
                .startInstantlyWithDuration(temporalAmount)
                .withEligibleVoters(votingModel.getNumberOfEligibleVoters())
                .build();
        ongoingElections.put(votingModel.getHoaId(), vote);
        electionEndCalls.schedule(new VoteEndCallable(vote, votingService: this),
                vote.getTimeKeeper().getDurationInSeconds(),
                TimeUnit.SECONDS);
    }

}
```

(a) `registerVoteStartingNow(...)` Before Refactoring

```
public void registerVoteStartingNow(VotingModel votingModel,
                                    TemporalAmount temporalAmount) {
    VoteBuilder voteBuilder;
    if (votingModel.getVotingType().equals(VotingType.ELECTIONS_VOTE)) {
        voteBuilder = new ElectionVoteBuilder()
                .withOptions(votingModel.getOptions());
    } else if (votingModel.getVotingType().equals(VotingType.REQUIREMENTS_VOTE)) {
        voteBuilder = new RequirementVoteBuilder();
    } else {
        throw new UnsupportedOperationException();
    }
    Vote vote = voteBuilder
            .forHoaWithId(votingModel.getHoaId())
            .startInstantlyWithDuration(temporalAmount)
            .withEligibleVoters(votingModel.getNumberOfEligibleVoters())
            .build();
    ongoingElections.put(votingModel.getHoaId(), vote);
    electionEndCalls.schedule(new VoteEndCallable(vote, votingService: this),
            vote.getTimeKeeper().getDurationInSeconds(),
            TimeUnit.SECONDS);
}
```

(b) `registerVoteStartingNow(...)` After Refactoring

3.6 Method Extraction - Requirements Microservice - `ReportController` and `RequirementController.getRequirements()` and `getReports()`

It was discovered that the two controller methods were using lambda filtering statements inside of themselves to certain lists of information. This violated the separation between controllers and services. The refactor cleans up the two methods' code and delegates that implementation to deliberate methods in their respective services.

```
@GetMapping("/getRequirements/{hoaId}")
public ResponseEntity<RequirementsResponseModel> getRequirements(@PathVariable("hoaId") int hoaId)
        throws Exception {
    try {
        if (hoaId != -1) {
            if (Util.hoaExists(hoaId)) {
                List<Requirements> requirementsList = requirementsService.getAll()
                        .stream().filter(o -> o.getHoaId() == hoaId)
                        .collect(Collectors.toList());
                return ResponseEntity.ok(new RequirementsResponseModel(requirementsList));
            } else {
                throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
            }
        } else {
            List<Requirements> requirementsList = requirementsService.getAll()
                    .stream().filter(o -> o.getHoaId() == hoaId)
                    .collect(Collectors.toList());
            return ResponseEntity.ok(new RequirementsResponseModel(requirementsList));
        }
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }
}
```

(a) `getRequirements()` Before Refactoring

```
public List<Requirements> getRequirementsByHoa(int hoaId){
    return getAll()
            .stream().filter(o -> o.getHoaId() == hoaId)
            .collect(Collectors.toList());
}
```

(b) Service Method Converting the Lambda

```
@GetMapping("/getRequirements/{hoaId}")
public ResponseEntity<RequirementsResponseModel> getRequirements(@PathVariable("hoaId") int hoaId) {
    try {
        if (hoaId != -1) {
            if (Util.hoaExists(hoaId)) {
                return ResponseEntity.ok(new RequirementsResponseModel(requirementsService.getRequirementsByHoa(hoaId)));
            } else {
                throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
            }
        }
        return ResponseEntity.ok(new RequirementsResponseModel(requirementsService.getRequirementsByHoa(hoaId)));

    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
    }
}
```

(c) `getRequirements()` Method After Refactoring

References

Aniche, M. (2015). *Java code metrics calculator (ck)* [Available in https://github.com/mauricioaniche/ck/].