



Universidade Federal de Viçosa - Campus Florestal

Projeto de Desenvolvimento

Disciplina: CCF 313 - Programação Orientada a Objetos

Professor: Fabrício Aguiar Silva

Lucas Takeshi - 2665

Victor Hugo - 3510

Florestal, 24 de março de 2022

Sumário

1- Introdução	3
2- Desenvolvimento	3
2.1- Funcionalidades	3
2.1.1- Morador	3
2.1.2- Conta	4
2.1.3- Tarefa	4
2.1.4- Decisão	4
2.2- Orientação a objetos	4
2.2.1- Modularidade	5
2.2.2- Encapsulamento	5
2.2.3- Herança	6
2.2.4- Tratamento de Exceções	6
2.2.5- Polimorfismo	7
3- Conclusão	7
4- Referências	9

1- Introdução

À medida que a disciplina de **Programação Orientada a Objetos** avançava, o professor **Fabício Aguiar Silva** juntamente com os monitores **Fábio Trindade Ramos** e **Gabriel Teixeira Pinto Coimbra**, propuseram a prática dos conceitos vistos em ambiente virtual de aula em um projeto que fosse capaz de resolver um problema da realidade. Além de reforçar os conceitos aprendidos a respeito da linguagem **Java**.

Este relatório consiste em descrever todo o trabalho desempenhado no projeto ao longo do período, chamado **iRep**. O projeto em questão busca desenvolver um sistema de **interesse real** da dupla. A dupla escolheu focar em problemas que as **repúblicas** passam no dia a dia, como **controle de contas, atribuir tarefas de casa, definir grandes tomadas de decisões**, entre outros.

A ideia deste projeto fornece uma grande oportunidade de colocar em práticas os grandes pilares da programação orientada a objetos e o modelo de arquitetura **MVC**, para, assim, poder organizar um sistema em módulos bem definidos, **reduzindo o acoplamento e aumentando a coesão**.

2- Desenvolvimento

O desenvolvimento deste relatório apresentará o projeto criado sob duas perspectivas diferentes: o código/projeto em si, apresentando as classes e suas funcionalidades e o projeto sob a ótica da programação orientada a objetos, ressaltando pontos de aplicação de um determinado conceito.

2.1- Funcionalidades

Esta seção (2.1) explica as entidades envolvidas no projeto e suas funcionalidades, exceto as funcionalidades de cadastro e de listar.

2.1.1- Morador

As funcionalidades implementadas foram pensadas na parte do front-end.

O usuário precisa ver as informações dele ou de outro morador e usado a função *exibirMoradorPorId()*, que pelo id do morador você conseguiria acessar as informações do morador, na parte do morador temos um *listarTarefasAtribuidas()*, que mostrar todas as tarefas que estão atribuída a ele e que ainda não foram feitas. **Essa função** busca em Tarefas o *idMorador* atribuído a cada tarefa e assim filtra somente aqueles que estão atribuídos a ele.

2.1.2- Conta

A intenção de criar a função *efetuarPagamentoConta()* foi pensada para que o usuário/morador pudesse escolher uma conta pelo seu *IdConta* e passar de “Em Aberto” para “Paga”, e remover da lista que usamos para mostrar as contas “Em Aberto” e usamos a função *exibirContasEmAberto()* para exibir.

2.1.3- Tarefa

Usamos a tarefa para ser uma lista de afazeres dos moradores, e assim usamos a função *efetuaAtribuicaoTarefa()* que usamos o *idMorador* e o *idTarefa* para vincular e atribuir a função, e quando finalizado chamamos a função *concluiTarefa()* e ela passa a ter uma flag sinalizando que está feita.

2.1.4- Decisão

Decisão seria a parte de reunião onde teríamos o *efetuavoto()* em que sempre que se passa *idDecisao* e o voto vai contando quantos votos para sim ou não foram feitos e quando estiver para finalizar é usado o *calculaResultado()* que verifica a quantidade de moradores e a quantidade de votos, e finaliza informando se foi a decisão foi aceita ou não.

2.2- Orientação a objetos

Sabe-se que o paradigma orientado a objetos busca diminuir o gap semântico entre o mundo real e o domínio do projeto em questão. Logo, a “modelagem” de qualquer projeto que aborde tal paradigma deve ser coesa com o mundo real. Portanto, a separação em módulos, o encapsulamento de estruturas, o uso de herança, tratamento de exceções e polimorfismo são práticas/pilares extremamente importantes para se aplicar o paradigma orientado a objetos. A seguir, serão descritas como tais práticas foram executadas ao longo do projeto.

2.2.1- Modularidade

A divisão em módulos menores, mais simples e com características específicas foi facilitada com a implantação da arquitetura de projetos MVC (Model-View-Controller).

Tal arquitetura forneceu a dupla extrema facilidade para execução da modularidade, uma vez que cada módulo é bem definido.

Neste caso, tem-se os principais módulos (Model-View-Controller), além dos módulos:

- **modelo.entidade**
- **modelo.exceção**
- **modelo.persistência**

Os três módulos “extras” fazem parte do módulo modelo, assim como apresentado pelo professor no projeto da **Eleição**, feito durante aula assíncrona. **Entidade** é responsável por conter as abstrações das entidades envolvidas no projeto, neste caso **Morador, Conta, Tarefa e Decisão**. **Exceção** é responsável por conter as várias exceções criadas para tratar algum trecho de código. **Persistência** é responsável por conter o padrão o *DAO (Data Access Object)*, que é um padrão de projeto que abstrai e encapsula os mecanismos de acesso a dados escondendo os detalhes de implementação de tratamento de dados.

2.2.2- Encapsulamento

Ainda falando sobre o padrão MVC, este ajuda bastante a esconder os detalhes de implementação da interface, ou seja, a encapsular o projeto. Definindo corretamente o uso dos módulos de uma projeto que utilize MVC, pode-se dizer que este projeto encontra-se encapsulado, uma vez que o MVC busca exatamente reduzir o acoplamento entre os módulos, escondendo, ao máximo, detalhes de um módulo de um outro qualquer.

Além do MVC, o projeto conta com **getters e setters** (só os necessários) para todas as entidades, além de modificadores de acesso (*public, private, protected*) corretamente definidos.

2.2.3- Herança

Sabe-se que o paradigma orientado a objetos busca ao máximo reaproveitar um código, evitando “inventar a roda”. O conceito de herança traz grandes inovações para a programação na orientação a objetos, podendo, uma determinada classe, adquirir características, além de suas próprias, de outras classes.

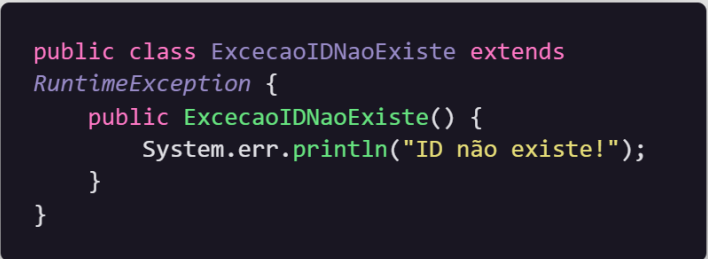
Neste projeto, o conceito de herança foi fortemente aplicado no tratamento de exceções e na persistência, como um todo.

Várias exceções foram criadas para atender uma determinada demanda no projeto, seja uma entrada inválida, ou até mesmo a inexistência de um dado no banco de dados. Todas as exceções criadas herdam de alguma determinada exceção, seja *RuntimeException*, seja *InputMismatchException*, dependendo da aplicação. Além disso, para o banco de dados, foi criada uma interface ***GenericDAO()*** para conter as implementações (contratos) básicas de todo *DAO* do projeto.

2.2.4- Tratamento de Exceções

Assim como dito anteriormente, algumas exceções foram criadas para atender uma determinada demanda no projeto. Esta prática se tornou possível graças ao conceito de herança da programação orientada a objetos.

Uma exceção é um evento excepcional ou erro durante a execução do programa. Abaixo, tem-se uma figura que apresenta uma exceção lançada quando um ID não é encontrado e logo em seguida uma figura apresentando o lançamento desta.



```
public class ExcecaoIDNaoExiste extends
RuntimeException {
    public ExcecaoIDNaoExiste() {
        System.err.println("ID não existe!");
    }
}
```

Figura 1 - *ExcecaoIDNaoExiste()* - lançada quando um ID não existe em uma pesquisa

```

public String exibirMoradorPorID(int idMorador) throws
ExcecaoIDNaoExiste{
    Morador morador = moradorDAO.pesquisa(idMorador);

    if (morador == null){
        throw new ExcecaoIDNaoExiste();
    }
    return (morador.toString());
}

```

Figura 2 - *moradorDAO.pesquisa()* retorna um **morador**, caso *IdMorador* seja válido, caso contrário, retorna **null**, lançando a exceção **ExcecaoIDNaoExiste()**

2.2.5- Polimorfismo

A capacidade de um objeto assumir a forma de outro é chamada Polimorfismo e este foi aplicado também na classe **GenericDAO()**. Uma vez que a definição da interface prevê um Objeto para algumas funções, como **pesquisa()** e **remove()**, que retornam um Object T, mas, na prática, pode retornar uma Conta, Morador, Tarefa ou Decisao.

```

public T pesquisa(int ID);
public T remove(int ID);

```

Figura 3 - Funções *pesquisa()* e *remove()* da interface GenericDAO

3- Conclusão

Por fim, com os trabalhos realizados acerca deste projeto, a dupla pode aprender, por meio da prática, como funcionam os projetos reais utilizando **Orientação a Objetos**. Além disso, com a utilização do padrão de projeto MVC, a separação em módulos coesos permitiu uma maior produtividade durante o desenvolvimento dos códigos.

A solução de um problema permitiu também produzir uma solução que realmente agrega valor no mundo real, podendo ser refatorada no futuro, para quem sabe, ser vendida.

A dupla também pode amadurecer o trabalho em equipe por meio da troca de conhecimentos e ideias, uma vez que este projeto contém ideias e implementações de ambos os alunos.

4- Referências

[1] Fabrício Aguiar - **Notas de Aula**. UFV-Campus Florestal. Disponível em: PVANet Moodle