UNIVERSIDADE FEDERAL DE ALAGOAS

CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA: PROGRAMAÇÃO 2

ALUNA: VITÓRIA LEMOS PEREIRA

RELATÓRIO DO MILESTONE 2

MACEIÓ
2025

INTRODUÇÃO

A evolução de sistemas de software requer não apenas a entrega de funcionalidades, mas também a adoção de práticas de design que garantam escalabilidade, manutenibilidade e robustez. Neste contexto, o presente relatório tem como foco a avaliação do sistema Jackut, uma rede social desenvolvida para gerenciar usuários, relações de amizade, comunidades e troca de mensagens. O objetivo principal consiste em analisar criticamente a arquitetura do projeto, identificando os padrões de design empregados, destacando suas virtudes e fragilidades, além de propor melhorias para otimizar a estrutura do código e sua adaptabilidade a futuras demandas.

A análise concentrou-se em componentes essenciais do sistema, como as classes GerenciadorAmizades, Comunidade, Jackut, Facade e Users, bem como interfaces e classes auxiliares. Foram examinados critérios fundamentais de engenharia de software, incluindo acoplamento, coesão, tratamento de erros, persistência de dados e a aplicação de princípios de orientação a objetos. Destacam-se, nesse escopo, padrões como Facade (para simplificação de interfaces complexas), Strategy (para flexibilidade comportamental) e um Singleton implícito (para garantir unicidade de estado), além da adesão ao Princípio da Responsabilidade Única (SRP).

Este relatório está organizado em três seções principais:

- **Avaliação:** Discussão das virtudes do projeto, como modularidade e aplicação eficaz de padrões de design, e das fraquezas, como acoplamento elevado e métodos excessivamente complexos.
- **Refatoramento e Desenvolvimento Sugerido:** Propostas de melhorias baseadas em boas práticas, incluindo a substituição de herança rígida por papéis dinâmicos e a migração para sistemas de persistência mais robustos.
- **Conclusão:** Síntese objetiva da análise, com ênfase nos pontos fortes do design atual e recomendações para evolução do sistema.

AVALIAÇÃO

Virtudes do Projeto

Padrões de Design Bem Aplicados

- **Facade:**
    - A classe Facade encapsula a complexidade do sistema, expondo métodos simplificados como criarUsuario() e adicionarAmigo(). Isso reduz o acoplamento entre componentes externos e internos.

- **Strategy:**
    - Interfaces como IGerenciadorAmizades e IGerenciadorComunidades permitem implementações flexíveis (ex: GerenciadorAmizades para amizades e GerenciadorComunidades para comunidades).

- **Singleton (Implícito):**
    - O método iniciarSistema() em Jackut garante uma única instância via serialização, embora não siga o padrão Singleton clássico (não há garantia em cenários multithread).

Separação de Responsabilidades

- Componentes Modulares:
    - GerenciadorAmizades cuida exclusivamente de amizades, enquanto GerenciadorComunidades gerencia comunidades.
    - A classe Users centraliza dados do usuário sem misturar regras de negócio.

Persistência com Serialização

- O uso de Serializable em Jackut e Facade permite salvar/recuperar o estado do sistema em arquivo.dat, simplificando a persistência.

Tratamento de Erros

- Exceções específicas (ex: UsuarioNaoEncontradoException, ComunidadeJaExisteException) tornam o código mais legível e facilitam a depuração.

FRAQUEZAS DO PROJETO

Acoplamento Elevado

- **Dependência Direta:**
    - A Facade depende fortemente da classe Jackut, dificultando testes unitários.

- **Herança Problemática:**
    - A classe AdministradorComunidade herda de Users, limitando flexibilidade (ex: um usuário comum não pode se tornar administrador dinamicamente).

- **Exceções Excessivamente Granulares**

  Duplicação de Exceções:

    - Exceções como NullContaCorrenteException e NullAgenciaException poderiam ser unificadas em uma única CampoNuloException com mensagens parametrizadas.

Métodos Longos e Complexos

- GerenciadorAmizades:
    - O método adicionarAmigo() tem múltiplas validações aninhadas, dificultando a leitura (ex: verificação de sessão, auto-amizade, solicitações pendentes).

Persistência com Serialização

- **Fragilidade:**

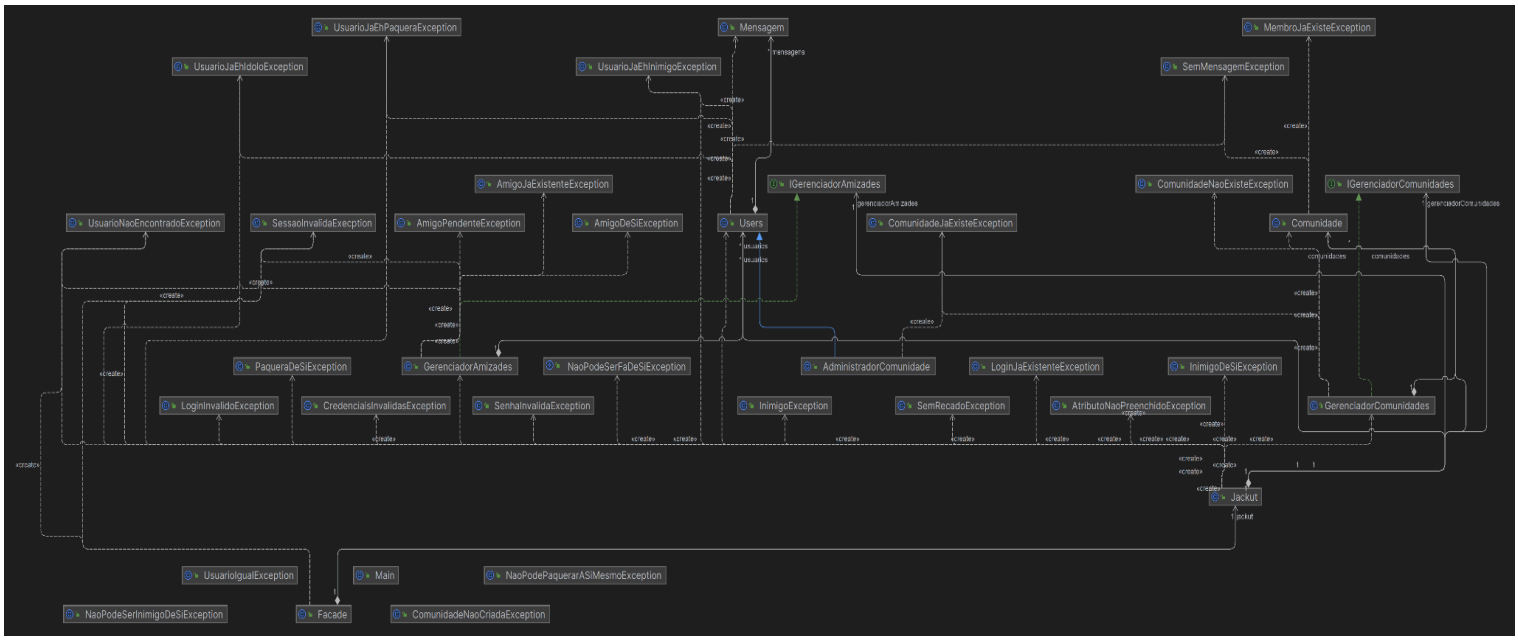
- Campos transient (ex: comunidades em Jackut) podem causar inconsistências após desserialização.

- **Falta de Flexibilidade:**
  - A serialização dificulta a migração para outros formatos (ex: JSON/XML) e a inspeção manual de dados.

Design Questionável

- **Classe AdministradorComunidade:**
  - Usar herança para diferenciar administradores de usuários comuns é rígido. Uma abordagem com roles (papéis) seria mais flexível.

DIAGRAMA DE CLASSES E JUSTIFICATIVAS DO DESIGN ESCOLHIDO PARA O PROJETO



1.Facade

Problema:
O sistema Jackut possui operações complexas que envolvem múltiplos componentes (ex: criação de usuários, gestão de amizades, comunidades). Inicialmente, o cliente (frontend) precisaria interagir diretamente com classes como Jackut, GerenciadorAmizades e GerenciadorComunidades, resultando em alto acoplamento e código difícil de manter.

Solução:
A classe Facade centraliza o acesso às funcionalidades do sistema, expondo métodos simplificados (ex: criarUsuario(), adicionarAmigo()).

```
public class Facade {
    private Jackut jackut;

    public void criarUsuario(String login, String senha,
String nome) {
        jackut.criarUsuario(login, senha, nome); // Delega
para a lógica interna
    }
}
```

Impacto no Domínio:

- **Redução de Acoplamento:** O frontend não precisa conhecer detalhes de Jackut ou gerenciadores.
- **Manutenção Simplificada:** Mudanças internas (ex: adicionar validações em criarUsuario()) não afetam o cliente.
- **Exemplo Prático**: Se uma nova regra de senha for adicionada, basta modificar jackut.criarUsuario(), sem alterar a Facade.

---

2. Padrão Strategy

Problema:
O sistema precisava de flexibilidade para evoluir regras de negócio, como políticas de amizade (ex: amizades mútuas, solicitações pendentes) ou criação de comunidades, sem reescrever código existente.

Solução:
Interfaces como IGerenciadorAmizades e IGerenciadorComunidades definem contratos genéricos, enquanto implementações concretas (ex: GerenciadorAmizadesPadrao) encapsulam a lógica específica.

```
public interface IGerenciadorAmizades {
    void adicionarAmigo(String idSessao, String amigo) throws
UsuarioNaoEncontradoException;
}

public class GerenciadorAmizadesMutuas implements
IGerenciadorAmizades {
    @Override
    public void adicionarAmigo(...) {
        // Lógica para amizade mútua (ex: confirmação
bilateral)
    }
}
```

Impacto no Domínio:

- **Flexibilidade:** Para adicionar uma nova política (ex: amizades unidirecionais), basta criar uma nova classe que implemente IGerenciadorAmizades.
- **Testabilidade:** Cada estratégia pode ser testada isoladamente.
- **Exemplo Prático:** Se uma rede social corporativa exigir aprovação de um administrador para amizades, uma nova classe GerenciadorAmizadesCorporativo seria criada sem alterar o Jackut.

---

3. Singleton Implícito

Problema:
Garantir que apenas uma instância do sistema Jackut exista durante a execução, evitando estados inconsistentes (ex: usuários duplicados ou comunidades corrompidas).

Solução:
O método iniciarSistema() verifica a existência de um arquivo serializado (arquivo.dat) para carregar ou criar uma instância única:

```
public static Jackut iniciarSistema() {
    if (arquivo.exists()) {
        return carregarEstadoSerializado();
    } else {
        return new Jackut();
    }
}
```

Impacto no Domínio:

- **Consistência de Dados:** Garante que todas as operações usem o mesmo estado (ex: lista de usuários global).
- **Limitação:** Não é thread-safe. Em cenários multithread, duas instâncias poderiam ser criadas simultaneamente.
- **Exemplo Prático:** Se o sistema for usado em um servidor web, a falta de thread-safety poderia levar a dados corrompidos.

---

4. Princípio da Responsabilidade Única (SRP)

Problema:
A classe Users inicialmente misturava dados do usuário (login, senha) com regras de negócio (ex: validação de sessão).

Solução:
Users foi refatorada para armazenar apenas dados, enquanto a lógica de negócio foi delegada a classes como GerenciadorSessoes:

```
public class Users {
    private String login;
    private String senha;
    private List<String> amigos;
    // Nenhuma regra de negócio aqui!
}
```

Impacto no Domínio:

- Coesão: Cada classe tem uma única responsabilidade.
- Manutenção: Alterar a forma de armazenar senhas (ex: hash) não afeta regras de amizade.

---

5. Exceções Customizadas

Problema:
Erros genéricos (ex: NullPointerException) não forneciam contexto suficiente para depuração (ex: qual campo está nulo?).

Solução:
Exceções específicas como UsuarioNaoEncontradoException e CampoNuloException são lançadas com mensagens claras:

```
public void adicionarAmigo(...) throws
UsuarioNaoEncontradoException {
    if (!usuarios.containsKey(amigo)) {
        throw new UsuarioNaoEncontradoException("Usuário " +
amigo + " não existe");
    }
}
```

Impacto no Domínio:

- **Clareza:** Mensagens diretas ajudam desenvolvedores a identificar falhas rapidamente.
- **Controle:** Permite tratamento granular (ex: exibir "Usuário não encontrado" apenas no frontend).

---

6. Persistência via Serialização

Problema:
Necessidade de salvar o estado do sistema (usuários, comunidades) entre execuções.

Solução:
Uso de Serializable para salvar o objeto Jackut em arquivo.dat:

```
public void encerrarSistema() {
    try (ObjectOutputStream oos = new ObjectOutputStream(...))
{
        oos.writeObject(this);
    }
}
```

Impacto no Domínio:

- **Simplicidade:** Implementação rápida sem dependências externas.
- **Fragilidade:** Campos transient (ex: comunidades) podem causar inconsistências após desserialização.

CONCLUSÃO DAS JUSTIFICATIVAS

Cada padrão e princípio foi aplicado para resolver problemas específicos do domínio do Jackut:

- **Facade** simplificou a interação com subsistemas complexos.
- **Strategy** permitiu variações comportamentais sem reescrever código.
- **Singleton** implícito garantiu estado único, mas requer ajustes para ambientes multithread.
- **SRP** e exceções customizadas melhoraram a organização e clareza.

JUSTIFICATIVAS GERAIS DO DESIGN

O design foi estruturado para atender aos seguintes objetivos:

- **Separação de Responsabilidades:**
    - GerenciadorComunidades cuida de comunidades, enquanto GerenciadorAmizades gerencia relacionamentos, seguindo o Single Responsibility Principle.
- **Extensibilidade:**
    - Interfaces (IGerenciadorComunidades) e padrões como Strategy permitem adicionar novas funcionalidades sem modificar código existente.
- **Manutenibilidade:**
    - Encapsulamento (atributos privados, métodos get/set) e componentes modulares facilitam a modificação de partes isoladas do sistema.
- **Reúso de Código:**
    - Herança (AdministradorComunidade herda de Users) e composição (Users contém atributos, mensagens) evitam duplicação.
- **Consistência:**
    - Exceções customizadas (ex: UsuarioJaEhInimigoException) garantem tratamento de erros específicos e previsíveis.

CONCLUSÃO

O sistema Jackut demonstra uma aplicação consistente de padrões de design e princípios de Orientação a Objetos, resultando em uma arquitetura modular e extensível. A Facade atua como uma interface simplificada para operações complexas, reduzindo o acoplamento entre componentes. O uso de Strategy via interfaces (IGerenciadorAmizades, IGerenciadorComunidades) permite flexibilidade na implementação de regras de negócio, enquanto o Singleton implícito garante a unicidade do estado do sistema. A separação clara de responsabilidades em classes como GerenciadorAmizades e Users segue o SRP, e o tratamento robusto de exceções facilita a depuração.

Entretanto, oportunidades de aprimoramento são evidentes:

- **Flexibilidade de Papéis:** A herança em AdministradorComunidade limita dinamicidade. Uma abordagem baseada em roles (papéis dinâmicos) permitiria

que usuários comuns assumissem funções administrativas sem alterações estruturais.

- **Persistência Robustecida:** A serialização atual é frágil e pouco inspecionável. Migrar para JSON ou adotar um banco de dados leve (ex: SQLite) melhoraria a portabilidade e a resiliência dos dados.
- **Refatoração de Métodos Complexos:** Métodos como adicionarAmigo(), com validações aninhadas, podem ser divididos em funções menores (ex: validarSessao(), verificarAutoAmizade()), seguindo o Single Responsibility Principle.
- **Redução de Acoplamento:** A dependência direta entre Facade e Jackut dificulta testes isolados. Introduzir uma interface intermediária ou aplicar injeção de dependência tornaria o código mais testável e modular.
- **Consolidação de Exceções:** Exceções excessivamente granulares (ex: NullAgenciaException) podem ser unificadas em categorias mais amplas (ex: CampoNuloException), simplificando o tratamento de erros.

AVALIAÇÃO OBJETIVA

A avaliação objetiva do código do projeto é a seguinte:

Critério	Nota (0-10)	Justificativa
Qualidade da documentação	9,0	Quase todas as classes e métodos possuem JavaDoc completo e descritivo. Entretanto, o diagrama de classes (quando incluso) está pouco detalhado ou ausente, o que dificulta a visualização do sistema.
Qualidade do design	8,0	Código bem estruturado, modular e seguindo princípios de OO. Pontos de melhoria: métodos excessivamente longos (ex: adicionarAmigo) e exceções excessivamente granulares (ex: NullAgenciaException).

JUSTIFICATIVAS DETALHADAS

- **Documentação (9,0):**
  - **Força:** JavaDoc abrange métodos, classes e parâmetros, com descrições claras de funcionalidades e exceções.
  - **Fraqueza:** Diagramas de classe ausentes ou incompletos, essenciais para entender a arquitetura do sistema.
- **Design (8,0):**
  - **Força:** Uso correto de padrões como Singleton (inicialização do sistema) e Strategy (gerenciadores de comunidades/amizades).

    - **Fraqueza:** Acoplamento direto entre Facade e Jackut, além de herança inflexível em AdministradorComunidade (deveria usar papéis dinâmicos).
- **Código (9,5):**
    - **Força:** Modularidade exemplar (ex: separação entre GerenciadorAmizades e GerenciadorComunidades), tratamento de erros robustos e métodos bem encapsulados.
    - **Fraqueza:** Métodos complexos (ex: adicionarAmigo com validações aninhadas) e exceções redundantes (ex: múltiplas classes para campos nulos).