UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação - IC
Programação 2 - Professor Mario Hozano

Milestone 2

Estudante: Kauê Patricius Montgomery Maranhão da Costa Montenegro

Matrícula: 23110439

Ciência da Computação - Quarto período.

Maceió-AL

INTRODUÇÃO

A Milestone 2 do projeto demandou uma ampla revisão e aprimoramento de sua estrutura, com o objetivo de integrar design patterns, melhorar a modularidade, fortalecer o tratamento de exceções e otimizar diversos outros aspectos essenciais para a evolução do sistema. Essas mudanças foram implementadas para aumentar a eficiência, facilitar a manutenção e tornar o código mais compreensível e escalável.

Este documento tem como propósito apresentar e justificar as decisões de design tomadas ao longo desse processo de reformulação. Para isso, são fornecidas explicações detalhadas sobre cada escolha, acompanhadas de diagramas ilustrativos que ajudam a visualizar a nova estrutura do projeto.

Dada a complexidade da arquitetura, especialmente o grande número de métodos e classes envolvidos, optou-se por gerar diferentes versões dos diagramas. Essa abordagem visa proporcionar uma melhor compreensão da organização do sistema, evitando que um único diagrama se torne excessivamente denso e difícil de interpretar.

Além disso, são discutidos os impactos dessas modificações no funcionamento do projeto, destacando benefícios como maior reutilização de código, aumento da clareza na implementação e um tratamento mais robusto de exceções, garantindo que erros sejam gerenciados de maneira eficiente e que o sistema se mantenha estável sob diversas condições.

Repositório do GitHub (Contendo as atividades e ambas as Milestones):
[Kcodesufal/P2-atividades]

O SISTEMA PRINCIPAL

Nesse primeiro tópico, abordaremos acerca de como as classes foram organizadas, quais as principais classes e também sua utilidade.

MAIN

A classe Main possuiu pouca, mas vital utilidade para o projeto. Nesse aspecto, não houveram muitas escolhas de design a serem aplicadas nela e sua serventia foi meramente de executar o EasyAccept e permitir que os testes/user-stories fossem processados, tendo eles como alvo a classe Facade.

FACADE

Facade, como o nome diz, agiu como classe de fachada para o sistema. Anteriormente utilizada para processar a maioria das operações, na milestone 2 foi modificada — de acordo com o feedback de um dos monitores — para que contivesse somente métodos básicos, necessários para o reconhecimento do EasyAccept, enquanto a maioria do trabalho “pesado” era delegado para a classe Sistema, que retornava os valores solicitados de volta para Facade. Essa classe, juntamente com Sistema, são um exemplo bem claro da aplicação de Facade como Design Patterns.

SISTEMA

A classe Sistema representa o núcleo funcional do Jackut, sendo responsável por toda a lógica de negócios, gerenciamento de estado e persistência de dados. Ela foi projetada e reestruturada seguindo os princípios de encapsulamento e coesão, concentrando todas as operações principais do sistema enquanto a Facade atua como interface simplificada. Principais características e responsabilidades:

Gerenciamento de Estado

O sistema foi projetado para manter três estruturas principais, de modo que a interação entre elas fosse prática e correspondesse aos requisitos das user_stories:

- usuarios: um mapeamento de logins para objetos da classe “Usuario”, permitindo o gerenciamento de perfis individuais. Falaremos ainda neste capítulo acerca dela, visto que é uma das estruturas fundamentais do projeto.
- sessoes: uma estrutura para controle das sessões ativas, garantindo que usuários autenticados possam interagir de maneira segura.
- comunidades: um registro de todas as comunidades existentes no sistema, possibilitando a criação e participação dos usuários nesses espaços. Os objetos pertencem à classe “Comunidade”. De forma similar, falaremos dela em breve.

Persistência de Dados

Para garantir que o estado do sistema seja preservado entre execuções, foram implementados métodos de serialização e desserialização:

- No construtor da classe, o sistema carrega o estado anterior armazenado nos arquivos "usuarios.ser" e "comunidades.ser", garantindo que dados previamente salvos sejam recuperados automaticamente.
- No encerramento do sistema (encerrarSistema()), os dados são armazenados novamente para evitar perda de informações.
- O sistema também pode ser completamente reinicializado através do método zerarSistema(), permitindo restaurar seu estado inicial.

Operações Principais

A classe Sistema centraliza diversas operações essenciais para o funcionamento do Jackut, incluindo:

- O controle de usuários, permitindo a criação, remoção e edição de perfis.
- A gestão das relações sociais, abrangendo funcionalidades como amizades, ídolos, paqueras e até inimigos.
- A administração das comunidades, permitindo a criação de novos grupos, a participação dos usuários e o envio de mensagens nesses ambientes.
- O funcionamento do sistema de mensagens, permitindo tanto recados pessoais quanto interações dentro das comunidades.

Tratamento de Exceções

Para garantir seu funcionamento e a aprovação nos casos testes, a classe Sistema define e lança exceções específicas para lidar com situações como tentativas de operações inválidas, evitando que usuários realizem ações não permitidas, acesso a recursos não existentes, impedindo o uso de dados que não estejam registrados no sistema e conflitos de informação, garantindo que inconsistências sejam identificadas e tratadas adequadamente.

Design e Padrões Aplicados

A estrutura do sistema segue algumas diretrizes de design patterns e boas práticas de programação:

> Embora o uso de Singleton não tenha sido implementado formalmente, o sistema foi concebido para ter uma única instância ativa durante sua execução. A lógica mais complexa não está concentrada na classe Sistema, mas é delegada a objetos de domínio, como Usuario e Comunidade. Isso garante uma boa separação de responsabilidades.
>
> Os mecanismos de serialização e desserialização são gerenciados internamente, sem expor detalhes da implementação.

Além disso, métodos privados, como getUsuario() e getUsuarioPorSessao(), centralizam a verificação de permissões, garantindo que apenas usuários autorizados possam acessar determinadas informações e evitando o “desperdício” de códigos com múltiplos if-elses idênticos.

USUÁRIO

A classe Usuario constitui a peça fundamental do sistema Jackut, representando cada indivíduo cadastrado na plataforma com todos os seus atributos, relacionamentos e interações sociais. Projetada para ser uma estrutura robusta e versátil, essa classe encapsula desde informações básicas de identificação até complexas redes de conexões entre usuários, seguindo princípios sólidos de orientação a objetos e design patterns para garantir organização e eficiência.
Cada instância de Usuario armazena dados essenciais como nome, login e senha, além de permitir a adição dinâmica de atributos extras por meio de um mapa flexível (atributosMap). Essa abordagem possibilita que o perfil do usuário seja personalizável sem exigir modificações constantes na estrutura da classe. A segurança das credenciais é garantida por métodos específicos, como comparaSenha(), que verifica a autenticidade do usuário sem expor a senha armazenada.
Um dos aspectos mais notáveis dessa classe é o gerenciamento de comunicação e relacionamentos. O sistema de mensagens é dividido em duas filas distintas: uma para recados pessoais (comunicados) e outra para mensagens de comunidades, ambas implementadas como estruturas FIFO (First-In, First-Out) para assegurar que as interações sejam lidas na ordem correta. Além disso, os relacionamentos sociais são tratados de forma especializada, utilizando classes auxiliares como Amigos, Idolos, Paqueras e Inimigos, cada uma responsável por suas próprias regras e comportamentos. Essa modularização facilita a manutenção e permite que novos tipos de interação sejam adicionados sem impactar o código existente.
A participação em comunidades também é gerenciada de forma eficiente, com métodos dedicados para vincular ou remover usuários de grupos, garantindo consistência nos dados. Quando um usuário é excluído do sistema, o método deletarUsuario() assegura que todas as referências a ele sejam devidamente removidas, evitando inconsistências nas listas de amigos, inimigos, mensagens e comunidades.
A classe ainda implementa Serializable, permitindo que os objetos sejam salvos em disco e recuperados posteriormente, um requisito essencial para a persistência dos dados entre sessões. Seus métodos são projetados para lançar exceções claras em situações inválidas, como tentar adicionar a si mesmo como amigo ou acessar atributos não definidos, melhorando a robustez do sistema.

Design e Padrões Aplicados

Graças ao método como foi estruturada (e reestruturada), a classe Usuario e as classes empregadas em Usuario (como Amigos, Fas e outras) aplicam múltiplos padrões de design. De acordo com o método que as classes de “Relacionamentos” foram desenvolvidas, os Design Patterns de Strategy e Composite foram aplicados em Usuario, de modo que a

interação com diferentes tipos de relacionamento pudesse facilmente possuir dinâmicas distintas, sem que Usuario ficasse com uma quantia desnecessariamente longa de código — A implementação de comportamentos específicos para cada tipo de relacionamento ficou delegada para suas classes específicas, ao invés de ocuparem espaço em Usuario. De mesmo modo, várias das exceções foram tratadas nos métodos contidos nas classes, ao invés de Usuario.

RELACIONAMENTO

A elaboração da classe abstrata Relacionamento partiu do princípio de aplicação do chamado Template Method, um design pattern específico que consiste na utilização de um modelo para a elaboração de outras classes, permitindo modularidade, criação rápida de novos tipos de “elementos” e facilidade de leitura. Com base nela, as classes Amigos, Inimigos, Fas, Idolos, Paqueras e Convites foram criadas, cada uma contendo seu conjunto de Exceptions e métodos inspirados em Relacionamento, com cada um deles podendo ou não ter sido modificados, de acordo com a necessidade.

COMUNICADO

A classe Comunicado serve como uma implementação mista de Recados e Mensagens. Como ambas possuem os mesmos requisitos na questão de dados, diferindo apenas no método de envio (Recados são enviados de forma privada, de uma pessoa para outra, enquanto mensagens são broadcasts para todos os membros de uma comunidade exceto o autor), foi julgado que seria de maior praticidade realizar sua implementação desse modo.

COMUNIDADE

No núcleo da implementação, encontramos quatro atributos fundamentais que definem a identidade e composição de cada comunidade. O dono, representado pelo login do usuário criador, estabelece uma hierarquia básica de administração. O nome serve como identificador único e imutável, enquanto a descrição oferece contexto sobre os propósitos e temas do grupo. O conjunto de membros, armazenado em um LinkedHashSet, garante não apenas a ausência de duplicatas, mas também preserva a ordem histórica de ingresso - detalhe importante para a experiência de usuário, coleta de dados e para os casos-testes. A inicialização de uma nova comunidade ocorre através de um construtor direto que exige os três elementos essenciais (dono, nome e descrição), incluindo automaticamente o criador como primeiro participante. Essa escolha arquitetural previne a existência de comunidades "órfãs" sem membros, eliminando um possível estado inconsistente do sistema. A estrutura se completa com métodos especializados para manipulação cuidadosa da lista de participantes, onde tanto a adição quanto a remoção de membros incorporam

verificações rigorosas. O método adicionarMembro valida a não duplicação antes de incluir um novo participante, lançando uma exceção específica quando detecta tentativas de reinserção. Na outra ponta, removerMembro assegura que apenas participantes existentes possam ser excluídos, protegendo a integridade dos dados.
Para acesso às informações da comunidade, a classe oferece alternativas diversificadas que atendem a diferentes necessidades do sistema. Enquanto getMembros formata a lista de participantes como string padronizada para exibição, getMembrosSet disponibiliza o conjunto bruto para operações internas mais complexas. Essa dualidade exemplifica o cuidado com a separação entre preocupações de apresentação e lógica de negócios. Os getters convencionais para nome, descrição e dono completam a interface pública, seguindo o princípio de menor privilégio ao expor apenas o estritamente necessário.
A implementação atual demonstra várias escolhas de design conscientes. O uso de LinkedHashSet para armazenar membros combina eficiência computacional (com operações O(1) para casos médios) com manutenção da ordem de inserção - qualidade valiosa tanto para interface do usuário quanto para depuração. A imutabilidade parcial (nome, descrição e dono não podem ser alterados após construção) reduz estados inconsistentes e simplifica o raciocínio sobre o comportamento do objeto. As exceções customizadas transformam falhas operacionais em mensagens significativas que podem ser tratadas de forma apropriada pelas camadas superiores do sistema.
A Comunidade mantém relações importantes com outras partes do sistema. Sua implementação de Serializable permite a persistência transparente junto com o estado geral da aplicação. A sincronia com a classe Usuario - onde cada instância mantém registro das comunidades a que pertence - cria uma consistência bidirecional que reforça a integridade dos dados. Apesar de sua aparente simplicidade, a estrutura foi pensada para permitir expansões futuras, como a introdução de cargos diferenciados entre membros ou sistemas de moderação, sem exigir reformulações drásticas (algo bastante importante nesse projeto e que, ao longo da produção do código, provou-se um desafio relativamente grande em algumas ocasiões em que boa parte do código teve de ser refeita múltiplas vezes).

Design e Padrões Aplicados

A classe Comunidade implementa o padrão Composite ao tratar todos os membros, incluindo o dono, de forma uniforme dentro da mesma estrutura de dados. Essa abordagem permite futuras extensões, como hierarquias mais complexas, sem necessidade de grandes reformulações no sistema.
O princípio Singleton aparece indiretamente na gestão dos identificadores únicos das comunidades. Cada grupo tem um nome imutável, garantindo unicidade, com a própria estrutura da classe evitando modificações posteriores. Essa estratégia reforça a integridade dos dados e previne conflitos.

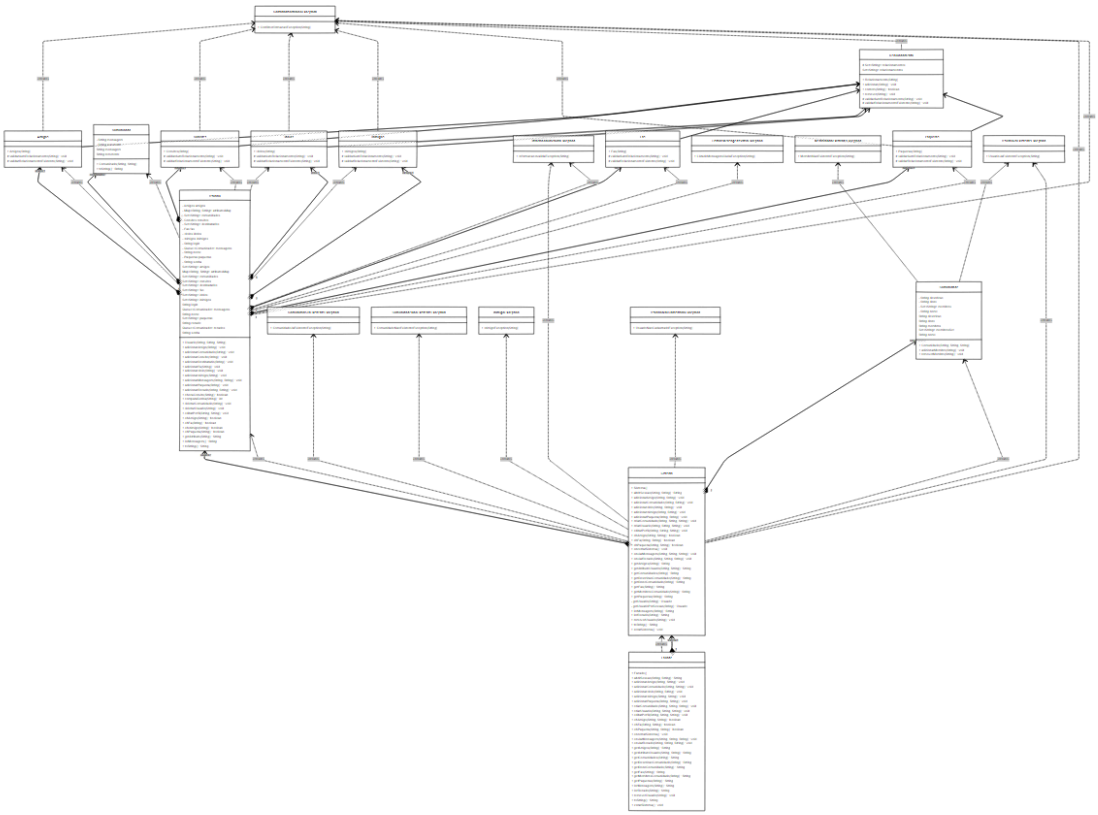
EXCEPTIONS

O uso de Exceptions ao longo do código foi feito de modo organizado. Exceptions novas foram criadas conforme necessário, enquanto algumas foram reutilizadas de acordo com a

flexibilidade de seu nome (ConflitoInformacaoException, por exemplo, serve para inúmeras ocasiões e ainda mantém uma boa consistência na sua aplicação).

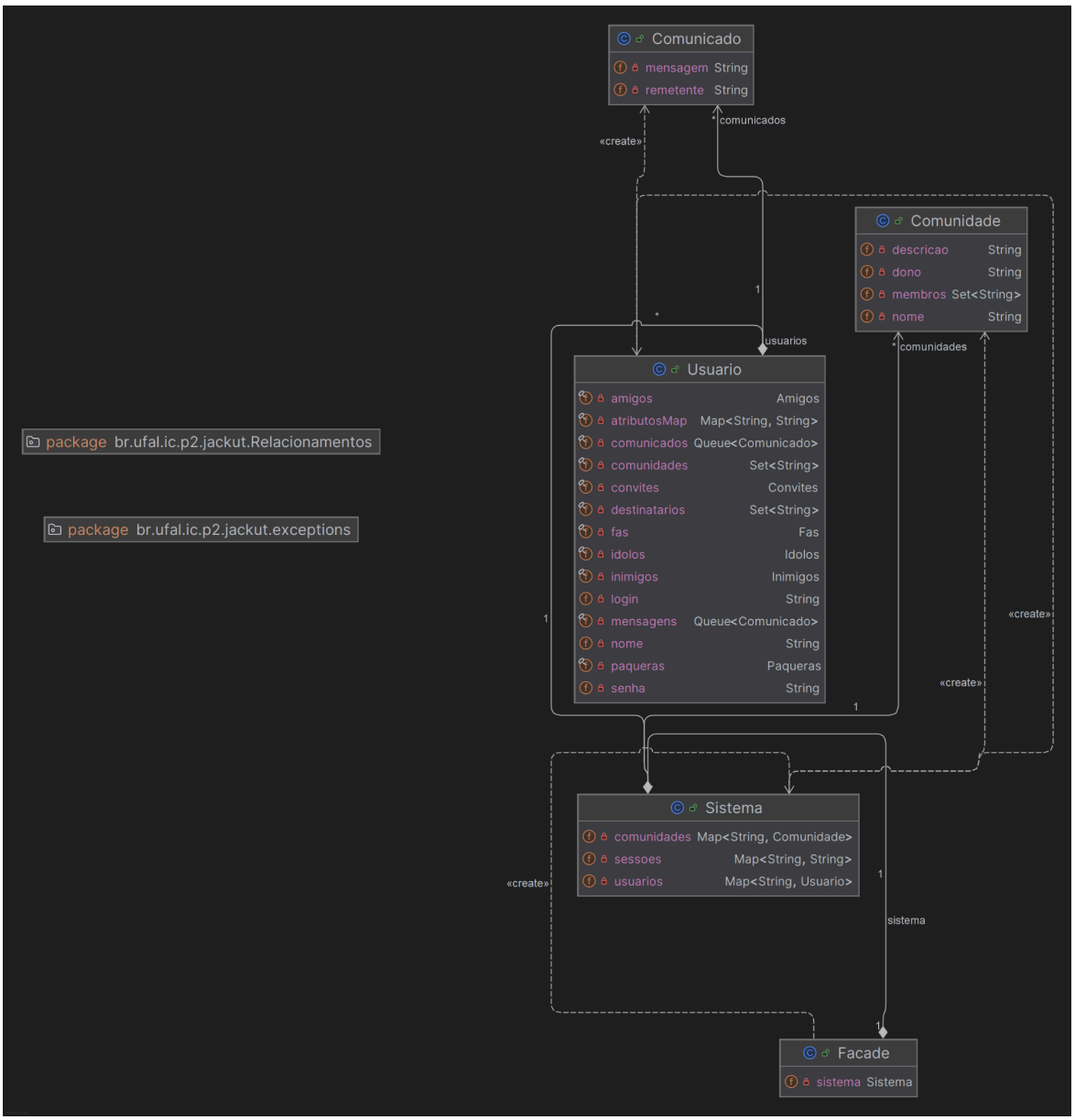
No capítulo a seguir, mostraremos os vários diagramas de classe e falaremos acerca de certas escolhas de design para algumas das user-stories.

DESIGN DO CÓDIGO



Como é fácil de perceber, o diagrama é “um pouco” grande demais para o documento. Por conta disso, ele será apresentado de forma simplificada e gradualmente será exibido de forma mais complexa. **O diagrama completo e uma versão somente com seus métodos, em qualidade alta, podem ser obtidas no github na pasta do milestone 2, em “diagramas”.**

Abaixo, há uma visualização do diagrama, contendo somente suas variáveis e relações entre classes. Note que todas as classes feitas com base em Relacionamento estão escondidas na pasta “Relacionamentos”, embora todas sejam empregadas em Usuario.
As Exceptions, por sua vez, são empregadas ao longo de todos os arquivos.



Por sua vez, podemos incluir uma versão com somente os métodos abaixo:

DECISÕES DE DESIGN

Ao elaborar o código para a Milestone 1, não estava muito ciente de como utilizar design Patterns. Isso acabou por me causar uma enorme perda de tempo para a elaboração do código da Milestone 2 — Fui forçado a reescrever boa parte do meu código, priorizando maior modularidade e eficácia. Isso significou modificar como Usuario lidava com certas variáveis (que posteriormente, na milestone 2 tornaram-se classes) e também modificar como Sistema lidava com o salvamento de arquivos e Strings.

Essa necessidade de reestruturação do código se mostrou imensamente necessária, principalmente durante a realização das user stories 8.1 e 9.1 e 9.2.
Mesmo após ter concluído o trabalho, ainda existem certos problemas que devem ser solucionados no futuro para melhorar ainda mais a modularidade do código e eficiência de certos algoritmos — Sempre há algo a se melhorar.

POSSÍVEIS OTIMIZAÇÕES

O caso mais gritante, em minha opinião, tange à user-story 9 e a forma como tratei o método de remoção de usuários. Após algum tempo indeciso entre utilizar soft-delete para alcançar a remoção em O(1) mas prejudicar a eficácia do código como um todo ou utilizar um sistema de busca para conseguir remover o usuário num tempo maior, mas mantendo o resto do código eficaz, optei pela segunda opção, isto pois:
A) Usuários apagarem sua conta é um evento raro, deleção em massa é extremamente improvável e portanto não há problema em usar isso.

B) Desacelerar todo o sistema para alcançar remoção em O(1) não compensaria, pois ainda consumiria memória até que os dados fossem processados novamente de forma posterior.

Dito isso, percorrer a lista de todos os usuários também não é nada eficaz. Por conta disso, implementei um sistema que prioriza buscar usuários que já interagiram com o usuário a ser deletado e então realiza a operação de remover os traços do usuário deletado neles.
No entanto, da forma que implementei, o pior caso acaba sendo bastante ineficaz, enquanto um resultado similar poderia ser alcançado com um pouco mais de gasto de memória e em O(N). Nesse aspecto, ainda existem pontos do código que merecem atenção e maior elaboração.

CONCLUSÃO

O código está totalmente funcional e passa nos casos testes sem demais problemas, embora haja pontos para possíveis melhorias. No caso de haver uma terceira milestone, estarei feliz em lidar com esses pequenos aspectos e melhorar o projeto ainda mais.
A aplicação dos design patterns foi realizada conforme solicitado e creia que atenda os requisitos estabelecidos pelo professor, aumentando a modularidade e organização do código, enquanto facilita sua leitura.