UNIVERSIDADE FEDERAL DE ALAGOAS — UFAL

Campus A. C. Simões

Ciência da Computação

Helio José Ribeiro Rêgo

RELATÓRIO DO MILESTONE 2

Maceió — AL

2025

Virtudes e Fraquezas

Virtudes

1. Arquitetura em Camadas

Separação de Responsabilidades:

Serviços como UsuarioService, ComunidadeService e RelacionamentoService isolam regras de negócio, seguindo o padrão Service Layer.

O DataRepository centraliza o acesso a dados, aderindo ao padrão Repository, o que facilita a manutenção e testes.

Facade Simplificada:

A classe Facade atua como uma interface unificada, reduzindo a complexidade para o cliente e encapsulando interações com serviços internos.

2. Reutilização de Componentes

Serviços compartilham o DataRepository, evitando duplicação de código e centralizando a lógica de acesso a dados.

3. Extensibilidade

A modularização permite adicionar novos recursos (ex: novos tipos de relacionamentos) sem impactar serviços existentes.

Fraquezas

1. Acoplamento entre Serviços

Dependências Circulares: Serviços como MensagemService dependem de UsuarioService e ComunidadeService, o que pode complicar testes unitários e aumentar fragilidade.

2. Gestão de Sessões

Complexidade Ocultas: A lógica de sessão está dividida entre SessaoService e DataRepository, dificultando o rastreamento de fluxos.

3. Desempenho em Escala

Estruturas de Dados Não Otimizadas: Uso de LinkedList para mensagens e HashMap para usuários pode ser ineficiente em cenários com milhares de registros.

Padrões de Projeto Utilizados

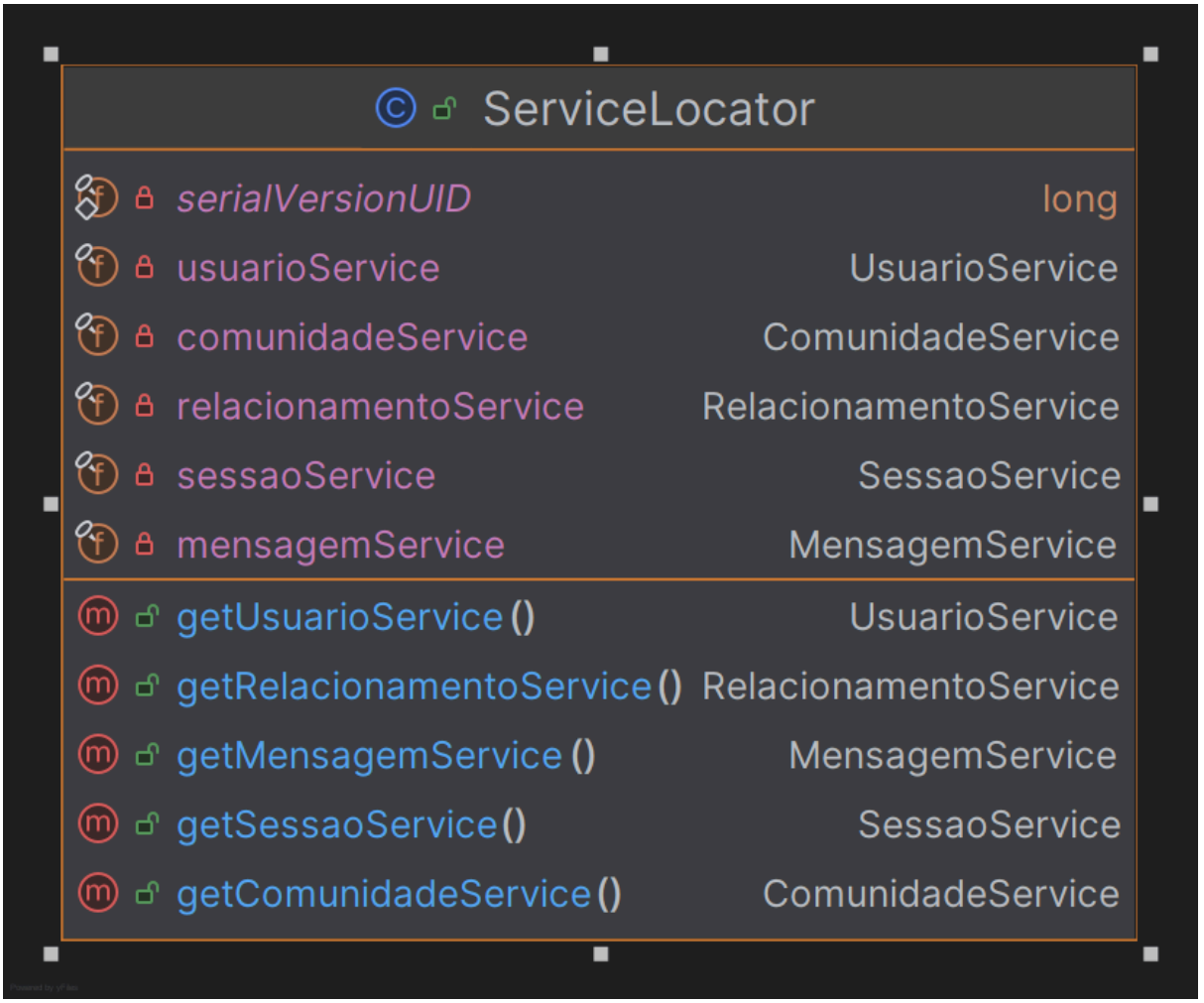
1. Padrão Service Locator

1.1 Definição e Propósito

O Service Locator é um padrão de design que encapsula os processos envolvidos na obtenção de um serviço com uma forte abstração de implementação. Ele fornece um ponto centralizado para gerenciar serviços e suas dependências.

1.2 Implementação no Sistema Jackut

No sistema Jackut, o Service Locator foi implementado através da classe ServiceLocator, segue abaixo o diagrama de classe:



1.3 Funcionamento no Sistema

- Inicialização: Na classe Facade, o ServiceLocator é inicializado ou carregado de um arquivo.
- Acesso aos Serviços: A Facade obtém referências aos serviços através do ServiceLocator.
- Persistência: O ServiceLocator é serializado para manter o estado do sistema.

1.4 Benefícios do Service Locator:

- Dependências Ocultas: As dependências não são explícitas nas assinaturas dos métodos.
- Dificuldade em Testes: Pode complicar testes unitários por criar uma dependência global.
- Violação do Princípio de Responsabilidade Única: O locator assume a responsabilidade de criar e gerenciar todos os serviços.

2. Padrão Service Layer

2.1 Definição e Propósito

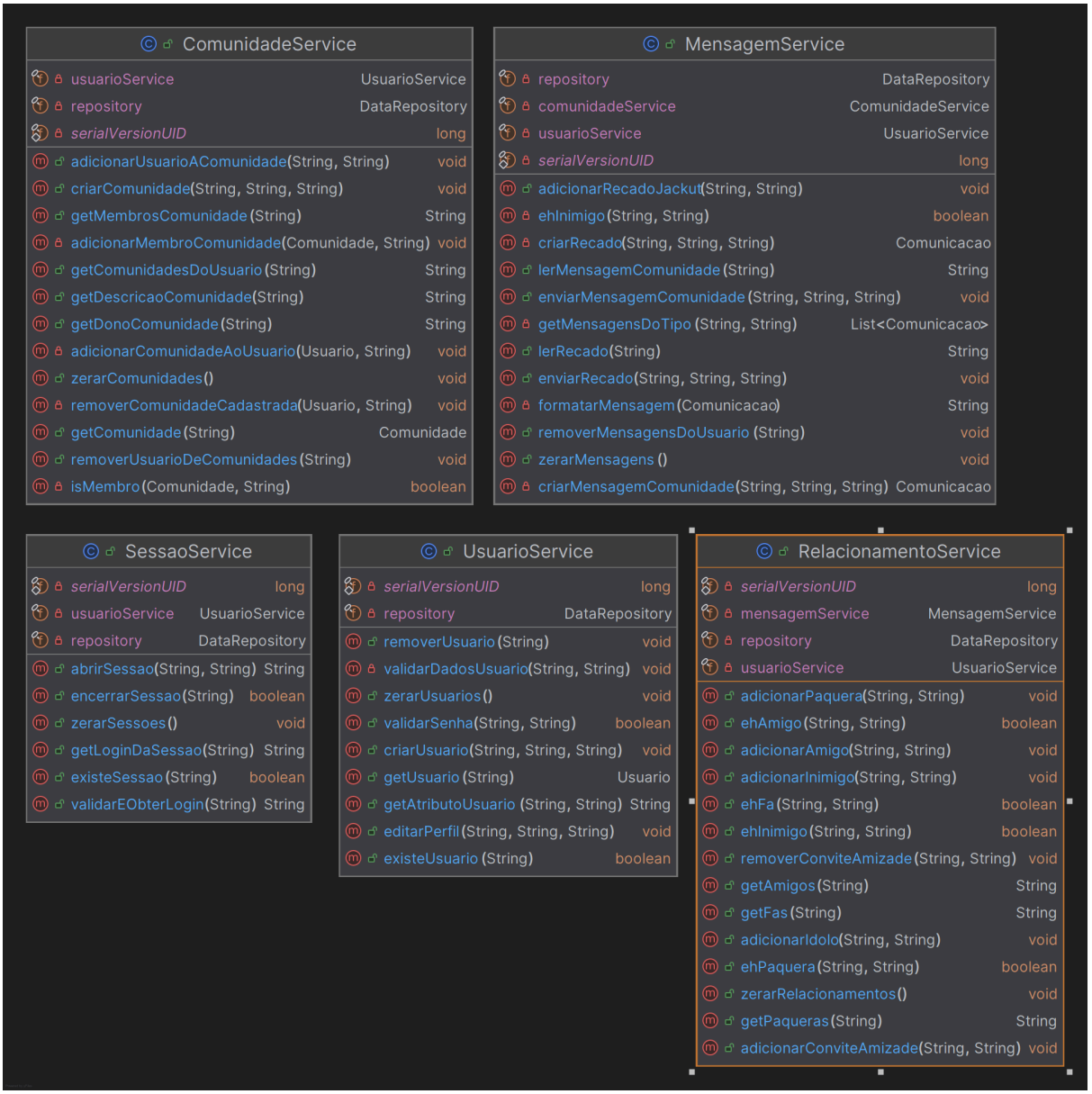
O Service Layer é um padrão arquitetural que define uma camada de aplicação que estabelece um conjunto de operações disponíveis e coordena a resposta da aplicação para cada operação. O principal objetivo é encapsular a lógica de negócio em uma camada separada da interface do usuário e da camada de acesso a dados.

2.2 Implementação no Sistema Jackut

No sistema Jackut, o Service Layer foi implementado através de várias classes de serviço, cada uma responsável por um domínio específico:

- UsuarioService: Gerencia operações relacionadas a usuários
- SessaoService: Gerencia sessões de usuários
- ComunidadeService: Gerencia comunidades
- MensagemService: Gerencia mensagens e recados
- RelacionamentoService: Gerencia relacionamentos entre usuários

Segue abaixo o diagrama de classes:



2.3 Funcionamento no Sistema

- Separação de Responsabilidades: Cada serviço encapsula a lógica de negócio relacionada a um domínio específico.
- Coordenação entre Serviços: Os serviços colaboram entre si para realizar operações complexas.
- Acesso a Dados: Os serviços utilizam o DataRepository para acessar e manipular os dados.

2.4 Benefícios do Service Layer

- Separação Clara de Responsabilidades: Cada serviço tem uma responsabilidade bem definida.

- Reutilização de Código: A lógica de negócio é encapsulada em serviços que podem ser reutilizados.
- Testabilidade: Facilita a criação de testes unitários para a lógica de negócio.
- Manutenibilidade: Torna o código mais fácil de entender e manter.
- Escalabilidade: Permite que diferentes partes do sistema evoluam independentemente.

3. Interação entre Service Locator e Service Layer

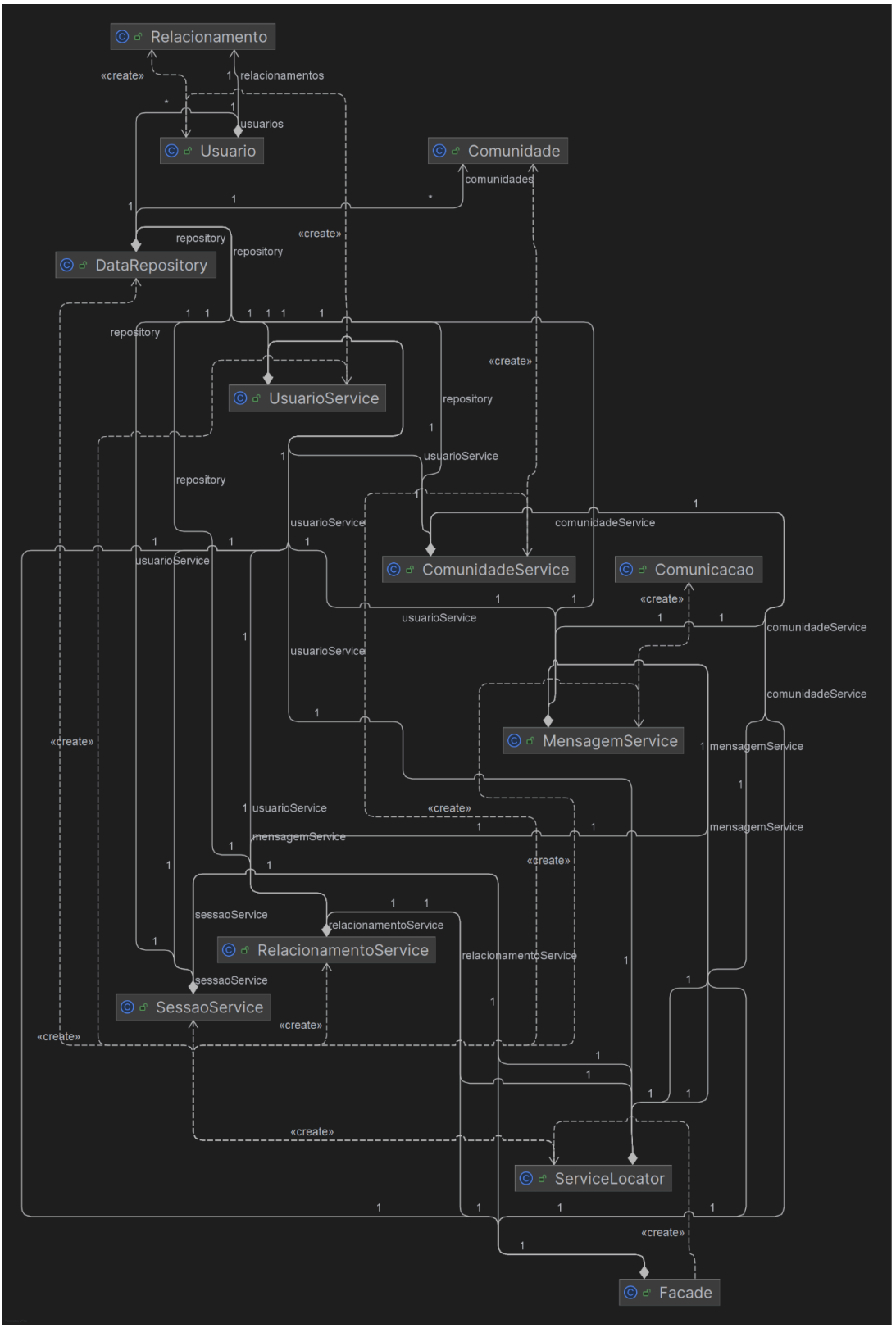
No sistema Jackut, o Service Locator e o Service Layer trabalham juntos da seguinte forma:

- O Service Locator gerencia as instâncias dos serviços e suas dependências.
- O Service Layer (implementado através das classes de serviço) encapsula a lógica de negócio
- A Facade utiliza o Service Locator para obter referências aos serviços e delega as operações para eles.

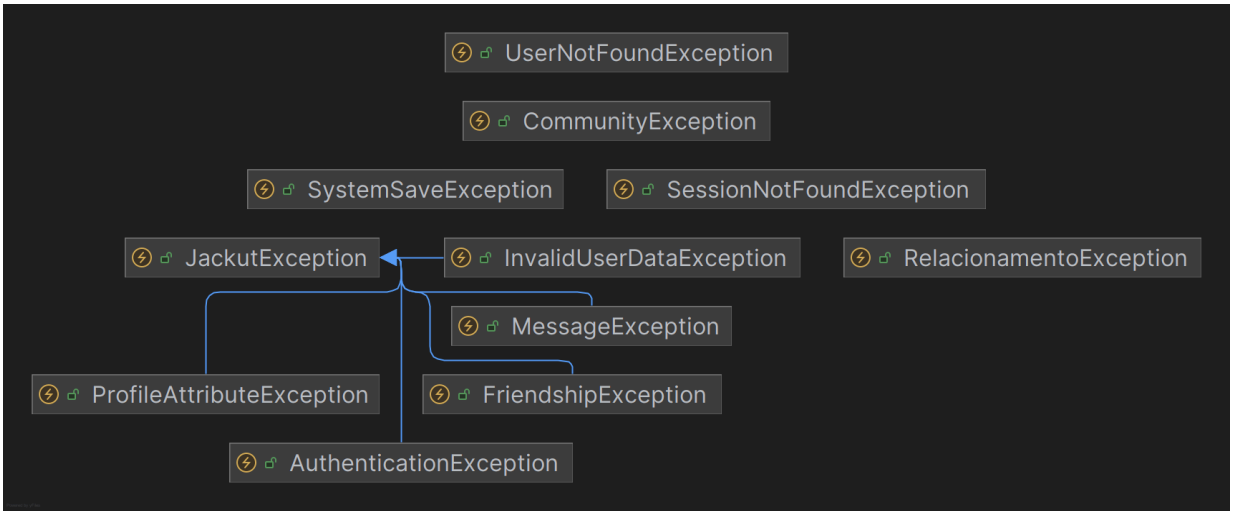
Esta combinação cria uma arquitetura em camadas bem definida:
[Cliente] → [Facade] → [Service Layer (via Service Locator)] → [Entidades/Repositório]

Diagrama de Classes

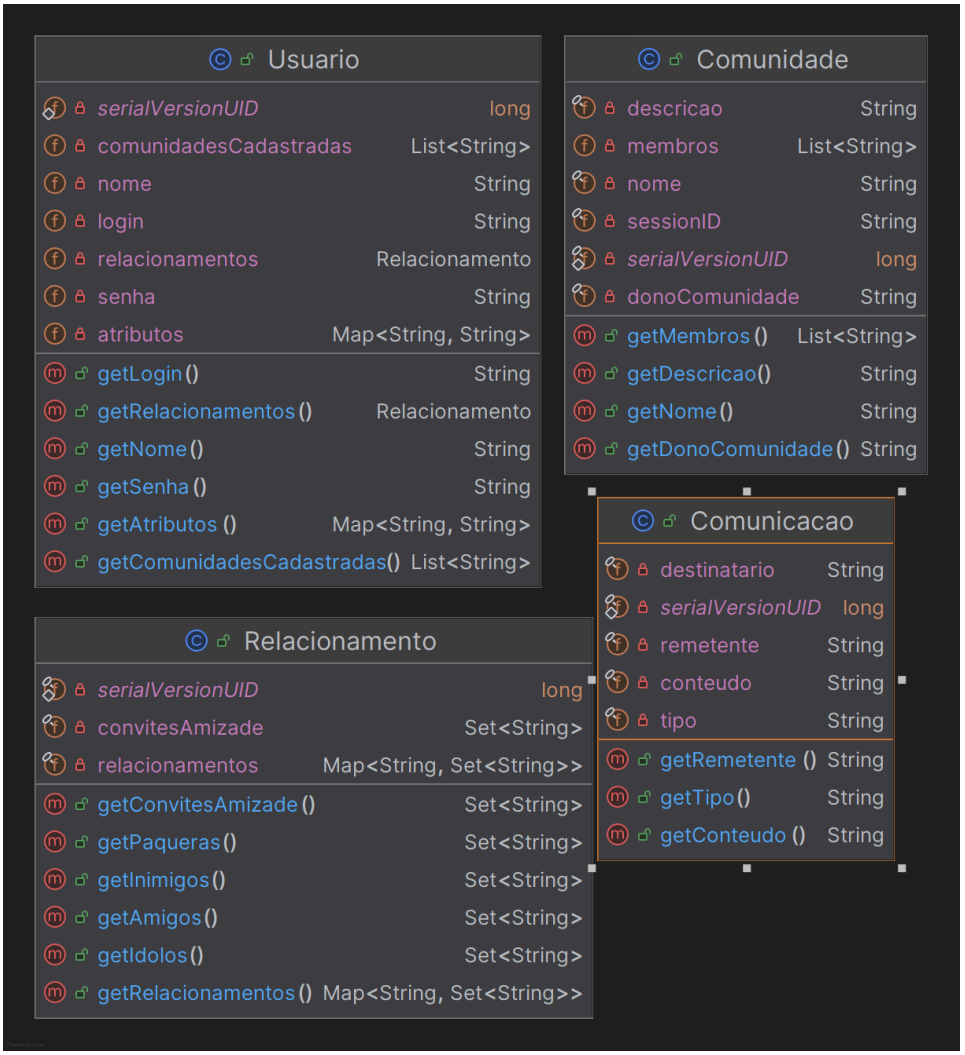
Completo



Pacote Exceptions

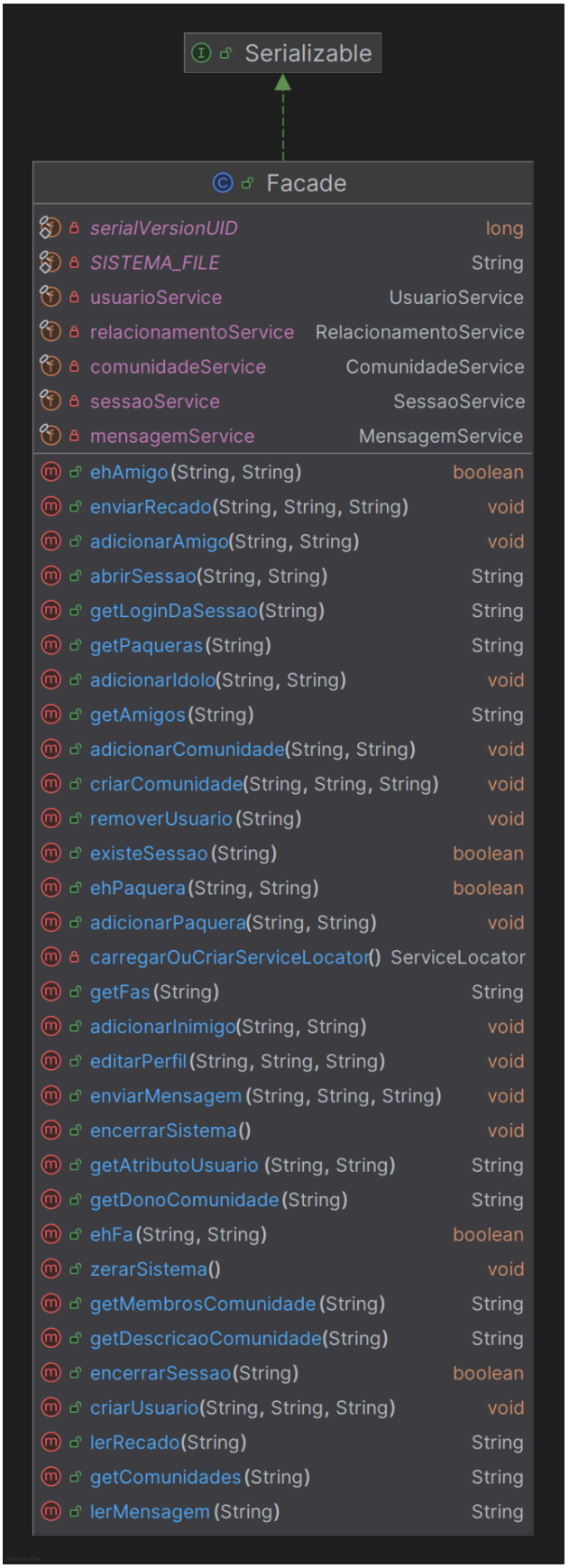


Pacote Entities



Pacote Services

Facade


Serializable
Facade
serialVersionUID long
SISTEMA_FILE String
usuarioService UsuarioService
relacionamentoService RelacionamentoService
comunidadeService ComunidadeService
sessaoService SessaoService
mensagemService MensagemService
ehAmigo(String, String) boolean
enviarRecado(String, String, String) void
adicionarAmigo(String, String) void
abrirSessao(String, String) String
getLoginDaSessao(String) String
getPaqueras(String) String
adicionarIdolo(String, String) void
getAmigos(String) String
adicionarComunidade(String, String) void
criarComunidade(String, String, String) void
removerUsuario(String) void
existeSessao(String) boolean
ehPaquera(String, String) boolean
adicionarPaquera(String, String) void
carregarOuCriarServiceLocator() ServiceLocator
getFas(String) String
adicionarInimigo(String, String) void
editarPerfil(String, String, String) void
enviarMensagem(String, String, String) void
encerrarSistema() void
getAtributoUsuario(String, String) String
getDonoComunidade(String) String
ehFa(String, String) boolean
zerarSistema() void
getMembrosComunidade(String) String
getDescricaoComunidade(String) String
encerrarSessao(String) boolean
criarUsuario(String, String, String) void
lerRecado(String) String
getComunidades(String) String
lerMensagem(String) String

Organização

O projeto JACKUT está organizado em três camadas principais.

Camada de Entidades: Modelos de dados como Comunicacao (mensagens entre usuários/comunidades), Comunidade (grupos de usuários), Relacionamento (conexões como amigos e inimigos) e Usuario (dados do usuário), focados apenas em armazenar informações.

Camada de Serviços: Classes como DataRepository (armazenamento centralizado), SessaoService (gerenciamento de sessões), MensagemService (envio/leitura de mensagens), RelacionamentoService (amizades, ídolos, etc.), ComunidadeService (criação/gestão de comunidades) e UsuarioService (operações de usuários), responsáveis pela lógica de negócio.

Camada de Integração:

- Facade: Interface simplificada para interações externas, seguindo o padrão Facade.
- ServiceLocator: Centraliza a criação e acesso aos serviços, aplicando o padrão Service Locator para gerenciar dependências e persistência de dados.

Essa estrutura separa claramente dados (entidades), regras (serviços) e integração (Facade/ServiceLocator), garantindo modularidade, facilidade de manutenção.

Correções e Implementações ocorridas entre o Milestone 1 para o Milestone 2

1. Remoção da Classe Sistema e Introdução do Service Layer

Problema Anterior:

A classe Sistema concentrava todas as operações (usuários, sessões, comunidades, mensagens), resultando em alto acoplamento e complexidade.

Refatoração:

Substituição por serviços especializados: UsuarioService, ComunidadeService, MensagemService, RelacionamentoService e SessaoService.

Benefícios:

Separação de responsabilidades: Cada serviço gerencia apenas sua área (ex: ComunidadeService lida com criação e membros).

Facilidade de manutenção: Mudanças em uma funcionalidade não afetam outras.

2. Entidades Mais Enxutas

Problema Anterior:

As entidades (como Usuario e Comunidade) continham lógicas complexas.

Refatoração:

Entidades passaram a armazenar apenas atributos e métodos básicos (ex: Usuario guarda login, senha, nome).

Lógica movida para serviços: Operações como adicionar amigos ou enviar mensagens foram transferidas para os serviços.

Benefícios:

Coesão: Cada classe tem uma única responsabilidade.

Reutilização: Serviços podem usar múltiplas entidades sem conflitos.

3. Padrão Service Locator

Problema Anterior:

Dependências entre componentes eram rígidas e difíceis de gerenciar.

Refatoração:

ServiceLocator: Centraliza a criação e acesso aos serviços, injetando o DataRepository (repositório de dados).

Benefícios:

Controle de dependências: Facilita a substituição de serviços (ex: para testes).

Persistência simplificada: O ServiceLocator é serializado em sistema.dat, mantendo o estado do sistema.

4. Introdução do Data Repository

Problema Anterior:

Dados eram gerenciados diretamente pelas classes e sistema, sem abstração.

Refatoração:

DataRepository: Classe responsável por armazenar e acessar dados (usuários, comunidades, mensagens).

Benefícios:

Isolamento do acesso a dados: Serviços não precisam saber como os dados são armazenados.

Persistência transparente: Serialização/recuperação de dados é centralizada.

5. Melhoria na Persistência

Problema Anterior:

A serialização era gerenciada pela Facade, misturando lógica de negócio com persistência.

Refatoração:

Facade delega a serialização ao ServiceLocator, que encapsula todos os serviços e o DataRepository.

Benefícios:

Consistência: Todos os dados são salvos/recuperados em um único fluxo.

Segurança: Redução de erros ao salvar estados parciais.

Implementações Milestone 2

User Story 5 (Criação de Comunidades)

Funcionalidade: Criar comunidades, validar nome único, descrição, dono e membros.
Classes Envolvidas:

- ComunidadeService:
  - Valida se a comunidade já existe (repository.existeComunidade(nome)).
  - Cria um objeto Comunidade com o dono (login), nome e descrição.
  - Adiciona o dono como membro (comunidade.addMembro(login)).
  - Atualiza o repositório (repository.adicionarComunidade()).

- Comunidade (Entidade):

User Story 6 (Adição a Comunidades)

Funcionalidade: Usuários entram em comunidades e verificam membros.
Classes Envolvidas:

- ComunidadeService:
  - Busca a comunidade.
  - Verifica se o usuário já é membro.
  - Adiciona o usuário à comunidade .
  - Atualiza a lista de comunidades do usuário.

- Usuario:
  - Lista de nomes de comunidades cadastradas.

User Story 7 (Envio de Mensagens a Comunidades)

Funcionalidade: Enviar mensagens a comunidades e validar recebimento.
Classes Envolvidas:

- MensagemService:
  - Cria a mensagem, instanciando Comunicacao..
  - Obtém os membros da comunidade

    - Adiciona a mensagem para cada membro.

- Comunicacao (Entidade):
    - Representa as mensagens.

User Story 8 (Novos Relacionamentos)

Funcionalidade: Adicionar ídolos, paqueras e inimigos.
Classes Envolvidas:

- RelacionamentoService:
    - Adicionar Idolos
    - Verificar se o ídolo já existe e se é inimigo.
    - Adiciona o ídolo à lista do usuário.
    - Adicionar Paquera
    - Verifica se a paquera é mútua e envia recados automáticos.
    - Adicionar Inimigo
    - Bloqueia futuras interações (ex: não permite envio de mensagens).
- Relacionamento
- Usuario

User Story 9 (Remoção de Conta)

Funcionalidade: Remover um usuário e apagar seus dados.
Classes Envolvidas:

- UsuarioService:
    - Remove o usuário do DataRepository.
- ComunidadeService:
    - Remove o usuário de todas as comunidades e exclui comunidades onde era dono.
- MensagemService:
    - Remove mensagens enviadas e recebidas pelo usuário.