Projeto da Disciplina – SOOS

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática – CEEI

Departamento de Sistemas e Computação - DSC

Disciplina: Laboratório de Programação II – Prof. Lívia Sampaio

Grupo: Gabriel Souto Maracajá – Mat. 115110977

Mattheus Brito Rodrigues – Mat. 115111167

Victor Emanuel Farias da Costa Borges – Mat. 115110681

Vinícius Brandão Araújo – Mat. 115111777

Sumário

Introdução

O projeto da disciplina consiste em implementar um sistema digital de saúde na linguagem Java. Apelidado de SOOS (Sistema Orientado a Objetos de Saúde), o sistema seria utilizado por funcionários de um hospital que estariam aptos a realizar diversas operações típicas de um hospital como cirurgias, transplantes de órgãos, gerenciamento de pacientes, entre outros.

O grande objetivo de desenvolver um sistema com tais características para um hospital está relacionado à praticidade do gerenciamento de informações. Afinal, por Java ser uma linguagem orientada a objetos, as operações de informações relacionadas aos funcionários, pacientes, medicamentos e procedimentos cirúrgicos seriam bem mais práticos, precisando apenas de simples comandos na máquina.

O hospital fictício que será gerenciado pelo projeto possui três hierarquias de funcionários: O diretor geral que faz parte do comitê gestor, isto é, pode gerenciar qualquer atividade; os médicos que fazem parte do corpo clínico, podendo acessar apenas as informações relacionadas aos medicamentos e procedimentos cirúrgicos; e por fim, os técnicos administrativos que seriam responsáveis por comandar o corpo profissional, ou seja, serão responsáveis pelo atendimento e cadastramento dos pacientes.

É importante ressaltar que o sistema permite apenas um diretor geral por hospital. Portanto, apenas médicos e ténicos administrativos podem ser criados após um diretor geral assumir o comando do hospital.

Milestone 1

A primeira etapa do projeto consistiu em implementar os pilares do sistema: funcionários, pacientes e medicamentos.

O primeiro caso de uso tinha como função iniciar o sistema como diretor geral e realizar o cadastro dos funcionários. Para organizar tudo isso, criamos um package *controller* com as classes *Controller* e *Facade* que irão gerenciar todo o sistema. Enquanto a *Facade* funcionará como uma “capa de proteção” para evitar erros e lançar exceções em entradas não identificadas, a *Controller* irá gerenciar todas as funções do hospital. Nesse primeiro caso, criamos os métodos:

liberaSistema(...) – Libera o sistema do hospital a partir da chave do diretor geral.

login(...) – Realiza o login do funcionário.

cadastraFuncionario(...) – Cadastra um funcionário no sistema a partir do cargo.

getInfoFuncionario(...) – Gera a matrícula do funcionário automaticamente.

fechaSistema() – Fecha o sistema do hospital.

logout() – Realiza o logout do funcionário.

Com isso, também tivemos que criar as classes que originariam os funcionários. Primeiramente, criamos um package *funcionario*, onde cada tipo de função diferente (*DiretorGeral, Medico* e *TecnicoAdministrativo*) são subclasses herdadas da classe-mãe *Funcionario*. Também criamos um enum *Funcoes* para definir cada atividade que o funcionário poderá exercer, além de uma classe *FuncionarioFactory*, responsável por criá-lo a partir de seus atributos.

Já no segundo caso de uso, nos deparamos com a modificação desses dados. Era necessário que o sistema pudesse modificar e excluir funcionários. Para o programa cobrir essas funções, criamos os seguintes métodos na *Controller*:

atualizaSenha(...) – Altera a senha do funcionário para um novo valor.

alteraNome(...) – Permite que o funcionário altere seu nome.

alteraData(…) – Permite que o funcionário altere sua data de nascimento.

excluiFuncionario(...) – Verifica se o funcionário tem as permissões para excluir outro e, caso tenha, o mesmo é excluído

verificaSenha(...) , verificaNome(...) , verificaData(...)

– Verifica se o novo valor é válido

A partir do terceiro caso, surgiram as questões envolvendo cadastro e atualização de pacientes e prontuários. Para essas funcionalidades, criamos um package pacientes onde implementamos as classes *Paciente* e *Prontuario*, que estão relacionadas através de composição. Além disso, tivemos a ideia de criar uma classe *ControllerProntuario* que contém todos os métodos envolvendo essas classes como: *criarPaciente(...), verificaProntuario(...), getInfoPaciente(...)*, entre outros. É importante ressaltar que todos esses métodos são chamado a partir da *Controller* citada anteriormente. Para acessar o prontuário de um paciente, por exemplo, teríamos o seguinte fluxo de acessos: *Facade* > *Controller* > *ControllerProntuario*.

O quarto e último caso do primeiro milestone estava relacionado aos medicamentos. Novamente, criamos um package separado farmacia onde implementamos a classe *Medicamento* com seus devidos atributos. Também criamos as duas classes *MedicamentoGenerico* e *MedicamentoDeReferencia* que estão conectadas com Medicamento por herança. Como um medicamento genérico ou de referência é um medicamento, optamos por usar esse tipo de reuso. Como os dois tipos se diferenciam apenas pelo seu cálculo de preço diferente (o genérico custa 60% do preço integral), criamos um método polimórfico *calculaPreco(...)* para nosso programa usurfruir do padrão GRASP de baixo acoplamento. Também criamos uma classe *Farmacia* que guarda esses medicamentos em um estoque (ArrayList) e contém todos os métodos relacionados aos mesmos. Além disso, criamos uma classe *FactoryMedicamento* que será chamada para criá-lo, um enum *Categoria* onde listamos todos os tipos de remédio, além de uma classe auxiliar *ComparaNome* que é chamada nos métodos de comparação dos medicamentos.

Milestone 2

A segunda etapa do projeto consistiu em implementar procedimentos cirúrgicos, cartões de desconto e um banco de órgãos para doações.

O quinto caso de uso tem como definição a criação de um banco de órgãos para que o hospital possa cadastrar órgãos disponíveis para doação. Nessa situação, criamos um package *bancodeorgaos* com apenas duas classes: *Orgao* e *BancoDeOrgaos*. Essa última, além de guardar os órgãos em um banco (ArrayList), também contém alguns métodos:

cadastraOrgao(...) – Cria um órgão e o adiciona no banco.

buscaOrgaos(...) – Procura um órgão no estoque.

buscaOrgaosCompativel(...) – Procura um órgão compatível de acordo com o sangue.

quantidadeDeOrgao(...) – Mostra a quantidade de órgãos disponíveis.

remove(…) – Remove um órgão do banco.

O sexto caso tinha como base a realização dos procedimentos cirúrgicos. Existem quatro tipos de procedimentos que o sistema do projeto deve interagir: consulta clínica, cirurgia bariátrica, redesignação sexual e transplante de órgãos. Criamos um package *procedimentos* com uma classe principal *Procedimentos* de quem as classes *TransplanteOrgao*, *RedesignacaoSexual*, *CirurgiaBariatrica* e *ConsultaClinica* herdam através da herança. Afinal, todos esses são procedimentos, destacando assim sua relação “é um”. Cada uma dessas classes tem pequenas diferenças quanto à suas informações, mas todas elas tem o método realizaCirurgia(...) em comum, ao qual é sobrescrito (Override).

Fora isso, temos também a classe *ControlleProcedimentos* que tem como objetivo administrar os procedimentos que serão realizados. Nela, temos alguns métodos:

procedimento(...) – Realiza um procedimento cirúrgico

trasnplantedeOrgaos(...) – Realiza um transplante de órgãos

Veja que como o procedimento do transplante de órgãos é bem diferente dos demais (afinal, depende do banco de órgãos), ele trabalha com um método em particular.

Já o sétimo e último caso de uso desse milestone explica o conceito do cartão de fidelidade. Cada paciente possui um para obter descontos nos procedimentos. Assim, criamos um package *cartaofidelidade* com as classes *CartaoPadrao*, *CartaoMaster* e *CartaoVip*, além de uma interface *CartaoDeFidelidade*. Essa interface contém a chamada de dois métodos presentes nas três classes, operando assim de forma polimórfica. Esses métodos são:

descontoservico(...) – Calcula o desconto que o paciente receberá.

creditobonus(...) – Calcula o crédito que o paciente ganhará com o procedimento.

Milestone 3

A terceira e última etapa do projeto consistia na exportação de arquivos externos ao programa, salvando dados em documentos externos ao código implementado.

O oitavo caso de uso se resumia a exportar a ficha dos pacientes, ou seja, seu prontuário para um arquivo de texto (extensão .txt). Além disso, o documento também deve conter o nome de cada médico, caso tenha realizado algum procedimento cirúrgico no paciente, como também a data em que foi realizada a operação.

Para realizar essa função, criamos um método *exportaFichaPaciente(...)* que recebe o ID do mesmo, salva a sua ficha em um arquivo com o título "*nomedopaciente_ano_mes_dia.txt*" e retorna uma string com o que foi escrito. Porém, a função de escrever os dados no arquivo foi realizada com o método auxiliar *salvaFicha(...)* que grava o prontuário com os comandos *BufferedWriter* e *FileWriter*. Procedemos da seguinte maneira:

BufferedWriter arquivoPaciente = new BufferedWriter(new FileWriter(nomeArquivo));

arquivoPaciente.write(fichaPaciente);

O nono e último caso tinha como proposta salvar todo o sistema do hospital para que, quando o diretor geral fechar o programa, os dados permaneçam salvos. Portanto, modificamos o método *iniciaSistema()* para que seja criado um arquivo "*system_data.dat*" caso ainda não exista nenhum salvo. Caso já exista, o metodo irá ler os dados salvos.

Também modificamos o método *fechaSistema()* de forma que ao fechar o programa, o mesmo salvará os dados no arquivo criado anteriormente. Assim, nenhuma informação será perdida quando o diretor geral realizar logout.

Classes de Teste

Em todos os casos de uso do projeto, optamos por criar classes de teste do tipo JUnit já que os testes disponibilizados pela disciplina no formato EasyAccept poderiam não ser suficientes para mostrar que o programa funciona corretamente e segundo as especificações.

Por isso, criamos um package *testes* com todas as JUnit Test Cases. Baseamos nossos testes em três divisões: a chamada dos métodos, para verificar se as funções estão sendo devidamente chamadas; as comparações de saída, verificando através do *assertEquals(...)* se o método está retornando o esperado; e por último a captura de erros, para verificar quando o programa falhará de acordo com nossas especificações, tais como nome vazio, quantidades negativas, entre outros.

Conclusão

Após implementarmos todos os nove casos de uso, além de testarmos o programa pelo EasyAccept e JUnit, há como garantir que nosso programa está pronto para o uso cotidiano. Apesar de não ser algo profissional, criamos nosso código com todos os recursos que aprendemos no decorrer do curso, tais como orientação a objetos, coleções, composição, herança, polimorfismo, interface, arquivos, entre outros. Assim, encerramos nosso projeto.