Documentação do Projeto Jackut

Introdução

O projeto Jackut foi desenvolvido para simular uma rede social com funcionalidades básicas, como criação de contas, edição de perfis, adição de amigos e envio de mensagens. A implementação foi baseada nos requisitos fornecidos pelas atividades propostas, com foco em modularidade, clareza e extensibilidade. A seguir, explico como os códigos foram construídos e as escolhas de design feitas para atender às atividades.

Estrutura do Projeto

O projeto foi dividido em três partes principais:

1. Classes principais:
    - Facade: Responsável por gerenciar as operações principais do sistema, como criação de usuários, gerenciamento de sessões, adição de amigos e envio de mensagens.
    - Users: Representa os usuários do sistema, armazenando informações como login, senha, atributos do perfil, amigos e mensagens.
2. Exceções personalizadas:
    - Criadas para lidar com erros específicos, como login inválido, usuário não encontrado, ou atributo não preenchido. Isso melhora a clareza do código e facilita a depuração.
3. Testes (EasyAccept):
    - Scripts de teste foram fornecidos para validar as funcionalidades implementadas. Cada funcionalidade foi desenvolvida para atender aos requisitos descritos nos testes.

Implementação e Escolhas de Design

1. Classe Facade

A classe Facade atua como a camada de controle do sistema, expondo métodos que encapsulam a lógica de negócios. Essa abordagem segue o padrão de design Facade, que simplifica a interação com subsistemas complexos.

- Gerenciamento de usuários e sessões:
    - Os usuários são armazenados em um mapa (Map<String, Users>), onde a chave é o login do usuário.
    - As sessões são gerenciadas por outro mapa (Map<String, String>), que associa um ID de sessão ao login do usuário.
- Persistência de dados:
    - Os dados dos usuários são salvos e carregados de um arquivo (users.dat) usando serialização. Isso garante que as informações sejam mantidas entre execuções do programa.
- Validação de entradas:

    - Métodos como validateLogin e validatePassword foram criados para garantir que os dados fornecidos pelos usuários sejam válidos antes de prosseguir com as operações.
- Exceções personalizadas:
    - Exceções como UserNotFoundException, InvalidLoginException e AttributeNotFilledException foram usadas para lidar com erros específicos, melhorando a clareza e a manutenção do código.

2. Classe Users

A classe Users representa os dados e comportamentos de um usuário no sistema. Ela foi projetada para ser serializável, permitindo que seus dados sejam persistidos.

- Atributos do perfil:
    - Os atributos do perfil são armazenados em um mapa (Map<String, String>), permitindo que novos atributos sejam adicionados dinamicamente.
- Gerenciamento de amigos:
    - Amigos são armazenados em uma lista (List<String>), enquanto pedidos de amizade pendentes são mantidos em outra lista (List<String>). Isso facilita a implementação de funcionalidades como envio e aceitação de pedidos de amizade.
- Mensagens:
    - As mensagens são armazenadas em uma fila (Queue<String>), garantindo que sejam lidas na ordem em que foram recebidas.
- Exceções personalizadas:
    - Exceções como AttributeNotFilledException e MessageException são usadas para lidar com casos como tentativa de acessar um atributo não preenchido ou ler mensagens de uma fila vazia.

3. Exceções Personalizadas

As exceções personalizadas foram criadas para substituir exceções genéricas (RuntimeException) e fornecer mensagens de erro mais específicas. Isso melhora a experiência do desenvolvedor e facilita a depuração.

- Exceções implementadas:
    - UserNotFoundException: Lançada quando um usuário não é encontrado.
    - InvalidLoginException: Lançada quando o login é inválido.
    - InvalidPasswordException: Lançada quando a senha é inválida.
    - AttributeNotFilledException: Lançada quando um atributo do perfil não está preenchido.
    - FriendshipException: Lançada para erros relacionados a amizades, como tentativa de adicionar um amigo já existente.
    - MessageException: Lançada para erros relacionados a mensagens, como tentativa de ler mensagens de uma fila vazia.

4. Testes (EasyAccept)

Os scripts de teste fornecidos foram usados como base para a implementação das funcionalidades. Cada funcionalidade foi desenvolvida para atender aos cenários descritos nos testes.

- User Story 1 (Criação de conta):

    - Implementação do método createUser para criar novos usuários.
    - Validação de login e senha para garantir que os dados sejam válidos.
- User Story 2 (Criação/Edição de perfil):
    - Implementação dos métodos editProfile e getUserAttribute para permitir que os usuários editem e acessem atributos do perfil.
    - Uso de AttributeNotFilledException para lidar com tentativas de acessar atributos não preenchidos.
- User Story 3 (Adição de amigos):
    - Implementação do método addFriend para gerenciar pedidos de amizade.
    - Uso de listas separadas para amigos e pedidos pendentes.
    - Validação para evitar que um usuário adicione a si mesmo como amigo.
- User Story 4 (Envio e leitura de recados):
    - Implementação dos métodos sendMessage e readMessage para gerenciar o envio e leitura de mensagens.
    - Uso de MessageException para lidar com tentativas de ler mensagens de uma fila vazia.

Escolhas de Design

1. Modularidade:
    - A separação entre Facade e Users garante que a lógica de negócios e os dados do usuário sejam gerenciados de forma independente.
2. Extensibilidade:
    - O uso de mapas para atributos do perfil permite que novos atributos sejam adicionados sem alterar a estrutura da classe Users.
3. Clareza:
    - A documentação no estilo Javadoc foi adicionada para todos os métodos, facilitando a compreensão do código.
4. Tratamento de erros:
    - Exceções personalizadas foram usadas para lidar com erros específicos, tornando o código mais robusto e fácil de depurar.

Conclusão

O projeto Jackut foi implementado com base nos requisitos fornecidos, utilizando boas práticas de programação e design de software. A modularidade, extensibilidade e clareza foram priorizadas para garantir que o sistema seja fácil de entender, manter e expandir. Se houver necessidade de ajustes ou novas funcionalidades, a estrutura atual permite que isso seja feito de forma eficiente.

Diagrama do Projeto Jackut