Alunos: Wallyngson Eliezer dos Santos Guedes | 117110835

Felipe Emanuel de Farias Nunes | 117211052

Victor Paz de Farias Braga | 117210688

Dácio Silva Bezerra | 118210572

**Relatório de Projeto LP2 | eDoe.com**

***Design geral:***

O design do nosso projeto foi adotado para possibilitar um simples entendimento com uso de nomenclaturas de métodos bastante claros, com o máximo de modularização possível, com todos os métodos e classes necessários para realizar com sucesso toda a especificação do sistema. Com relação a complexidade das diferentes partes do sistema, escolhemos um design simples, onde temos um controlador que gerencia todos os itens e todos os usuários do sistema, além de delegar funcionalidades para as camadas menores, criamos uma Interface de Itens, onde existem classes abaixo que implementam os métodos da interface e os seus métodos específicos. Com relação aos usuários optamos por uma classe abstrata de Usuario e criar duas classes filhas chamadas Doador e Receptor, optamos por esse design pois o doador e o receptor são tipos especiais de Usuario.

O código está cheio de exceções checadas, segundo o Expert da Informação, é justamente onde temos a informação que estamos checando as exceções, caso haja alguma lançamos a exceção, logo após capturamos e a tratamos, gerando assim uma baixa taxa de quebrar o código.

Dividimos todo o código em pacotes, com as classes que compõem diretamente o sistema em seu pacote, temos o pacote dos itens, o pacote dos usuários, o pacotes com os testes de unidades, um para a persistência e outro chamado util com todas as partes utilitárias, tais como, uma classe chamada validador, que valida todas as informações necessárias, um exemplo é, se o usuário já existe, entre muitos outros, temos também todos os comparadores organizados, gerando assim um bom entendimento para qualquer programador, professor ou aluno que por ventura venha a olhar o projeto, gerando um código simples, com um design simples e um entendimento rápido.

As próximas seções explicam com mais clareza cada caso e como foi a solução dada aos mesmos.

***Caso 1***:

No caso 1, as especificações determinam que devemos poder criar, atualizar, pesquisar, atualizar e deletar usuários.

Os usuários podem ser de dois tipos, Doados ou Receptores.  
Os receptores são adicionados através da leitura de um arquivo .csv que é lido em um método especial para esse fim e acessa diretamente e exclusivamente o método responsável por criar usuários receptores.  
Já os usuários doadores são adicionados diretamente pelo usuário através do método responsável.  
Aqui tratamos o usuário como uma classe abstrata que dá origem a duas classes filhas concretas, Doador e Receptor.

Esses usuários são armazenados em uma coleção do tipo HashMap, onde a chave é a numeração do documento do usuário (obviamente), e os valores desse mapa são os próprios objetos usuário.  
 No método de pesquisa passamos o número de documento ou nome do usuário que se quer encontrar. (no caso de passar o nome e termos dois ou mais nomes iguais, se retornam os usuários que tiverem nomes iguais ordenados pelo nome de forma decrescente alfabeticamente falando). No caso de passar o número da documentação tudo fica mais fácil, apenas buscamos ele nas chaves do mapa. Já no caso de receber o nome, temos que colocar os usuários em uma lista e iterar em busca do nome.  
Por fim, o método retorna o toString() do(s) usuário(s) encontrado(s).

Também temos o método de atualização, que recebe o id do usuário e assim, como no método anterior, acessa o usuário correspondente e é capaz de alterar seu email, telefone ou nome, (ele pode fazer isso com um de cada vez ou com todos ao mesmo tempo, depende de quantos parâmetros válidos forem passados para a atualização) recebendo o nome do atributo que quer alterar e o novo valor.

Por fim temos o método que remove usuarios do do mapa de usuários.   
Esse é bem simples, é passado a String que representa o id do usuario, atraves dele

o programa acessa a chave correspondente no mapa de usuários, caso o usuario exista, e simplesmente o remove retornando um booleano true caso haja sucesso na operação.

***Caso 2:***

No caso 2, nos é pedido para cadastrar itens a serem doados, mantendo uma coleção de descritores, só será permitido cadastrar os itens caso o existir o descritor de itens já exista, caso não o descritor deve ser criado, não se pode existir dois descritores iguais, devem ser únicos, e devemos também associar esses itens criados a um usuario doador passado por parâmetro (de acordo com o seu documento), aqui criamos uma interface chamada de Item, e criamos a classe ItemDoavel, que tem seus métodos específicos de item de doação, fazemos todas as verificações de exceções na classe Item antes de criarmos, verificamos se os nomes ou tags são vazias ou nulas, verificamos também se a quantidade é maior que zero, se tudo estiver correto, criamos o item doável, assim que criamos o item doável deve se gerar um id único para esse item, os itens são iguais se seus nomes e tags forem iguais. Criamos um contador idUnico geral na classe Controller sempre que criamos um item adicionamos +1 nesse contador, assim nenhum item possui o mesmo id, quando criamos o item retornamos o id único, e enviamos o objeto ItemDoavel ao Doador e adicionamos a seu Mapa de Itens a serem doados, assim mantemos uma Coleção de Itens no Controller que possui todos os itens que podem ser doados no sistema, mas em cada doador mantemos apenas os seus itens a serem doados, assim possibilitando melhor acessar e modificar esses itens.

Além disso criamos métodos específicos de atualização de dados dos itens (atualizaItem), um método de remoção de itens (removeItens, que remove o item tanto da coleção geral com da coleção associada a cada doador), tal como o de exibição (exibeItem) que exibe um item já cadastrados, importante salientar que se um item já estiver cadastrado e tentarmos o cadastrar novamente mudando apenas a sua quantidade, o item deve ser cadastrado novamente, porém com um id diferente e aquele que já tinha antes deve ser excluído da coleção geral. Todas essas funcionalidades relacionadas aos itens apenas são possíveis pois existe o identificador único e sempre que vamos fazer alguma coisa é atraves do seu identificador único.

***Caso 3:***

No caso 3, o sistema já conhece os itens a serem doados, portanto, nos é pedido para listar os itens de maneiras distintas. Para listarmos de maneiras distintas, como pedido, fizemos um *package* de nome “útil”, para organizar estas classes subsidiárias ao problema. A maneira padrão de comparação é a comparação por nome. Por isso, a interface Item estende primordialmente a classe *Comparable* e compara os itens por quantidade. Essa utilidade se encaixa na necessidade inicial do caso, que é listar os itens por sua quantidade. Além desse padrão, temos que listar os descritores, ordenando-os por seus nomes, ação feita pelo método *listaDescritorDeItensParaDoacao().* Por fim, o caso requer a pesquisa ordenada por nomes, de itens cuja descrição seja igual à descrição do parâmetro do método. Para essa função incumbimos o método *pesquisaItemPorDescricao(),* que vai chamar a classe subsidiária *ComparadorNome* para poder ordenar a lista de acordo com o pedido.

***Caso 4:***

O caso 4 requer o avanço do sistema com a utilização dos usuários receptores e o cadastro de seus itens necessários. Esses itens são iguais – até determinado ponto – dos itens doáveis, contudo, mais a frente eles vão se diferenciar pois um necessita contar pontos e outro não. Por isso a interface, podemos ter um mapa de itens, sejam de que natureza for (por conta da interface) e podemos acessá-los a qualquer momento, além de usuários doadores possuírem apenas itens doáveis e usuários receptores, apenas itens necessários, facilitando o processo. Os itens necessários a priori, se comportam da mesma forma que os doáveis, com id, descrição, etc.

Para cadastrar um item necessário, nós passamos as informações para a construção do objeto ItemNecessario, e uma informação extra para o método privado que *adicionaItem(),* onde este irá escolher qual item cadastrar de acordo com a informação, se doador ou se necessário, ele irá cadastrar um ou outro tipo de item.

Outra necessidade do caso é a listagem desses itens necessários por meio de seu id. Essa funcionalidade é subsidiada pela classe *ComparadorID* e implementada no Controller pelo método *listaItensNecessarios().*

A forma de atualizar e remover os itens necessários e doáveis, são idênticos, por isso, utilizam-se dos mesmos métodos para atualização, sem necessidade de duplicação de código.

***Caso 5:***

Para o caso 5, era necessário criar um sistema para realizar casamentos entre itens necessários e itens doáveis e para isso criamos o método Match que funciona a partir do descritor do item e suas tags, o usuário receptor inicia a funcionalidade declarando o que deseja receber e o sistema busca um usuário doador com que encaixe com as especificações desejadas pelo receptor. A estratégia para realizar o casamento considera o descritor dos itens em primeira ordem e as suas tags em segundo momento ao verificar se as mesmas fazem o casamento com o item requisitado.

O sistema faz as listagem dos possíveis matches a partir de um sistema de pontuação, em que itens com o mesmo descritor iniciam com 20 pontos e tags na mesma posição somam 10 pontos, já tags em posições diferentes somam 5 pontos. Os possíveis matches são ordenados a partir de sua pontuação com a classe ComparaItemPorPontuacao e caso a pontuação seja a mesma deve se ordenar a partir do id do item que está sendo doado a partir da classe ComparadorId. No fim os possíveis matches devem ser exibidos.

***Caso* *6:***

No caso 6 era requisitado que adicionássemos as funcionalidades de realizar doações e manter o histórico dessas doações para que os mesmos fossem listados de forma ordenada a partir de sua data. Para isso criamos uma classe de Doacao para armazenar as mesmas e posteriormente poder ordená-las e listá-las.

O primeiro método criado na classe foi o realizaDoacao que ao receber o id do Item necessário (IdItemNec), o id do item doado (IdItemDoado) e a data da doação, realizava o procedimento da doação dos itens, caso o item doado ou o item necessário chegasse a 0 unidades após a realização da doação o mesmo era excluido do sistema. Após a doação ser realizada com sucesso, ela era armazenada em uma lista para posterior ordenação e exibição.

O outro método utilizado era o listaDoacoes que ao verificar a lista de todas as doações realizadas ordenava a mesma pela data (na classe ComparaPorData, do mais antigo para o mais recente) e caso houvesse doações realizadas no mesmo dia a mesma era ordenada pelo nome do item doado (em ordem alfabética), e posteriormente listando todas.

***Caso 7:***

No caso 7, deveríamos utilizar a persistência de dados para armazenar todos os dados necessários ao sistema para manter o estado do mesmo sempre que o programa fosse fechado. Ao utilizar a classe serializable e implementar ela em todos as classes necessárias para manter os dados salvos, utilizamos a pasta data para guardar os arquivos .dat necessários ao sistema para sua manutenção. O sistema utiliza de dois métodos para inicializar e finalizar o sistema, o primeiro carrega os dados .dat que foram guardados ao fechar o programa, e o segundo salva os dados necessários para o sistema. Os dados guardados no sistema são as listas de Descritores, Itens e Usuarios que são as principais coleções utilizadas no nosso sistema.

***Considerações:***

Fizemos todo o projeto em Português, tanto métodos, quanto comentários, quanto nomenclaturas.

***Link para o repositório no Github:***

<https://github.com/wallyngson/eDoe.com/>

***Linguagem Usada:***

Java (<https://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>);

***Testes de Unidade:***

JUnit 5 (<https://junit.org/junit5/>);

***Testes de Aceitação:***

EasyAccept (<http://easyaccept.sourceforge.net/>);