UNIVERSIDADE FEDERAL DE ALAGOAS –
UFAL CAMPUS A. C. SIMÕES

CURSO DE CIÊNCIA DA COMPUTAÇÃO 2024.2

RIAN ANTONIO DA SILVA GAIÃO

PROGRAMAÇÃO ORIENTADA A OBJETOS: PROJETO JACKUT

MILESTONE 2

MAIO, 2025

UNIVERSIDADE FEDERAL DE ALAGOAS –
UFAL CAMPUS A. C. SIMÕES

CURSO DE CIÊNCIA DA COMPUTAÇÃO 2024.2

RIAN ANTONIO DA SILVA GAIÃO

Relatório referente ao experimento: Ex.: Um projeto de programação orientada a objetos , solicitado pelo professor Mário Horanzo. De cunho avaliativo para composição da média final.

PROGRAMAÇÃO ORIENTADA A OBJETOS: PROJETO JACKUT

MILESTONE 2

RESUMO

O projeto Jackut consiste em uma rede de relacionamentos inspirada no Orkut, desenvolvida em Java e seguindo princípios de POO. O sistema permite cadastro de usuários, gestão de perfis, amizades mútuas, envio de mensagens e participação em comunidades. Foi implementado uma arquitetura modular com facade para integração com testes automatizados (EasyAccept), garantindo 100% de aprovação nos testes de aceitação. Usando estrutura de dados e design de projetos orientados a objetos para maior eficiência e elegância do serviço. Resultados validaram a robustez do sistema em operações básicas e complexas.

Palavras chave: programação orientada a objetos, java, relatório, projeto

INTRODUÇÃO

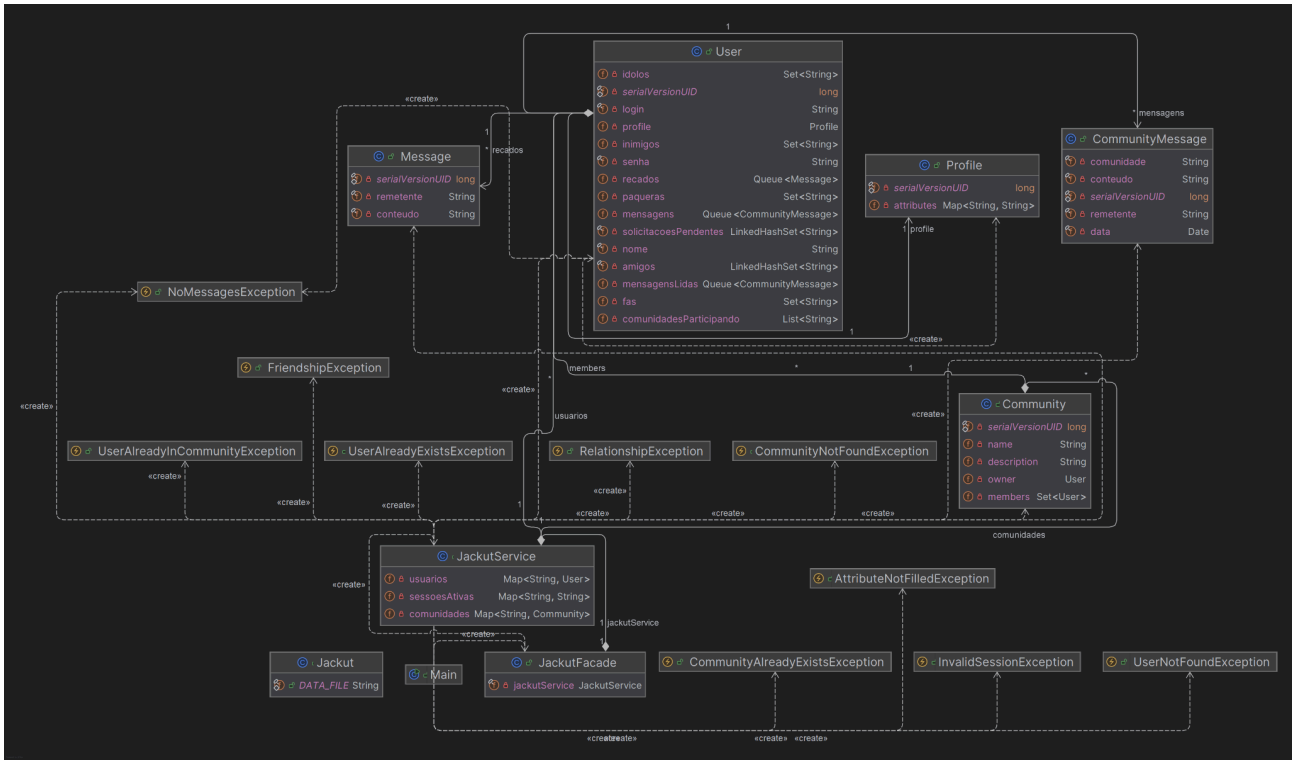
Este documento descreve o Milestone 2 do projeto Jackut, uma rede de relacionamentos desenvolvida em Java como parte de um trabalho acadêmico em POO (Programação Orientada a Objetos). O projeto será entregue em etapas (milestones), cada uma incorporando novas funcionalidades definidas em User Stories providenciados pelo orientador do projeto. Neste papel há breve elaborações das entidades escritas/adicionadas nesta milestone, focando mais no processo de decisões de design e diagrama de classes UML.

A milestone 2 do projeto Jackut cobre os user stories: US5 até o US9. (esses arquivos podem ser encontrados no diretório do projeto, junto com um javadoc com descrições de cada função e classe relevante.)

Link do repositório no github: https://github.com/ryanthecomic/Projeto-Jackut

VISÃO GERAL

DIAGRAMA UML



(Diagrama UML gerado a partir da ferramenta diagramas do Intellij, no centro do diagrama está a entidade User.)

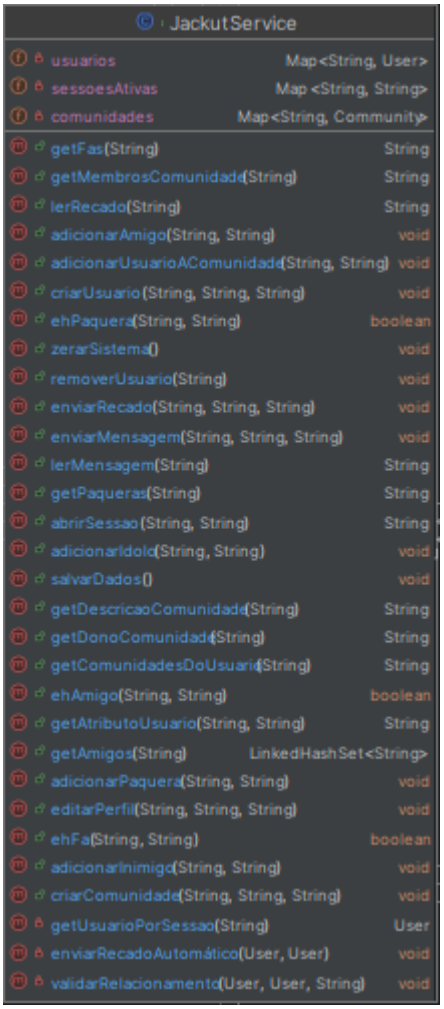
Função Main

A função Main gera uma interface simples que foi essencial para o desenvolvimento contínuo do projeto, como estamos na etapa de milestone 1, achei apropriado deixar na entrega da primeira milestone para testes mais simples dos comandos do easyaccept.jar.

Mudança na Separação de Responsabilidades

Anteriormente, o projeto teria vários serviços diferentes para cada contexto diferente: Serviço de Usuário, Serviço do Jackut, Serviço de Dados, Serviço de Comunidades e etc… Porém, ao decorrer do projeto, mostrou-se que isso seria desnecessário além da organização e não iria influenciar na experiência do usuário de forma alguma, já que todas as exceções estão separadas e a comunicação entre diversos serviços dentro de um só aplicativo não tem efeito nenhum na estruturação dos dados do projeto.

User Service -> Jackut Service



Motivação da Mudança:

UserService tinha um escopo limitado: Geria apenas operações de usuário (criação, login, perfil).

Dispersão de lógica: Funcionalidades como comunidades e mensagens ficariam em serviços separados (ex: CommunityService, MessageService), causando: Acoplamento entre serviços. Dificuldade na coordenação de operações que envolvem múltiplas entidades (ex: enviar mensagem a comunidade + atualizar histórico do usuário).Solução com JackutService fornece um serviço unificado: Agrega todas as operações do sistema em uma única classe:Usuários, comunidades, mensagens, sessões, amizades.

Vantagens: Menos acoplamento: Evita chamadas entre serviços.Persistência simplificada: Um único ponto para salvar/carregar dados.

Consistência: Operações complexas (ex: adicionarUsuário Comunidade) acessam diretamente os repositórios.

User Story 5_1 - Criação de Comunidades

Decisões:

- Nome da comunidade é a chave primária (deve ser único)
- O criador automaticamente se torna dono e primeiro membro
- Validações implementadas:
  - `CommunityAlreadyExistsException` para nomes duplicados
  - `InvalidSessionException` para sessões inválidas

Implementação:

```
public void criarComunidade(String idSessao, String nome, String descricao) {
    User dono = validarSessao(idSessao);
    if (comunidades.containsKey(nome)) {
        throw new CommunityAlreadyExistsException();
    }
    comunidades.put(nome, new Community(nome, descricao, dono));
}
```

User Story 6_1 - Adição de Membros a Comunidades

Decisões:

- Membros podem ser adicionados a comunidades existentes
- Ordem de exibição: dono primeiro, depois ordem de adição
- Validações:
  - `CommunityNotFoundException` para comunidades inexistentes
  - `UserAlreadyInCommunityException` para membros duplicados

Implementação:

```
public void adicionarUsuarioAComunidade(String idSessao, String nomeComunidade) {
    Community comunidade = validarComunidade(nomeComunidade);
    User usuario = getUsuarioPorSessao(idSessao);
    if (comunidade.getMembers().contains(usuario)) {
        throw new UserAlreadyInCommunityException();
    }
    comunidade.addMember(usuario);
}
```

User Story 7_1 - Mensagens para Comunidades

Decisões:

- Qualquer usuário pode enviar mensagens para comunidades públicas

- Mensagens são distribuídas para todos os membros

- Separação clara entre:

  - `Message` (recados privados entre usuários)

  - `CommunityMessage` (mensagens para comunidades)

Implementação:

```
private Map<String, Queue<CommunityMessage>> mensagensPorUsuario =
new HashMap<>();

public void enviarMensagem(String idSessao, String comunidade,
String conteudo) {
    Community comunidadeObj = validarComunidade(comunidade);
    User remetente = getUsuarioPorSessao(idSessao);

        CommunityMessage msg = new CommunityMessage(comunidade,
remetente.getLogin(), conteudo);
    comunidadeObj.getMembers().forEach(membro -> {
        mensagensPorUsuario
                    .computeIfAbsent(membro.getLogin(), k -> new
LinkedList<>())
            .add(msg);
    });
}
```

User Story 8_1 - Relacionamentos Avançados (Fã-Ídolo, Paquera, Inimigo)

Usuários agora têm várias listas de tipos diferentes de relacionamentos.

Decisões:

Tipos de Relacionamento:

- Fã-Ídolo: Público e unilateral (ex.: usuário A segue B, mas B não segue A).
- Paquera: Privado e mútuo (notificação automática quando ambos se adicionam).
- Inimigo: Bloqueia qualquer interação futura.

Validações:

- Auto-relacionamento proibido (ex.: não pode ser fã de si mesmo).
- Inimizade prevalece sobre outros relacionamentos.

Exemplo de Implementação (Paquera):

```
public void adicionarPaquera(String idSessao, String paqueraLogin)
{
    User usuario = getUsuarioPorSessao(idSessao);
        User alvo = validarUsuario(paqueraLogin); // Lança
UserNotFoundException

    if (usuario.ehPaquera(paqueraLogin)) {
        throw new RelationshipException("Usuário já é paquera!");
    }
    usuario.adicionarPaquera(paqueraLogin);

    // Notificação automática se for mútuo
    if (alvo.ehPaquera(usuario.getLogin())) {
        enviarRecadoAutomático(usuario, alvo); // "X é seu paquera
- Recado do Jackut"
    }
}
```

User Story 9_1 - Remoção de Contas

Limpeza Completa:

- Remove o usuário de todas as comunidades (como dono ou membro).
- Apaga mensagens, relacionamentos e sessões associadas.

Persistência:

- Atualiza o arquivo de dados após a remoção.

Implementação:

```
public void removerUsuario(String idSessao) {
    User usuario = getUsuarioPorSessao(idSessao);
    String login = usuario.getLogin();

    // 1. Remove de comunidades
    for (Community comunidade : comunidades.values()) {
        if (comunidade.getOwner().equals(usuario)) {
                comunidades.remove(comunidade.getName()); // Exclui
comunidades onde é dono
        } else {
            comunidade.getMembers().remove(usuario); // Remove como
membro
        }
    }

    // 2. Limpa relacionamentos em outros usuários
    for (User outroUsuario : usuarios.values()) {
        outroUsuario.getAmigos().remove(login);
        outroUsuario.getPaqueras().remove(login);
        // (...)
    }
       usuarios.remove(login); // Remove do sistema
    salvarDados(); // Persiste as alterações
}
```

Persistência de Dados agora salva mensagens e comunidades.

Implementação:

```
public void salvarDados() {
    Map<String, Object> dados = new HashMap<>();
    dados.put("usuarios", usuarios);
    dados.put("comunidades", comunidades);
    dados.put("mensagens", mensagensPorUsuario);
    Jackut.salvar(dados); // Serializa para arquivo
}

public void carregarDados() {
    Map<String, Object> dados = Jackut.carregar();
    if (dados != null) {
        this.usuarios = (Map<String, User>) dados.get("usuarios");
                    this.comunidades   =   (Map<String,   Community>)
dados.get("comunidades");
    }
}
```

Conclusão

O Projeto Jackut, desenvolvido como parte da disciplina de Programação Orientada a Objetos, demonstrou a aplicação prática de princípios fundamentais de POO, como encapsulamento, coesão, baixo acoplamento e reutilização de código. Através da implementação das User Stories 5 a 9, foram incorporadas funcionalidades essenciais para uma rede social, incluindo:

- Gestão de Comunidades (US5 e US6):Criação de comunidades com donos e membros.Validações robustas para evitar duplicações e garantir consistência

- .Mensagens em Comunidades (US7):Sistema de envio e distribuição de mensagens para todos os membros.Separação clara entre mensagens privadas e públicas.

- Relacionamentos Avançados (US8):Implementação de fãs-ídolos, paqueras e inimigos.Lógica de notificação automática para paqueras mútuas.Remoção de Contas

- (US9):Limpeza completa de dados associados ao usuário.Atualização em cascata para manter a integridade do sistema.