

# Assignment 1 Part 2 - Scenario

## Homeowner's Association

Student name	Student number
Rafael Petouris	4776968
Jelt Jongsma	5496594
Vladimir Petkov	5447194
Bram Snelten	5519365
Alex Brown	5171709
Roland Bockholt	5534283



*Figure 1: Image of a house model titled Homeowner's Association [1].*

CSE2115

Group 19a

Delft University of Technology, The Netherlands

23 December 2022

# Contents

<b>1</b>	<b>Design Patterns</b>	<b>1</b>
1.1	Decorator pattern   Votes . . . . .	1
1.1.1	Description . . . . .	1
1.1.2	Class diagram . . . . .	2
1.1.3	Implementations . . . . .	2
1.2	Builder pattern   Activity . . . . .	7
1.2.1	Description . . . . .	7
1.2.2	Class diagram . . . . .	7
1.2.3	Implementations . . . . .	8
	<b>Bibliography</b>	<b>15</b>

# 1. Design Patterns

## 1.1 Decorator pattern | Votes

### 1.1.1 Description

Problem: we have 2 types of votes. The type of vote should change depending on the context. If we were to add new combinable types of votes, we would have to make separate classes for each possible permutation of votetype. This is a very bad solution because:

1. This introduces code clones.
2. Adding another feature means implementing a lot more classes for each new combination.
3. If one feature is faulty, all related combinations are also faulty.

This implementation clearly violates the open-close principle, therefore we decided to implement Decorator pattern instead.

The Decorator Pattern attaches additional responsibilities to a vote dynamically. A vote is an election vote or a proposal vote. These responsibilities can now be added without affecting other objects. This flexible form of subclassing using a Decorator fits our problem well. Consequently making sure we can scale future types of vote responsibly. Although I must mention a downside of using a Decorator in this context; large quantities of little vote objects can be created if overused, which adds up to a lot of memory being allocated for stack frames, instead of memory being allocated for useful object data.

Taking a look at figure 1.1 will help with the following structure explanation.

We have a `TypelessVote` 1.4 class which contains only an `Id` of the user that cast the vote. `TypelessVote` implements the `Vote` 1.5 interface. The abstract class `specialVote` 1.6 implements and is an aggregation of the `Vote` interface. A `specialVote` contains a `Vote`, but a `Vote` can exist without a `specialVote`. `SpecialVote` class contains the `Vote` attribute which will be an instance of a `TypelessVote`. Now we will discuss the two existing concrete decorators: `ElectionVote` 1.7 and `ProposalVote` 1.8. They both extend the `specialVote` abstract class. The `ElectionVote` class contains the `Id` of the election and the `Id` of the applicant who is being voted for. The `ProposalVote` class contains an `Id` (in this case a `ProposalPk`) to identify which rule proposal is being voted for. In addition the `ProposalVote` class contains a `Decision` 1.9, which is an enumeration of the three values: `ABSTAIN`, `ACCEPT` and `REJECT`. Wrapping these concrete Decorator classes around a `TypelessVote` instance will generate a useful vote object.

### 1.1.2 Class diagram

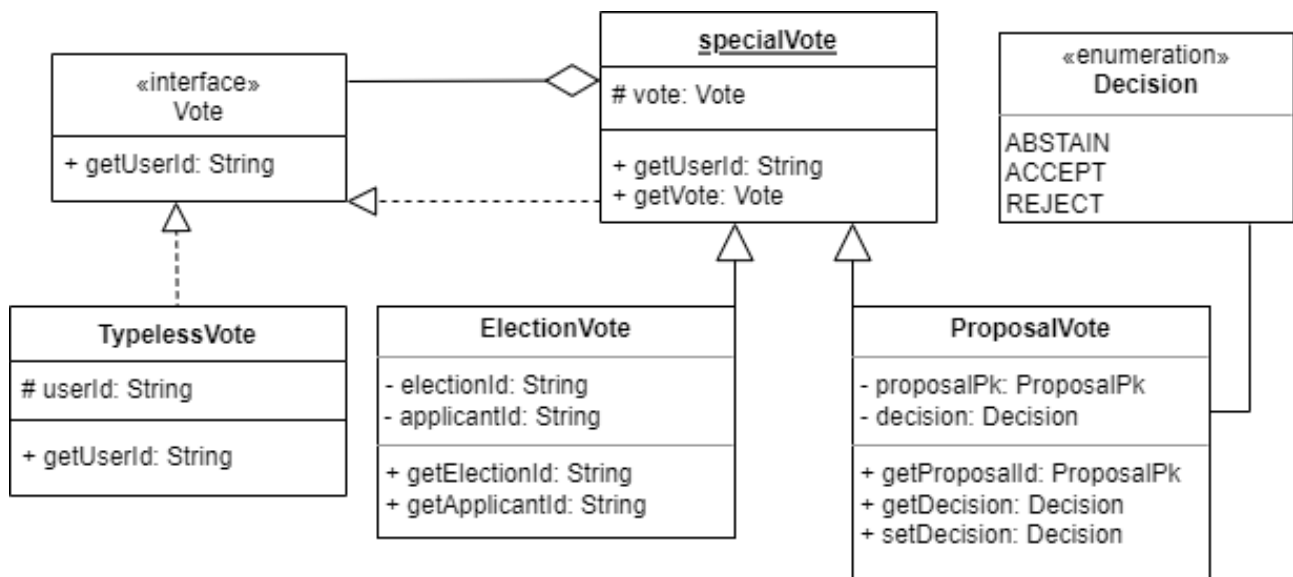


Figure 1.1: Class diagram for the votes system using a decorator pattern.

### 1.1.3 Implementations

I started by creating a new "vote" directory in the domain directory of the HOA microservice. So that the Decorator pattern can be easily distinguished from all the other classes.

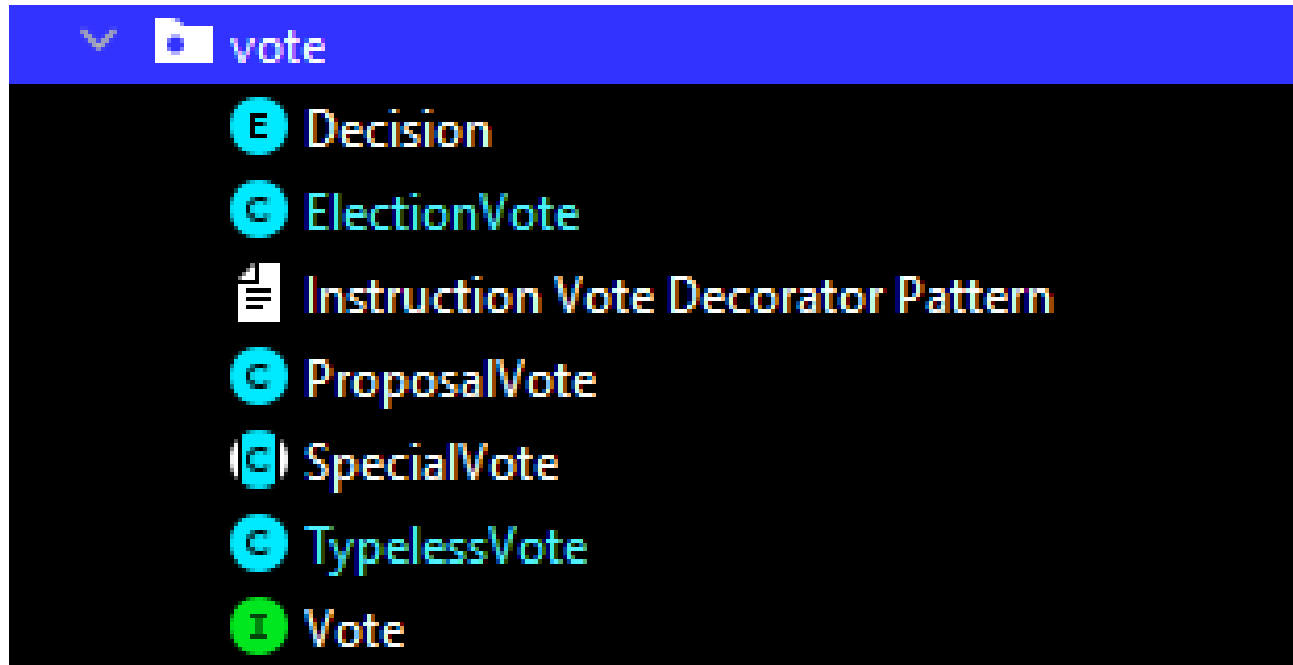


Figure 1.2: Vote directory located in `hoa-microservice/src/main/java/nl/tudelft/sem/hoa/domain`

I also included a text file which contains an instruction on how to properly build a vote object using this Decorator pattern.

```

1 //how to create a vote using the decorator pattern
2 //first create a Vote
3
4 Vote voteFromPiet = new typelessVote(piet.getUserId());
5 Vote voteFromJan = new typelessVote(jan.getUserId());
6 Vote voteFromKlaas = new typelessVote(klaas.getUserId());
7
8
9 //then make the vote of correct type
10 SpecialVote proposalVote = new ProposalVote(vote1, f382d92c, ABSTAIN);
11 SpecialVote electionVote = new ElectionVote(voteFromJan, 371648ccd, boardMemberJan.getId());
12 SpecialVote electionVote = new ElectionVote(voteFromKlaas, 371648ccd, boardMemberWitek.getId());
13
14 //then you can use all method in ElectionVote class etc
15 electionVote.getUserId();
16 electionVote.getElectionId();

```

Figure 1.3: Instruction usage Decorator

The rest of this section will be screenshots of the in section 1.1.1 described class implementations.

```

1 package nl.tudelft.sem.hoa.domain.vote;
2
3 public class TypelessVote implements Vote {
4     protected transient String userId;
5
6     A TypelessVote itself is useless. But it is needed to complete the decorator pattern.
7     Params: userId – ID of the member that has voted this vote
8
9     public TypelessVote(String userId) { this.userId = userId; }
10
11     public String getUserId() { return this.userId; }
12 }

```

Figure 1.4: TypelessVote class

```

1 package nl.tudelft.sem.hoa.domain.vote;
2
3 public interface Vote {
4     String getUserId();
5 }

```

Figure 1.5: Vote interface

```

1  package nl.tudelft.sem.hoa.domain.vote;
2
3  import java.util.Objects;
4
5  public abstract class SpecialVote implements Vote {
6      protected transient Vote vote;
7
8      public SpecialVote(Vote vote) { this.vote = vote; }
11
12     public String getUserId() { return this.getVote().getUserId(); }
15
16     public Vote getVote() { return this.vote; }
19
20     @Override
21     public boolean equals(Object o) {
22         if (this == o) {
23             return true;
24         }
25         if (!(o instanceof SpecialVote)) {
26             return false;
27         }
28         SpecialVote that = (SpecialVote) o;
29         return getVote().equals(that.getVote());
30     }
31
32     @Override
33     public int hashCode() { return Objects.hash(getVote()); }
36 }

```

Figure 1.6: *SpecialVote* abstract class

```

1 package nl.tudelft.sem.hoa.domain.vote;
2
3 public class ElectionVote extends SpecialVote {
4
5     private transient String electionId;
6     private transient String applicantId;
7
8     An ElectionVote is a vote for the board elections of one specific HOA.
9     Params: vote – The vote component that the decorator pattern uses.
10    electionId – ID to determine the election
11    applicantId – ID of the applicant which this vote is for
12
13    public ElectionVote(Vote vote, String electionId, String applicantId) {
14        super(vote);
15        this.electionId = electionId;
16        this.applicantId = applicantId;
17    }
18
19    public String getElectionId() { return electionId; }
20
21    public String getApplicantId() { return applicantId; }
22 }

```

Figure 1.7: ElectionVote concrete Decorator class

```

1  package nl.tudelft.sem.hoa.domain.vote;
2
3  import nl.tudelft.sem.hoa.domain.proposals.ProposalPk;
4
5  public class ProposalVote extends SpecialVote {
6
7      private transient ProposalPk proposalPk;
8      private transient Decision decision;
9
10     A ProposalVote is a vote for a proposal.
11     Params: vote – vote component used in the decorator pattern
12     proposalPk – Id to determine which proposal this is about
13     decision – ENUM Type can be: ABSTAIN, REJECT, ACCEPT
14
15     public ProposalVote(Vote vote, ProposalPk proposalPk, Decision decision) {
16         super(vote);
17         this.proposalPk = proposalPk;
18         this.decision = decision;
19     }
20
21     public ProposalPk getProposalId() { return proposalPk; }
22
23     public Decision getDecision() { return decision; }
24
25     public void setDecision(Decision decision) { this.decision = decision; }
26 }

```

Figure 1.8: ProposalVote concrete Decorator class

```

1  package nl.tudelft.sem.hoa.domain.vote;
2
3  public enum Decision {
4      ABSTAIN,
5      ACCEPT,
6      REJECT
7  }

```

Figure 1.9: Decision enumeration implementation



## 1.2 Builder pattern | Activity

### 1.2.1 Description

Problem: How can we create different representations of a complex object like the responses for activities for example? How can we generalize the creation of potentially different types of responses? How can we simplify it? In the scenario we are presented with modularity and abstraction are key. Therefore, it's important to think about how to encapsulate creating and assembling complex objects like activity responses in separate objects. The activity responses present precisely such a challenge and this is why we decided to use the Builder creational design pattern for them specifically. The Builder design pattern comes to our rescue as it separates the responses' creation, parameters configuration (the actual response option chosen and the responder's name), and other functionality, providing a much more flexible code structure that can potentially be extended much more conveniently.

This allows us to vary the responses' internal representation (we can easily add new member attributes like for example date), encapsulate the code for construction and representation, and overview the different steps of construction (which inevitably also increases control over them).

Potential downsides of this approach are that we must create distinct concrete response builders for each concrete type of response we'd like to build. For example, if I want also EventOrganisationResponses apart from ActivityResponses. This can complicate dependency injections. Nevertheless, the advantages this approach brings for our specific case significantly outnumber the disadvantages as we do not have too many different response types.

In the sections underneath (1.2.2 and 1.2.3) we have portrayed the UML class diagram of the responses builder design pattern to ease the visualization of the implementation. Additionally, you can also trace the actual code of it through the implementation screenshots provided underneath that include the full classes' implementation.

### 1.2.2 Class diagram

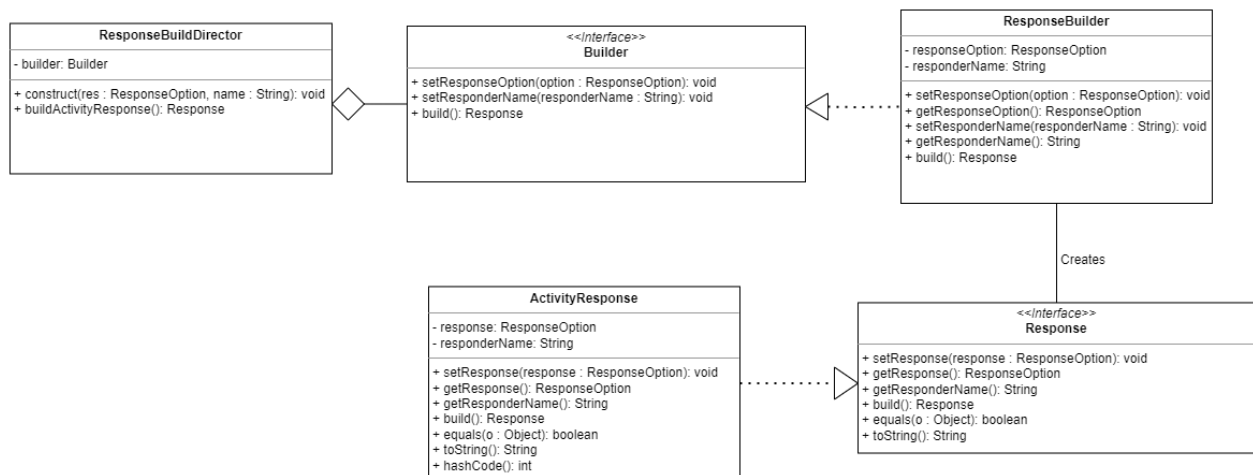


Figure 1.10: Class diagram for the responses system using a builder pattern.

### 1.2.3 Implementations

We created a concrete `ActivityResponse` class that implements a generic `Response` interface. Additionally, We made a generic `Builder` class with a concrete `ResponseBuilder` that produces `ActivityResponse` objects. The `Builder` creates `Response` objects and this makes it very generic. For the last step, we made a `ResponseBuildDirector` that encapsulates the generic builder and allows the developer to augment the encased builder through it, which is the class actually responsible for building the response to store. That way we solved all of the above-mentioned problems and dealt with complex objects like the activity responses, separating every step from the user's request, through the environment setup and configuration, to the concrete building of a response that is to be then used in the application context. Please note that the actual response options have been defined as an enum of `going`, `interested`, and `not interested` within the `ResponseOption` enum class.

```
1 package nl.tudelft.sem.activity.domain;
2
3 public class ResponseBuildDirector {
4     private final transient Builder builder;
5
6     public ResponseBuildDirector(Builder builder) {
7         this.builder = builder;
8     }
9
10    public void construct(ResponseOption res, String name) {
11        this.builder.setResponseOption(res);
12        this.builder.setResponderName(name);
13    }
14
15    public Response buildActivityResponse() {
16        return this.builder.build();
17    }
18 }
19
```

8 usages Vladimir Plamenov Petkov

4 usages

3 usages Vladimir Plamenov Petkov

2 usages Vladimir Plamenov Petkov

2 usages Vladimir Plamenov Petkov

Figure 1.11: Code implementation for the *ResponseBuildDirector* used.

```

1 package nl.tudelft.sem.activity.domain;
2
3 19 usages 1 implementation Vladimir Plamenov Petkov
public interface Builder {
4 27 usages 1 implementation Vladimir Plamenov Petkov
    void setResponseOption(ResponseOption option);
5
6 26 usages 1 implementation Vladimir Plamenov Petkov
    void setResponderName(String responderName);
7
8 22 usages 1 implementation Vladimir Plamenov Petkov
    Response build();
9 }
10

```

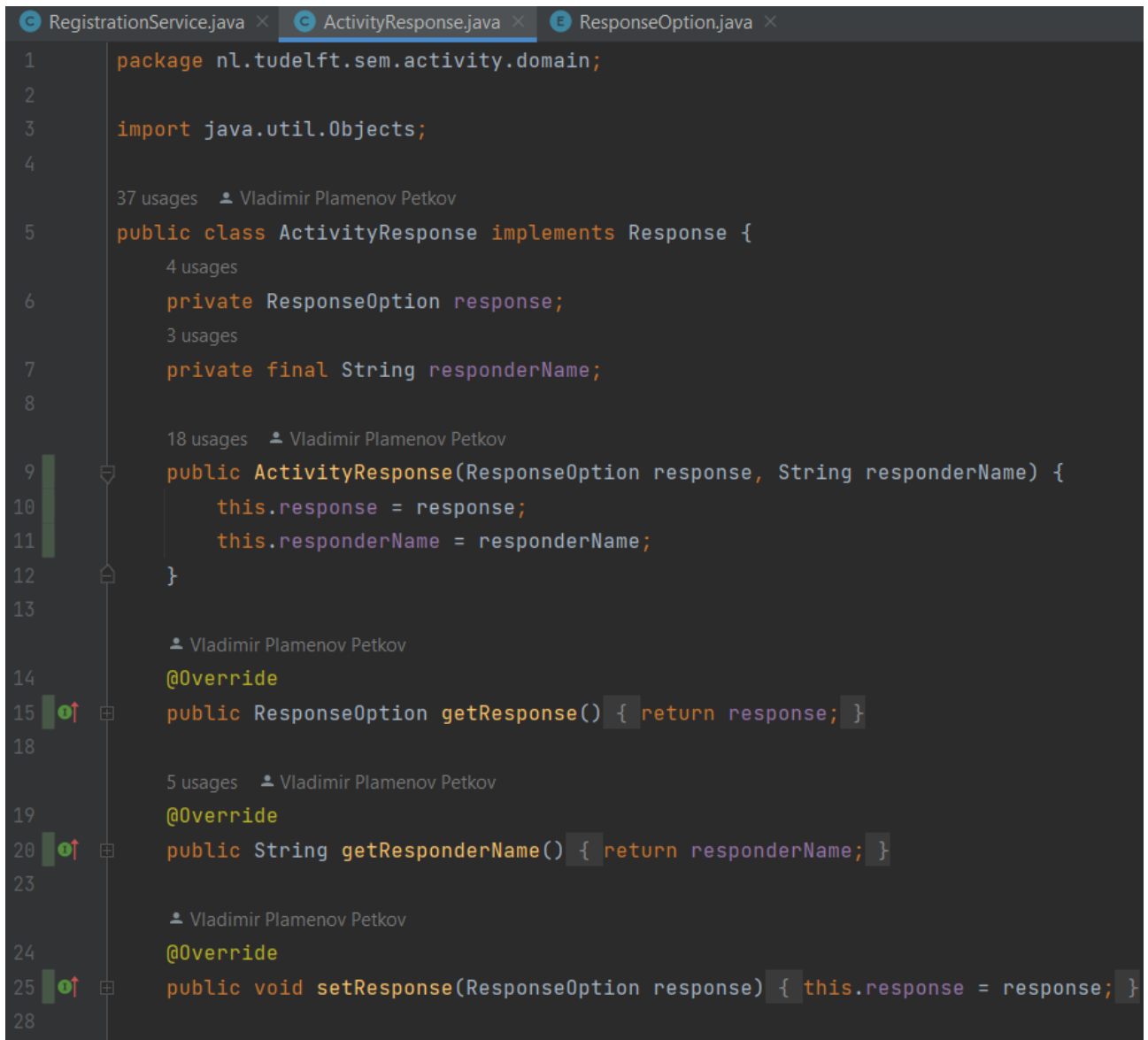
Figure 1.12: Code implementation for the Builder used.

```
RegistrationService.java × ResponseBuilder.java × Response.java × ActivityResponse.java × ResponseOption.java ×
1 package nl.tudelft.sem.activity.domain;
2
3 /PMD.BeanMembersShouldSerialize/
4 public class ResponseBuilder implements Builder {
5     3 usages
6     private ResponseOption option;
7
8     2 usages Vladimir Plamenov Petkov
9     public ResponseOption getOption() { return this.option; }
10
11     27 usages Vladimir Plamenov Petkov
12     @Override
13     public void setResponseOption(ResponseOption option) { this.option = option; }
14
15     3 usages
16     private String responderName;
17
18     2 usages Vladimir Plamenov Petkov
19     public String getResponderName() { return this.responderName; }
20
21     26 usages Vladimir Plamenov Petkov
22     @Override
23     public void setResponderName(String responderName) { this.responderName = responderName; }
24
25     22 usages Vladimir Plamenov Petkov
26     @Override
27     public Response build() {
28         return new ActivityResponse(this.option, this.responderName);
29     }
30 }
31 }
```

Figure 1.13: Code implementation for the ResponseBuilder used.

```
1 package nl.tudelft.sem.activity.domain;
2
3 1 implementation  Vladimir Plamenov Petkov
4 public interface Response {
5     1 implementation  Vladimir Plamenov Petkov
6     public ResponseOption getResponse();
7
8     5 usages  1 implementation  Vladimir Plamenov Petkov
9     public String getResponderName();
10
11     1 implementation  Vladimir Plamenov Petkov
12     public void setResponse(ResponseOption response);
13
14     1 implementation  Vladimir Plamenov Petkov
15     @Override
16     public boolean equals(Object obj);
17
18     1 implementation  Vladimir Plamenov Petkov
19     @Override
20     public String toString();
21 }
22 }
```

Figure 1.14: Code implementation for the Response used.



```
1 package nl.tudelft.sem.activity.domain;
2
3 import java.util.Objects;
4
5 37 usages Vladimir Plamenov Petkov
6 public class ActivityResponse implements Response {
7     4 usages
8     private ResponseOption response;
9     3 usages
10    private final String responderName;
11
12    18 usages Vladimir Plamenov Petkov
13    public ActivityResponse(ResponseOption response, String responderName) {
14        this.response = response;
15        this.responderName = responderName;
16    }
17
18    Vladimir Plamenov Petkov
19    @Override
20    public ResponseOption getResponse() { return response; }
21
22    5 usages Vladimir Plamenov Petkov
23    @Override
24    public String getResponderName() { return responderName; }
25
26    Vladimir Plamenov Petkov
27    @Override
28    public void setResponse(ResponseOption response) { this.response = response; }
```

Figure 1.15: Code implementation for the ActivityResponse used part 1.

```

29      Vladimir Plamenov Petkov
30      @Override
31      public boolean equals(Object o) {
32          if (o instanceof ActivityResponse) {
33              Response r = (ActivityResponse) o;
34              return this.getResponderName().equals(r.getResponderName()) && this.getResponse().equals(r.getResponse());
35          } else {
36              return false;
37          }
38      }

39      Vladimir Plamenov Petkov
40      @Override
41      public int hashCode() { return Objects.hash(response, responderName); }
42
43      Vladimir Plamenov Petkov
44      @Override
45      public String toString() { return getResponse().toString() + " " + getResponderName(); }
46  }
47
48  }
49

```

Figure 1.16: Code implementation for the ActivityResponse used part 2.

```

3  Vladimir Plamenov Petkov
4  public enum ResponseOption {
5      15 usages
6      GOING,
7      25 usages
8      INTERESTED,
9      8 usages
10     NOT_INTERESTED
11 }

```

Figure 1.17: Code implementation for the ResponseOption enum used.



## Bibliography

- [1] Nick Youngson. *Homeowners association*. Retrieved: 27 November 2022. URL: <https://pix4free.org/assets/library/2021-10-24/originals/homeowners-association.jpg>.