RELATÓRIO FINAL: SISTEMA DE GESTÃO DE CONDOMÍNIO 22/11/2025

Sistema de gestão de condomínio

Grupo:

Ricardo Cronemberger Cruz Ruben Pereira

José Francisco Paes Landim Sobrinho

Gustavo Nunes da Silva Pereira

Yuri Cavalcante Veloso Soares

João Guilherme Aragão Malta

1. Introdução

1.1. Visão Geral:

O sistema de gestão de condomínio é uma aplicação desenvolvida em java, com interface de console, com propósito de automatizar as principais tarefas administrativas e operacionais de um condomínio residencial. O sistema atende a necessidade de organizar desde o cadastro de moradores e a alocação de apartamentos até o controle financeiro e manutenção.

O projeto foi desenvolvido aplicando-se os princípios de programação orientada a objetos, garantindo uma base de código modular e escalável, com foco na integridade dos dados e no tratamento adequado de erros de entrada e de negócio.

1.2. Público Alvo:

O sistema é direcionado à administração do condomínio (Gestores, Síndicos e equipe de funcionários).

Funcionalidades implementadas:

Gestão de cadastros: Cadastro de moradores e apartamentos, com associação de residentes e unidades específicas.

Controle de acesso: Registro detalhado de entrada e saída de visitantes, associados ao morador visitado.

Reservas de áreas comuns: Agendamento de áreas (Academia, Piscina e Salão de festas, com validação de conflitos de horário.

Controle financeiro: Registro de pagamentos.

Manutenção: Abertura e acompanhamento de chamados de manutenção (Status: aberto, em andamento e fechado) com registro de custo.

Persistência de dados: Salvamento e carregamento de dados essenciais via arquivo (.txt).

2. Estrutura do código:

A estrutura do código foi organizada em classes de domínio e classes de controle/menu, facilitando a separação de responsabilidades. As seguintes abstrações e princípios foram fundamentais:

2.1. Abstração e Herança

Classe abstrata Pessoa: Abstrai os atributos e validações comuns a todos os indivíduos do sistema (nome, documento e telefone).

Classe Abstrata AreaComum: Define o comportamento fundamental de qualquer área que possa ser reservada, como a lógica central de verificarDisponibilidade baseada em tempo.

Classes Filhas: Academia, Piscina e SalaoDeFestas herdam de AreaComum.

2.2. O princípio foi garantido definindo todos os atributos das classes de domínio (Apartamento, Morador, Pagamento, etc.) como private.

O acesso aos dados é mediado por getters e setters.

Em classes como Morador, os setters contêm validações (ex: verifica se a quantidade de pets é não negativa, lançando CampoInvalidoException).

A classe Apartamento garante a imutabilidade da sua lista interna de moradores ao retornar Collections.unmodifiableList no método getMoradores(), impedindo manipulações externas não autorizadas.

2.3. Polimorfismo

O polimorfismo é presente na herança de classes e pelo uso de enums:

Classes de Área: Embora as classes filhas de AreaComum atualmente compartilhem a mesma implementação de reserva, elas podem facilmente receber lógicas polimórficas (ex: Piscina proibir reservas fora de temporada).

Métodos Comuns: A sobrescrita do método toString() em várias classes (Visitante, ChamadoManutencao, etc.) permite que cada objeto se apresente de forma personalizada, mas seja chamado de forma genérica.

2.4. Classe Considerada Importante: ControleFinanceiro.java

A classe ControleFinanceiro é crucial para o requisito de relatórios. Ela demonstra a aplicação de boas práticas de código moderno em Java, utilizando o paradigma funcional de Streams para processar coleções de forma eficiente e concisa.

O uso de stream().filter().mapToDouble().sum() no ControleFinanceiro ilustra uma implementação de alta qualidade, garantindo que os cálculos financeiros sejam realizados de maneira clara e eficiente.

3. Implementação Funcional e Tratamento de Erros

3.1. Funcionalidades Implementadas

Associação Bidirecional: Moradores são associados a Apartamentos e etc, e essa relação é corretamente restabelecida durante o carregamento dos dados pela classe Persistencia.

Lógica de Reserva: A classe GerenciadorReservas implementa a lógica de validação de conflito de horário, e a classe Reserva aplica o limite de duração de 8 horas e verifica a regra de cancelamento com multa (48h).

Controle de Chamados: A classe ChamadoManutencao implementa o ciclo de vida do chamado (ABERTO -> EM_ANDAMENTO -> FECHADO), com validações de transição de status usando OperacaoInvalidaException.

Relatório Financeiro em TXT: O método salvarRelatorioFinanceiroTxt() da classe ControleFinanceiro gera um resumo das finanças e o salva em um arquivo de texto.

3.2 Funcionalidades não implementadas e dificuldades

3.2.1 Geração de PDF

A funcionalidade de geração automática de relatórios em formato PDF não foi implementada por três motivos principais:

1. Dependência de biblioteca externa
   A criação de PDFs em Java normalmente exige bibliotecas específicas como *iText* ou *Apache PDFBox*, que não fazem parte do JDK. A integração dessas ferramentas aumenta a complexidade do projeto.
2. Tempo demasiado grande para integração e testes
   Devido ao cronograma ter um limite para entrega, optou-se por priorizar o funcionamento pleno das funcionalidades centrais do sistema (cadastros, reservas, financeiro, visitantes, etc.), conforme descrito no escopo oficial do trabalho. A geração de PDF exigiria tempo adicional para configurar fontes, layouts, tratamento de exceções e testes.
3. Alternativa já existente e funcional
   Para garantir que os relatórios pudessem ser gerados, foi implementada uma alternativa simples e eficiente: exportação em arquivos TXT.

3.2.2 Integração com Google Drive

A integração com o Google Drive também não foi adicionada ao sistema pelos seguintes motivos:

1. Requer configuração de API externa
   Para enviar arquivos ao Drive é necessário configurar o Google Cloud Console, ativar a API do Drive e gerar credenciais do tipo OAuth 2.0.
2. Processo de autenticação complexo
   O fluxo de OAuth envolve redirecionamento de usuário, tokens de acesso, tokens de atualização e armazenamento seguro das credenciais — o que aumenta muito a complexidade.

3.2.3 Interface Gráfica

A implementação de uma interface gráfica (GUI), como JavaFX ou Swing, também não foi realizada. As razões foram:

1. Prioridade na lógica de negócio e POO
   O objetivo principal do trabalho é demonstrar domínio de conceitos como abstração, herança, composição, polimorfismo, persistência e tratamento de erros — todos presentes na implementação do sistema.
2. Interface em console atende aos requisitos
   O menu em console permite testar toda a lógica, sem aumento significativo de código ou dependências externas, mantendo o foco no essencial.
3. Possível melhoria futura
   A arquitetura do projeto (conforme diagrama de classes) permite uma futura migração para JavaFX ou Swing, caso seja desejado evoluir o sistema.

4. Experiência Individual e Referências

4.1. Experiência do Aluno: Ricardo

Minha Contribuição: Fui responsável pela arquitetura inicial da classe ControleVisitante e contribuição em diversas outras classes.

O que mais gostei: A parte que mais gostei foi a criação do método de salvarVisitantes na classe "ControleVisitante", que é responsável pela persistência, ou seja, salvar a lista de objetos Visitante em um txt.

Maiores Dificuldades: Tive uma grande dificuldade em identificar erros de sintaxe, procurando onde estava meu erro de fechamento do método e em associar minha parte ao resto do código.

Aprendizado Principal: Fortaleci meu entendimento sobre as vantagens da Abstração e Herança para evitar duplicação de código, e aprendi sobre a importância das exceções customizadas para mapear regras de negócio específicas em classes Java.

4.2. Experiência do Aluno: Yuri Cavalcante

Minha Contribuição: fiz o sistema manutenção e o sistema principal, além de ajudar com os testes dos outros sistemas.

O que mais gostei: desenvolver um sistema de gerenciamento foi muito legal principalmente pela parte de finalizar os testes

Maiores Dificuldades: bugs, primeiro corrigir os bugs foi muito difícil, mas a parte mais difícil foi achar o porque algumas funcionalidades não estavam executando de forma esperada, mas acabou que o difícil foi achar só a primeira, as outras estavam com problemas parecidos e acabou que uma puxou a outra

Aprendizado Principal: trabalhar em equipe, acho que foi o que mais agregou. realizar uma demanda para entregar em um prazo com uma equipe, tendo tarefas definidas.

4.3. Experiência do Aluno: Gustavo Nunes

Minha Contribuição: Fui responsável pela criação da classe ControleFinanceiro e contribui no desenvolvimento da classe MenuPagamentos.

O que mais gostei: A parte que mais gostei foi o processo de desenvolvimento das classes, pois por meio dele pude conhecer novas bibliotecas do Java e também novas maneiras de codificar.

Maiores Dificuldades: minha maior dificuldade nesse processo foi entender como integrar minhas classes ao resto do código e também gerar um arquivo txt sem que comprometesse o todo o código.

Aprendizado Principal: O trabalho em equipe foi um dos principais aprendizados, compreender como realizar um desenvolvimento produtivo com coordenação sem gerar erros ou problemas no código.

4.4. Experiência do Aluno: João Guilherme Aragão Malta

Minha Contribuição: Fui responsável pela implementação das classes Reserva, GerenciadorReserva e AreaComum, bem como suas subclasses (aplicando o conceito de herança). Também colaborei no desenvolvimento da lógica de interação na classe MenuReservas.

O que mais gostei: Ver a lógica de negócios funcionando na prática, especialmente a orquestração entre o gerenciador e as diferentes áreas comuns para efetivar uma reserva com sucesso.

Maiores Dificuldades: O tratamento de exceções e a depuração (debugging) durante o processo de reserva, garantindo que o sistema lidasse corretamente com conflitos e erros de validação.

Aprendizado Principal: A importância da modularização e do polimorfismo para criar um código escalável, facilitando a comunicação entre diferentes objetos do sistema.

4.5. Experiência do Aluno: José Francisco Paes Landim Sobrinho

Minha Contribuição: Atuei diretamente na estrutura de moradores e apartamentos, garantindo que o relacionamento entre ambas as entidades. Também implementei o sistema de persistência em múltiplos arquivos, permitindo que os dados fossem mantidos mesmo após o encerramento da aplicação. Outra parte importante do meu trabalho foi montar os menus e integrar funcionalidades, possibilitando uma navegação clara e funcional.

O que mais gostei: A parte que mais gostei foi pensar na organização do projeto, estruturando as classes e discutindo com os integrantes do grupo sobre quais estratégias e funcionalidades seriam implementadas. Trabalhar com persistência foi especialmente enriquecedor, pois exigiu a aplicação prática de conceitos importantes e trouxe uma sensação de evolução no entendimento de um projeto real de gerenciamento.

Maiores Dificuldades: Enfrentei algumas dificuldades ao longo do desenvolvimento, principalmente relacionadas à parte técnica da persistência. Entre os principais desafios, destacam-se: manter a sequência correta dos IDs ao carregar dados já existentes, gerenciar vários arquivos de persistência simultaneamente sem perder consistência e resolver erros que surgiam progressivamente, exigindo análise cuidadosa, testes e ajustes constantes no código.

Aprendizado Principal: Este projeto permitiu consolidar conhecimentos importantes de programação orientada a objetos e desenvolvimento de sistemas. Entre os aprendizados mais significativos, destaco: aplicar conceitos de POO na prática, de forma integrada e funcional, resolver problemas complexos de lógica, entendendo melhor o ciclo de desenvolvimento, melhorar minhas habilidades de debug, aprendendo a identificar, compreender e corrigir erros e entender a real importância das validações, organização do código e, principalmente, do trabalho em equipe para o sucesso do projeto.

5. Melhorias Futuras

5.1 Curto Prazo

Implementar geração de relatórios em PDF com iText

Adicionar mais validações de dados (CPF, telefone)

Melhorar interface console com formatação de tabelas

Implementar backup automático dos arquivos

5.2 Médio Prazo

Criar interface gráfica com JavaFX

Adicionar sistema de autenticação (login de moradores/admin)

Implementar notificações de vencimento de pagamentosDashboard com gráficos de ocupação de áreas comuns

5.3 Longo Prazo

Migrar para banco de dados relacional (PostgreSQL/MySQL)

Desenvolver aplicativo mobile para moradores

Integração com sistemas de pagamento online

API REST para integração com outros sistemas

Sistema de comunicação interna (avisos, enquetes).

6. Conclusão

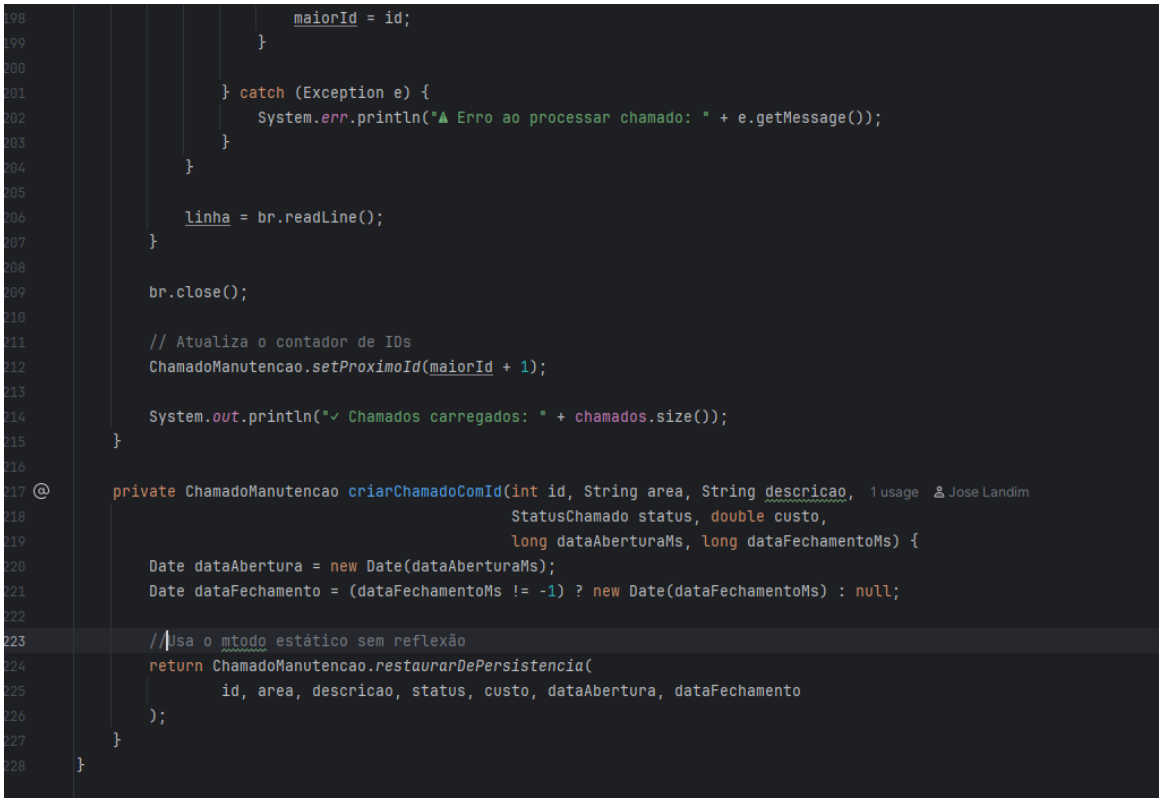
O projeto Sistema de Administração de Condomínios "Vista Alegre" atendeu com sucesso aos requisitos funcionais e não funcionais propostos. A aplicação dos princípios de POO (herança, polimorfismo, encapsulamento e abstração) foi realizada de forma justificada e consistente ao longo de todo o código. A arquitetura escolhida, com clara separação entre entidades, gerenciadores e interface, facilita a manutenção e evolução futura do sistema. O tratamento robusto de exceções garante que o sistema não quebre com entradas inválidas ou situações inesperadas. O sistema está funcional e pronto para uso, com persistência de dados garantindo que informações não sejam perdidas entre execuções. As funcionalidades de reservas, controle de visitantes, gestão financeira e manutenção atendem às necessidades reais de um condomínio. Este projeto consolidou o aprendizado prático dos conceitos de Programação Orientada a Objetos e demonstrou como boas práticas de desenvolvimento resultam em código mais robusto, legível e manutenível.

7. Impressão de uma classe importante

```java
import java.io.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Locale;

public class ControleFinanceiro {  7 usages  Jose Landim +1

    private List<Pagamento> pagamentos;  7 usages
    private List<ChamadoManutencao> chamados;  7 usages

    public ControleFinanceiro() {  3 usages  Gustavo19-code
        this.pagamentos = new ArrayList<>();
        this.chamados = new ArrayList<>();
    }

    public List<Pagamento> getPagamentos() {  10 usages  Gustavo19-code
        return pagamentos;
    }

    public List<ChamadoManutencao> getChamados() {  10 usages  Gustavo19-code +1
        return chamados;
    }









    private Morador buscarMoradorPorDocumento(List<Morador> moradores, String documento) {  1 usage  Jose Landim
        for (Morador m : moradores) {
            if (m.getDocumento().equals(documento)) {
                return m;
            }
        }
        return null;
    }

    // ============================================================
    // VERIFICAR E APLICAR MULTAS AUTOMATICAMENTE
    // ============================================================
    public void verificarPagamentosAtrasados() {  1 usage  Jose Landim
        int multasAplicadas = 0;
```

```
        for (Pagamento p : pagamentos) {
            if (p.estaAtrasado() && p.getStatus() != Pagamento.Status.atrasado) {
                p.verificarAtraso();
                multasAplicadas++;
            }
        }

        if (multasAplicadas > 0) {
            System.out.println("⚠ " + multasAplicadas + " multa(s) aplicada(s) por atraso.");
        }
    }

    // ============================================================
    // ADICIONAR PAGAMENTO COM VERIFICAÇÃO
    // ============================================================
    public void adicionarPagamento(Pagamento p) {  no usages  Jose Landim
        pagamentos.add(p);
        p.verificarAtraso(); // Verifica se já está atrasado ao adicionar
    }

    // ============================================================
    // SALVAR COM MULTAS E STATUS CORRETO
    // ============================================================
    public void salvarPagamentosComMultas(String arquivo) throws IOException {  1 usage  Gustavo19-code +1
        BufferedWriter bw = new BufferedWriter(new FileWriter(arquivo));

        for (Pagamento p : pagamentos) {
            String linha =
                    p.getValor() + ";" +
                            p.getMorador().getDocumento() + ";" +
                            p.getMorador().getNome() + ";" +
                            p.getStatus() + ";" +
                            p.getId() + ";" +
                            p.getMulta(); //salva a multa

            bw.write(linha);
            bw.newLine();
        }
    }

    bw.close();
    System.out.println("✓ Pagamentos atualizados em " + arquivo);
}

// ============================================================
// CARREGAR COM MULTAS
// ============================================================
public void carregarPagamentosComMultas(String arquivo, List<Morador> moradores) throws IOException {  1 usage  Gustavo19-code +1
    File f = new File(arquivo);
    if (!f.exists()) {
        return;
    }

    BufferedReader br = new BufferedReader(new FileReader(f));
    String linha = br.readLine();

    while (linha != null) {
        linha = linha.trim();

        if (!linha.isEmpty()) {
            String[] partes = linha.split( regex: ";");

            double valor = Double.parseDouble(partes[0]);
            String documento = partes[1];
            String nomeIgnorado = partes[2];
            Pagamento.Status status = Pagamento.Status.valueOf(partes[3]);
            int id = Integer.parseInt(partes[4]);
            double multa = (partes.length > 5) ? Double.parseDouble(partes[5]) : 0.0;

            Morador m = buscarMoradorPorDocumento(moradores, documento);

            if (m != null) {
                Pagamento p = new Pagamento(m, id, valor, status);
                p.setMulta(multa); //restaura a multa salva
                pagamentos.add(p);
                m.getPagamentos().add(p);
            }
        }
```

```
            }

            linha = br.readLine();
        }

        br.close();
        System.out.println("✓ Pagamentos carregados: " + pagamentos.size());
    }

    // ============================================================
    // PERSISTÊNCIA DE CHAMADOS
    // ============================================================

    public void salvarChamados(String arquivo) throws IOException {  1 usage  Jose Landim +1
        BufferedWriter bw = new BufferedWriter(new FileWriter(arquivo));

        for (ChamadoManutencao c : chamados) {
            // Formato: ID;AREA;DESCRICAO;STATUS;CUSTO;DATA_ABERTURA_MS;DATA_FECHAMENTO_MS
            //Usa ponto como separador decimal
            String linha = String.format(Locale.US,  format: "%d;%s;%s;%s;%.2f;%d;%d",
                    c.getId(),
                    c.getAreaAfetada(),
                    c.getDescricao().replace( target: ";",  replacement: ","), // Remove ; da descrição
                    c.getStatus().name(),
                    c.getCusto(),
                    c.getDataAbertura().getTime(),
                    c.getDataFechamento() != null ? c.getDataFechamento().getTime() : -1
            );
            bw.write(linha);
            bw.newLine();
        }

        bw.close();
        System.out.println("✓ Chamados salvos: " + chamados.size());
    }

    public void carregarChamados(String arquivo) throws IOException {  1 usage  Jose Landim
        File f = new File(arquivo);
        if (!f.exists()) {
            System.out.println("⚠ Arquivo de chamados não encontrado.");
            System.out.println("⚠ Arquivo de chamados não encontrado.");
            return;
        }

        chamados.clear();
        int maiorId = 0;

        BufferedReader br = new BufferedReader(new FileReader(f));
        String linha = br.readLine();

        while (linha != null) {
            linha = linha.trim();

            if (!linha.isEmpty()) {
                try {
                    String[] partes = linha.split( regex: ";");
                    if (partes.length != 7) {
                        linha = br.readLine();
                        continue;
                    }

                    int id = Integer.parseInt(partes[0]);
                    String area = partes[1];
                    String descricao = partes[2];
                    StatusChamado status = StatusChamado.valueOf(partes[3]);

                    //ubstitui vírgula por ponto antes de converter
                    String custoStr = partes[4].replace( target: ",",  replacement: ".");
                    double custo = Double.parseDouble(custoStr);

                    long dataAberturaMs = Long.parseLong(partes[5]);
                    long dataFechamentoMs = Long.parseLong(partes[6]);

                    // Cria o chamado e restaura seus dados
                    ChamadoManutencao c = criarChamadoComId(id, area, descricao, status,
                            custo, dataAberturaMs, dataFechamentoMs);
                    chamados.add(c);

                    if (id > maiorId) {
                        maiorId = id;
```



8. Bibliotecas Utilizadas e Referências

8.1. Bibliotecas

A solução foi desenvolvida utilizando apenas as bibliotecas padrão do Java Development Kit (JDK), versão 17 ou superior, não havendo dependências externas complexas.

java.io.*: Utilizado para a persistência dos dados de Moradores e Apartamentos em arquivos de texto (FileWriter, BufferedReader).

java.util.*: Utilizado para coleções de dados (List, ArrayList), manipulação de datas (Date, Calendar) e ferramentas de entrada (Scanner).

java.util.stream: Utilizado na classe ControleFinanceiro para realizar operações de agregação (filtragem e soma) de forma funcional e eficiente.

8.2. Referências

Materiais do Curso

Slides da disciplina sobre POO

Exemplos de código fornecidos em aula

Discussões em sala de aula

Oracle Java Documentation

Java Date and Time

Java I/O Tutorial