Alunos: Wallyngson Eliezer dos Santos Guedes | 117110835

**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***:

O caso 1 pede que seja criado uma entidade que represente os diferentes tipos de funcionários no sistema. Os funcionários podem ser Médicos (M edic ), Técnico (Technician ) e Diretor (Director ). Para representar esses diferentes tipos, optamos por criar uma classe abstrata chamada Employee que contém todos os comportamentos em comum entre esses tipos, além da definição do método generateRegistration(), que cria um código de identificação único para cada objeto. O uso do polimorfismo com herança possibilitou o alto reuso de código, sendo necessário apenas escrever o comportamento de um método (generateRegistration() ), já que cada tipo tem comportamento diferente para a criação do ID único.

Para gerenciar os funcionários, foi criada uma classe chamada C linicalTeam . Nesta classe, há

uma coleção que armazena todos os funcionários. Essa coleção é um mapa, onde a chave é o código de identificação, único para cada usuário.

***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, um banco de órgãos foi criado. Para isso, criamos a entidade O rgan que

encapsula um órgão, contendo informações como o nome e o tipo sanguíneo do órgão. Para

armazenálos, a classe O rgan Bank foi criada. Ela possui um ArrayList para armazenar as referências dos órgãos, bem como métodos de consulta, adição e exclusão de um órgão no sistema.

Caso 6:

No caso 6 era requisitado que adicionássemos a funcionalidade de realizar um determinado

procedimento. Para isso era necessário fazer a checagem se o usuário em sessão é um medico, essa funcionalidade está na classe C linicalTeam , que tem o método c heckMedic , responsável por fazer esta checagem. Os procedimentos pré definidos são: transplante de órgãos ( OrganTransplant ), cirurgia bariátrica (B ariatricSurgery ), redesignação sexual (S exualReassignment ), consulta médica ( MedicalAppointment ).

Foi acordado que cada procedimento teria uma classe própria, porém, foi percebida a presença de reuso de e a possibilidade de polimorfismo. Foi decidido então o uso de uma interface, chamada P rocedureIF , pois cada procedimento tem um padrão de comportamento, mas que varia entre os diferentes procedimentos. No caso, a interface contém os métodos performProcedure() responsável por realizar um procedimento em um paciente (recebido no construtor da classe), getProcedureType() responsável por retornar o tipo de procedimento, getProcedurePoints() responsável por retornar o número de pontos que o procedimento fornece e os métodos getDate() e g etMedic() responsáveis por retornar respectivamente a data em que foi realizado e o médico que realizou o procedimento. Como cada procedimento tem um médico, data, tipo e pontos específicos o uso de uma interface é viável. No caso de performProcedure() , o comportamento é apenas registrar o procedimento, adicionar o gasto e realizar (por exemplo, no caso de uma cirurgia bariátrica, é necessário reduzir a massa do paciente, cada procedimento tem seu comportamento específico).

No topo do sistema, há a classe Procedures , onde há o controle da realização de procedimento, esta classe possui um padrão strategy , pois existe um atributo ProcedureIF procedure ,

que assume uma instancia do tipo das classes de procedimentos, dependendo do procedimento

requisitado na entrada pelo usuário. Também possui um padrão factory , pois o procedimento é criado interativamente, evitando que os dados de um paciente se acumulem para outro procedimento.

A classe GeneralController faz a checagem com o banco de órgãos (se necessário), coleção de pacientes, etc. Para evitar o alto acoplamento da classe P rocedures com essas classes foi optado por fazer as checagens necessárias na classe GeneralController.

Caso 7:

No caso 7, deveríamos colocar no nosso sistema a funcionalidade de um cartão fidelidade,

que possui pontos, que são obtidos quando um procedimento é realizado. Esse cartão, que pertence ao paciente (Patient ), poderia ter três tipos de fidelidade: padrão, master e v ip . Como, segundo a especificação, há a possibilidade de evolução de um tipo de fidelidade para outro, desde que os pontos necessários para isso sejam atingidos, optamos por fazer um s trategy , que facilita a mudança de tipo (no caso, de fidelidade). O Strategy é um padrão de design , que reúne polimorfismo, composição e interface, de tal forma que possam haver mudanças no tipo de uma instância facilmente. Devido a isso, uma mesma referência pode assumir várias formas.

Sendo assim, criamos a interface I Fidelity , que representa o que uma implementação de fidelidade faz. No caso, possui os métodos de calcular desconto e de calcular bônus. E ainda as classes Standard , Master e V ip , que implementam essa interface, de acordo com seus valores de desconto e de bônus.

Após isso, criamos a entidade FidelityCard , que encapsula o que um cartão fidelidade deve fazer. Há um atributo para guardar a quantidade de pontos, e uma composição com I Fidelity , que representa o tipo de fidelidade do cartão, onde F idelityCard é o c omposite . A partir daí é utilizado o strategy , para que haja a mudança de acordo com a quantidade de pontos do cartão.

Como pela lógica um paciente deve possuir um cartão fidelidade, existe agora uma composição entre Patient e FidelityCard , com métodos delegadores.

Caso 8:

No caso de uso 8 foi pedida a funcionalidade de exportar a ficha de um paciente para um

arquivo de texto que deve ficar dentro da pasta “fichas\_pacientes”. Adaptamos então a classe Patients , para que fosse possível essa exportação.

Criamos o método para tal funcionalidade na classe P atients , já que esta é a classe que controla os pacientes. No corpo do método, temos um F ile que representa a pasta “ fichas\_pacientes” ,

e caso a pasta não exista, ela é criada. Temos ainda um F ile que representa a ficha do paciente em si, com o nome do arquivo sendo “Nome\_Paciente\_Data.txt”, conforme especificado; e um FileWritter e um BufferedWritter , para a escrita da ficha no arquivo.

Sendo assim, temos um s tream de texto, em que a fonte é o sistema, o destino é o arquivo na pasta especificada, a direção é de saída (escrita) e há processamento intermediário ( Buffer ).

Para a escrita, ajustamos o t oString() de M edicalRecord , que agora contem os dados do paciente (t oString() de P atient ), os gastos e pontos do cartão fidelidade, e ainda lista todos os procedimentos realizados, que foram adaptados para receber o nome do médico que os realizou e a data.

Finalizando, os métodos em F acade e G eneralController apenas delegam para o método criado em P atients . Este último, então, escreve o t oString() de M edicalRecord no arquivo. Se o arquivo já existir, ele é substituído, como foi pedido.

Caso 9:

No caso de uso 9 foi requisitado que a equipe adicionasse a funcionalidade de salvar o estado do sistema para persistência de todos os dados após o encerramento. Para isso, foi necessário implementar a interface serializable em todas as classes com o nível 1 ou mais (sendo a façade o nível 0). Foi feita a criação da pasta “/system\_data/” para armazenar o arquivo contendo o estado do sistema.

Foi entendido que, o atributo “GeneralController controller ”, na classe Façade , além de fazer a gestão e aplicar o padrão Controller , tinha todos os dados salvos em seus atributos (que são classes de coleções), no caso, os dados de medicamentos estão salvos no atributo “Pharmacy

pharmacy ”, do banco de órgãos em “OrganBank organBank”, etc. Para isso, a classe facade tem os métodos “iniciaSistema” e “fechaSistema”:

No primeiro caso, o método tem um trycatch, onde no escopo do comando é verificado se há

algum sistema já carregado na pasta indicada (“system \_data /”), se não houver então o controller é iniciado sem nenhum dado salvo. Caso haja um arquivo de sistema para ser carregado, é aberto um novo stream de dados ( ObjectInputStream ), onde a fonte é o arquivo e o destino é o sistema, a direção é de entrada, o tipo de dados é object e não há processamento intermediário. Depois é dado o comando .readObject() , onde o objeto é lido e atribuído ao atributo controller .

No segundo caso o raciocínio relativo a qual atributo ser armazenado no mesmo local é o mesmo. No escopo do método “fechaSistema” há um trycatch , que checa se há um arquivo carregado

na pasta, se não houver é criado um novo arquivo soos.dat vazio, onde serão armazenados os dados do sistema. Depois disso, é criado um stream do tipo ObjectOutputStream , onde a fonte é o sistema e o destino o arquivo, a direção é de saída, o tipo de dados é object e não há processamento intermediário.

No comando .writeObject() o sistema é salvo no arquivo soos.dat .

***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/