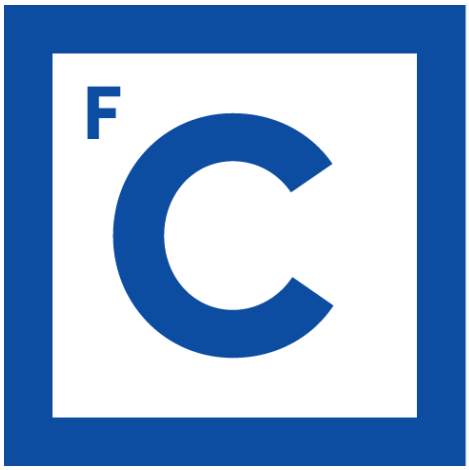


Ciências ULisboa

Construção de Sistemas de Software

Democracia2.0 Fase 1

2022/2023

Relatório

Miguel Agostinho fc53475

Guilherme Garrilha fc53838

João Pena fc54477

Relativamente aos requisitos funcionais / Casos de Uso pedidos nesta fase, foram todos implementados, sendo que os únicos requisitos não funcionais que não foram cumpridos, não são desejados nesta fase i.e autenticação com OAUTH, e uma API Rest ter sido usada. Fora isso, todos os requisitos não funcionais são cumpridos.

Nesta fase utilizámos simplesmente a camada de business, onde particionámos por outras subpackages, nomeadamente Entities, para as entidades do Projeto (Citizen, Vote, Theme, Project, etc), por repositórios a serem utilizados pelas entities (CitizenRepository etc), em services que servem para realizar operações para as Entities ou para os casos de usos pedidos (CitizenService, ClosePollService). Também criámos exceções customizadas para os casos específicos. Fora do business acrescentámos um Enum a utils, chamado de StatusPoll.

Na próxima fase ponderamos em acrescentar controllers à camada Presentation.

Relativamente às decisões tomadas nas estratégias e mapeamentos, mapeamos o seguinte:

Classe Citizen:

```java
/**
 * Class that represents a Citizen entity with their respective information such as their name and
 * ID, along with getters and setters, furthermore being able to get their Delegates and setting
 * them as well. Also being defined are the associations to tables, along with strategies.
 */
@Entity
@Table(name = "citizen")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.STRING)
public class Citizen {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @OneToMany(mappedBy = "citizen", cascade = CascadeType.ALL)
  private List<CitizenThemeDelegate> citizenThemeDelegates;

  /** Default constructor for citizen. */
  public Citizen() {}

  /**
   * Constructor for citizen
   *
   * @param name citizen's name
   */
  public Citizen(@NonNull String name) {
    this.name = name;
    this.citizenThemeDelegates = new ArrayList<CitizenThemeDelegate>();
  }
```

Em relação à classe-entidade Citizen, fizemos @Entity para formarmos uma tabela com a entidade Citizen. @Table cria a tabela com o nome citizen, usamos a estratégia SINGLE_TABLE em inheritance, que cria uma única tabela para todas as entidades filhas de Citizen. Usámos esta estratégia visto que temos ainda algumas classes e subclasses, logo não queremos preencher a database com múltiplas tabelas para cada subclasse. Adicionando a isso, temos vários Casos de Uso em que há maior utilidade de carregar

os Citizens todos sendo de maior facilidade fazê-lo com uma única tabela. @DiscriminatorColumn possibilita-nos ter uma coluna para cada entidade na tabela.

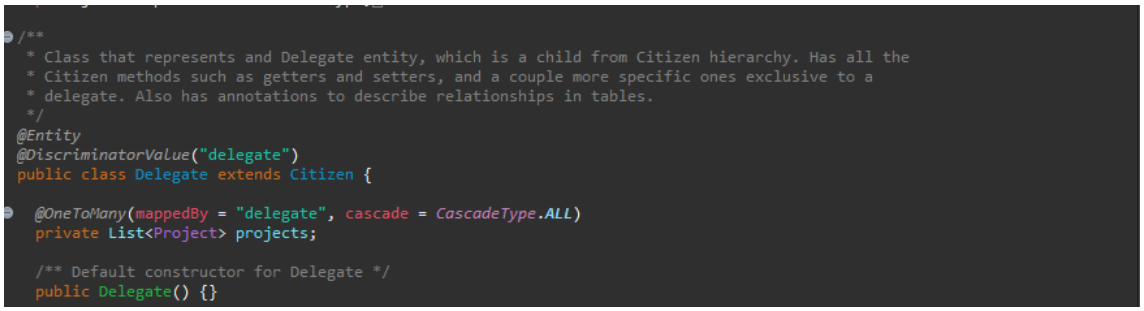
@Id serve para tornar o parâmetro a chave primária (comum a todas as classes), @Generated Value com estratégia GenerationType.IDENTITY, usando IDENTITY gera valores de chave primária automaticamente, sendo assim desnecessário o input manual por parte do programador (não sendo hardcoded), levando assim a que cada coluna tenha uma chave primária diferente, simplifica código e é mais fácil trabalhar com a base de dados. @OneToMany utilizámos esta anotação para indicar que para cada cidadão podem existir vários objetos citizenThemeDelegate, que são vários pares de temas e delegados únicos (resulta de uma ligação ternária entre cidadão-tema-delegado). MappedBy indica o dono da relação, que neste caso é citizen, e cascade tem CascadeType.ALL como valor de Cascade, todas as operações realizadas em Citizen são propagadas para as outras tabelas dependentes, por exemplo o delete.

Classe citizenThemeDelegate:

```java
@Entity
public class CitizenThemeDelegate {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "voter_id")
    private Citizen citizen;

    @ManyToOne
    @JoinColumn(name = "delegate_id")
    private Delegate delegate;

    @ManyToOne
    @JoinColumn(name = "theme_id")
    private Theme theme;

    /** Default constructor for CitizenThemeDelegate. */
    public CitizenThemeDelegate() {}

    /**
```

Em relação à classe-entidade CitizenThemeDelegate, fizemos @Entity para formarmos uma tabela com a entidade CitizenThemeDelegate. @GeneratedValue com estratégia GenerationType.IDENTITY, usando IDENTITY gera valores de chave primária automaticamente, levando assim a que cada coluna tenha uma chave primária diferente. Estabelecendo uma relação @ManyToOne, para muitos cidadãos há um votador. O mesmo se aplica para os delegados e temas. @JoinColumn junta uma associação de entidades, neste caso fazendo sentido visto que é uma ligação ternária.

Classe Delegate:



```java
/**
 * Class that represents and Delegate entity, which is a child from Citizen hierarchy. Has all the
 * Citizen methods such as getters and setters, and a couple more specific ones exclusive to a
 * delegate. Also has annotations to describe relationships in tables.
 */
@Entity
@DiscriminatorValue("delegate")
public class Delegate extends Citizen {

  @OneToMany(mappedBy = "delegate", cascade = CascadeType.ALL)
  private List<Project> projects;

  /** Default constructor for Delegate */
  public Delegate() {}
```

@DiscriminatorValue especifica o value de uma Discriminator Column.

@OneToMany indicando que para um delegado pode haver vários projetos.

Classe Poll:

```java
 * verifications to check whether or not it's an ongoing poll according to dates, or if a citizen
 * has voted. Also has annotations for tables and strategies.
 */
@Entity
public class Poll {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private List<Long> idList;

  private int positiveVotes;

  private int negativeVotes;

  private String name;

  private LocalDateTime endDate;

  @Enumerated(EnumType.STRING)
  private StatusPoll statusPoll;

  @ElementCollection
  @CollectionTable(name = "delegate_votes", joinColumns = @JoinColumn(name = "poll_id"))
  @MapKeyColumn(name = "delegate_column")
  @Column(name = "vote_column")
  private Map<Long, Boolean> delegateVotes;

  @OneToOne
  @JoinColumn(name = "project_id")
  private Project project;

  /** Default constructor for Poll. */
  public Poll() {}
```

@Enumerated para persistir o enum na base de dados. Persistido em string, sendo mais estável, no entanto mais lento. Considerando o tamanho do projeto, preferimos a estabilidade.

@ElementCollection especifica uma coleção de elementos, neste caso necessitado para poder persistir um map de ids de delegados e as suas escolhas/votos, sendo estas valores Booleanos.

@CollectionTable especifica uma tabela em separado que mapeia a coleção.

@MapKeyCollumn na tabela possui-se uma coluna para keys (Long), e outra para values(Boolean) associados a keys. Basicamente transcrever um Map para Table. Daí ter-se usado @Column.

@OneToOne porque não pode existir poll sem existir projeto, definindo uma relação binária.

Classe Project:

```java
@Entity
public class Project {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String title;
  private String text;
  private byte[] pdf;
  private List<Long> signatures;
  private LocalDateTime endDate;
  private boolean ongoing;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "delegate_id")
  private Delegate delegate;

  @OneToOne(mappedBy = "project", cascade = CascadeType.ALL)
  private Poll poll;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "theme_id")
  private Theme theme;

  /** Default constructor for Project. */
  public Project() {}
```

@ManyToOne porque para vários projetos há um só delegado. O FetchType.LAZY serve para não carregar dados desnecessários, carregando assim o objeto só quando necessário. @OneToOne explicitando que para um projeto tem de haver um poll (outra parte da relação binária da classe Poll), o cascade é utilizado dado que o projeto é o dono da relação (pelo mappedBy) logo toda a operação tem de ser propagada. @ManyToOne especificando que vários projetos podem ter um só tema.

Classe Theme:

```java
@Entity
public class Theme {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @OneToMany(mappedBy = "theme")
  private List<Project> projects;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "parent_theme_id")
  private Theme parentTheme;

  @OneToMany(mappedBy = "parentTheme", cascade = CascadeType.ALL)
  private List<Theme> subthemes;

  /** Default constructor for theme. */
  public Theme() {}

  /**
   * Constructor for theme, with the respective name.
   *
   * @param name name being given to theme.
   */
```

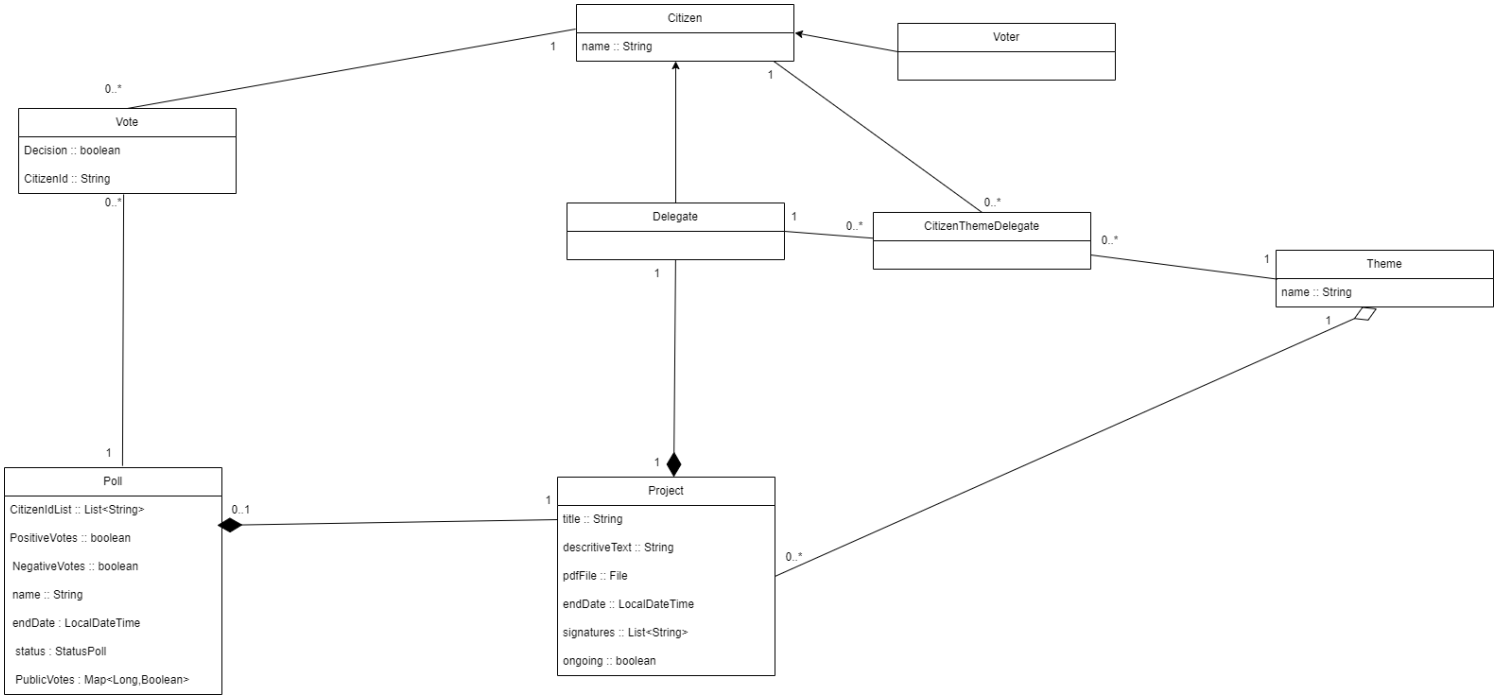
@OneToMany com mappedBy indicando que theme é a dona da relação, indicando também um tema pode ter vários projetos. @ManyToOne com fetch type Lazy, indicando que o objeto é só carregado quando necessário, indicando que vários temas podem ter unicamente uma classe pai. Seguidamente, @OneToMany, indicando que parentTheme é a classe dona (pelo mappedBy), indica que para um tema, pode-se ter vários subtemas. O cascadetype com ALL, indica que toda a operação tem de ser propagada.

Classe Voter:

```java
@Entity
@DiscriminatorValue("voter")
public class Voter extends Citizen {

  /** Default constructor for Voter */
  public Voter() {}

  /**
   * Constructor for a voter given a name
   *
   * @param name name being given
   */
  public Voter(@NonNull String name) {
    super(name);
  }
}
```

Desenhos do Projeto

Modelo de Domínio



SSD Caso de Uso J

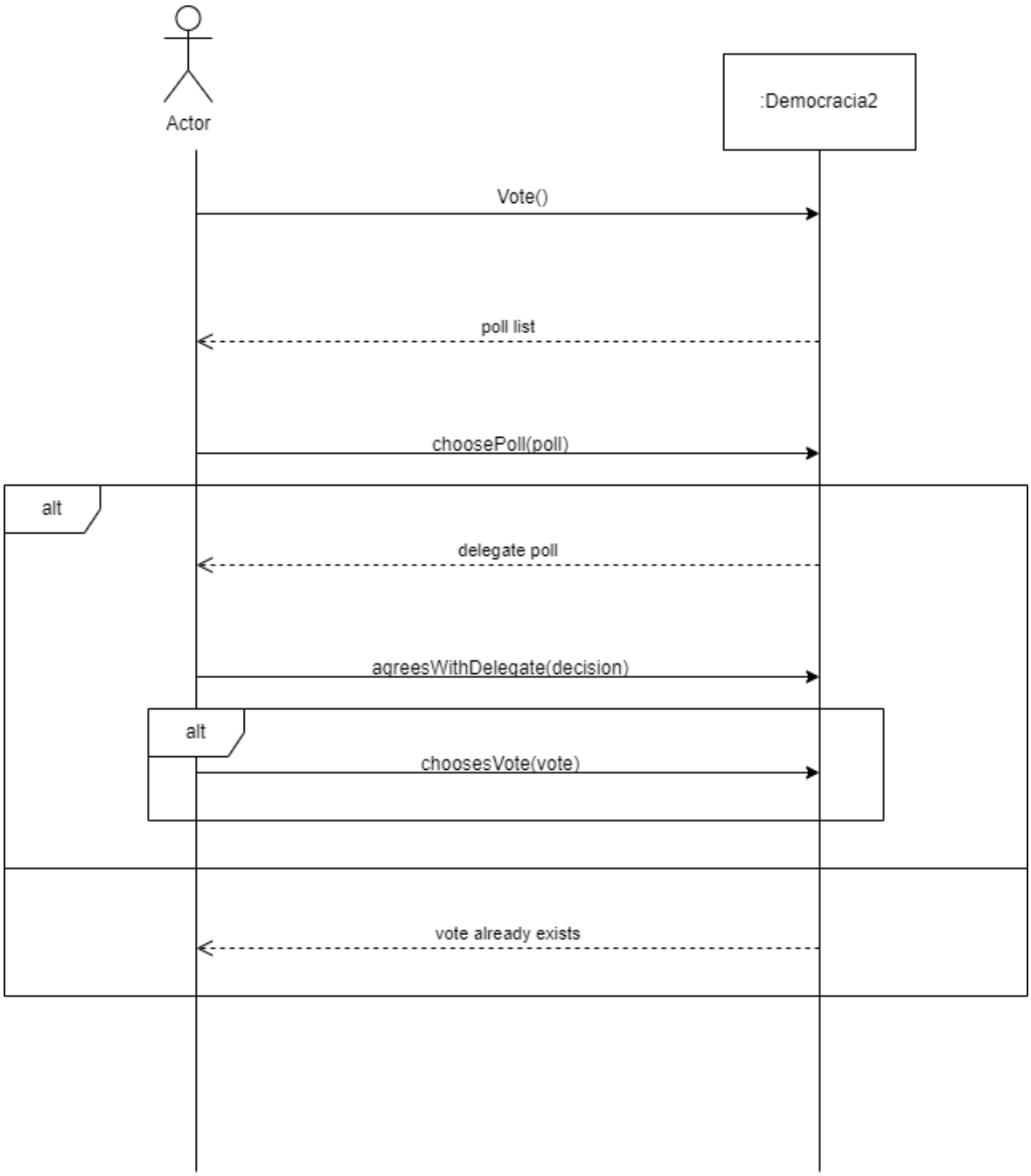


Diagrama de Classes