Universidade Federal de Campina Grande
Centro de Ciências e Tecnologia
Departamento de Sistemas e Computação
Disciplina: Laboratório de Programação II
Professora: Lívia Sampaio

Relatório de Projeto

Sistema Orientado a Objetos para a Saúde (SOOS)

Equipe:

Aramis Sales Araujo
Elton Dantas de Oliveira Mesquita
Gabriel de Araujo Coutinho
Mainara Cavaltanti de Farias

Campina Grande
2016

Introdução

O Sistema Orientado a Objetos para a saúde (SOOS) é um sistema que possui parceria com o Sistema Único de Saúde (SUS), ele será utilizado por gestores e colaboradores de clínicas particulares e tem como objetivo auxiliar nos procedimentos internos inerentes à organização e seus departamentos, além de fornecer meios que possam agilizar e conduzir a gerência dos recursos humanos e físicos do hospital.

Através deste relatório pretendemos informar detalhadamente como o nosso SOOS foi implementado. O relatório será dividido pelos casos de uso, abordando além do design utilizado no projeto, os principais tópicos estudados nas disciplinas de Programação II e Laboratório de Programação II.

Sobre o design

Para este projeto, foram seguidos dois padrões de design de orientação a objetos: *Model-Controller* e *Façade* Design Patterns. Os primeiros são oriundos de um design ainda mais complexo, o *Model-View-Controller* (MVC), cujas ideias centrais são a reusabilidade de código e a separação de conceitos e/ou tarefas.

A utilização desse modelo no nosso projeto permitiu solucionar o problema através da separação das tarefas de acesso aos dados da lógica de negócio, introduzindo um componente entre os dois: o controlador.

Com o padrão *Façade* pudemos simplificar a utilização de um subsistema complexo apenas implementando uma classe que fornece uma interface única e mais razoável, porém se desejarmos acessar as funcionalidades de baixo nível do sistema isso é perfeitamente possível.

Vale ressaltar que o padrão *Façade* não "encapsula" as interfaces do sistema, ele apenas fornece uma interface simplificada para acessar as suas funcionalidades.

Com relação as Exceptions, criamos uma package chamada *exceptions*, nela possui todas as classes criadas para o encapsulamento de determinada situação de exceção. Por exemplo, *BancoOrgaoException* encapsula todas as exceções relacionadas ao banco de órgãos. Além disso, foi criada no mesmo pacote uma classe *VerificaExcecao*, ela possui métodos estáticos para verificação de parâmetros específicos.

Caso de uso 1

O primeiro caso de uso tem como objetivo fazer com que o diretor geral inicie o sistema do SOOS e realize o cadastro de novos colaboradores, para que os mesmos possam, posteriormente, utilizar o sistema.

Para que o sistema seja inicializado criamos um método chamado *liberaSistema,* que verifica se o sistema está bloqueado ou não. Se o sistema já foi desbloqueado, então uma exceção será lançada, pois o sistema só poderá ser liberado uma única vez. O desbloqueio é efetuado através de uma chave, que deve ser igual a "c041ebf8". Então, o sistema é liberado e uma matrícula para o diretor é gerada - a matrícula é o retorno do método. O *liberaSistema* está presente na *Controller* e recebe os dados da *Façade*, que realiza um *forwarding.*

Determinado usuário pode ter acesso a todas as funções inerentes ao seu cargo através do método *login,* que também foi implementado na *Controller*. Caso o método não lance nenhuma exceção, o usuário será logado.

Para definir os diferentes tipos de cargo do usuário (Diretor Geral, Médico ou Técnico administrativo) criamos um *Enumeration*, pois o código se torna mais reusável e legível.

O cadastro de funcionários ocorre por meio do método *cadastraFuncionario* do *Controller,* que recebe o nome, o cargo e a data de nascimento do mesmo como parâmetro e então essas informações são passadas à *Factory de Funcionários,* responsável pela criação do funcionário. Uma senha e uma matrícula são geradas automaticamente durante esse processo.

Após sua criação, o funcionário é adicionado ao banco de funcionários -Classe responsável por armazenar e gerenciar os funcionários- instanciado no *Controller*. Todas as informações envolvendo datas foram implementadas com a classe *LocalDate,* que oferece mecanismos que facilitam a manipulação desse tipo de dado.

Caso de uso 2

O caso de uso 2 remete às funcionalidades de acesso e alteração nas informações de funcionários, além da exclusão de um funcionário do sistema.

Para o acesso a informações de um funcionário, implementamos o método *getInfoFuncionario*, que tem como parâmetros a matrícula do funcionário que se deseja acessar a informação e o atributo desejado.

Esse método acessa o funcionário no banco de funcionários através da matrícula, e realiza o *get* do atributo desejado no funcionário.

Para a atualização de informações, implementamos o método *atualizaInfoFuncionario*, que na *Facade* e no *Controller* é um método que possui sobrecarga. Tal sobrecarga se deve ao fato de um usuário logado não precisar informar sua matrícula para atualização de suas informações e considerando também que o diretor pode atualizar as informações de qualquer funcionário. Nessas funcionalidades o nível de acesso requerido é respeitado, bem como as restrições para os novos valores atribuídos às informações do funcionário.

Em especial para a mudança da senha, existe um método qual recebe como parâmetro a matricula do funcionário logado, senha atual que será autenticada e uma nova senha.

Novas senhas que não façam parte do conjunto alfanumérico ou que não tenham a extensão entre 8 e 12 caracteres são tratadas como exceção.

Para fazer a exclusão de um funcionário do sistema, implementamos o método *excluiFuncionario*, que no *Controller* que realiza a autenticação do diretor que deve logado (visto que apenas este possui autorização para exclusão) e delega para um método com o mesmo nome no *bancoFuncionarios*. Esse método recebe como parâmetro a matrícula do funcionário que deseja excluir e a senha do diretor, realizando a remoção do funcionário.

Caso de uso 3

O caso de uso 3 implica na implementação das funcionalidades de cadastro e acesso de informações de pacientes.

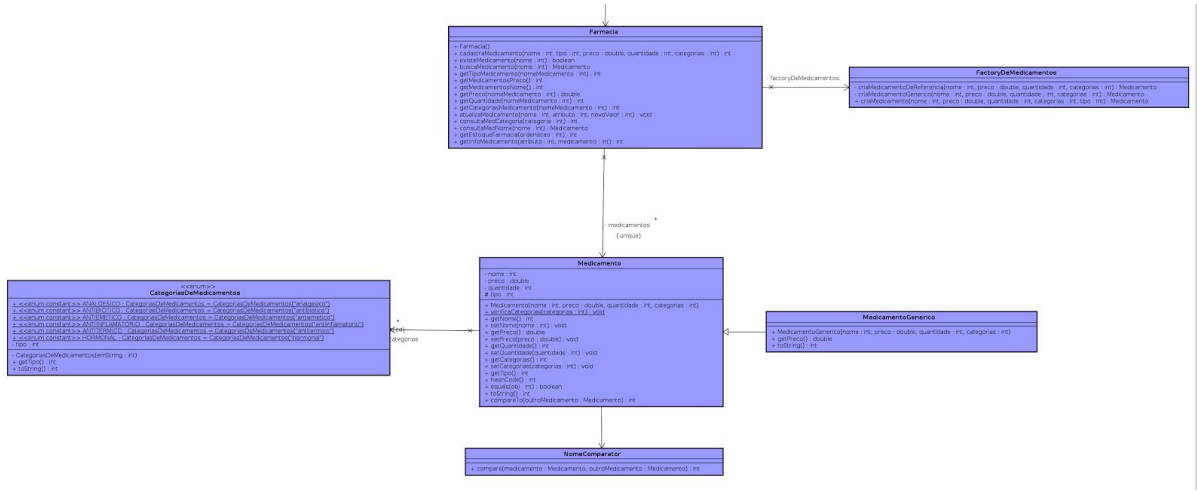
O cadastro de pacientes se dá através do método *cadastraPaciente* qual delega a criação e o armazenamento ao *bancoProntuarios*, no qual ocorrem todos os testes com relação a validade dos parâmetros. Esse método possui como parâmetros nome, data de nascimento, peso, tipo sanguíneo, sexo biológico e gênero, além um ID.

O ID é gerado a partir da utilização da classe UUID que nos garante a exclusividade de um ID gerado. Após a criação do paciente, este é imediatamente atribuído a um prontuário, devido a intrínseca relação entre eles. Por fim o prontuário é adicionado ao banco de prontuários, o qual é ordenado a cada atualização.

O acesso a uma informação de um paciente é semelhante ao dos funcionários, com o diferencial de o ID do paciente servir como forma de acesso. Também difere, pois o acesso é feito a partir do prontuário e não diretamente ao paciente. Só então a informação desejada é atingida.

Caso de uso 4

O objetivo deste caso de uso é gerenciar medicamentos para permitir a disponibilização deles nos tratamentos dos pacientes, essa gerência é feita pelos técnicos administrativos.



Para a criação e o gerenciamento de medicamentos foi implementada a classe *Farmacia*.

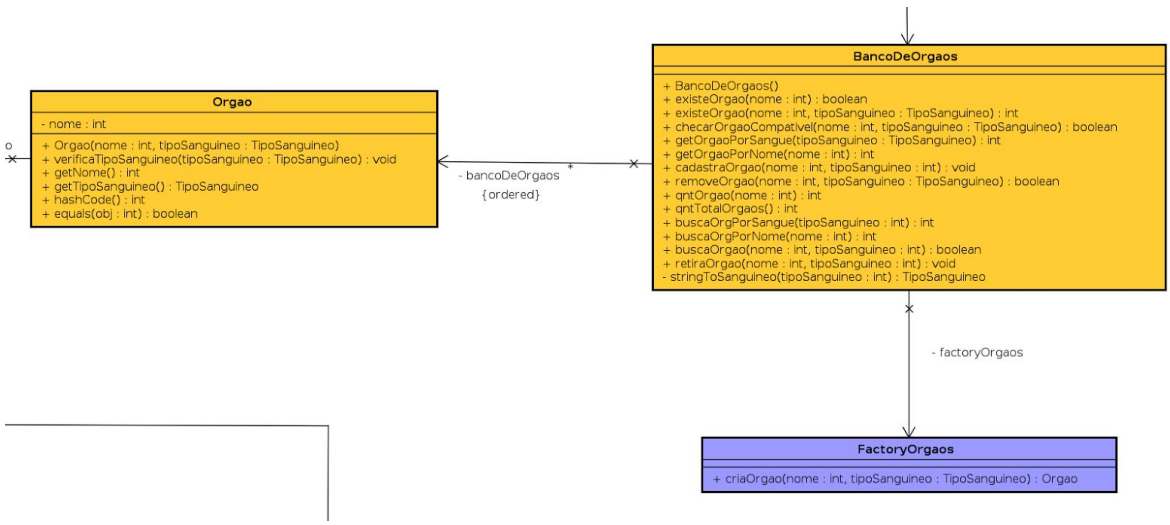
A *Farmacia* tem como uma de suas funções retornar todos os medicamentos em ordem alfabética ou pelo menor preço, para isso implementamos a interface *Comparable* na classe *Medicamento* considerando a ordem pelo menor preço, para ordenar alfabeticamente criamos a classe *NomeComparator* que funciona como um comparador.

A *Farmacia* possui um *Set* (já que não podem existir medicamentos repetidos) para armazenar todos os medicamentos cadastrados. A definição do medicamento é feita através da classe *Medicamento*, ela possui uma coleção *List* de categorias, essas categorias (analgésico, antibiótico, antiemético, anti-inflamatório, antitérmico e hormonal) são representadas por um *Enumeration* chamado *CategoriasDeMedicamentos*, pois com esse recurso tem-se mais facilidade para tratar, guardar e usar as constantes.

Os medicamentos podem ser de dois tipos: medicamento genérico e medicamento de referência, ambos possuem as mesmas informações e diferem apenas no cálculo do preço (caso o medicamento seja genérico terá um desconto de 40% do preço especificado), por esse motivo criamos a classe *Medicamento* que representa um medicamento de referência, e também a classe *MedicamentoGenerico* que herda de medicamento e faz a sobrescrita dos métodos *getPreco* e *toString* para representar um medicamento do tipo genérico.

Caso de uso 5

A responsabilidade desse caso de uso é cadastrar órgãos disponibilizados por doadores para a realização de transplantes no SOOS, os cadastros são permitidos apenas a funcionários médicos.



Para armazenar e gerenciar os órgãos fornecidos por doadores foi implementada a classe *BancoDeOrgaos*, ela possui uma coleção *List* (já que podem existir órgãos repetidos) para armazenar todos os órgãos cadastrados. Os órgãos são definidos pela classe *Orgao* que possui como atributos um nome e um tipo sanguíneo (enumerator).

Caso de uso 6

Nesse caso de uso foram implementados procedimentos aos quais os pacientes cadastrados no SOOS podem ser submetidos. Os médicos são os responsáveis pela execução desses procedimentos, que devem ser armazenados no prontuário do paciente.

Os tipos de procedimentos são: Cirurgia bariátrica, Consulta clínica, Redesignação sexual e Transplante de órgãos. A título de referência, esses tipos foram colocados em uma *Enumerator TipoProcedimento*.

Um procedimento é realizado através do método *realizaProcedimento,* que tem como parâmetros o nome do procedimento, o nome do paciente e os medicamentos que serão utilizados. Caso o procedimento seja uma consulta clínica ou um transplante de órgãos há uma sobrecarga do método, pois para uma consulta clínica são necessários apenas o nome do procedimento e o ID do paciente, já para o transplante de órgãos é necessário, além dos parâmetros já existentes, o nome do órgão.

Se a farmácia não possuir os medicamentos especificados, o procedimento não é realizado e uma exceção é lançada.

Para a implementação dos procedimentos foi criada uma Interface *Procedimento*, em que são definidos os métodos *realizaProcedimento* e *getPreco*.

As classes *CirurgiaBariatrica*, *ConsultaClinica*, *RedesignacaoSexual* e *TransplanteDeOrgaos* implementam essa Interface e cada uma possui um comportamento distinto para o método *realizaProcedimento* e sua própria representação através do *toString*. Com o intuito de manter um histórico de procedimentos, a classe *Prontuario* tem uma coleção de procedimentos realizados implementada como *List,* visto que a ordem em que os procedimentos foram realizados é importante.

Em todos as formas do método *realizaProcedimento* há manipulação dos dados do paciente. Sejam informações pessoais como sexo biológico e peso, nos casos de redesignação sexual e cirurgia bariátrica respectivamente, ou apenas contabilização dos gastos do paciente como na consulta clínica.

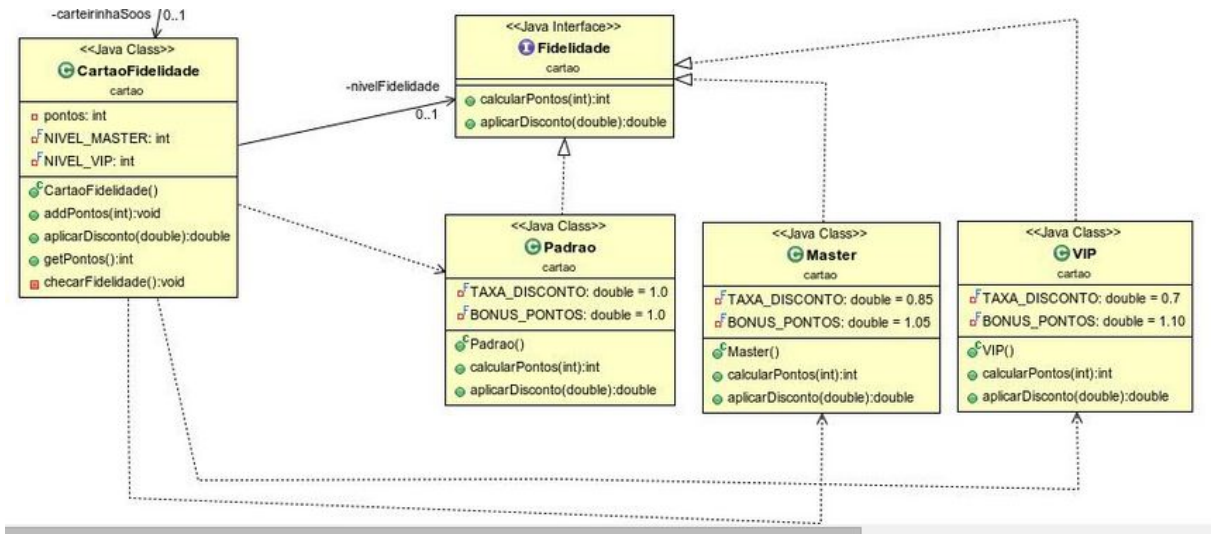
Caso de uso 7

Implementação do cartão fidelidade que tem como objetivo acumular pontos e fornecer descontos e benefícios durante a interação do paciente com o SOOS.

Os cartões fidelidade possuem três níveis de fidelidade: *Padrao*, *Master* e *VIP*. Para a implementação dos mesmos utilizamos o padrão strategy (como mostrado no diagrama abaixo), pois cada em cada nível de fidelidade do cartão observamos formas distintas de calcular os pontos que serão adicionados e também o desconto aplicado.

Sendo assim, foi criada a Interface *Fidelidade* que define os métodos que as classes *Padrao*, *Master* e *VIP* terão que implementar. A classe *CartaoFidelidade* é responsável por fazer a troca dinâmica do tipo de nível de fidelidade de acordo com sua pontuação.

O gasto total que é mantido no paciente considera os descontos aplicados pelo cartão fidelidade.



Caso de uso 8

O objetivo desse caso de uso é exportar a ficha de um prontuário de um paciente para disponibilizar o registro de operações realizadas com o paciente especificado, os técnicos administrativos executam esse passo.

Para fornecer ao paciente e médicos informações sobre o paciente, foi criado o método *exportarFichaPaciente*, nele todas as informações de um paciente são salvas. Em sua implementação foi utilizado a *Stream BufferedWriter* para escrever os dados do paciente em um arquivo de texto. Caso o sistema gere uma nova ficha do mesmo paciente e na mesma data, o sistema sobrescreve a ficha existente no diretório.

Todas as fichas são salvas em um diretório chamado *"fichas_pacientes"* que é criado automaticamente no caso de não existir na raíz da execução do Soos.

Cada prontuário possui um método *getFichaPaciente* que retorna ao controler uma representação em String da ficha do paciente que então é levada à classe *FileMannager* que realizará a escrita da ficha em um arquivo de texto.

Caso de uso 9

Nesse último caso de uso o objetivo é salvar e carregar todas as informações presentes no sistema para garantir a persistência dos dados após finalizar o programa.

Para fechar e salvar os dados do sistema foi criado o método *fechaSistema*, em sua implementação é feito o *Forwarding* dos métodos *exportarFuncionarios*, *exportarProntuarios, exportarFarmacia* e *exportarBancoOrgaos* da classe *FileMannager*, todos esses métodos criam um arquivo de extensão *.dat* para salvar suas informações de forma serial.

Todos esses métodos utilizam o método *writeObject* da classe *ObjectOutputStream*, em consequência disso, as classes *BancoFuncionarios*, *BancoProntuarios, Farmacia* e *BancoDeOrgao* implementam a interface *Serializable* e serão salvos em seus respectivos arquivos, permitindo a recuperação dos estados desses objetos de forma modular. A escolha por esta modularização se dá para garantir a independência entre esses objetos.

Desta maneira o sistema é concluído e todos os dados podem ser carregados sempre que o sistema for inicializado e salvos quando for finalizado.