Relatório - Jackut

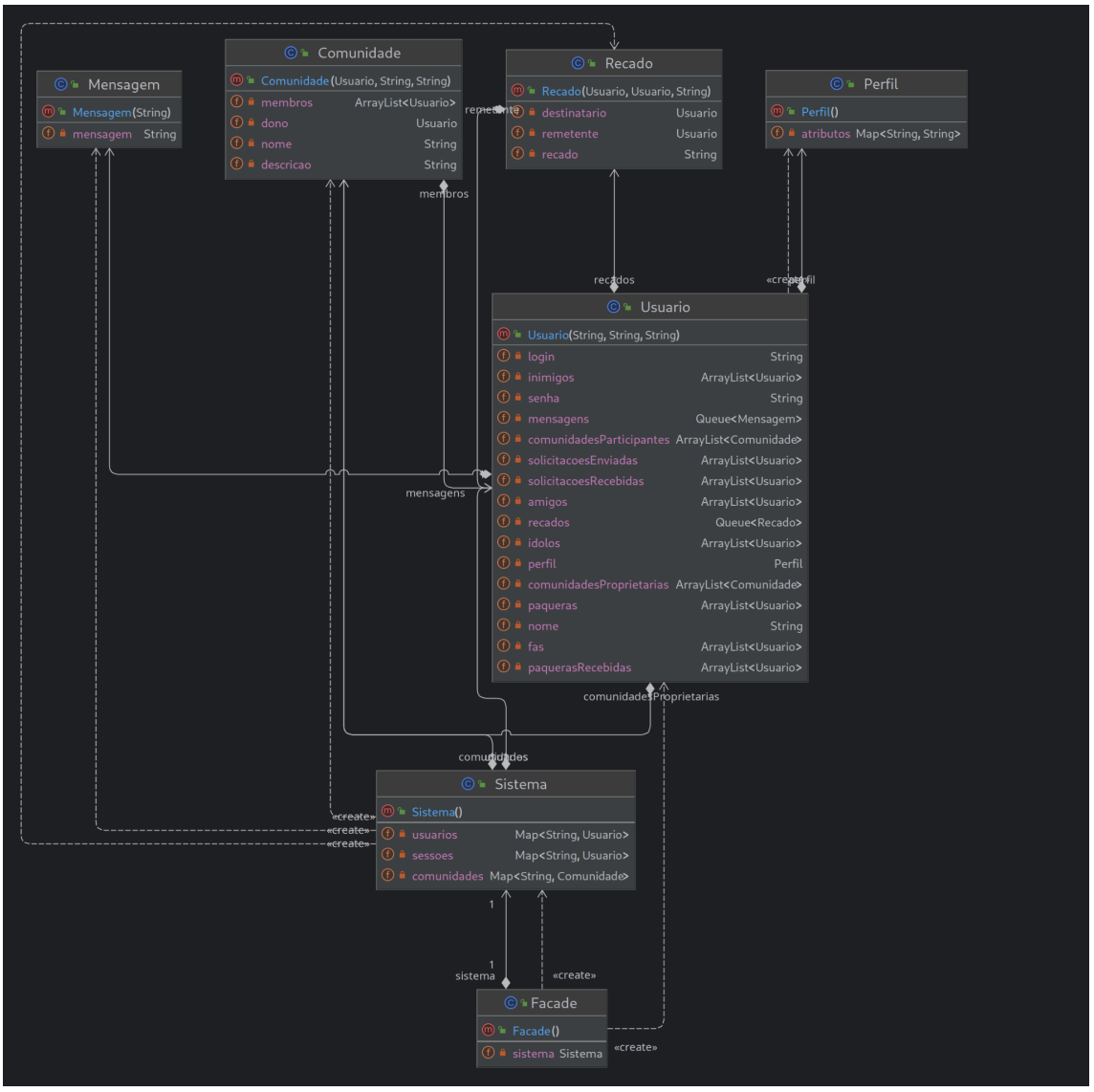
Alunos: João Lucas e Mateus dos Santos

Introdução:

Neste relatório, iremos descrever sobre o processo que fizemos para refatorar o código da milestone 1 e também detalhar o que criamos de novo para a implementação do milestone 2. Também iremos argumentar o porquê da escolha do nosso design, explicando as decisões que tivemos tanto do refatoramento quanto das novas funcionalidades. E mais uma vez, criamos o diagrama de classe, agora atualizado com o estado atual do projeto.

Sobre o banco de dados, continuamos utilizando um arquivo de texto ao invés de um banco de dados relacional, pois acreditamos que mesmo com as alterações e adições deste novo milestone, ainda não se prova necessário a utilização de um banco.

Desenvolvimento:



No Jackut, criamos essencialmente 7 classes principais:

- **Facade:**

Facade é a classe que irá fazer a comunicação com o easyAccept, contendo todos os métodos que o easyAccept reconhece. Além disso, a classe possui uma instância da classe Sistema, que iremos abordá-la logo após. Os métodos da classe Facade apenas chamam outros métodos das classes que irão desempenhar aquela determinada ação, livrando-se de qualquer regra de negócio para ela mesma.

- **Sistema:**

A classe Sistema é responsável por gerenciar as variáveis que persistem em um nível de estado de "sessão" (diferente do nível de "banco", que guarda dados entre sessões), sendo elas as sessões dos

usuários logados, a lista de usuários cadastrados, e a lista de comunidades criadas no sistema. É nesta classe que usuários que foram validados a estarem logados, terão suas sessões salvas no sistema.

Para essas variáveis, foram utilizados *HashMaps*, identificando cada variável com seu identificador único para a localização posterior de cada elemento. Por exemplo, usuários cadastrados têm suas informações atreladas ao seu login, que é único, para quando a classe Sistema precisar acessar esse usuário, só precisará saber o login do usuário para conseguir as informações do usuário.

Podemos destacar as seguintes funções da classe Sistema:

- ***Sistema:***

  É no construtor da classe Sistema em que é utilizado as *classes utilitárias* (que abordaremos logo após) para a leitura dos arquivos de texto, recuperando todas as informações dos usuários e das comunidades que foram salvas no momento em que o sistema foi desligado.

- ***abrirSessão:***

  É nesse método que é atribuído àquele usuário que foi corretamente logado, o seu id de sessão. Optamos nesse projeto em utilizar, em cada usuário que está com sua sessão ativa, um id no formato UUID *(Universally Unique IDentifier)*, uma vez que, para cada usuário logado no sistema, ele deve possuir um id único que irá identificá-lo no sistema. Então, após ser gerado um UUID aleatório, atrelamos ele a instância do usuário logado.

- ***removerUsuário:***

  No método de remover usuário, um usuário é desligado do sistema, removendo o mesmo de todas as instâncias em que ele é encontrado no decorrer do sistema, como em comunidades, relacionamentos e recados. Apesar de, nos testes do *easyAccept*, não ter sido testado de fato, o "***removerUsuário*** ele remove o usuário de todos os seus relacionamentos, até mesmo de possíveis solicitações de amizades realizadas.

- **Usuário:**

A classe de Usuário é a classe onde guardamos as informações mais importantes do usuário. É nela que guardamos seu nome, login, senha, seu perfil, seus relacionamentos, comunidades que participa e em que é o dono, como também suas mensagens e seus recados não lidos. Nesses últimos, é importante destacar que:

❖ O ***perfil*** é uma classe à parte, para ser mais fácil de gerenciar seus atributos, que podem ser quaisquer;

❖ Todos seus ***relacionamentos*** são *ArrayList*s de usuários, uma vez que não se via necessário criar uma complexidade no código para tentar generalizar tantas funcionalidades distintas que cada uma tinha, criando uma classe distinta para cada uma, ou muito menos criar uma classe geral para que todas herdem da mesma;

❖ Tanto as ***solicitacaoEnviadas*** como ***solicitacoesRecebidas*** *são ArrayLists de usuários também; e, tanto os* ***recados*** como as ***mensagens*** são uma fila de suas respectivas classes à parte, *Recado* e *Mensagem*.

- **Perfil:**

A classe Perfil é utilizada pelo usuário, responsável por guardar seus atributos, que, pela estrutura solicitada, pode ter qualquer chave e valor. Portanto, foi criada uma classe para gerenciar essa relação entre os atributos e o usuário. Seus métodos são simples getters e o seu construtor.

- **Comunidade:**

A classe de ***Comunidade*** é a que gerencia os dados mais importantes que são necessários para gerir uma instância de uma comunidade.

Uma comunidade possui nome, descrição criador e membros, portanto, nome e descrição são Strings, criador é um Usuário e membros é uma lista de Usuários.

Não foi necessário atrelar à comunidade, uma lista de mensagens que são enviadas à mesma, uma vez que a comunidade só serve de intermediário para os usuários enviarem de forma coletiva, as mensagens da comunidade. Portanto, vale ressaltar que, se em outra implementação do milestone, onde seria necessário resgatar o *log* da comunidade e salvar as mensagens enviadas à ela, seria necessária a mudança da classe ***Mensagem***, e adicionar uma lista delas à classe *Comunidade.*

Em seu *construtor*, a classe Comunidade já adiciona seu criador à lista de membros, uma vez que esse é um comportamento esperado de uma comunidade, essa regra de negócio ficou por responsabilidade da própria classe, que neste caso foi implementada no seu construtor.

- **Recado/Mensagem:**

Ambas as classes de recado e mensagem são utilizadas pelo ***Usuário***, de forma que o usuário possui uma fila de cada um, onde cada recado possui seu *remetente*, *destinatário* e *recado*, e onde cada mensagem possui apenas *mensagem* como atributo, uma vez que é utilizada no contexto de comunidades, não foi necessário guardar quem foi o usuário remetente.

Nós optamos por utilizar da estrutura de dados ***fila*** pois, uma vez que a estrutura do easyAccept é a de, após adicionar n *recados/mensagens,* ler os *recados/mensagens* a partir do primeiro que foi adicionado, esse é o clássico padrão de FIFO *(First In, First Out)*, portanto, escolhemos de fazer uma Queue de *recados/mensagens* para o usuário. Esta classe possui como métodos integrantes apenas getters para o uso da instância do usuário ou da comunidade.

Classes utilitárias (utils):

As classes utilitárias, localizadas na pasta ***utils*** desse projeto foram criadas principalmente para refatorar partes do código que o ***Sistema*** realizava, mas que não era necessariamente utilizada apenas por ela. Portanto, criamos classes estáticas que desempenhavam funções genéricas. Por isso, criamos três classes utilitárias:

- **UtilsString:**

Essa classe serve para formatar a string em um padrão extremamente utilizado pelo projeto, que é o padrão ***{valor1,valor2,valor3}*** onde é formatado várias strings dentro de chaves, separadas por "," e sem espaço entre elas. Nesta classe, é criado um método que recebe um *ArrayList* de um tipo genérico ***<T>***, onde esse tipo precisa apenas sobrescrever a função ***toString*** para ser convertida para o formato que queremos.

No caso, essa função é utilizada pelas classes: **Usuário** e **Comunidade**, onde no caso de *Usuário* ela é utilizada em diversos contextos, como o de retornar os *fãs*, as *paqueras* ou os *amigos* do usuário, e para cada uma delas, foi feito um **@Override** na função *toString* para retornar o atributo delas em que seria utilizado nessa função de formatação de string.

- **UtilsFileWriter/UtilsFileReader:**

Essas duas classes são utilizadas majoritariamente pelo *Sistema* para realizar a leitura e escrita dos arquivos. Neles é feito uma refatoração para com que os métodos estejam desacoplados um do outro, facilitando

assim a adição de novas persistências de dados, criando somente a lógica por trás da escrita e leitura deste arquivo.

É também utilizado nessas classes o **tipo** personalizado criado na pasta ***types***, que é o ***RelacoesTypes***. Esse ***enum*** foi criado para facilitar na criação e leitura dos relacionamentos dos usuários, uma vez que cada um é escrito no banco de dados, com um *usuário1* e um *usuário2* e um tipo de relacionamento, que está dentro desse ***enum.*** A criação desse tipo também facilita em uma futura adição de novas relações para a criação/leitura desses tipos.

Exceções:

As exceções criadas nesse projeto, são simples exceções herdadas de RuntimeException que realizam a exceção específica que é às dada. Porém, podemos citar exceções em específico que conseguem resumir funcionalidades que seriam feitas por diversas exceptions com funcionalidades muito similares, trazendo uma duplicidade alta de código:

- **LoginSenhaInvalidosException:**

Essa exceção cuida de 3 exceções ao mesmo tempo: “Login inválido.”, “Senha inválida.” e “Login ou senha inválidos.” Para fazer isso, colocamos uma string de tipo para determinar qual exceção seria feita ao usuário, sendo “login” para a de login inválido, “senha” para a de senha inválida, e “any” para as duas, uma vez que, no contexto da aplicação, foi pedido que, ao errar qualquer um dos campos no ***abrirSessao***, não ser informado qual foi o inválido, por isso o uso do “any”.

- **UsuarioAutoRelacaoException:**

Essa exceção cuida de todas as exceções de auto relacionamentos possíveis. Ele tem como parâmetro um tipo de relação, e diferencia a mensagem por esse tipo. O tipo “amizade” é diferente dos outros pois, pelos testes implementados pelo easyAccept, amizade tem um retorno de exceção ligeiramente diferente dos demais, mas a exceção trata isso com uma condicional feita com *ternary operator* para diferenciar amizade dos demais relacionamentos como “*inimigo*”, “*paquera*” ou “*fã*”.

- **UsuarioJaTemRelacaoException:**

Essa exceção, similar a exceção de “*UsuarioAutoRelacao*”, ela cuida de todas as exceções que falam sobre o usuário já possuir uma relação com um outro usuário. A forma que isso é feito é que, reconhecendo que os testes esperam um determinado comportamento das exceções, foi dado um parâmetro em que a string restante é completamente idêntica, mudando apenas o tipo do relacionamento em que a exceção é lançada.