Universidade Federal de Alagoas - UFAL

Instituto de Computação - IC
Disciplina: Programação Orientada a Objetos
Aluno: Josenilton Ferreira da Silva Junior

Relatório do Milestone 2

Maceió, 6 de Maio de 2025

Introdução

Este relatório apresenta uma análise detalhada do projeto Jackut, desenvolvido como parte da disciplina de Programação Orientada a Objetos. O projeto tem como objetivo implementar uma rede social com funcionalidades como criação de usuários, gerenciamento de comunidades, envio de mensagens e relacionamentos sociais.
O documento está estruturado da seguinte forma:

1. Avaliação: Apresenta os pontos positivos e negativos do design e implementação do projeto, com base no código e nas ferramentas utilizadas.
2. Refatoramento e Desenvolvimento Realizado: Descreve as melhorias realizadas no código e as funcionalidades implementadas.
3. Conclusão: Apresenta uma visão geral do trabalho realizado e uma avaliação objetiva do projeto.

Avaliação

Virtudes do Projeto

1. Estrutura Geral:
   O projeto apresenta uma estrutura funcional, com classes que representam os principais conceitos do sistema, como Users, Community e Facade.
2. Persistência de Dados:
   A persistência foi implementada utilizando serialização de objetos, permitindo que os dados sejam salvos e carregados entre execuções do sistema.
3. Interface de Testes:
   O projeto utiliza a ferramenta EasyAccept para a execução de testes de aceitação, o que facilita a validação das funcionalidades implementadas.
4. Padrão Facade:
   A classe Facade centraliza as operações do sistema, simplificando a interação com as funcionalidades principais.

Fraquezas do Projeto

1. God Class:
   A classe Facade concentra grande parte da lógica de negócio, o que a torna uma "God Class". Isso viola o princípio de alta coesão e dificulta a manutenção e extensão do sistema.
2. Falta de Modularidade:
   Apesar de algumas classes auxiliares, como UserFactory e CommunityFactory, o projeto carece de uma separação mais clara entre responsabilidades, resultando em alto acoplamento entre as classes.
3. Tratamento de Exceções:
   O tratamento de exceções é limitado. Em muitos casos, mensagens genéricas são lançadas, e não há um sistema robusto para lidar com erros de forma amigável ao usuário.
4. Persistência Limitada:
   A persistência foi implementada com serialização de objetos, o que dificulta a leitura e manipulação direta dos dados. O uso de formatos como JSON ou XML poderia trazer maior flexibilidade.
5. Falta de Padrões de Projeto:
   Embora o padrão Facade tenha sido utilizado, outros padrões, como Observer e Factory Method, foram implementados de forma limitada ou não explorados em todo o seu potencial.
6. Comentários e Documentação:
   Apesar de algumas classes possuírem comentários, a documentação é inconsistente e não cobre todos os aspectos do sistema.

Refatoramento e Desenvolvimento Realizado

No projeto, foram utilizados três padrões de projeto principais para estruturar e organizar o sistema: Facade, Observer e Factory Method.

---

1. Facade

O padrão Facade foi implementado na classe Facade, que centraliza todas as operações do sistema. Essa classe atua como uma interface única para os clientes (como a ferramenta de testes EasyAccept), encapsulando a complexidade das operações internas.
A Facade gerencia funcionalidades como:

- Criação de usuários e comunidades.
- Gerenciamento de sessões.
- Envio de mensagens e recados.
- Relacionamentos sociais (amigos, ídolos, paqueras e inimigos).
- Persistência de dados.

Esse padrão simplifica a interação com o sistema, fornecendo uma API coesa e reduzindo o acoplamento entre os clientes e as classes internas.

---

2. Observer

O padrão Observer foi utilizado para implementar notificações automáticas em comunidades. A classe Community atua como o sujeito (subject), enquanto a classe Users implementa a interface Observer.
Sempre que uma mensagem é enviada para uma comunidade, todos os membros registrados como observadores são notificados. Isso garante que os usuários recebam atualizações em tempo real sobre as atividades nas comunidades das quais participam.
Exemplo de uso:

- A classe Community possui métodos como addObserver e notifyObservers, que gerenciam e notificam os observadores.
- A classe Users implementa o método update, que é chamado quando uma notificação é recebida.

---

3. Factory Method

O padrão Factory Method foi utilizado para encapsular a criação de objetos complexos, como usuários e comunidades. As classes UserFactory e CommunityFactory foram criadas para centralizar a lógica de instanciamento.
Benefícios:

- Promove a reutilização de código.
- Facilita a manutenção, permitindo que alterações na lógica de criação sejam feitas em um único local.
- Reduz a complexidade da classe Facade, delegando a criação de objetos às fábricas.

Exemplo:

- A criação de um novo usuário é feita através do método UserFactory.createUser, que retorna uma instância da classe Users.
- A criação de uma nova comunidade é feita através do método CommunityFactory.createCommunity.

Esses padrões de projeto foram fundamentais para melhorar a organização, modularidade e extensibilidade do sistema, garantindo que novas funcionalidades possam ser adicionadas sem comprometer a integridade do código existente.

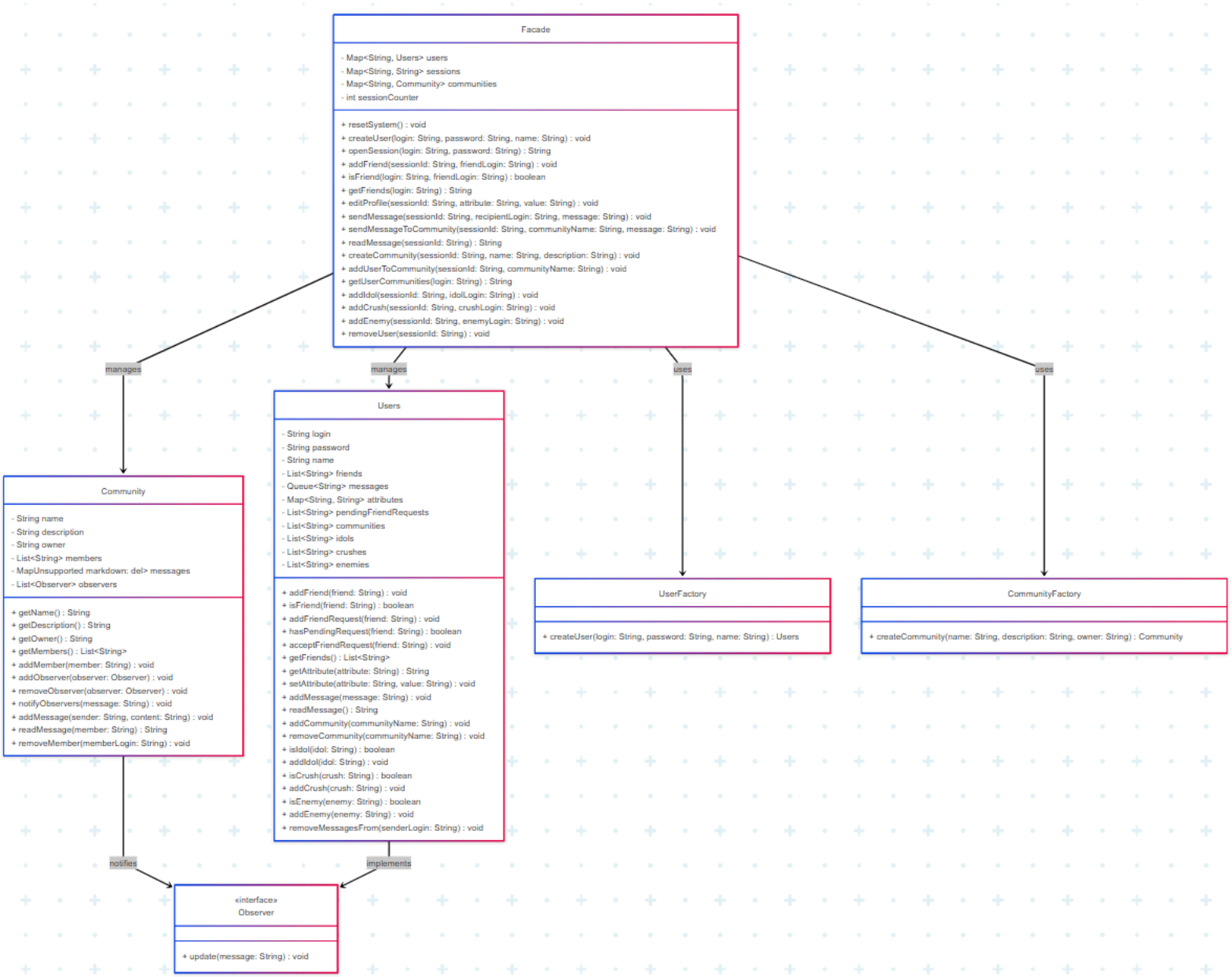
Refatoramento

1. Melhoria na Estrutura de Dados:
   Foram realizadas pequenas melhorias na organização das classes, como a separação de responsabilidades em métodos menores dentro da classe Facade.
2. Tratamento de Exceções:
   Algumas exceções foram revisadas para garantir mensagens mais claras, mas ainda há espaço para melhorias.
3. Persistência:
   Não houve mudanças significativas na persistência, que continua utilizando serialização de objetos.

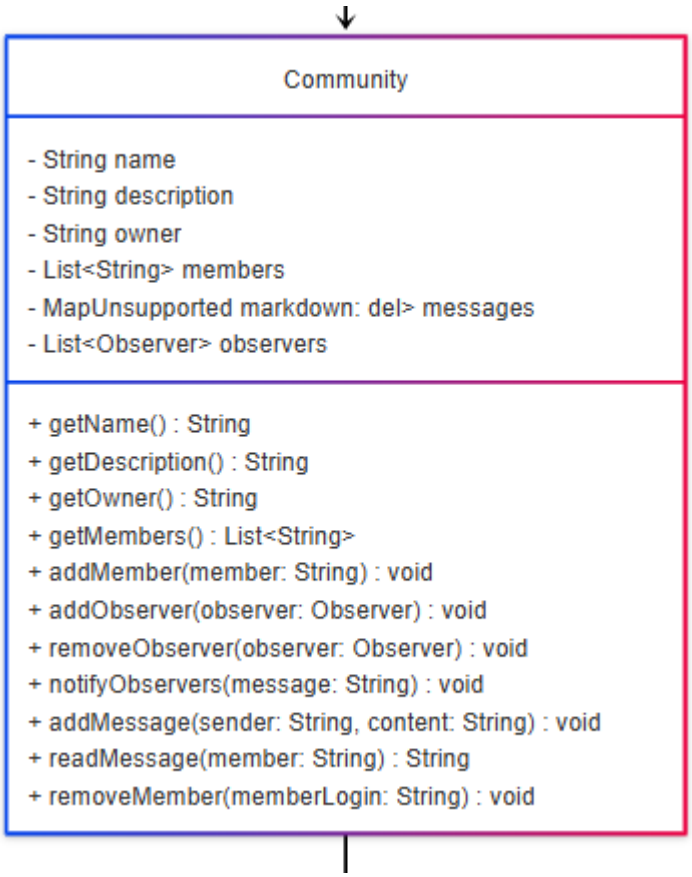
Funcionalidades Implementadas

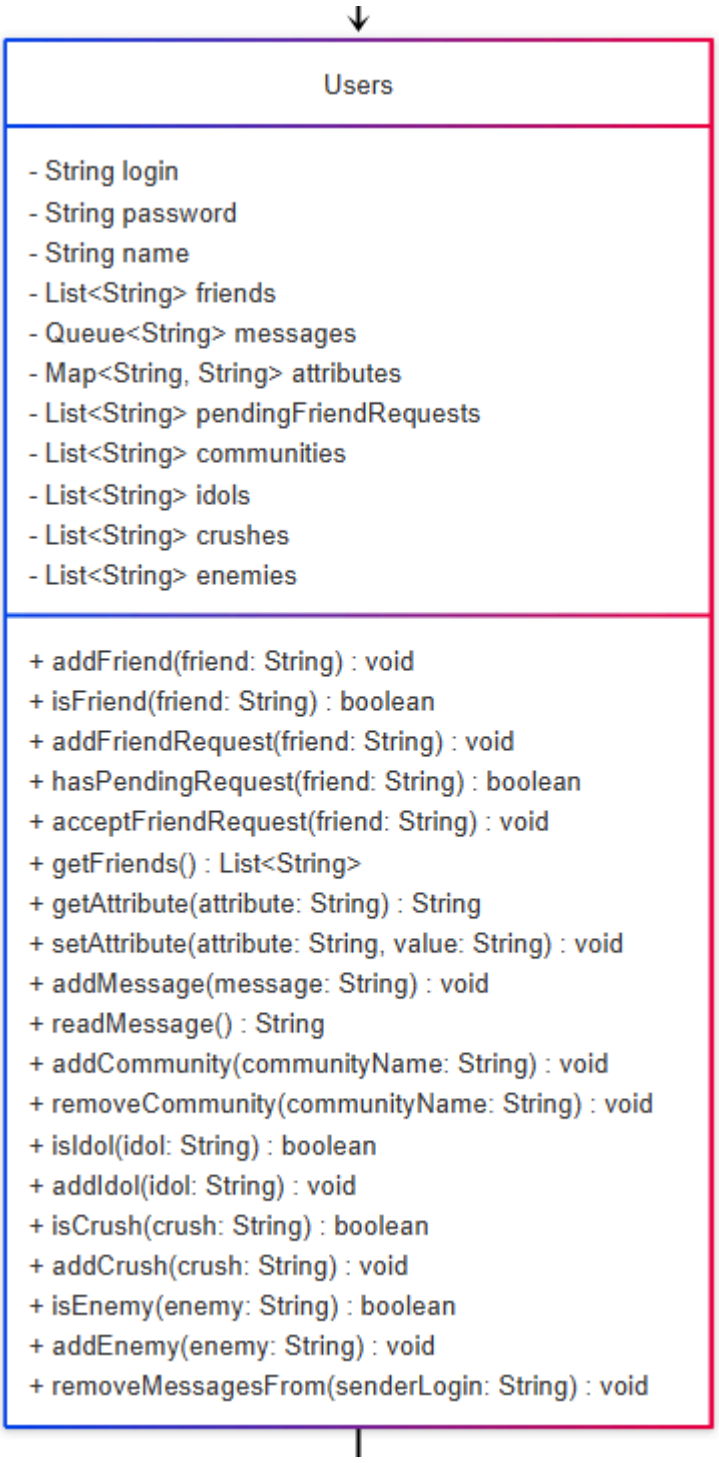
1. Gerenciamento de Comunidades:
    - Adição de métodos para criar comunidades, adicionar membros e enviar mensagens.
    - Implementação de notificações básicas para membros de comunidades.
2. Relacionamentos Sociais:
    - Adição de funcionalidades para gerenciar amigos, ídolos, paqueras e inimigos.
3. Mensagens:
    - Implementação de envio e leitura de mensagens entre usuários e comunidades.

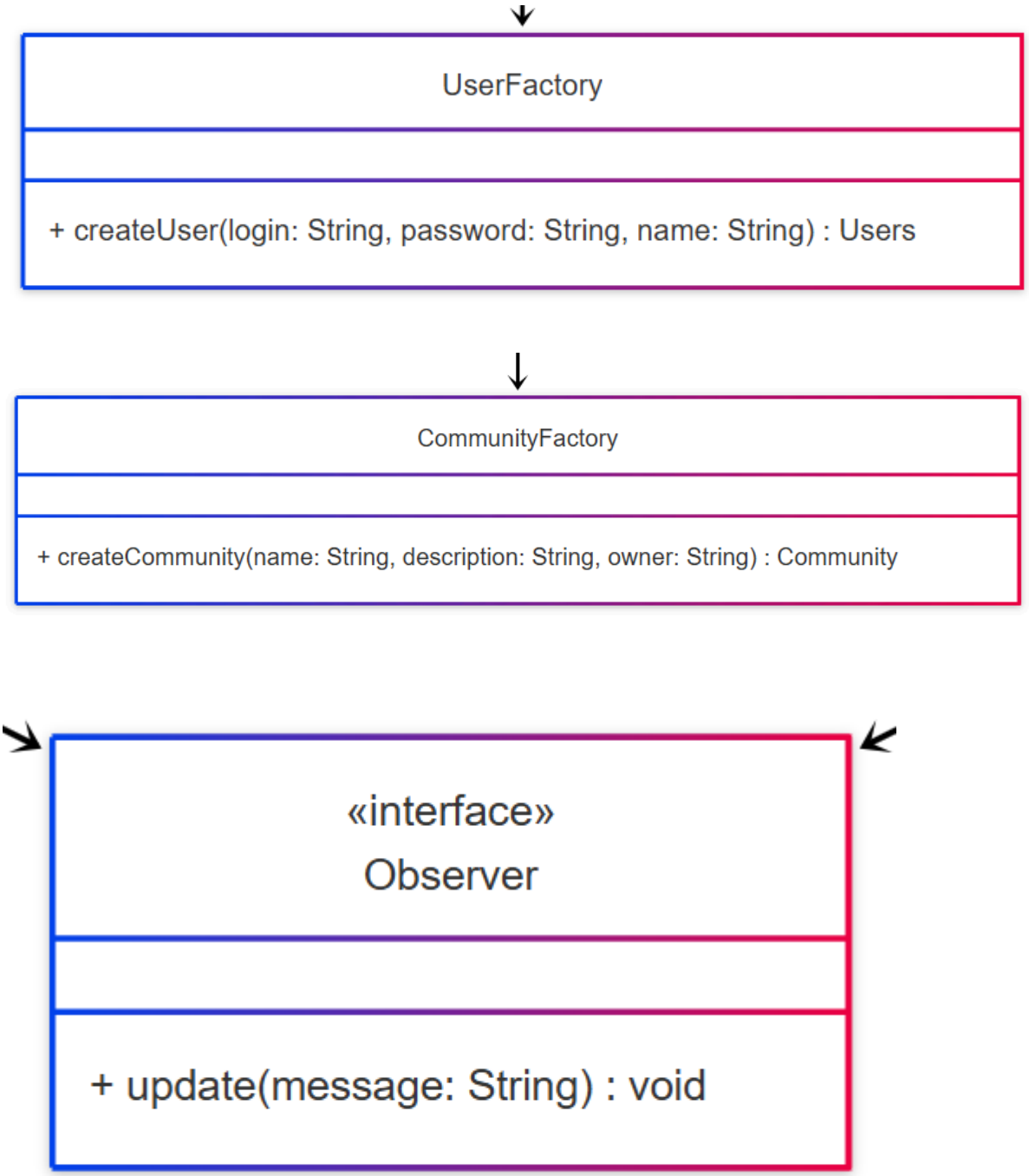
Diagrama de Classes:




Facade
- Map<String, Users> users
- Map<String, String> sessions
- Map<String, Community> communities
- int sessionCounter
+ resetSystem() : void
+ createUser(login: String, password: String, name: String) : void
+ openSession(login: String, password: String) : String
+ addFriend(sessionId: String, friendLogin: String) : void
+ isFriend(login: String, friendLogin: String) : boolean
+ getFriends(login: String) : String
+ editProfile(sessionId: String, attribute: String, value: String) : void
+ sendMessage(sessionId: String, recipientLogin: String, message: String) : void
+ sendMessageToCommunity(sessionId: String, communityName: String, message: String) : void
+ readMessage(sessionId: String) : String
+ createCommunity(sessionId: String, name: String, description: String) : void
+ addUserToCommunity(sessionId: String, communityName: String) : void
+ getUserCommunities(login: String) : String
+ addIdol(sessionId: String, idolLogin: String) : void
+ addCrush(sessionId: String, crushLogin: String) : void
+ addEnemy(sessionId: String, enemyLogin: String) : void
+ removeUser(sessionId: String) : void


Community
- String name
- String description
- String owner
- List<String> members
- MapUnsupported markdown: del> messages
- List<Observer> observers
+ getName() : String
+ getDescription() : String
+ getOwner() : String
+ getMembers() : List<String>
+ addMember(member: String) : void
+ addObserver(observer: Observer) : void
+ removeObserver(observer: Observer) : void
+ notifyObservers(message: String) : void
+ addMessage(sender: String, content: String) : void
+ readMessage(member: String) : String
+ removeMember(memberLogin: String) : void


Users
- String login
- String password
- String name
- List<String> friends
- Queue<String> messages
- Map<String, String> attributes
- List<String> pendingFriendRequests
- List<String> communities
- List<String> idols
- List<String> crushes
- List<String> enemies
+ addFriend(friend: String) : void
+ isFriend(friend: String) : boolean
+ addFriendRequest(friend: String) : void
+ hasPendingRequest(friend: String) : boolean
+ acceptFriendRequest(friend: String) : void
+ getFriends() : List<String>
+ getAttribute(attribute: String) : String
+ setAttribute(attribute: String, value: String) : void
+ addMessage(message: String) : void
+ readMessage() : String
+ addCommunity(communityName: String) : void
+ removeCommunity(communityName: String) : void
+ isIdol(idol: String) : boolean
+ addIdol(idol: String) : void
+ isCrush(crush: String) : boolean
+ addCrush(crush: String) : void
+ isEnemy(enemy: String) : boolean
+ addEnemy(enemy: String) : void
+ removeMessagesFrom(senderLogin: String) : void



Conclusão

O desenvolvimento do projeto Jackut foi uma experiência enriquecedora, que proporcionou a aplicação prática de conceitos fundamentais de Programação Orientada a Objetos. Apesar de algumas limitações iniciais, como o alto acoplamento e a centralização de responsabilidades na classe Facade, o projeto demonstrou ser funcional e extensível, atendendo aos requisitos das User Stories propostas.