

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

LABORATORIO DE PROGRAMAÇÃO II

CARLOS WILSON

DANIYEL NEGROMONTE – MAT:

IGOR MATHEUS CASTOR DINIZ PINHEIRO – MAT: 114210164

MATHEUS GOMES MAIA – MAT: 114210417

DOCUMENTAÇÃO E DESCRIÇÃO DE DESIGN DA REDE SOCIAL +POP

CAMPINA GRADE – PB

20 DE NOVEMBRO DE 2015

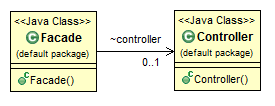
1 – Introdução

Foi solicitado aos alunos que cursam a disciplina de laboratório de programação II, a implementação de um código que definisse o funcionamento de uma rede social, a qual se denomina +Pop, para tal 10 passos foram repassados em forma de descrição e especificação de código, além disso foram disponibilizados códigos de teste. Tal rede social possui uma proposta diferente das demais redes sócias, buscando a iteração entre usuários com diferentes níveis de popularidade. Desse modo a característica marcante da rede social deve ser a popularidade dos usuários. Neste trabalho será descrito o passo a passo da criação do código bem como em quais teorias foram fundamentadas as decisões de design e padrão de codificação.

2 – Desenvolvimento

Para inicio de projeto, foi solicitado uma arquitetura estrutural baseado em Controller associado a Façade, onde a façade irá encapsular todo o subsistema em uma visão simplificada. O papel da façade é se fazer de interface de alto nível para assim descomplicar o acesso aos metodos. A Façade é responsável por repassar da melhor forma as chamadas de métodos para Controller, realizar impressões e capturar exceções. O uso de façade e controller ajuda a descomplicar o acesso aos métodos do programa e isso será útil à interface gráfica e aos testes. Somando-se a Façade, implementamos uma classe Controller, que por sua vez se torna responsável por delegar as chamadas e armazenar estruturas de dados. O Controller faz uma “ponte” entre a façade e as demais classes, é no Controller que se faz verificações básicas e se começa operar toda a lógica do sistema.

Para relacionar a Facade com o Controller em termos de código utilizamos uma composição que se trata de uma relação do tipo “Façade tem um Controller”, isso permite chamadas de métodos de controller dentro de Facade.



2.1 – Cadastro, Login e Logout de Usuários.

É solicitado no primeiro passo um código eficaz para cadastro de usuários, para isso se fez necessário a criação de uma classe chamada Usuário, com os respectivos parâmetros, e métodos necessários para tal classe inicialmente como um construtor e o métodos *gets e setters*. É importante ter em mente quem deverá cadastrar o usuário, a mesma tarefa não pode ser feita pelo próprio usuário, não faria sentido alguém que não está cadastrado, auto cadastrar-se. Vimos agora pouco que a Classe Controller fica responsável por controlar operações do programa, uma dessas operações será o cadastro de usuários, bem como o login e logout, além disso tomasse como base um dos padrões GRASP de design, o Creator, o qual é utilizado quando temos duas classes X e Y, e devemos atribuir a Y a tarefa de criar X quando Y registra X. O qual é exatamente o caso em questão.

Um fato importante sobre esta etapa é o reconhecimento de quando algum usuário está online, isso será importante porque foi requisitado que apenas um usuário esteja logado por vez, de modo que se um usuário A estiver logado, nenhum outro usuário poderá logar-se. Este reconhecimento é feito utilizando uma composição, associando Controller e Usuario, e a relação será do tipo Controller “tem” um usuarioLogado por vez.

2.2 – Pesquisar e atualizar Informações de Usuário.

Neste passo, deve-se utilizar mais dos padrões GRASP, o *Expert Information*, baseado em atribuir uma responsabilidade a classe que possui a informação necessária para preencher tal responsabilidade, neste caso quem tem as informações necessárias para os métodos que irão atualizar informações dos usuários, é o próprio usuário, isso nos leva a ter um baixo acoplamento, uma alta coesão, e as informações do Usuário permanecem “ocultas”, de modo que só a mesma, pode alterar. Controller fará apenas uma chamada destes métodos através da instancia do usuário que está logado no momento, que por sua vez será utilizado pela Façade.

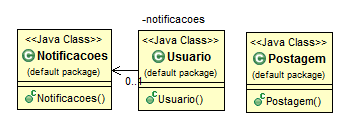
2.3 – Postar mensagens no mural.

Neste passo teremos a criação de mais uma classe, a classe Postagem com seus respectivos atributos e métodos relacionados. Foi requisitado a implementação de dois métodos que se correlacionam com uma postagem, contudo pelo padrão de *Expert Information*, esses métodos não devem estar dentro da classe postagem, os tais são o curtir e rejeitar um post, logo quem deve curtir ou rejeitar é um usuário, e numa uma própria postagem.

2.4 – Adicionar e remover amigos.

Para adicionar e remover um amigo, foi necessário o uso de Collections, neste caso, utilizou-se ArrayList, visto que tem-se uma lista que está em constante alteração. As listas criadas foram, lista de amigos, onde ficam os usuários os quais foram aceitos o pedido de amizade, e outra lista para armazenar todos os pedidos de amizade, a fim de que o usuário tenha acesso a todos os pedidos quando logar. Além disso foi criada uma composição dentro de Usuario com uma nova classe Notificações, através de *forwarding* foram delegados alguns métodos lógicos para dentro desta classe, sendo esses métodos, apenas os que, pelo conceito de *Expert Infromation* (já definido em linhas anteriores) fazem sentido estarem lá. Nela ainda são armazenadas notificações, sejam elas de curtidas ou de pedidos de amizade. É interessante pensar que até agora temos seguido padrões GRASP de design. Visto que temos um baixo acoplamento de código, onde as classes não estão interligadas a ponto de que pequenas alterações gerem grandes dores de cabeço, em todo o código, e ainda temos uma alta coesão, pelo motivo de estarmos fornecendo responsabilidades certas as classes certas.

Se por acaso o método adicionaAmizade( ) estivesse dentro de postagem, teríamos um código incoerente, prejudicando a funcionalidade de todo o resto do código.



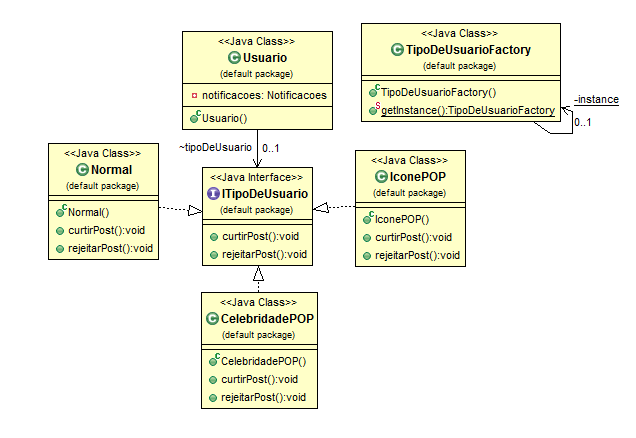
2.5 e 2.6 – Popularidade e Mudanças nos tipos de Usuário.

Conceitos muito importantes foram utilizados nestes passos, os passos 5 e 6 foram implementados juntos. Inicialmente percebesse que irão existir tipos de usuário diferentes de acordo com o numero de popularidade (atributo de cada usuário), cada tipo de usuário terá um tipo de comportamento para os métodos rejeitarPost e curtirPost, isso remete a ideia de métodos polimórficos.

Importante perceber que os tipos de Usuário (Normal, Ícone e Celebridade) não serão filhas de Usuário, ou seja, não teremos uma relação de herança, pois no passo 6 é definida uma mudança dinâmica no tipo de Usuário, e herança não permite essa dinamicidade. Logo, utilizamos um tipo de design denominado *Strategy* , que definem uma família de algoritmos, encapsulando cada um, sendo eles intercambiáveis, permitindo que os algoritmos variem independentemente das classes que a utilizam, em termos de código, uma variável do tipo *Interface* é instanciada em uma classe concreta, e através do retorno de uma *Factory*, classes concretas que implementam a mesma interface poderão ser instanciadas. Para utilizar tal design foi criado uma variável em Usuário que inicializa um tipo de interface, a ITipoDeUsuario, tal variável só é instanciada quando a FactoryTipoDeUsuario é chamada atribuindo uma classe concreta a tal variável em Usuário, neste passo as classes concretas se referem exatamente aos tipos de Usuario, e os métodos da interface, os quais as classes concretas teriam que sobre escrever são o curtir e o rejeitarPost.

Neste ponto encontramos dois novos conceitos, o primeiro deles de *Interface*, que serve como um “contrato” onde todas as classes que a implementarem terão que sobre-escrever seus métodos e o outro é o de Factory, design que delega a obrigação de “construção” a uma classe especifica. Pode ainda ser extraído mais um conceito neste passo, que é o de *Classe Singleton* , uma classe que não precisa ser instanciada, ela “trabalha” pra ela mesmo, inclusive se auto-instanciando, podendo ser acessada de qualquer classe que deseje, em nosso código unimos o conceito de *Strategy, Factory, Interface e classe Singleton* neste passo.

Segue abaixo um figura representativa do caso citado.



2.7 – Ranking de Usuarios e Trending Topics.

Neste ponto, outra classe *singleton* foi criada, a BancoHashtags, visto que varias classes precisavam de um acesso a mesma, e não faria muito sentido instancia-la em tantos lugar, desse modo através de forwarding foram delegados métodos específicos que trabalham e operam com as hashtags. Uma nova Collection surgiu neste ponto, a *HashMap*, que possui uma chave e um valor associado. Utilizamos esse tipo de collection para facilitar a ordenação do treeding topics, associando uma hashtag ao valor de vezes que a mesma se repete. Outra “novidade” apresentada, é a implementação de interfaces de ordenação, essenciais para quando se tem fins de ordenação, sendo elas a *Comparable<T>* e *a Comparator<T>*, ao serem implementadas os métodos devem ser sobre escritos da maneira que se deseja a ordem, neste caso, a ordem se baseará nos pontos x2p.

2.8 – Feed de Noticias.

Não há nenhuma grande novidade neste passo, apenas foram utilizados conceitos já descritos anteriormente. Neste ponto houve a criação de uma nova classe, a FeedNoticias, e por sua vez através de composição foi criado uma instancia em Usuário, já que “usuário tem um feed de noticias”. Métodos específicos para atualizar esses feeds foram criados.

2.9 e 2.10 – Importar e Exportar Posts em Arquivos e persistência de Arquivos .

Por fim, um conhecimento sobre manipulação de arquivos teve que ser exercido. A implementação desse passo, apenas mostra a importância da persistência de informações, de modo que o usuário possa deslogar, mas suas informações permanecerem.

3.0 – Adicionais.

Priorizando ainda o baixo acoplamento, e percebendo que a essa altura a classe Controller, estava responsável por muitas coisas, criamos algumas classes auxiliares, e por meio de composição e *forwarding*, diminuiu-se o acoplamento da classe controller.

Foram criadas as classes, AuxiliarAmizades, através de GRASP – *Expert Information*, delegamos para essa classe métodos relacionados a operações logicas que envolviam amizade entre os usuários. AuxiliarOperações, responsável pela parte de operações da rede social como login, logout e cadastrar usuário. AuxiliarValidadores, que tem por responsabilidade verificar a validade de e-mails, e datas, não permitindo e-mails inválidos e datas invalidas sejam cadastradas no sistema. E por fim uma classe BancoDeDados, que terá todos os Usuário da rede social, e todas as suas informações bem como métodos responsáveis por capturar essas informações.

