UNIP – UNIVERSIDADE PAULISTA

CIÊNCIA DA COMPUTAÇÃO (CC)

RIBEIRÃO PRETO/SP

Sistema de Gerenciamento para Pet Shop

Autores:

Luiz Fernando Carvalho Vilarinho da Silva -- RA: R149FG1

Pedro Augusto Nicolau Maximo -- RA: R057HJ9

Guilherme do N L Olympio -- RA: R084698

Caina Cesar de Souza -- RA: G9549H5

ATIVIDADES PRÁTICAS SUPERVISIONADAS (APS)

Professores Responsáveis:

Prof. Dr. Kleython José Coriolano Cavalcanti de Lacerda

Prof. Dr. Lucas Lins de Lima

Prof. Dr. Bruno Azevedo

Youtube: https://youtu.be/Nu9uI4kPcmw

Maio – 2025

Sumário

Maio – 2025

1. Resumo

Este projeto desenvolve um sistema de gerenciamento para pet shops baseado na Programação Orientada a Objetos (POO), utilizando Java. O objetivo é oferecer uma solução acessível para pequenos estabelecimentos do setor, automatizando processos como cadastro de clientes e animais, agendamento de serviços, controle de produtos e gestão financeira.

A escolha de Java se deve à sua robustez e ampla adoção, possibilitando uma arquitetura modular e reutilizável. O sistema é estruturado em classes interligadas, cada uma com responsabilidades específicas, aplicando os conceitos de encapsulamento, herança, polimorfismo e abstração. As principais classes incluem Cliente, Pets, Produto, Serviço, Agendamento e Financeiro.

A metodologia prioriza boas práticas de programação, como reutilização de código e separação de responsabilidades, garantindo um sistema eficiente e sustentável. Os desafios técnicos envolvem a modelagem correta das entidades, a implementação da herança e a integração do módulo financeiro.

Além da funcionalidade, o projeto enfatiza documentação clara e profissional, permitindo futuras expansões, como interface gráfica e integração com banco de dados. O desenvolvimento possibilita ao programador uma vivência completa de análise, modelagem, implementação e documentação, consolidando conhecimentos teóricos em um contexto prático.

2. Introdução

A transformação digital vem impactando significativamente a forma como pequenos negócios operam e se relacionam com seus clientes. No setor de serviços voltados ao cuidado animal, como pet shops, essa realidade não é diferente. A crescente demanda por serviços especializados, aliada ao aumento do número de animais domésticos nos lares brasileiros, exige soluções tecnológicas capazes de organizar e automatizar processos internos, melhorar a eficiência do atendimento e proporcionar uma experiência mais satisfatória ao consumidor final.

Segundo dados da Associação Brasileira da Indústria de Produtos para Animais de Estimação (ABINPET), o Brasil ocupa uma das primeiras posições no ranking mundial em população de animais de estimação, movimentando bilhões de reais por ano em produtos e serviços relacionados ao segmento pet. Apesar do crescimento expressivo desse mercado, a realidade de muitos pet shops de pequeno porte ainda é marcada pela ausência de ferramentas tecnológicas adequadas à gestão do negócio. Muitos empreendedores utilizam planilhas simples ou mesmo registros manuais, o que torna o controle de clientes, serviços e finanças um processo moroso, sujeito a erros e ineficiências operacionais.

Neste cenário, destaca-se a necessidade de soluções digitais acessíveis, funcionais e personalizadas para a realidade de pequenos estabelecimentos. Este trabalho tem como objetivo principal o desenvolvimento de um sistema de gerenciamento para pet shops, estruturado com base nos princípios da Programação Orientada a Objetos (POO). A escolha por esse paradigma deve-se à sua capacidade de representar entidades do mundo real, como clientes, animais, produtos e serviços, de forma modular, coesa e reutilizável, favorecendo a organização do código e a manutenibilidade do sistema.

A linguagem Java foi adotada para o desenvolvimento da aplicação por suas características robustas, seguras e amplamente difundidas no meio acadêmico e profissional. Além disso, sua sintaxe orientada a objetos facilita a aplicação prática dos conceitos fundamentais da POO, como encapsulamento, herança, polimorfismo e abstração. O uso da linguagem também possibilita, futuramente, a integração do sistema com interfaces gráficas, bancos de dados e aplicações web, ampliando sua usabilidade e alcance.

O projeto foi desenvolvido com foco na resolução de um problema real: a dificuldade de gestão enfrentada por pequenos pet shops. Com base nesse propósito, foram implementadas funcionalidades que envolvem o cadastro de clientes e de seus respectivos animais, o registro de serviços prestados, o controle de produtos comercializados e a realização de agendamentos. Um módulo financeiro simples foi criado para registrar os valores associados aos atendimentos realizados, permitindo ao gestor um controle básico da receita gerada.

O público-alvo deste sistema inclui micro e pequenos empreendedores do setor pet, que buscam uma solução prática e sem custos elevados para melhorar sua organização interna, bem como estudantes de programação que desejam aprofundar seus conhecimentos em desenvolvimento orientado a objetos por meio de um caso real e aplicado. Durante o desenvolvimento, foram utilizadas ferramentas como o NetBeans para implementação do código, e diagramas de classes foram elaborados de forma conceitual com base na UML (Unified Modeling Language), com o objetivo de estruturar as relações entre os elementos do sistema antes da implementação.

Dessa forma, este projeto propicia uma experiência completa de construção de software, unindo teoria e prática e estimulando o uso de boas práticas de codificação. A introdução de conceitos técnicos em um contexto aplicado contribui significativamente para a formação de desenvolvedores mais preparados para os desafios do mercado de trabalho. Ao final, espera-se obter um sistema funcional, modular,

extensível e documentado de forma clara e objetiva, demonstrando não apenas a viabilidade técnica da proposta, mas também seu potencial de contribuição social e educacional.

3. Tema Escolhido

A escolha do tema deste projeto — o desenvolvimento de um sistema de gerenciamento para pet shop — baseia-se tanto na observação de uma demanda real de mercado quanto na viabilidade técnica e acadêmica da proposta. Com o crescimento expressivo do setor pet no Brasil e a multiplicação de estabelecimentos de pequeno porte voltados ao atendimento de animais de estimação, tornou-se evidente a carência de ferramentas específicas que atendam às necessidades gerenciais desses negócios de maneira acessível e eficiente.

Grande parte dos pet shops de bairro enfrenta desafios operacionais relacionados à organização de informações, controle de agendamentos, histórico de atendimento, estoque de produtos e movimentação financeira. Muitas dessas tarefas ainda são realizadas de forma manual ou com o uso de planilhas básicas, o que compromete a agilidade e a confiabilidade dos dados. Nesse contexto, a automação desses processos por meio de um sistema computacional representa não apenas uma solução prática, mas também uma oportunidade de agregar valor ao negócio por meio da melhoria na qualidade dos serviços prestados e na experiência do cliente.

A proposta do sistema desenvolvido neste trabalho justifica-se, portanto, pela possibilidade de preencher essa lacuna com uma solução simples, funcional e adaptável à realidade dos pequenos empreendedores do setor. A ideia é oferecer uma ferramenta que não exija altos investimentos em tecnologia, mas que seja suficientemente completa para atender às principais demandas do cotidiano do pet shop, promovendo maior controle, organização e segurança das informações.

Do ponto de vista acadêmico, a escolha do tema também se mostra pertinente por permitir a aplicação prática dos conteúdos estudados ao longo do curso, especialmente no que se refere à

Programação Orientada a Objetos (POO). O projeto proporciona uma oportunidade concreta de consolidar conhecimentos em modelagem de sistemas, implementação de classes e métodos, tratamento de dados e boas práticas de codificação. Além disso, favorece o desenvolvimento de habilidades como análise crítica, resolução de problemas e documentação técnica — competências valorizadas tanto no ambiente acadêmico quanto no mercado profissional.

Outro fator que reforça a relevância do tema é a possibilidade de expansão futura do sistema. Como o projeto foi concebido com uma estrutura modular, ele pode ser aprimorado e adaptado com relativa facilidade para incluir novas funcionalidades, como integração com bancos de dados, interfaces gráficas, acesso remoto ou relatórios gerenciais mais detalhados. Dessa forma, o sistema deixa de ser apenas um exercício acadêmico e passa a configurar-se como uma base sólida para aplicações reais ou projetos de extensão.

Em suma, a escolha do tema atende a múltiplos objetivos: responde a uma necessidade concreta do mercado, proporciona um ambiente adequado para aplicação dos conhecimentos teóricos, estimula o desenvolvimento de competências técnicas e oferece margem para evolução e aprofundamento do projeto. Por esses motivos, considera-se que o desenvolvimento de um sistema de gerenciamento para pet shop representa uma proposta válida, coerente e de grande potencial acadêmico e prático. Fundamentos da Programação Orientada a Objetos.

4. Dissertação

A crescente demanda por eficiência nos processos internos de pequenos negócios tem incentivado a adoção de soluções tecnológicas simples, mas funcionais. Entre esses estabelecimentos, os pet shops se destacam por oferecer múltiplos serviços e produtos voltados ao cuidado de animais de estimação, exigindo uma gestão organizada e dinâmica. No entanto, muitos desses empreendimentos ainda operam sem ferramentas adequadas para gerenciar suas atividades, o que pode comprometer o atendimento e a administração geral.

Neste cenário, este trabalho propõe o desenvolvimento de um sistema de gerenciamento específico para pet shops, com foco na automação de processos como cadastro de clientes e animais, controle de serviços e agendamentos, e acompanhamento financeiro. O projeto utiliza os princípios da Programação Orientada a Objetos (POO), aplicada por meio da linguagem Java, conhecida por sua robustez, portabilidade e ampla aceitação acadêmica e profissional.

Além de atender a uma demanda prática do mercado, o projeto oferece uma oportunidade de aplicar conceitos teóricos em um caso real, permitindo consolidar conhecimentos sobre estruturação de sistemas, boas práticas de codificação e documentação técnica. Dessa forma, o desenvolvimento do sistema contribui tanto para o aprimoramento da formação profissional quanto para a criação de uma solução acessível e útil para pequenos negócios do setor pet.

4.1 Fundamentos da Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de desenvolvimento amplamente utilizado na construção de sistemas de software, especialmente quando há a necessidade de representar estruturas complexas de maneira modular, reutilizável e próxima da realidade. Neste projeto, a POO foi essencial para modelar os processos e entidades típicas de um pet shop, como clientes, animais, produtos, serviços e agendamentos.

Diferente da programação estruturada, que se baseia em funções e fluxos sequenciais, a POO organiza o código a partir de objetos, que são instâncias de classes. Cada classe encapsula dados (atributos) e comportamentos (métodos), permitindo que o sistema seja desenvolvido de maneira coesa e extensível. No contexto deste trabalho, esse paradigma não apenas facilitou o desenvolvimento e a organização do código, como também reforçou boas práticas como o reuso, a manutenção e a clareza do sistema.

4.1.1 Conceitos Gerais

Os quatro pilares fundamentais da Programação Orientada a Objetos — encapsulamento, herança, polimorfismo e abstração — foram aplicados de forma prática e didática ao longo do projeto. A seguir, cada conceito é apresentado de forma resumida:

- Encapsulamento: Refere-se à proteção dos dados internos das classes, permitindo que sejam acessados ou modificados apenas por meio de métodos específicos. No projeto, por exemplo, a classe Cliente mantém atributos como nome, CPF e telefone como privados,

acessíveis apenas via métodos públicos (getters), garantindo maior controle sobre os dados inseridos.

- Herança: Permite que uma classe herde atributos e métodos de outra. Embora o sistema atual não implemente herança entre classes específicas (como Cachorro e Gato a partir de Pets), a estrutura está preparada para isso. A classe Pets, por exemplo, pode futuramente servir de base para especializações.
- Polimorfismo: Consiste na possibilidade de um mesmo método ter comportamentos diferentes dependendo do contexto ou da classe que o implementa. Esse conceito pode ser utilizado, por exemplo, para diferenciar serviços como banho e consulta, caso sejam criadas subclasses de Servico com métodos específicos.
- Abstração: Trata-se de representar entidades do mundo real de forma simplificada no sistema, ocultando detalhes internos e expondo apenas o que é necessário. As classes do projeto, como Produto ou Agendamento, apresentam esse princípio ao isolar a complexidade da lógica de cadastro e associação de dados.

4.1.2 – Aplicação da POO no Sistema

O sistema de gerenciamento para pet shop foi desenvolvido integralmente com base nos conceitos de POO. A modelagem das classes procurou representar de maneira fiel a realidade de uma loja de serviços e produtos voltados para animais domésticos. Por meio da abstração, as principais entidades foram representadas como objetos independentes e inter-relacionados.

A modularidade permitida pelo paradigma orientado a objetos possibilitou uma separação clara das responsabilidades, onde cada classe tem um papel bem definido. Isso facilita a manutenção e futura

expansão do sistema, além de contribuir para a legibilidade do código. Como exemplo prático, a classe Agendamento atua como elo entre outras classes como Cliente, Pets e Servico, exemplificando o uso da composição entre objetos.

Além disso, o uso de encapsulamento garantiu a integridade dos dados sensíveis, como CPF e dados financeiros. Métodos públicos controlam a entrada e saída dessas informações, protegendo o sistema contra uso incorreto ou malicioso. Embora o sistema ainda esteja em estágio inicial, sua arquitetura orientada a objetos proporciona uma base sólida para avanços futuros, como a implementação de interface gráfica, persistência em banco de dados ou integração com API externa.

4.2 – Classes e Relacionamentos

O sistema de gerenciamento para pet shop foi construído com base em classes que representam entidades reais do domínio do problema. Essas classes interagem entre si de forma a espelhar o funcionamento de um pet shop, permitindo operações como cadastro de clientes e pets, agendamentos de serviços, registro de produtos e controle financeiro. A relação entre essas classes foi pensada de forma modular, favorecendo a legibilidade, a manutenção e a escalabilidade do sistema.

As principais classes são: Cliente, Pets, Produto, Servico, Agendamento e Financeiro. Cada uma dessas entidades foi modelada com atributos e métodos específicos para cumprir sua função dentro do sistema.

4.2.1 – Classe Cliente, Pets e Agendamento

A classe Cliente é responsável por armazenar as informações básicas do cliente, como nome, CPF, telefone e endereço. Essa classe possui uma composição com a classe Pets, indicando que um cliente pode possuir vários animais cadastrados. Os pets são mantidos em uma lista interna, protegida por encapsulamento, e acessada apenas por métodos definidos na própria classe.

```
public class Cliente {
    private final String nome;      // Nome completo do cliente
    private final String cpf;       // CPF do cliente formatado corretamente
    private final String telefone;  // Telefone do cliente
    private final String endereco;  // Endereço do cliente
    private final List<Pets> pets;   // Lista de pets associados ao cliente
```

A classe Pets, por sua vez, representa os animais de estimação vinculados a um cliente. Ela armazena informações como nome, espécie, idade e outras características relevantes. Embora atualmente a classe Pets seja genérica, a estrutura permite expansão para heranças futuras, como Cachorro, Gato ou outras subclasses específicas.

Já a classe Agendamento é o elo entre cliente, pet e o serviço que será prestado. Essa classe agrupa em um único objeto os dados de quem solicitou o serviço, para qual pet, qual tipo de serviço e em qual data. Essa associação múltipla é fundamental para o funcionamento do sistema, e representa um relacionamento de composição entre as demais entidades.

4.2.2 – Classe Servico e Financeiro

A classe Servico é utilizada para definir os tipos de serviços prestados pelo pet shop, como banho, tosa, consulta veterinária, entre outros. Cada serviço possui atributos como descrição e valor. O sistema foi desenvolvido com flexibilidade para permitir que esses serviços sejam criados dinamicamente, ou seja, o código não restringe os tipos de serviço possíveis.

```
public class Servico {
    private final String nomeServico; // Nome do serviço realizado
    private final Pets pet; // Pet que receberá o serviço
    private final int tempo; // Tempo estimado para a realização do serviço em minutos
    private final double preco; // Preço do serviço com base no porte do pet
```

O valor de cada serviço é utilizado em conjunto com a classe Financeiro, responsável pelo controle financeiro do sistema. Essa classe acumula os valores de todos os serviços finalizados, permitindo o acompanhamento da receita gerada pelo pet shop.

```
private double recebimento;      // Valor total recebido
private int servicoFeitos;       // Quantidade de serviços realizados
private String metodoPagamento;  // Método de pagamento utilizado
private LocalDate dataRegistro;  // Data do registro financeiro
private double despesas = 0.0;   // Total de despesas associadas

public Financeiro(double recebimento, int servicoFeitos, String metodoPagamento, LocalDate dataRegistro) {
    if (recebimento < 0) throw new IllegalArgumentException("O valor de recebimento não pode ser negativo.");
    if (servicoFeitos < 0) throw new IllegalArgumentException("Quantidade de serviços feitos não pode ser negativa.");
    if (metodoPagamento == null || metodoPagamento.isBlank()) throw new IllegalArgumentException("Método de pagamento inválido.");
    if (dataRegistro.isAfter(LocalDate.now())) throw new IllegalArgumentException("Data futura não permitida.");
```

Essa integração entre Agendamento, Servico e Financeiro é essencial para o funcionamento do sistema, pois garante que a execução de um serviço reflita diretamente na movimentação financeira. Além disso, essa estrutura modular facilita futuras implementações, como integração com métodos de pagamento, emissão de notas ou visualização em dashboards.

4.3 – Funcionalidades do Sistema

O sistema de gerenciamento para pet shop foi desenvolvido com o objetivo de atender às principais demandas operacionais desse tipo de estabelecimento. As funcionalidades foram planejadas de forma a abranger o ciclo completo de atendimento, desde o cadastro do cliente até o controle financeiro. Todas as ações disponíveis foram implementadas utilizando os conceitos de Programação Orientada a Objetos e organizadas de forma modular para facilitar o entendimento, manutenção e futura expansão do sistema.

As funcionalidades estão divididas por áreas de atuação: cadastro, agendamento, serviços e produtos, controle financeiro e relatórios via terminal. A seguir, descreve-se detalhadamente cada uma dessas funcionalidades.

4.3.1 – Funcionalidades por Módulo

Cadastro de Clientes e Pets

O sistema permite o registro completo dos clientes, incluindo informações como nome, CPF, telefone e endereço. Durante o cadastro, é possível associar um ou mais animais de estimação a cada cliente. O sistema garante a integridade dos dados e mantém a associação entre cliente e pets por meio de composição de objetos.

- Cadastro de cliente com validação de CPF.
- Cadastro de pets com atributos como nome, espécie, idade e outros.
- Associação de pets ao cliente correspondente.

Registro de Serviços e Produtos

O sistema permite o cadastro de serviços (como banho, tosa e consulta) e produtos (como rações, brinquedos e medicamentos). Cada serviço contém uma descrição e um valor associado. Os produtos, por sua vez, possuem nome e valor unitário.

- Cadastro de serviços com preços configuráveis.
- Inclusão de produtos vendidos no pet shop.

Agendamento de Serviços

A funcionalidade de agendamento permite combinar cliente, pet, serviço e data. O objetivo é organizar a agenda de atendimentos e evitar conflitos ou sobreposição. Essa funcionalidade também prepara os dados para alimentar o módulo financeiro.

- Seleção de cliente e pet previamente cadastrados.
- Escolha do tipo de serviço e data de execução.
- Geração de instância da classe Agendamento.

Controle Financeiro

O sistema realiza o controle financeiro básico, somando os valores dos serviços prestados. Toda vez que um serviço é agendado, seu valor pode ser registrado no módulo financeiro, permitindo acompanhar o total faturado.

- Acúmulo de receita com base nos serviços prestados.
- Visualização de faturamento total no terminal.

Impressão de Dados e Relatórios

Embora o sistema ainda não possua uma interface gráfica, todas as informações podem ser exibidas no terminal por meio de métodos toString() e relatórios simples. Isso permite que o usuário visualize clientes, pets, agendamentos, produtos e totais financeiros diretamente pela saída padrão.

- Relatórios de clientes e pets cadastrados.
- Listagem de agendamentos por data ou cliente.
- Exibição do faturamento acumulado.

4.3.2 – Fluxo do Usuário

O fluxo de uso do sistema ocorre geralmente da seguinte forma:

1. Cadastro do cliente;
2. Cadastro de pets vinculados ao cliente;
3. Registro dos serviços disponíveis;
4. Agendamento de um serviço com cliente, pet, tipo de serviço e data;
5. Registro do valor no módulo financeiro;
6. Visualização dos dados e relatórios.

Essa sequência representa o funcionamento básico de um pet shop, e o sistema foi desenvolvido com o intuito de simular esse ciclo de maneira fiel. Cada etapa é realizada por meio de interações com objetos Java, sem a necessidade de interface gráfica, mas com foco no comportamento lógico do sistema.

4.4 – Códigos de Implementação

Nesta seção, são apresentados trechos representativos do código-fonte desenvolvido para o sistema de gerenciamento de pet shop. O objetivo é demonstrar a aplicação dos conceitos da Programação Orientada a Objetos (POO), como encapsulamento, herança, abstração e polimorfismo, além de evidenciar decisões de design adotadas ao longo do projeto.

Todos os códigos foram escritos em linguagem Java, utilizando pacotes organizados e nomenclatura clara, o que facilita a manutenção e a escalabilidade do sistema. Os exemplos a seguir foram escolhidos por representarem partes cruciais da estrutura e funcionamento do sistema.

4.4.1 – Trechos Comentados

Classe Cliente – Encapsulamento e Composição

A classe Cliente é responsável por armazenar os dados de um cliente e manter a lista de seus pets associados. Os atributos são declarados como private para garantir o encapsulamento, e os acessos são feitos via métodos públicos (getters e setters).

```java
public class Cliente {
    private final String nome;      // Nome completo do cliente
    private final String cpf;       // CPF do cliente formatado corretamente
    private final String telefone;  // Telefone do cliente
    private final String endereco;  // Endereço do cliente
    private final List<Pets> pets;   // Lista de pets associados ao cliente

    public Cliente(String nome, String cpf, String telefone, String endereco) {
        validarCampoObrigatorio(nome, "Nome");
        validarCPF(cpf);
        validarCampoObrigatorio(telefone, "Telefone");
        validarCampoObrigatorio(endereco, "Endereço");

        this.nome = nome.trim();
        this.cpf = formatarCPF(cpf);
        this.telefone = telefone.trim();
        this.endereco = endereco.trim();
        this.pets = new ArrayList<>(); // Inicializa a lista de pets do cliente
    }

    // Getters
    public String getNome() {
        return nome;
    }
    public String getCpf() {
        return cpf;
    }
    public String getTelefone() {
        return telefone;
    }
    public String getEndereco() {
        return endereco;
```

Classe Pets – Estrutura Base para Heranças Futuras

Embora o sistema ainda não implemente subclasses como Cachorro ou Gato, a estrutura da classe Pets foi desenvolvida com base em uma possível herança futura, respeitando o princípio da extensibilidade.

```java
public class Pets {
    private String nome;
    private String especie;
    private int idade;

    public Pets(String nome, String especie, int idade) {
        this.nome = nome;
        this.especie = especie;
        this.idade = idade;
    }

    @Override
    public String toString() {
        return nome + " (" + especie + "), " + idade + " anos";
    }
}
```

Classe Agendamento – Composição de Entidades

A classe Agendamento conecta as entidades Cliente, Pets e Servico, funcionando como ponto central do fluxo de operação do sistema.

```java
public class Agendamento {
    private Cliente cliente;
    private Pets pet;
    private Servico servico;
    private LocalDate data;

    public Agendamento(Cliente cliente, Pets pet, Servico servico, LocalDate data) {
        this.cliente = cliente;
        this.pet = pet;
        this.servico = servico;
        this.data = data;
    }

    @Override
    public String toString() {
        return "Agendamento de " + cliente.getNome() + " para o pet " + pet.getNome()
                + " no dia " + data + " - Serviço: " + servico.getNomeServico();
    }
}
```

Classe Financeiro – Registro de Receita

O controle financeiro é feito por meio da classe Financeiro, que registra o total faturado a partir dos serviços realizados.

```
public class Financeiro {
    private double totalFaturado = 0;

    public void registrarReceita(double valor) {
        totalFaturado += valor;
    }

    public double getTotalFaturado() {
        return totalFaturado;
    }
}
```

Esses trechos representam apenas parte da implementação. O código completo está organizado em pacotes dentro do projeto Java, respeitando os princípios da modularidade, clareza e reaproveitamento de código. A implementação técnica do sistema reforça os objetivos pedagógicos do trabalho, proporcionando uma aplicação prática dos conceitos aprendidos.

4.5 – Análise dos Resultados

A análise dos resultados obtidos com o sistema de gerenciamento para pet shop visa avaliar a eficácia da aplicação, a coerência entre o planejamento e a execução, bem como verificar o funcionamento das principais funcionalidades implementadas.

O sistema foi executado com sucesso no ambiente Java, utilizando NetBeans. A interação com o usuário foi realizada exclusivamente via

terminal, por meio de impressões formatadas que simulam relatórios simples, exibição de dados e validação de ações.

4.5.1 – Execução e Saídas no Terminal

Ao cadastrar clientes, pets, serviços e realizar agendamentos, o sistema demonstrou comportamento esperado, preservando a integridade dos dados e mantendo os relacionamentos corretos entre as classes. A imagem abaixo representa um exemplo real da execução do sistema:

```
=== PETSHOP ===
1. Cadastrar Pet
2. Listar Pets
3. Novo Agendamento
4. Ver Hist�rico de Agendamentos
5. Outros (Financeiro e Produtos)
6. Sair
Op��o: Escolha uma op��o: 1

Nome do cliente: Luiz
CPF do cliente: 444.999.888-10
Telefone do cliente: 16 99999-9999
Endere�o do cliente: R.123
? Cliente cadastrado com sucesso!
Nome do pet: Chico
Esp�cie (Cachorro/Gato): Cachorro
Peso (kg): 5
Data de nascimento (dd/MM/yyyy): 01/03/2024
? Pet cadastrado com sucesso!

=== PETSHOP ===
1. Cadastrar Pet
2. Listar Pets
3. Novo Agendamento
4. Ver Hist�rico de Agendamentos
5. Outros (Financeiro e Produtos)
6. Sair
Op��o: Escolha uma op��o: 2

? Lista de Pets Cadastrados e seus Donos:
```

```
=== PETSHOP ===
1. Cadastrar Pet
2. Listar Pets
3. Novo Agendamento
4. Ver Hist�rico de Agendamentos
5. Outros (Financeiro e Produtos)
6. Sair
Op��o: Escolha uma op��o: 2

? Lista de Pets Cadastrados e seus Donos:

Cliente:
Nome: Luiz
CPF: 444.999.888-10
Telefone: 16 99999-9999
Endere�o: R.123
Total de Pets: 1


Pets de Luiz (1):
----------------------------------------
Nome: Chico
Esp�cie: Cachorro
Idade: 1 anos
Peso: 5,00 kg
Porte: Pequeno
Nascimento: 01/03/2024
```

Nessa execução, é possível observar:

- Impressão do cliente e seu(s) pet(s) associado(s);
- Lista de serviços cadastrados com seus respectivos valores;
- Confirmação de agendamentos realizados com data, pet e serviço;

A formatação clara das saídas permitiu ao usuário interpretar facilmente as informações, mesmo em um ambiente sem interface gráfica.

4.5.2 – Avaliação das Funcionalidades

As principais funcionalidades do sistema foram validadas com sucesso:

Funcionalidade	**Resultado Obtido**
Cadastro de cliente e pet	Dados armazenados corretamente
Registro de serviços e produtos	Inserção e exibição via terminal
Agendamento de serviços	Relacionamento funcional entre cliente, pet e data
Impressão de relatórios	Apresentação textual clara das entidades

A arquitetura baseada em POO mostrou-se eficaz, permitindo a separação de responsabilidades e facilidade de manutenção do código. A associação entre classes, o uso de listas e métodos de acesso (getters, setters) garantiram o bom funcionamento mesmo em cenários com múltiplos dados relacionados.

4.5.3 – Relatório Técnico

Introdução – Classe Cliente

O gerenciamento de clientes em um pet shop exige organização e confiabilidade na coleta de dados. A classe Cliente foi desenvolvida para estruturar o armazenamento de informações dos clientes e seus respectivos pets, garantindo registros seguros e consistentes. Este módulo permite o cadastro estruturado de clientes, validando dados essenciais como nome, CPF, telefone e endereço, além de possibilitar a associação de múltiplos pets sem comprometer a integridade dos dados.

Problemas e Requisitos

- Problemas Identificados

- Cadastro de clientes com dados incompletos.
- CPF inválido ou mal formatado.
- Conflitos na associação de múltiplos pets ao cliente.
- Alterações indevidas nos dados cadastrados.

- Requisitos Pensados

- Validação dos dados pessoais antes da criação do cliente.
- Formatação automática do CPF.

- Gestão segura da lista de pets, impedindo modificações externas.

- Mensagens explicativas para erros encontrados.

Introdução – Classe Pet

O cadastro de pets em um sistema exige precisão na coleta de dados para garantir registros consistentes e evitar inconsistências. A classe Pets foi desenvolvida para representar as características essenciais dos animais, com validações rigorosas para assegurar que apenas informações corretas sejam armazenadas. Além do cadastro seguro, este módulo realiza cálculos automáticos para definir o porte do pet com base no peso, prevenindo erros na categorização.

Problemas e Requisitos

- Problemas Identificados

- Cadastro de pets sem nome válido.

- Restrição na aceitação de espécies (apenas cães e gatos).

- Peso inadequado (zero ou negativo).

- Datas de nascimento incorretas (futuras).

- Requisitos Pensados

- Validação completa dos atributos antes do cadastro.

- Apenas cães e gatos devem ser aceitos como entrada.

- Definição automática do porte do pet com base no peso.

- Mensagens de erro claras e informativas.

Introdução – Classe Serviço

O gerenciamento de serviços em um pet shop exige precisão nos registros e transparência na precificação dos atendimentos. A classe Serviço foi desenvolvida para estruturar a oferta de serviços aos pets, garantindo que preços e tempos sejam calculados corretamente com base no porte do animal. Este módulo padroniza a criação de serviços, assegurando que cada atendimento siga regras pré-definidas e prevenindo erros através de validações rigorosas.

Problemas e Requisitos

- Problemas Identificados

- Cálculo incorreto de preços para diferentes portes de pet.

- Tempo de atendimento indefinido para serviços.

- Registro de serviços inválidos.

- Prevenção contra erros de entrada.

- Requisitos Pensados

- Padronização de preços dos serviços conforme o porte do pet.

- Definição automática do tempo estimado para cada serviço.

- Cadastro restrito a serviços disponíveis no pet shop.

- Mensagens de erro claras e explicativas.

Introdução – Classe Produto

O gerenciamento de produtos em um pet shop exige organização e controle rigoroso dos itens disponíveis para venda. A classe Produto foi desenvolvida para estruturar o armazenamento e manipulação de produtos, permitindo a administração eficiente de estoque, preços e categorias. Este módulo assegura que todas as informações sejam verificadas antes da inserção no sistema e possibilita operações como reajustes de preço e controle de estoque.

Problemas e Requisitos

- Problemas Identificados

- Cadastro de produtos sem nome válido.
- Estoque negativo ou inconsistente.
- Preços inadequados ou zerados.
- Categorização incorreta.

- Requisitos Pensados

- Validação dos atributos essenciais antes da criação do produto.
- Ajuste automático do preço ao aplicar um desconto.
- Modificação segura do estoque, impedindo valores inválidos.
- Mensagens de erro claras e explicativas.

Introdução – Classe Agendamento

O gerenciamento de serviços em um pet shop exige organização e controle de horários para garantir um atendimento eficiente e evitar conflitos operacionais. A classe Agendamento foi desenvolvida com o propósito de estruturar e validar marcações de serviços, assegurando que os registros sigam regras operacionais predefinidas. O módulo previne inconsistências como agendamentos em datas inválidas, horários fora do expediente ou registros duplicados, através de verificações robustas de integridade dos dados.

Problemas e Requisitos

- Problemas Identificados

- Agendamentos inválidos (datas no passado).
- Horários fora do expediente.
- Falta de padronização nos registros.
- Ausência de mensagens claras de erro.

- Requisitos Pensados

- Validação da data e hora do agendamento antes da confirmação.
- Respeito aos horários de funcionamento do pet shop.
- Prevenção contra modificações indevidas em registros validados.
- Mensagens de erro detalhadas e intuitivas.

Introdução – Classe Financeiro

A classe Financeiro foi criada para gerenciar de forma eficiente as operações financeiras de um pet shop, garantindo controle preciso dos recebimentos, despesas e saldo disponível. O módulo assegura integridade nos registros e evita inconsistências nos dados financeiros.

Problemas e Requisitos

• Problemas Identificados

- Cadastro de valores incorretos (valores negativos).
- Datas de registro inconsistentes (datas futuras).
- Falta de padronização nos métodos de pagamento.
- Cálculo inadequado do saldo financeiro.

- Requisitos Pensados

- Validação dos valores financeiros antes da inserção no sistema.

- Cálculo automático do saldo, subtraindo despesas dos recebimentos.

- Registro correto das despesas, garantindo que valores inválidos não sejam aceitos.

- Mensagens de erro claras e explicativas para facilitar correções.

5 – Considerações Finais

O desenvolvimento deste sistema de gerenciamento para pet shop representou uma aplicação prática valiosa dos princípios da Programação Orientada a Objetos (POO) no contexto acadêmico. O projeto teve como foco principal a criação de uma solução funcional, escrita em Java, capaz de gerenciar clientes, pets, serviços, produtos e agendamentos de forma organizada e modular.

Os resultados obtidos com a execução do sistema demonstraram a eficácia das decisões de modelagem e a solidez da estrutura construída. Mesmo utilizando apenas recursos de entrada e saída via terminal, foi possível simular as operações principais de um pequeno negócio do ramo pet, validando a usabilidade da aplicação e a coerência dos relacionamentos entre as classes.

O projeto atendeu aos objetivos propostos, permitindo que os pilares da POO — encapsulamento, herança, polimorfismo e abstração — fossem incorporados de forma clara ao longo da implementação. O sistema foi construído com atenção a boas práticas de programação, como separação de responsabilidades, reutilização de código e organização por pacotes.

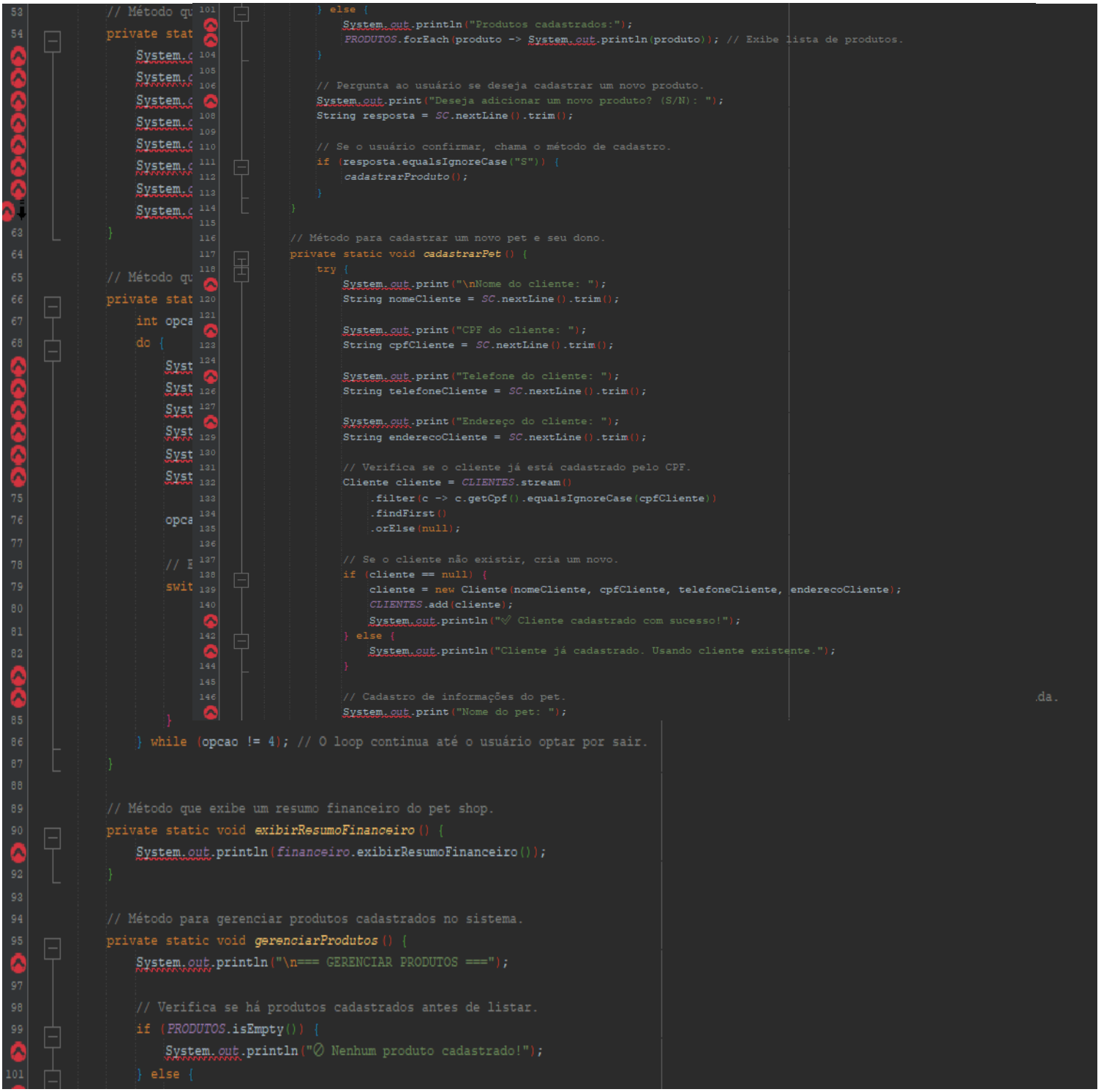
Embora tenha limitações como a ausência de persistência em banco de dados e interface gráfica, a base do sistema está preparada para futuras evoluções. Entre as possibilidades estão a criação de uma interface visual (GUI), armazenamento de dados em arquivos ou banco de dados SQL, integração com sistema de login, e geração de relatórios interativos.

Conclui-se, portanto, que o projeto cumpriu seu papel acadêmico e ainda pode servir como ponto de partida para soluções reais de software voltadas a pequenos empreendedores do ramo pet, oferecendo praticidade e controle por meio da tecnologia.

6 – Linhas de código

6.1 Classe principal – PetShop

```java
package aps3;

import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.*;

// Classe principal do sistema de pet shop, responsável pela gestão de clientes, agendamentos, produtos e financeiro.
public class PetShop {

    // Scanner usado para entrada de dados do usuário.
    // Definido como constante para evitar múltiplas instâncias ao longo da execução.
    private static final Scanner SC = new Scanner(System.in);

    // Listas para armazenar clientes, agendamentos e produtos cadastrados no sistema.
    private static final List<Cliente> CLIENTES = new ArrayList<>();
    private static final List<Agendamento> AGENDAMENTOS = new ArrayList<>();
    private static final List<Produto> PRODUTOS = new ArrayList<>();

    // Objeto responsável pela gestão financeira, inicializado com valores padrão.
    private static final Financeiro financeiro = new Financeiro(0, 0, "Indefinido", LocalDate.now());

    // Lista imutável contendo os serviços oferecidos pelo pet shop.
    private static final List<String> SERVIÇOS_VALIDOS = List.of(
        "Banho", "Tosa Tesoura", "Tosa Máquina", "Tosa Bebê",
        "Tosa Higiênica", "Corte de Unha", "Limpeza de Ouvido",
        "Hidratação", "Remoção de Subpelos"
    );

    // Método principal do sistema, responsável pelo fluxo de interação com o usuário.
    public static void main(String[] args) {
        // Utilização do try-with-resources para garantir que o Scanner seja fechado corretamente ao final da execução.
        try (SC) {
            int opcao;
            do {
                exibirMenu(); // Exibe o menu principal ao usuário.
                opcao = lerOpcao(); // Método para capturar a opção escolhida.

                // Estrutura de decisão baseada na opção do usuário.
                switch (opcao) {
                    case 1 -> cadastrarPet(); // Método para cadastrar um novo pet.
                    case 2 -> listarPets(); // Método para listar todos os pets cadastrados.
                    case 3 -> realizarAgendamento(); // Método para criar um novo agendamento.
                    case 4 -> exibirHistoricoAgendamentos(); // Método para visualizar agendamentos passados.
                    case 5 -> exibirOutros(); // Submenu com outras funcionalidades.
                    case 6 -> System.out.println("Saindo... Obrigado por usar o sistema!"); // Encerra o programa.
                    default -> System.out.println("Opção inválida! Escolha uma opção válida."); // Mensagem de erro para entrada inválida.
                }
            } while (opcao != 6); // Loop continua até que o usuário escolha a opção de sair.
```



```
    // Método qu
    private stat
        System.
        System.
        System.
        System.
        System.
        System.
        System.
        System.
    }

    // Método qu
    private stat
        int opca
        do {
            Syst
            Syst
            Syst
            Syst
            Syst
            Syst

            opca

            // E
            swit
            }
        } while (opcao != 4); // O loop continua até o usuário optar por sair.
    }

    // Método que exibe um resumo financeiro do pet shop.
    private static void exibirResumoFinanceiro() {
        System.out.println(financeiro.exibirResumoFinanceiro());
    }

    // Método para gerenciar produtos cadastrados no sistema.
    private static void gerenciarProdutos() {
        System.out.println("\n=== GERENCIAR PRODUTOS ===");

        // Verifica se há produtos cadastrados antes de listar.
        if (PRODUTOS.isEmpty()) {
            System.out.println("⚠ Nenhum produto cadastrado!");
        } else {
            System.out.println("Produtos cadastrados:");
            PRODUTOS.forEach(produto -> System.out.println(produto)); // Exibe lista de produtos.
        }

        // Pergunta ao usuário se deseja cadastrar um novo produto.
        System.out.print("Deseja adicionar um novo produto? (S/N): ");
        String resposta = SC.nextLine().trim();

        // Se o usuário confirmar, chama o método de cadastro.
        if (resposta.equalsIgnoreCase("S")) {
            cadastrarProduto();
        }
    }

    // Método para cadastrar um novo pet e seu dono.
    private static void cadastrarPet() {
        try {
            System.out.print("\nNome do cliente: ");
            String nomeCliente = SC.nextLine().trim();

            System.out.print("CPF do cliente: ");
            String cpfCliente = SC.nextLine().trim();

            System.out.print("Telefone do cliente: ");
            String telefoneCliente = SC.nextLine().trim();

            System.out.print("Endereço do cliente: ");
            String enderecoCliente = SC.nextLine().trim();

            // Verifica se o cliente já está cadastrado pelo CPF.
            Cliente cliente = CLIENTES.stream()
                .filter(c -> c.getCpf().equalsIgnoreCase(cpfCliente))
                .findFirst()
                .orElse(null);

            // Se o cliente não existir, cria um novo.
            if (cliente == null) {
                cliente = new Cliente(nomeCliente, cpfCliente, telefoneCliente, enderecoCliente);
                CLIENTES.add(cliente);
                System.out.println("✅ Cliente cadastrado com sucesso!");
            } else {
                System.out.println("Cliente já cadastrado. Usando cliente existente.");
            }

            // Cadastro de informações do pet.                                                    .da.
            System.out.print("Nome do pet: ");
```

```
        } else {
            System.out.println("Produtos cadastrados:");
            PRODUTOS.forEach(produto -> System.out.println(produto)); // Exibe lista de produtos.
        }

        // Pergunta ao usuário se deseja cadastrar um novo produto.
        System.out.print("Deseja adicionar um novo produto? (S/N): ");
        String resposta = SC.nextLine().trim();

        // Se o usuário confirmar, chama o método de cadastro.
        if (resposta.equalsIgnoreCase("S")) {
            cadastrarProduto();
        }
    }

    // Método para cadastrar um novo pet e seu dono.
    private static void cadastrarPet() {
        try {
            System.out.print("\nNome do cliente: ");
            String nomeCliente = SC.nextLine().trim();

            System.out.print("CPF do cliente: ");
            String cpfCliente = SC.nextLine().trim();

            System.out.print("Telefone do cliente: ");
            String telefoneCliente = SC.nextLine().trim();

            System.out.print("Endereço do cliente: ");
            String enderecoCliente = SC.nextLine().trim();

            // Verifica se o cliente já está cadastrado pelo CPF.
            Cliente cliente = CLIENTES.stream()
                .filter(c -> c.getCpf().equalsIgnoreCase(cpfCliente))
                .findFirst()
                .orElse(null);

            // Se o cliente não existir, cria um novo.
            if (cliente == null) {
                cliente = new Cliente(nomeCliente, cpfCliente, telefoneCliente, enderecoCliente);
                CLIENTES.add(cliente);
                System.out.println("✅ Cliente cadastrado com sucesso!");
            } else {
                System.out.println("Cliente já cadastrado. Usando cliente existente.");
            }

            // Cadastro de informações do pet.
            System.out.print("Nome do pet: ");
```

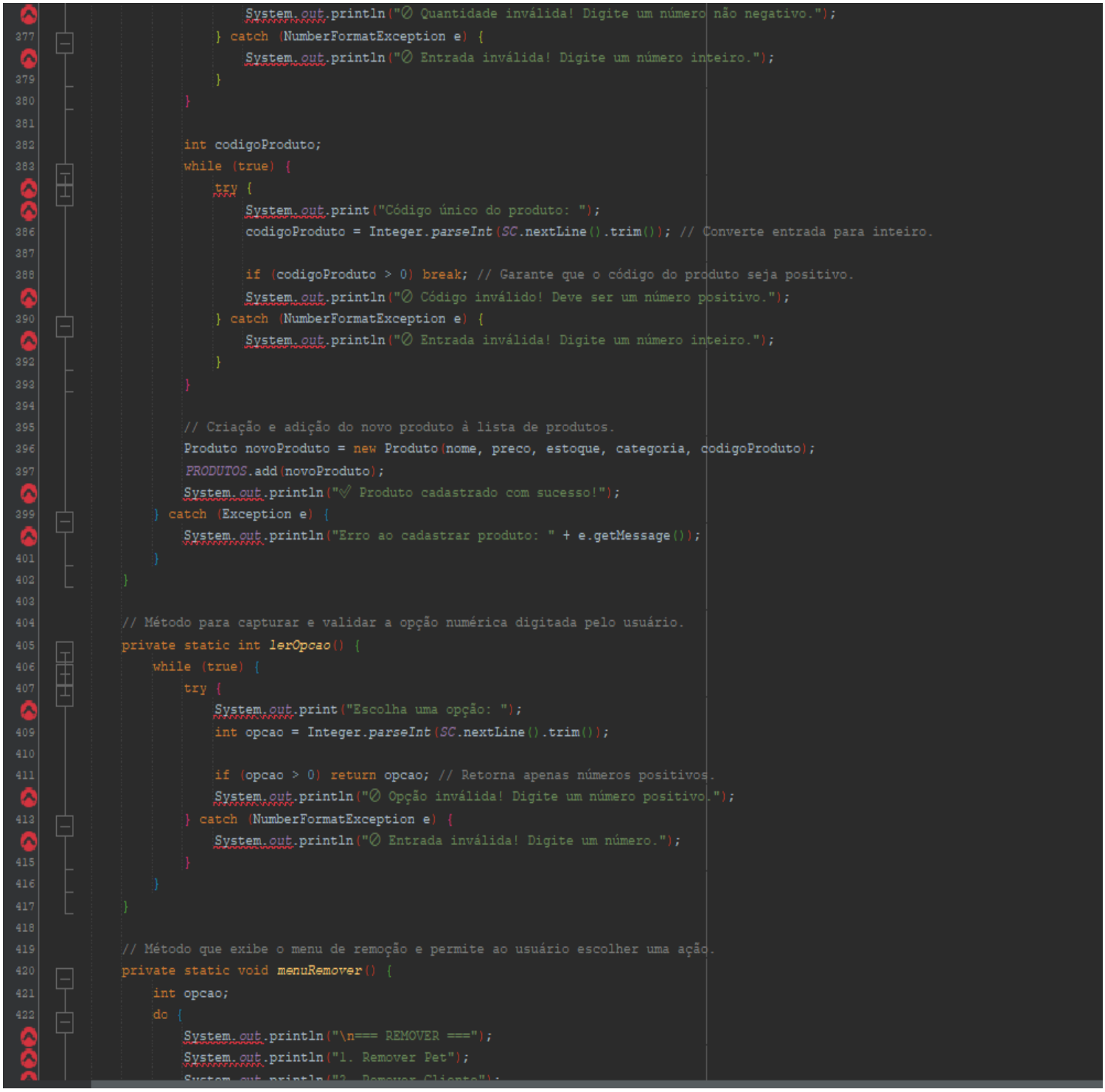
```
            String nomePet = SC.nextLine().trim();

            System.out.print("Espécie (Cachorro/Gato): ");
            String especie = SC.nextLine().trim();

            // Validação da espécie para evitar entradas inválidas.
            while (!especie.equalsIgnoreCase("Cachorro") && !especie.equalsIgnoreCase("Gato")) {
                System.out.println("Espécie inválida! Apenas Cachorro ou Gato são permitidos.");
                System.out.print("Espécie (Cachorro/Gato): ");
                especie = SC.nextLine().trim();
            }

            // Leitura do peso do pet e da data de nascimento com validação.
            float peso = lerFloat("Peso (kg): ", 0.1f, 100f);
            LocalDate nascimento = lerData("Data de nascimento (dd/MM/yyyy): ");

            // Criação e associação do pet ao cliente.
            Pets pet = new Pets(nomePet, especie, peso, nascimento);
            cliente.adicionarPet(pet);
            System.out.println("✅ Pet cadastrado com sucesso!");
        } catch (Exception e) {
            System.out.println("Erro ao cadastrar pet: " + e.getMessage());
        }
    }

    // Método que exibe a lista de pets cadastrados no sistema.
    private static void listarPets() {
        // Verifica se a lista de clientes está vazia, indicando que não há pets cadastrados.
        if (CLIENTES.isEmpty()) {
            System.out.println("\n⊘ Nenhum cliente e pet cadastrado!");
            return; // Retorna imediatamente para evitar processamento desnecessário.
        }

        System.out.println("\n📋 Lista de Pets Cadastrados e seus Donos:");

        // Itera sobre todos os clientes cadastrados.
        for (Cliente cliente : CLIENTES) {
            // Exibe as informações do cliente utilizando seu método `toString()`.
            System.out.println("\n" + cliente.toString());

            // Chama o método `listarPets()` do objeto Cliente, que provavelmente retorna informações sobre os pets do cliente.
            System.out.println(cliente.listarPets());
        }
    }

    // Método responsável por realizar um novo agendamento de serviço para um pet.
    private static void realizarAgendamento() {
```

```
            System.out.println("∅ Nenhum cliente e pet cadastrado! Cadastre antes de agendar.");
            return;
        }

        try {
            // Obtém a data e hora do serviço, utilizando métodos auxiliares para garantir formatos corretos.
            LocalDate data = lerData("Data do serviço (dd/MM/yyyy): ");
            LocalTime hora = lerHora();

            // Verifica se o horário do agendamento está dentro do funcionamento do pet shop.
            if (!isHorarioValido(data, hora)) {
                System.out.println("∅ Horário de agendamento inválido. O pet shop está fechado nesse horário.");
                return;
            }

            // Exibe a lista de pets cadastrados para facilitar a escolha do usuário.
            listarPets();

            // Seleciona o pet para o qual será feito o agendamento.
            Pets pet = selecionarPet();

            // Captura o serviço escolhido pelo usuário.
            String servico = lerServico();

            // Calcula o preço automaticamente com base no serviço e no porte do pet.
            double valor = Servico.calcularPrecoAutomatico(servico, pet.getPortePet());

            // Cria um novo objeto `Agendamento` e adiciona à lista de agendamentos.
            AGENDAMENTOS.add(new Agendamento(pet, data, hora, servico, valor));

            // Atualiza os registros financeiros do pet shop após o agendamento ser concluído.
            financeiro.setServicoFeitos(financeiro.getServicoFeitos() + 1);
            financeiro.setRecebimento(financeiro.getRecebimento() + valor);

            // Exibe uma mensagem de sucesso com o valor do serviço.
            System.out.printf("√ Agendamento realizado com sucesso! Valor: R$ %.2f%n", valor);
        } catch (Exception e) {
            // Captura possíveis erros e exibe uma mensagem informativa ao usuário.
            System.out.println("Erro ao realizar agendamento: " + e.getMessage());
        }
    }

    // Método que verifica se um horário de agendamento é válido conforme o horário de funcionamento do pet shop.
    private static boolean isHorarioValido(LocalDate data, LocalTime hora) {
        // Obtém o dia da semana correspondente à data informada.
        DayOfWeek diaDaSemana = data.getDayOfWeek();

        // O pet shop está fechado aos domingos, então retorna `false` imediatamente.
        if (diaDaSemana == DayOfWeek.SUNDAY) {
            return false;
```

```
244             // O pet shop está fechado aos domingos, então retorna `false` imediatamente.
                if (diaDaSemana == DayOfWeek.SUNDAY) {
246                 return false;
247             } else if (diaDaSemana == DayOfWeek.SATURDAY) {
248                 // Aos sábados, o funcionamento é das 09:00 às 13:00.
249                 return hora.isAfter(LocalTime.of(8, 59)) && hora.isBefore(LocalTime.of(13, 1));
250             } else { // De segunda a sexta-feira
251                 // Horário válido: das 08:00 às 18:00.
252                 return hora.isAfter(LocalTime.of(7, 59)) && hora.isBefore(LocalTime.of(18, 1));
253             }
254         }
255
256         // Método que permite ao usuário escolher um serviço válido a partir da lista de opções disponíveis.
257         private static String lerServico() {
                System.out.println("\n📌 Serviços disponíveis:");
259
260             // Itera sobre a lista de serviços válidos e exibe cada um com seu número correspondente.
261             for (int i = 0; i < SERVIÇOS_VALIDOS.size(); i++) {
                    System.out.printf("%d. %s%n", i + 1, SERVIÇOS_VALIDOS.get(i)); // Exibe os serviços numerados a partir de 1.
263             }
264
265             while (true) {
                    System.out.print("Escolha o número do serviço desejado: ");
267                 String input = SC.nextLine().trim(); // Captura e remove espaços extras da entrada do usuário.
268
269                 try {
270                     int escolha = Integer.parseInt(input); // Converte a entrada para um número inteiro.
271
272                     // Verifica se a escolha está dentro do intervalo válido.
273                     if (escolha >= 1 && escolha <= SERVIÇOS_VALIDOS.size()) {
274                         return SERVIÇOS_VALIDOS.get(escolha - 1); // Retorna o serviço correspondente à escolha do usuário.
275                     } else {
                            System.out.println("⚠ Número inválido! Escolha um número entre 1 e " + SERVIÇOS_VALIDOS.size() + ".");
277                     }
278                 } catch (NumberFormatException e) {
                        System.out.println("⚠ Entrada inválida! Digite um número."); // Trata o erro caso a entrada não seja um número válido.
280                 }
281             }
282         }
283
284         // Método que permite ao usuário selecionar um pet a partir dos pets cadastrados no sistema.
285         private static Pets selecionarPet() {
                System.out.print("Digite o nome do pet: ");
287             String nomePet = SC.nextLine().trim(); // Captura a entrada e remove espaços extras.
288
289             // Percorre todos os clientes cadastrados.
290             for (Cliente cliente : CLIENTES) {
291                 // Percorre os pets associados ao cliente atual.
292                 for (Pets pet : cliente.getPets()) {
```

```
        // Percorre todos os clientes cadastrados.
        for (Cliente cliente : CLIENTES) {
            // Percorre os pets associados ao cliente atual.
            for (Pets pet : cliente.getPets()) {
                // Compara o nome do pet ignorando maiúsculas e minúsculas.
                if (pet.getNomePet().equalsIgnoreCase(nomePet)) {
                    return pet; // Retorna o pet encontrado.
                }
            }
        }

        // Caso nenhum pet seja encontrado, uma exceção é lançada para indicar erro.
        throw new NoSuchElementException("∅ Pet não encontrado!");
    }

    // Método que solicita e valida uma data digitada pelo usuário.
    private static LocalDate lerData(String mensagem) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy"); // Define o formato esperado para a data.
        LocalDate data = null;

        while (data == null) {
            System.out.print(mensagem);
            String dataStr = SC.nextLine().trim(); // Captura a entrada e remove espaços extras.

            try {
                data = LocalDate.parse(dataStr, formatter); // Converte a entrada para um objeto `LocalDate`.
            } catch (DateTimeParseException e) {
                System.out.println("∅ Data inválida! Use o formato dd/MM/yyyy."); // Trata erro caso a entrada não corresponda ao formato esperado.
            }
        }

        return data; // Retorna a data válida.
    }

    // Método que solicita e valida um horário digitado pelo usuário.
    private static LocalTime lerHora() {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm"); // Define o formato esperado para a hora.

        while (true) {
            try {
                System.out.print("Hora do serviço (HH:mm): ");
                String horaStr = SC.nextLine().trim(); // Captura a entrada e remove espaços extras.

                LocalTime hora = LocalTime.parse(horaStr, formatter); // Converte a entrada para um objeto `LocalTime`.
                return hora; // Retorna o horário válido.
            } catch (DateTimeParseException e) {
                System.out.println("∅ Hora inválida! Use o formato HH:mm."); // Exibe erro caso a entrada não seja válida.
            }
        }
    }
```

```
            String horaStr = SC.nextLine().trim(); // Captura a entrada e remove espaços extras.

            LocalTime hora = LocalTime.parse(horaStr, formatter); // Converte a entrada para um objeto `LocalTime`.
            return hora; // Retorna o horário válido.
        } catch (DateTimeParseException e) {
            System.out.println("❌ Hora inválida! Use o formato HH:mm."); // Exibe erro caso a entrada não seja válida.
        }
    }
}

// Método que solicita e valida um número decimal dentro de um intervalo definido.
private static float lerFloat(String mensagem, float min, float max) {
    while (true) {
        try {
            System.out.print(mensagem);
            float valor = Float.parseFloat(SC.nextLine().trim()); // Converte a entrada para um `float`.

            // Verifica se o número está dentro do intervalo permitido.
            if (valor >= min && valor <= max) return valor;

            System.out.println("❌ Valor fora do intervalo permitido! Tente novamente.");
        } catch (NumberFormatException e) {
            System.out.println("❌ Entrada inválida! Digite um número válido."); // Exibe erro caso a entrada não seja um número válido.
        }
    }
}

// Método para cadastrar um novo produto no sistema.
private static void cadastrarProduto() {
    try {
        System.out.print("\nNome do produto: ");
        String nome = SC.nextLine().trim(); // Captura o nome do produto, removendo espaços extras.

        System.out.print("Categoria (Higiene/Alimentação/Brinquedos): ");
        String categoria = SC.nextLine().trim(); // Captura a categoria do produto.

        // Obtém o preço do produto dentro do intervalo permitido (0.1 a 10.000).
        float preco = lerFloat("Preço (R$): ", 0.1f, 10000f);

        int estoque;
        while (true) {
            try {
                System.out.print("Quantidade disponível no estoque: ");
                estoque = Integer.parseInt(SC.nextLine().trim()); // Converte entrada para inteiro.

                if (estoque >= 0) break; // Garante que o estoque não seja negativo.
                System.out.println("❌ Quantidade inválida! Digite um número não negativo.");
            } catch (NumberFormatException e) {
                System.out.println("❌ Entrada inválida! Digite um número inteiro.");
```



```
                System.out.println("⚠ Quantidade inválida! Digite um número não negativo.");
            } catch (NumberFormatException e) {
                System.out.println("⚠ Entrada inválida! Digite um número inteiro.");
            }
        }

        int codigoProduto;
        while (true) {
            try {
                System.out.print("Código único do produto: ");
                codigoProduto = Integer.parseInt(SC.nextLine().trim()); // Converte entrada para inteiro.

                if (codigoProduto > 0) break; // Garante que o código do produto seja positivo.
                System.out.println("⚠ Código inválido! Deve ser um número positivo.");
            } catch (NumberFormatException e) {
                System.out.println("⚠ Entrada inválida! Digite um número inteiro.");
            }
        }

        // Criação e adição do novo produto à lista de produtos.
        Produto novoProduto = new Produto(nome, preco, estoque, categoria, codigoProduto);
        PRODUTOS.add(novoProduto);
        System.out.println("✅ Produto cadastrado com sucesso!");
    } catch (Exception e) {
        System.out.println("Erro ao cadastrar produto: " + e.getMessage());
    }
}

// Método para capturar e validar a opção numérica digitada pelo usuário.
private static int lerOpcao() {
    while (true) {
        try {
            System.out.print("Escolha uma opção: ");
            int opcao = Integer.parseInt(SC.nextLine().trim());

            if (opcao > 0) return opcao; // Retorna apenas números positivos.
            System.out.println("⚠ Opção inválida! Digite um número positivo.");
        } catch (NumberFormatException e) {
            System.out.println("⚠ Entrada inválida! Digite um número.");
        }
    }
}

// Método que exibe o menu de remoção e permite ao usuário escolher uma ação.
private static void menuRemover() {
    int opcao;
    do {
        System.out.println("\n=== REMOVER ===");
        System.out.println("1. Remover Pet");
        System.out.println("2. Remover Cliente");
```

```
        do {
            System.out.println("\n=== REMOVER ===");
            System.out.println("1. Remover Pet");
            System.out.println("2. Remover Cliente");
            System.out.println("3. Voltar");
            System.out.print("Opção: ");

            opcao = lerOpcao(); // Obtém a opção do usuário.

            switch (opcao) {
                case 1 -> removerPet();
                case 2 -> removerCliente();
                case 3 -> System.out.println("Voltando...");
                default -> System.out.println("Opção inválida! Escolha uma opção válida.");
            }
        } while (opcao != 3); // Continua até que o usuário escolha "Voltar".
    }

    // Método que permite ao usuário remover um pet cadastrado.
    private static void removerPet() {
        if (CLIENTES.isEmpty()) {
            System.out.println("∅ Nenhum pet cadastrado para remover.");
            return;
        }

        listarPets(); // Exibe a lista de pets cadastrados.

        System.out.print("Digite o nome do pet para remover: ");
        String nomePet = SC.nextLine().trim(); // Captura o nome do pet a ser removido.

        Cliente clientePet = null;
        Pets petRemover = null;

        // Percorre os clientes e seus pets para encontrar o pet desejado.
        for (Cliente cliente : CLIENTES) {
            for (Pets pet : cliente.getPets()) {
                if (pet.getNomePet().equalsIgnoreCase(nomePet)) {
                    clientePet = cliente;
                    petRemover = pet;
                    break;
                }
            }
            if (petRemover != null) break;
        }

        // Se o pet não for encontrado, exibe uma mensagem de erro.
        if (petRemover == null) {
            System.out.println("∅ Pet não encontrado.");
            return;
```

```
            return;
        }

        System.out.printf("Tem certeza que deseja remover %s? (S/N): ", petRemover.getNomePet());
        String confirmar = SC.nextLine().trim();

        if (confirmar.equalsIgnoreCase("S")) {
            // Remove o pet da lista de pets do cliente.
            List<Pets> petsModificaveis = new ArrayList<>(clientePet.getPets());
            if (petsModificaveis.remove(petRemover)) {
                clientePet.getPets().clear();
                clientePet.getPets().addAll(petsModificaveis);
                System.out.println("✅ Pet removido com sucesso!");

                // Se o cliente não tiver mais pets, pergunta se deseja removê-lo também.
                if (clientePet.getPets().isEmpty()) {
                    System.out.printf("Cliente %s não possui mais pets. Deseja removê-lo? (S/N): ", clientePet.getNome());
                    String confirmaCliente = SC.nextLine().trim();
                    if (confirmaCliente.equalsIgnoreCase("S")) {
                        CLIENTES.remove(clientePet);
                        System.out.println("✅ Cliente removido com sucesso!");
                    }
                }
            } else {
                System.out.println("⚠ Erro ao remover pet.");
            }
        } else {
            System.out.println("❌ Remoção cancelada.");
        }
    }

    // Método para remover um cliente e todos os seus pets cadastrados.
    private static void removerCliente() {
        if (CLIENTES.isEmpty()) {
            System.out.println("⚠ Nenhum cliente cadastrado para remover.");
            return;
        }

        listarClientes(); // Exibe a lista de clientes cadastrados.

        System.out.print("Digite o CPF do cliente para remover: ");
        String cpf = SC.nextLine().trim(); // Captura o CPF do cliente.

        Cliente clienteRemover = null;

        // Busca o cliente pelo CPF informado.
        for (Cliente cliente : CLIENTES) {
            if (cliente.getCpf().equalsIgnoreCase(cpf)) {
```

```
        // Busca o cliente pelo CPF informado.
        for (Cliente cliente : CLIENTES) {
            if (cliente.getCpf().equalsIgnoreCase(cpf)) {
                clienteRemover = cliente;
                break;
            }
        }

        // Se o cliente não for encontrado, exibe mensagem de erro.
        if (clienteRemover == null) {
            System.out.println("∅ Cliente não encontrado.");
            return;
        }

        System.out.printf("Tem certeza que deseja remover o cliente %s e todos os seus pets? (S/N): ", clienteRemover.getNome());
        String confirmar = SC.nextLine().trim();

        if (confirmar.equalsIgnoreCase("S")) {
            // Remove o cliente do sistema.
            CLIENTES.remove(clienteRemover);
            System.out.println("✅ Cliente removido com sucesso!");
        } else {
            System.out.println("❌ Remoção cancelada.");
        }
    }

    // Método que lista todos os clientes cadastrados no sistema.
    private static void listarClientes() {
        // Verifica se a lista de clientes está vazia. Caso esteja, exibe uma mensagem e retorna.
        if (CLIENTES.isEmpty()) {
            System.out.println("∅ Nenhum cliente cadastrado.");
            return; // Retorna imediatamente para evitar execução desnecessária.
        }

        System.out.println("\n📋 Lista de Clientes:");

        // Utiliza `forEach` para percorrer e imprimir cada cliente da lista.
        CLIENTES.forEach(System.out::println);
    }

    // Método que exibe o histórico de agendamentos feitos no pet shop.
    private static void exibirHistoricoAgendamentos() {
        // Verifica se há agendamentos registrados. Caso não haja, exibe uma mensagem e retorna.
        if (AGENDAMENTOS.isEmpty()) {
            System.out.println("∅ Nenhum agendamento realizado ainda.");
            return;
        }

        System.out.println("\n📖 Histórico de Agendamentos:");
```

6.2 Classe cliente

```java
package aps3;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Objects;

/**
 * Representa um cliente do pet shop com informações cadastrais e pets associados.
 */
public class Cliente {
    private final String nome;      // Nome completo do cliente
    private final String cpf;       // CPF do cliente formatado corretamente
    private final String telefone;  // Telefone do cliente
    private final String endereco;  // Endereço do cliente
    private final List<Pets> pets;   // Lista de pets associados ao cliente

    /**
     * Construtor da classe Cliente, garantindo validações adequadas.
     *
     * @param nome Nome completo (não pode ser vazio)
     * @param cpf CPF válido no formato XXX.XXX.XXX-XX
     * @param telefone Telefone válido com DDD
     * @param endereco Endereço completo
     * @throws IllegalArgumentException Se algum parâmetro for inválido
     */
    public Cliente(String nome, String cpf, String telefone, String endereco) {
        validarCampoObrigatorio(nome, "Nome");
        validarCPF(cpf);
        validarCampoObrigatorio(telefone, "Telefone");
        validarCampoObrigatorio(endereco, "Endereço");

        this.nome = nome.trim();
        this.cpf = formatarCPF(cpf);
        this.telefone = telefone.trim();
        this.endereco = endereco.trim();
        this.pets = new ArrayList<>(); // Inicializa a lista de pets do cliente
    }

    // Getters
```

```java
    // Getters
    public String getNome() {
        return nome;
    }
    public String getCpf() {
        return cpf;
    }
    public String getTelefone() {
        return telefone;
    }
    public String getEndereco() {
        return endereco;
    }

    /**
     * Retorna uma lista imutável dos pets do cliente.
     * Evita modificações externas na lista original.
     * @return Lista de pets associada ao cliente
     */
    public List<Pets> getPets() {
        return Collections.unmodifiableList(pets);
    }

    /**
     * Adiciona um pet ao cliente, garantindo que ele não seja nulo.
     * @param pet Instância válida de Pet
     * @throws NullPointerException Se o pet for nulo
     */
    public void adicionarPet(Pets pet) {
        pets.add(Objects.requireNonNull(pet, "Pet não pode ser nulo"));
    }

    /**
     * Lista todos os pets associados ao cliente.
     * @return Relatório formatado com detalhes dos pets
     */
    public String listarPets() {
        if (pets.isEmpty()) {
            return "Nenhum pet cadastrado.";
```

```
78              return "Nenhum pet cadastrado.";
79          }
80
81          StringBuilder sb = new StringBuilder();
82          sb.append(String.format("\nPets de %s (%d):", nome, pets.size()));
83
84          for (Pets pet : pets) {
85              sb.append("\n----------------------------------------");
86              sb.append("\n").append(pet.gerarResumo()); // Chama `gerarResumo()` para obter detalhes do pet
87          }
88          return sb.toString();
89      }
90
91      /**
92       * Retorna uma representação formatada do cliente.
93       * @return String com detalhes do cliente
94       */
95      @Override
96      public String toString() {
97          return String.format("""
98              Cliente:
99              Nome: %s
100             CPF: %s
101             Telefone: %s
102             Endereço: %s
103             Total de Pets: %d
104             """,
105             nome, cpf, telefone, endereco, pets.size()
106         );
107     }
108
109     // Métodos de validação
110
111     /**
112      * Valida um campo obrigatório, garantindo que não esteja vazio ou nulo.
113      * @param valor Valor informado
114      * @param nomeCampo Nome do campo para mensagem de erro
115      * @throws IllegalArgumentException Se o campo estiver vazio
116      */
```

```
114         * @param nomeCampo Nome do campo para mensagem de erro
115         * @throws IllegalArgumentException Se o campo estiver vazio
116         */
117        private void validarCampoObrigatorio(String valor, String nomeCampo) {
118            if (valor == null || valor.isBlank()) {
119                throw new IllegalArgumentException(nomeCampo + " é obrigatório.");
120            }
121        }
122
123        /**
124         * Valida um CPF, garantindo que tenha 11 dígitos.
125         * @param cpf CPF informado pelo usuário
126         * @throws IllegalArgumentException Se o CPF for inválido
127         */
128        private void validarCPF(String cpf) {
129            String cpfLimpo = cpf.replaceAll("[^0-9]", "");
130            if (cpfLimpo.length() != 11) {
131                throw new IllegalArgumentException("CPF inválido. Deve conter 11 dígitos.");
132            }
133        }
134
135        /**
136         * Formata um CPF para o padrão XXX.XXX.XXX-XX.
137         * @param cpf CPF informado pelo usuário
138         * @return CPF formatado
139         */
140        private String formatarCPF(String cpf) {
141            String cpfLimpo = cpf.replaceAll("[^0-9]", "");
142            return cpfLimpo.replaceAll("(\\d{3})(\\d{3})(\\d{3})(\\d{2})", "$1.$2.$3-$4");
143        }
144    }
145
```

6.3 Classe Pets

```java
package aps3;

import java.time.LocalDate;
import java.time.Period;
import java.time.format.DateTimeFormatter;

/**
 * Representa um pet cadastrado no sistema com suas características básicas.
 */
public class Pets {
    private String nomePet;
    private String especie;
    private float pesoPet;
    private LocalDate dataNascimento;
    private String portePet;

    /**
     * Construtor da classe Pet, realizando validações essenciais.
     * @param nomePet Nome do pet
     * @param especie Espécie (Cachorro ou Gato)
     * @param pesoPet Peso do pet (deve ser maior que zero)
     * @param dataNascimento Data de nascimento (não pode ser futura)
     */
    public Pets(String nomePet, String especie, float pesoPet, LocalDate dataNascimento) {
        if (nomePet == null || nomePet.isBlank()) {
            throw new IllegalArgumentException("Nome do pet não pode ser vazio.");
        }
        validarEspecie(especie); // Verifica se a espécie é válida
        if (pesoPet <= 0) throw new IllegalArgumentException("Peso inválido! Deve ser maior que zero.");
        if (dataNascimento.isAfter(LocalDate.now())) throw new IllegalArgumentException("Data futura não permitida.");

        this.nomePet = nomePet;
        this.especie = especie;
        this.pesoPet = pesoPet;
        this.dataNascimento = dataNascimento;
        definirPorte(); // Define o porte automaticamente com base no peso
    }

```

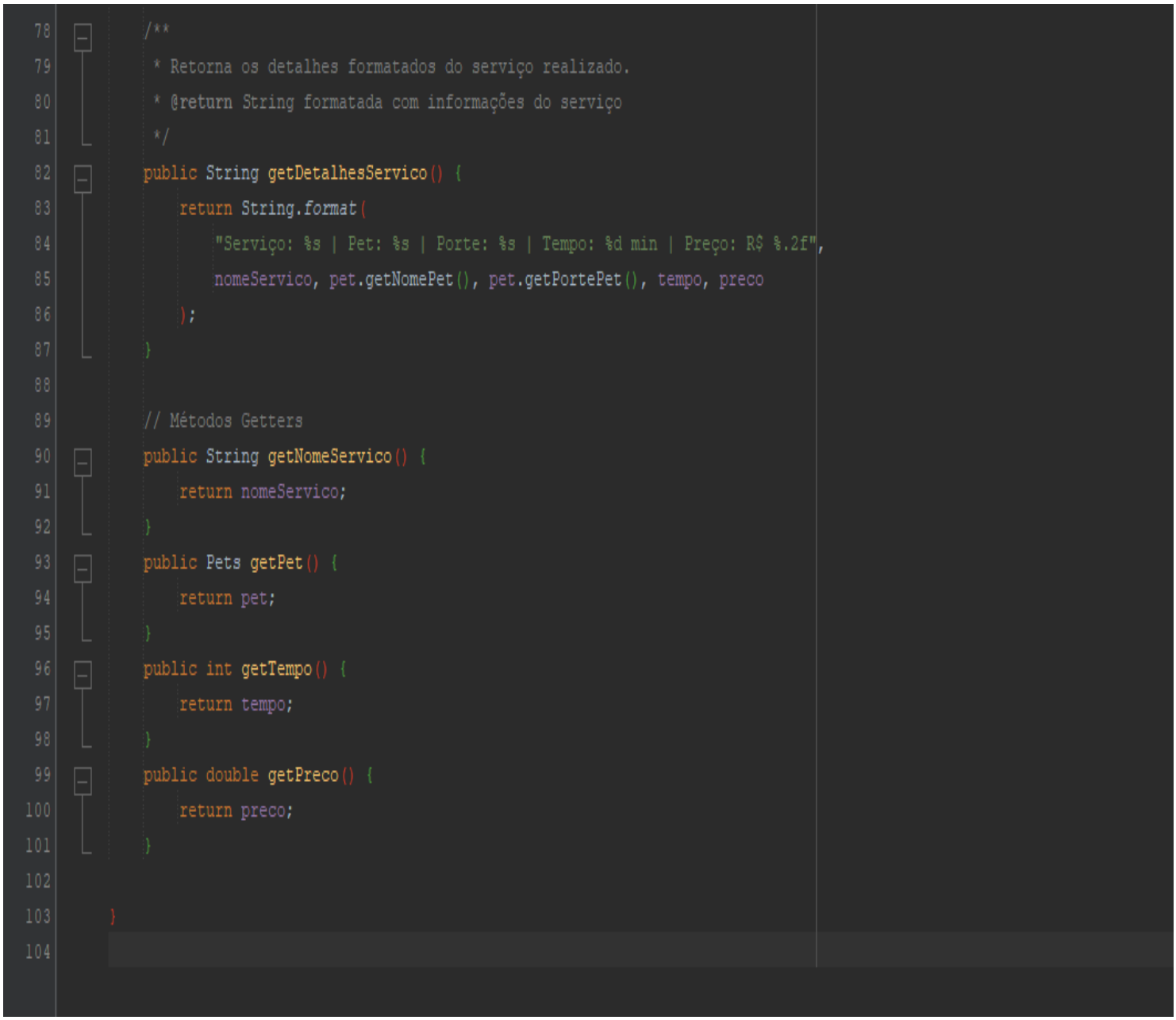
```java
        } else if (pesoPet <= 25) {
            this.portePet = "Médio";
        } else {
            this.portePet = "Grande";
        }
    }

    /**
     * Gera um resumo formatado sobre o pet.
     * @return String com as informações do pet
     */
    public String gerarResumo() {
        DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        return String.format("""
            Nome: %s
            Espécie: %s
            Idade: %d anos
            Peso: %.2f kg
            Porte: %s
            Nascimento: %s
            """,
            nomePet, especie, Period.between(dataNascimento, LocalDate.now()).getYears(),
            pesoPet, portePet, dataNascimento.format(fmt)
        );
    }

    /**
     * Valida a espécie do pet.
     * @param especie Espécie informada
     */
    private void validarEspecie(String especie) {
        if (!especie.equalsIgnoreCase("Cachorro") && !especie.equalsIgnoreCase("Gato")) {
            throw new IllegalArgumentException("Espécie inválida! Apenas Cachorro ou Gato são permitidos.");
        }
    }

    // Getters e Setters com validações

    public String getNomePet() {
```

```java
    // Getters e Setters com validações

    public String getNomePet() {
        return nomePet;
    }
    public void setNomePet(String nomePet) {
        if (nomePet == null || nomePet.isBlank()) {
            throw new IllegalArgumentException("Nome do pet não pode ser vazio.");
        }
        this.nomePet = nomePet;
    }

    public String getEspecie() {
        return especie;
    }
    public void setEspecie(String especie) {
        validarEspecie(especie);
        this.especie = especie;
    }

    public float getPesoPet() {
        return pesoPet;
    }
    public void setPesoPet(float pesoPet) {
        if (pesoPet <= 0) throw new IllegalArgumentException("Peso inválido! Deve ser maior que zero.");
        this.pesoPet = pesoPet;
        definirPorte(); // Atualiza o porte ao modificar o peso
    }

    public LocalDate getDataNascimento() {
        return dataNascimento;
    }
    public void setDataNascimento(LocalDate dataNascimento) {
        if (dataNascimento.isAfter(LocalDate.now())) {
            throw new IllegalArgumentException("Data futura não permitida.");
        }
        this.dataNascimento = dataNascimento;
        definirPorte(); // Atualiza o porte ao modificar a data de nascimento
```

6.4 Classes Serviço

```java
package aps3;

import java.util.Map;

/**
 * Representa um serviço oferecido para um pet no pet shop.
 */
public class Servico {
    private final String nomeServico; // Nome do serviço realizado
    private final Pets pet; // Pet que receberá o serviço
    private final int tempo; // Tempo estimado para a realização do serviço em minutos
    private final double preco; // Preço do serviço com base no porte do pet

    // Tabela de preços por serviço e porte do pet
    private static final Map<String, Map<String, Double>> precosPorServico = Map.of(
        "Banho", Map.of("Pequeno", 60.0, "Médio", 80.0, "Grande", 130.0),
        "Tosa Tesoura", Map.of("Pequeno", 100.0, "Médio", 130.0, "Grande", 160.0),
        "Tosa Máquina", Map.of("Pequeno", 85.0, "Médio", 110.0, "Grande", 120.0),
        "Tosa Bebê", Map.of("Pequeno", 140.0, "Médio", 165.0, "Grande", 180.0),
        "Tosa Higiênica", Map.of("Pequeno", 55.0, "Médio", 65.0, "Grande", 100.0),
        "Corte de Unha", Map.of("Pequeno", 15.0, "Médio", 15.0, "Grande", 15.0),
        "Limpeza de Ouvido", Map.of("Pequeno", 10.0, "Médio", 10.0, "Grande", 10.0),
        "Hidratação", Map.of("Pequeno", 90.0, "Médio", 120.0, "Grande", 150.0),
        "Remoção de Subpelos", Map.of("Pequeno", 30.0, "Médio", 50.0, "Grande", 70.0)
    );

    // Tabela de tempo fixo por serviço
    private static final Map<String, Integer> tempoPorServico = Map.of(
        "Banho", 60,
        "Tosa Tesoura", 180,
        "Tosa Máquina", 80,
        "Tosa Bebê", 180,
        "Tosa Higiênica", 70,
        "Corte de Unha", 20,
        "Limpeza de Ouvido", 20,
        "Hidratação", 60,
        "Remoção de Subpelos", 120
    );

```

```
    });

    /**
     * Construtor da classe Servico, validando e atribuindo valores.
     * @param nomeServico Nome do serviço a ser prestado
     * @param pet Pet que receberá o serviço
     * @throws IllegalArgumentException Se o serviço não existir ou pet for null
     */
    public Servico(String nomeServico, Pets pet) {
        if (nomeServico == null || !precosPorServico.containsKey(nomeServico)) {
            throw new IllegalArgumentException("Serviço inválido! Escolha um dos serviços disponíveis.");
        }
        if (pet == null) {
            throw new IllegalArgumentException("Pet não pode ser nulo.");
        }
        this.nomeServico = nomeServico;
        this.pet = pet;
        this.tempo = tempoPorServico.getOrDefault(nomeServico, 60); // Tempo padrão de 60 min caso não esteja na lista
        this.preco = calcularPreco(); // Calcula preço com base no porte do pet
    }

    /**
     * Método privado para calcular o preço do serviço com base no porte do pet.
     * @return O preço correspondente ao porte do pet
     */
    private double calcularPreco() {
        return precosPorServico.get(nomeServico).getOrDefault(pet.getPortePet(), 0.0);
    }

    /**
     * Método estático para calcular o preço sem instanciar um objeto Servico.
     * @param nomeServico Nome do serviço desejado
     * @param portePet Porte do pet
     * @return O preço do serviço conforme o porte
     */
    public static double calcularPrecoAutomatico(String nomeServico, String portePet) {
        if (nomeServico == null || portePet == null) return 0.0;
        return precosPorServico.getOrDefault(nomeServico, Map.of()).getOrDefault(portePet, 0.0);
    }

    /**
```



```java
    /**
     * Retorna os detalhes formatados do serviço realizado.
     * @return String formatada com informações do serviço
     */
    public String getDetalhesServico() {
        return String.format(
            "Serviço: %s | Pet: %s | Porte: %s | Tempo: %d min | Preço: R$ %.2f",
            nomeServico, pet.getNomePet(), pet.getPortePet(), tempo, preco
        );
    }

    // Métodos Getters
    public String getNomeServico() {
        return nomeServico;
    }
    public Pets getPet() {
        return pet;
    }
    public int getTempo() {
        return tempo;
    }
    public double getPreco() {
        return preco;
    }

}

```

6.5 Classe Produtos

```java
package aps3;

/**
 * Representa um produto do pet shop, incluindo nome, preço, estoque e categoria.
 */
public class Produto { // Renamed to singular form for consistency
    private final String nome;      // Nome do produto
    private double preco;           // Preço do produto
    private int estoque;            // Quantidade disponível em estoque
    private final String categoria; // Categoria do produto (Ex: Higiene, Alimentação, Brinquedos)
    private final int codProduto;   // Código único do produto

    /**
     * Construtor da classe Produto, garantindo validações essenciais.
     * @param nome Nome do produto (não pode ser vazio)
     * @param preco Preço do produto (deve ser positivo)
     * @param estoque Quantidade disponível no estoque (não pode ser negativo)
     * @param categoria Categoria do produto (não pode ser vazia)
     * @param codProduto Código único do produto
     * @throws IllegalArgumentException Se algum parâmetro for inválido
     */
    public Produto(String nome, double preco, int estoque, String categoria, int codProduto) {
        if (nome == null || nome.isBlank()) throw new IllegalArgumentException("Nome do produto não pode ser vazio.");
        if (preco <= 0) throw new IllegalArgumentException("Preço inválido! Deve ser maior que zero.");
        if (estoque < 0) throw new IllegalArgumentException("Estoque não pode ser negativo.");
        if (categoria == null || categoria.isBlank()) throw new IllegalArgumentException("Categoria inválida.");

        this.nome = nome;
        this.preco = preco;
        this.estoque = estoque;
        this.categoria = categoria;
        this.codProduto = codProduto;
    }

    /**
     * Adiciona unidades ao estoque do produto.
     * @param quantidade Quantidade a ser adicionada (deve ser positiva)
     * @return Mensagem informando o novo estoque
```

```java
     * @param quantidade Quantidade a ser adicionada (deve ser positiva)
     * @return Mensagem informando o novo estoque
     */
    public String adicionarEstoque(int quantidade) {
        if (quantidade > 0) {
            this.estoque += quantidade;
            return "Estoque atualizado! Novo total: " + this.estoque;
        }
        return "Quantidade inválida! Informe um valor positivo.";
    }

    /**
     * Remove unidades do estoque do produto, garantindo que não fique negativo.
     * @param quantidade Quantidade a ser removida
     * @return Mensagem informando o novo estoque ou erro se insuficiente
     */
    public String removerEstoque(int quantidade) {
        if (quantidade > 0 && this.estoque >= quantidade) {
            this.estoque -= quantidade;
            return "Estoque atualizado! Novo total: " + this.estoque;
        }
        return "Estoque insuficiente ou quantidade inválida!";
    }

    /**
     * Aplica um desconto percentual ao preço do produto.
     * @param descontoPercentual Percentual de desconto (entre 1 e 100)
     * @return Mensagem com o novo preço ou erro se percentual for inválido
     */
    public String aplicarDesconto(double descontoPercentual) {
        if (descontoPercentual > 0 && descontoPercentual <= 100) {
            this.preco *= (1 - (descontoPercentual / 100));
            return "Novo preço com desconto: R$ " + String.format("%.2f", this.preco);
        }
        return "Porcentagem de desconto inválida! Informe um valor entre 1% e 100%.";
    }

```

```java
     * @param novoPreco Novo preço do produto (deve ser positivo)
     * @return Mensagem informando o novo preço ou erro se for inválido
     */
    public String reajustePreco(double novoPreco) {
        if (novoPreco > 0) {
            this.preco = novoPreco;
            return "Novo preço ajustado: R$ " + String.format("%.2f", this.preco);
        }
        return "Preço inválido! O valor deve ser maior que zero.";
    }

    /**
     * Verifica se há estoque suficiente para atender uma demanda específica.
     * @param quantidade Quantidade desejada
     * @return True se há estoque suficiente, False caso contrário
     */
    public boolean verificarEstoqueSuficiente(int quantidade) {
        return quantidade > 0 && this.estoque >= quantidade;
    }

    /**
     * Retorna uma representação textual do produto.
     * @return String formatada com detalhes do produto
     */
    @Override
    public String toString() {
        return String.format("Produto: %s | Código: %d | Categoria: %s | Preço: R$ %.2f | Estoque: %d",
            nome, codProduto, categoria, preco, estoque);
    }

    // Getters
    public String getNome() {
        return nome;
    }
    public double getPreco() {
        return preco;
    }
    public int getEstoque() {
```

```
107     public String getNome() {
108         return nome;
109     }
110     public double getPreco() {
111         return preco;
112     }
113     public int getEstoque() {
114         return estoque;
115     }
116     public String getCategoria() {
117         return categoria;
118     }
119     public int getCodProduto() {
120         return codProduto;
121     }
122
123     public void setPreco(double preco) {
124         this.preco = preco;
125     }
126
127     public void setEstoque(int estoque) {
128         this.estoque = estoque;
129     }
130
131 }
132
```

6.6 Classe Agendamento

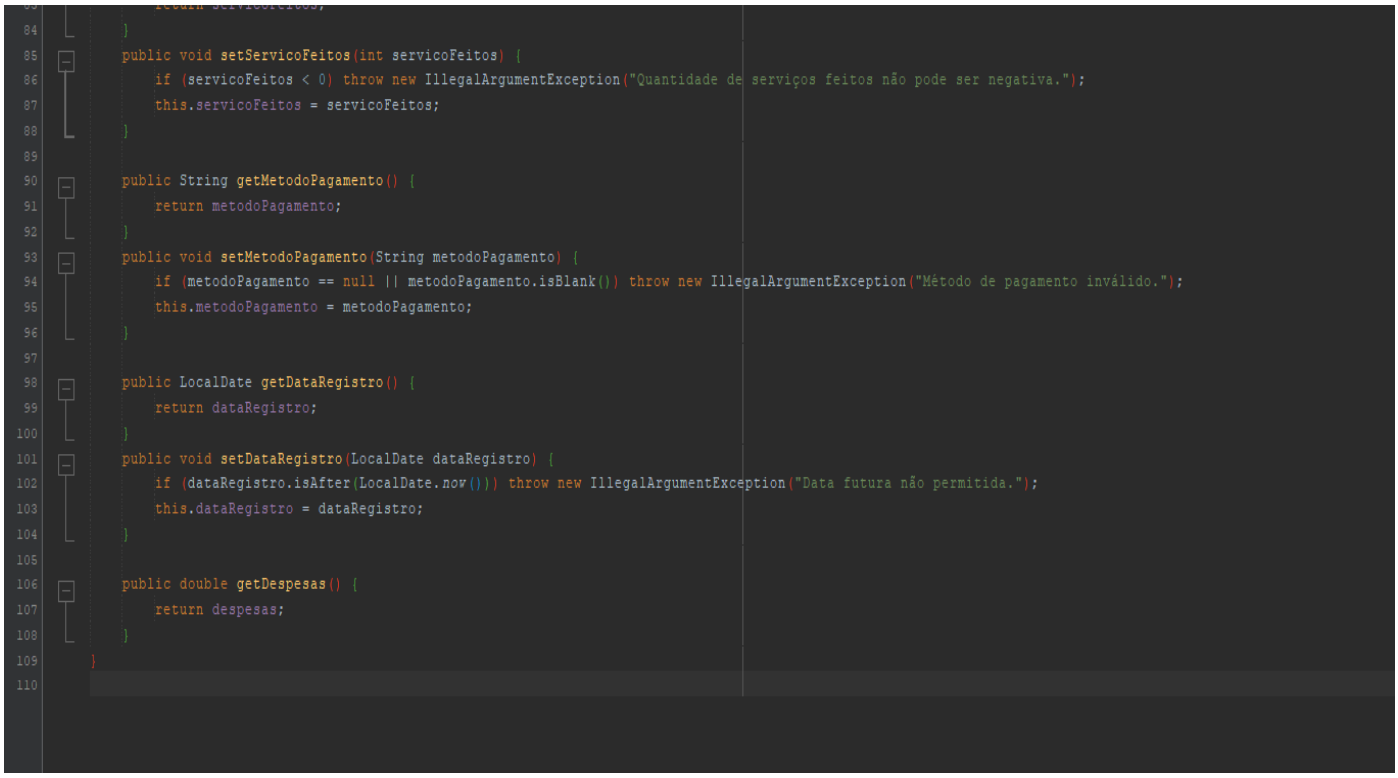
```java
package aps3;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalTime;

/**
 * Representa um agendamento de serviço para um pet no pet shop.
 */
public class Agendamento {
    private final Pets pet;      // Pet que será atendido
    private final LocalDate data;   // Data do agendamento
    private final LocalTime hora;   // Hora do agendamento
    private final String servico;   // Nome do serviço agendado
    private final double valor;     // Valor do serviço

    /**
     * Construtor da classe Agendamento, garantindo a inicialização dos atributos.
     *
     * @param pet Pet que receberá o serviço
     * @param data Data do agendamento (não pode ser no passado)
     * @param hora Hora do agendamento
     * @param servico Nome do serviço a ser realizado
     * @param valor Valor do serviço
     * @throws IllegalArgumentException Se a data do agendamento for no passado ou fora do horário de funcionamento
     */
    public Agendamento(Pets pet, LocalDate data, LocalTime hora, String servico, double valor) {
        if (data.isBefore(LocalDate.now()) || (data.isEqual(LocalDate.now()) && hora.isBefore(LocalTime.now()))) {
            throw new IllegalArgumentException("A data e hora do agendamento devem estar no presente.");
        }
        if (!isHorarioValido(data, hora)) {
            throw new IllegalArgumentException("Horário de agendamento inválido. O pet shop está fechado nesse horário.");
        }

        this.pet = pet;
        this.data = data;
        this.hora = hora;
        this.servico = servico;
        this.valor = valor;
    }
```

```java
     * Verifica se a data e hora do agendamento estão dentro do horário de funcionamento do pet shop.
     *
     * @param data Data do agendamento
     * @param hora Hora do agendamento
     * @return true se o horário for válido, false caso contrário
     */
    private boolean isHorarioValido(LocalDate data, LocalTime hora) {
        DayOfWeek diaDaSemana = data.getDayOfWeek();
        if (null == diaDaSemana) { // Segunda a sexta
            return hora.isAfter(LocalTime.of(7, 59)) && hora.isBefore(LocalTime.of(18, 1)); // 08:00 a 18:00
        } else return switch (diaDaSemana) {
            case SUNDAY -> false;
            case SATURDAY -> hora.isAfter(LocalTime.of(8, 59)) && hora.isBefore(LocalTime.of(13, 1));
            default -> hora.isAfter(LocalTime.of(7, 59)) && hora.isBefore(LocalTime.of(18, 1));
        }; // Fechado aos domingos
        // 09:00 a 13:00
        // Segunda a sexta
        // 08:00 a 18:00


    }

    /**
     * Retorna os detalhes do agendamento de forma formatada.
     *
     * @return String com informações do agendamento
     */
    public String getDetalhesAgendamento() {
        return String.format("Data: %s | Hora: %s | Pet: %s | Serviço: %s | Valor: R$ %.2f",
                data, hora, pet.getNomePet(), servico, valor);
    }

    // Getters
    public Pets getPet() {
        return pet;
    }
    public LocalDate getData() {
        return data;
    }
    public LocalTime getHora() {
        return hora;
    }
    public String getServico() {
        return servico;
    }
```

6.7 Classe Financeiro

```java
package aps3;

import java.time.LocalDate;

/**
 * Gerencia o controle financeiro do pet shop, incluindo recebimentos, despesas e métodos de pagamento.
 */
public class Financeiro {
    private double recebimento;      // Valor total recebido
    private int servicoFeitos;       // Quantidade de serviços realizados
    private String metodoPagamento;  // Método de pagamento utilizado
    private LocalDate dataRegistro;  // Data do registro financeiro
    private double despesas = 0.0;   // Total de despesas associadas

    /**
     * Construtor da classe Financeiro, garantindo validações essenciais.
     * @param recebimento Valor recebido (não pode ser negativo)
     * @param servicoFeitos Número de serviços realizados (não pode ser negativo)
     * @param metodoPagamento Método de pagamento utilizado (não pode ser vazio)
     * @param dataRegistro Data do registro financeiro (não pode ser futura)
     * @throws IllegalArgumentException Se algum parâmetro for inválido
     */
    public Financeiro(double recebimento, int servicoFeitos, String metodoPagamento, LocalDate dataRegistro) {
        if (recebimento < 0) throw new IllegalArgumentException("O valor de recebimento não pode ser negativo.");
        if (servicoFeitos < 0) throw new IllegalArgumentException("Quantidade de serviços feitos não pode ser negativa.");
        if (metodoPagamento == null || metodoPagamento.isBlank()) throw new IllegalArgumentException("Método de pagamento inválido.");
        if (dataRegistro.isAfter(LocalDate.now())) throw new IllegalArgumentException("Data futura não permitida.");

        this.recebimento = recebimento;
        this.servicoFeitos = servicoFeitos;
        this.metodoPagamento = metodoPagamento;
        this.dataRegistro = dataRegistro;
    }

    /**
     * Adiciona uma despesa ao controle financeiro.
     * @param valor Valor da despesa (não pode ser negativo)
     * @throws IllegalArgumentException Se o valor for inválido
     */
    public void adicionarDespesa(double valor) {
        if (valor < 0) throw new IllegalArgumentException("O valor da despesa não pode ser negativo.");
        despesas += valor;
```

```
41          if (valor < 0) throw new IllegalArgumentException("O valor da despesa não pode ser negativo.");
42          despesas += valor;
43      }
44
45      /**
46       * Calcula o saldo financeiro atual.
47       * @return Saldo final (recebimento menos despesas)
48       */
49      public double getSaldoFinal() {
50          return recebimento - despesas;
51      }
52
53      /**
54       * Exibe um resumo financeiro formatado.
55       * @return String com informações do financeiro
56       */
57      public String exibirResumoFinanceiro() {
58          return String.format(
59              """
60              📊 Resumo Financeiro:
61              - Total Recebido: R$ %.2f
62              - Total de Serviços Realizados: %d
63              - Método de Pagamento: %s
64              - Data do Registro: %s
65              - Despesas: R$ %.2f
66              - Saldo Final: R$ %.2f
67              """,
68              recebimento, servicoFeitos, metodoPagamento, dataRegistro, despesas, getSaldoFinal()
69          );
70      }
71
72      // Getters e Setters
73
74      public double getRecebimento() {
75          return recebimento;
76      }
77      public void setRecebimento(double recebimento) {
78          if (recebimento < 0) throw new IllegalArgumentException("O valor de recebimento não pode ser negativo.");
79          this.recebimento = recebimento;
80      }
81
82      public int getServicoFeitos() {
83          return servicoFeitos;
```



```
    }
    public void setServicoFeitos(int servicoFeitos) {
        if (servicoFeitos < 0) throw new IllegalArgumentException("Quantidade de serviços feitos não pode ser negativa.");
        this.servicoFeitos = servicoFeitos;
    }

    public String getMetodoPagamento() {
        return metodoPagamento;
    }
    public void setMetodoPagamento(String metodoPagamento) {
        if (metodoPagamento == null || metodoPagamento.isBlank()) throw new IllegalArgumentException("Método de pagamento inválido.");
        this.metodoPagamento = metodoPagamento;
    }

    public LocalDate getDataRegistro() {
        return dataRegistro;
    }
    public void setDataRegistro(LocalDate dataRegistro) {
        if (dataRegistro.isAfter(LocalDate.now())) throw new IllegalArgumentException("Data futura não permitida.");
        this.dataRegistro = dataRegistro;
    }

    public double getDespesas() {
        return despesas;
    }
}
```

7 – Bibliografia

- DEITEL, H. M.; DEITEL, P. J. *Java: como programar*. 10. ed. São Paulo: Pearson Education do Brasil, 2016.
- ECKEL, B. *Thinking in Java*. 4th ed. Prentice Hall, 2006.
- ORACLE. *The Java™ Tutorials*. Disponível em: https://docs.oracle.com/javase/tutorial/. Acesso em: 24 maio 2025.
- SOMMERVILLE, I. *Engenharia de Software*. 10. ed. São Paulo: Pearson Education do Brasil, 2011.
- PRESSMAN, R. S. *Engenharia de Software*. 8. ed. São Paulo: McGraw-Hill, 2016.

8 – Ficha Técnica

UNIP
UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: Caiuã Leão de Souza TURMA: CC3P18 RA: G954945

CURSO: Ciência da computação CAMPUS: Rib. Preto - Vargas SEMESTRE: 3 TURNO: noite

CÓDIGO DA ATIVIDADE: 76B9 SEMESTRE: 3º Semestre ANO GRADE: 2025

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
26/03	Estudo de Java	8	Caiuã Leão		
01/04	Estudo de empacotamento	8	Caiuã Leão		
09/04	Estudo de Matrizes	8	Caiuã Leão		
20/04	Estudo de Abstração	8	Caiuã Leão		
27/04	Estudo de Polimorfismo	8	Caiuã Leão		
06/05	Pesquisa de Polimorfismo	8	Caiuã Leão		
14/05	Pesquisa de Herança	8	Caiuã Leão		
17/05	Pesquisa de Classes	8	Caiuã Leão		
23/05	Construção da Classe Principal	8	Caiuã Leão		
24/05	Edição de Video	8	Caiuã Leão		
25/05	Criação de documentação	8	Caiuã Leão		

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

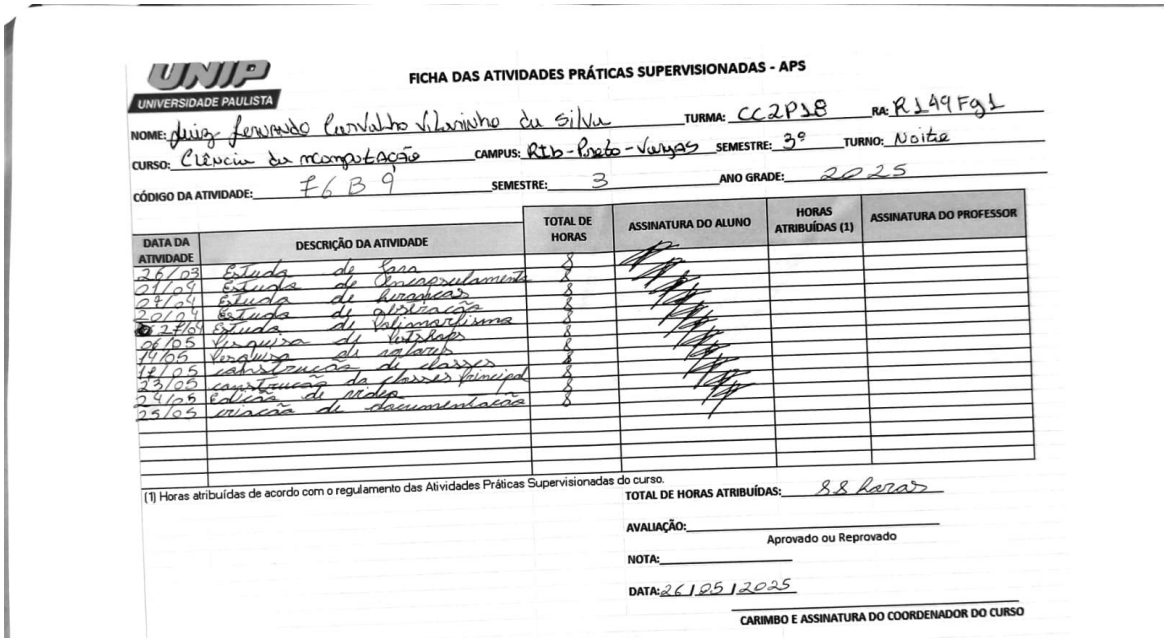
TOTAL DE HORAS ATRIBUÍDAS: 88 horas

AVALIAÇÃO: _____ Aprovado ou Reprovado

NOTA: _____

DATA: 26/05/25

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO



UNIP
UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: Luiz Fernando Carvalho Vilarinho da Silva TURMA: CC2P18 RA: R149FG1

CURSO: Ciência da computação CAMPUS: Rib - Preto - Vargas SEMESTRE: 3º TURNO: Noite

CÓDIGO DA ATIVIDADE: 76B9 SEMESTRE: 3 ANO GRADE: 2025

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
26/03	Estudo de Java	8			
01/04	Estudo de Encapsulamento	8			
27/04	Estudo de Heranças	8			
20/04	Estudo de Abstração	8			
27/04	Estudo de Polimorfismo	8			
06/05	Pesquisa de Interfaces	8			
14/05	Pesquisa de Matrizes	8			
17/05	Construções de classes	8			
23/05	Construção da Classe Principal	8			
24/05	Edição de video	8			
25/05	Criação de documentação	8			

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 88 horas

AVALIAÇÃO: _____ Aprovado ou Reprovado

NOTA: _____

DATA: 26/05/2025

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO

UNIP
UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: Guilherme do N. L. Olympio TURMA: CC3P18 RA: R08469-8

CURSO: Ciências da Computação CAMPUS: Rib Preto vagas SEMESTRE: 3 TURNO: Noite

CÓDIGO DA ATIVIDADE: 76B9 SEMESTRE: 3° ANO GRADE: 2025

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
26/03	Estudo do JAVA	8	Guilherme Olympio		
01/04	Estudo de encapsulamento	8	Guilherme Olympio		
09/04	Estudo de Heranças	8	Guilherme Olympio		
20/04	Estudo de Abstração	8	Guilherme Olympio		
23/04	Estudo de Polimorfismo	8	Guilherme Olympio		
06/05	Pesquisa de petShops	8	Guilherme Olympio		
14/05	Pesquisa de Valores	8	Guilherme Olympio		
13/05	Construção de classes	8	Guilherme Olympio		
23/05	Construção da classe principal	8	Guilherme Olympio		
24/05	Edição de vídeo	8	Guilherme Olympio		
25/05	Criação de documentação	8	Guilherme Olympio		

[1] Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 88 Horas

AVALIAÇÃO: _____ Aprovado ou Reprovado

NOTA: _____

DATA: 26/05/2025

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO

UNIP
UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: Pedra Augusta Nicolau maxima TURMA: CC3A18 RA: R057HJ9

CURSO: Ciencia da computacão CAMPUS: Rib Preta vagas SEMESTRE: 3 TURNO: naturna

CÓDIGO DA ATIVIDADE: 76B9 SEMESTRE: 3 ANO GRADE: 2025

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
26/03	Estudo do Java	8	Pedra Augusta		
01/04	Estudo de Encapsulamento	8	Pedra Augusta		
09/04	Estudo de Heranças	8	Pedra Augusta		
20/04	Estudo de abstração	8	Pedra Augusta		
23/04	Estudo de Polimorfismo	8	Pedra Augusta		
06/05	Pesquisa de PetShops	8	Pedra Augusta		
14/05	Pesquisa de valores	8	Pedra Augusta		
13/05	construção de classes	8	Pedra Augusta		
23/05	construção da classe Principal	8	Pedra Augusta		
24/05	Edição de video	8	Pedra Augusta		
25/05	criação de documentação	8	Pedra Augusta		

[1] Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

TOTAL DE HORAS ATRIBUÍDAS: 88 horas

AVALIAÇÃO: _____ Aprovado ou Reprovado

NOTA: _____

DATA: 26/05/2025

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO