

## 1 Descrição do problema

O sistema PetCare Manager deve permitir o cadastro e gerenciamento de **Cientes, Pets, Serviços e Agendamentos**. Deve armazenar os dados em arquivo com **cabeçalho** e controle de exclusão com lógica por **lápide**.

## 2 Objetivo do trabalho

- Desenvolver um sistema que permita o CRUD de Clientes, Pets, Serviços e Agendamentos.
- Garantir persistência em arquivos binários com controle de exclusão lógica.
- Fornecer documentação contendo DCU, DER e Arquitetura Proposta.

## 3 Requisitos funcionais

- RF01: Incluir Cliente.
- RF02: Incluir Pet.
- RF03: Incluir Serviço.
- RF04: Criar Agendamento.
- RF05: Listar registros ativos.
- RF06: Editar registros ativos.
- RF07: Excluir registros (lógica com lápide).

## 4 Requisitos não funcionais

- RNF01: O sistema não poderá utilizar console como interface.
- RNF02: GUI estática em JavaFX.
- RNF03: Persistência obrigatória em arquivos binários com cabeçalho
- RNF04: Documentação obrigatória (DCU + DER + Arquitetura).

## 5 Atores

- **Funcionário:** gerencia inserções, edições e exclusões.

## 6 Diagrama de Caso de Uso

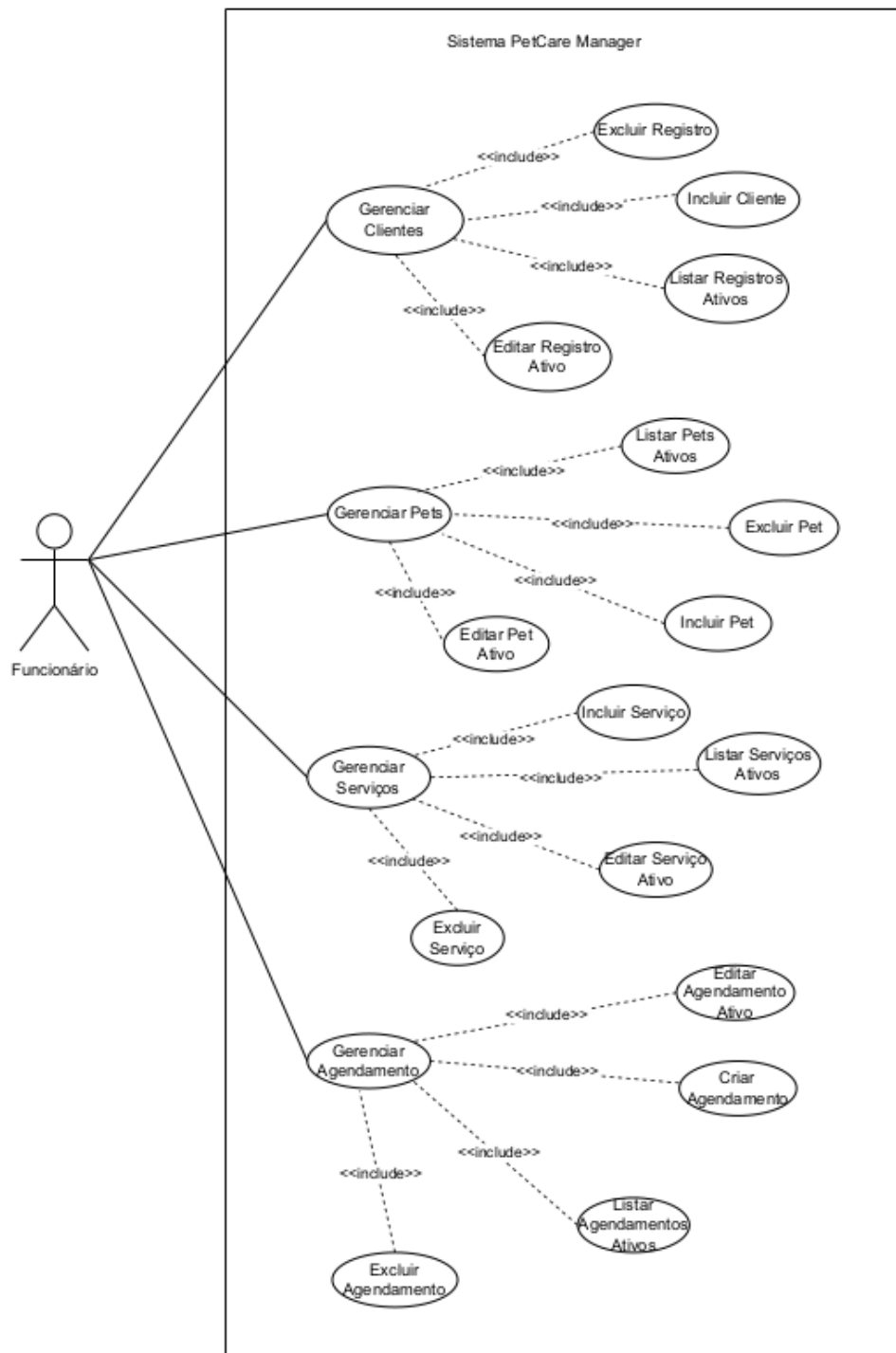


Figura 1: DCU: PCM

## 7 Diagrama Entidade-Relacionamento

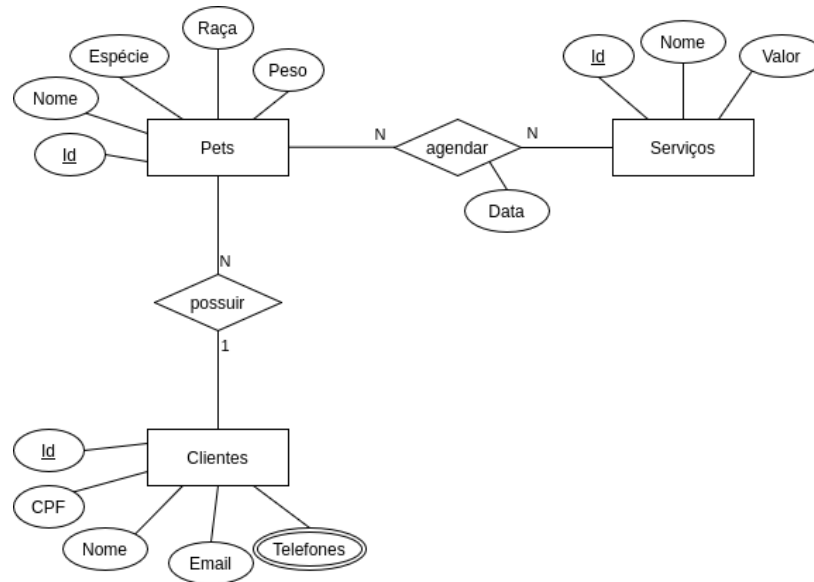


Figura 2: DER: PCM

## 8 Arquitetura Proposta

O sistema seguirá o padrão **MVC + DAO**, onde:

- **Model:** classes de domínio (Cliente, Pet, Serviço, Agendamento).
- **DAO:** acesso a arquivos binários com cabeçalho e lápide
- **Controller:** regras de negócio
- **View:** interface em JavaFX

## 9 Formulário

### 9.1 Estrutura para representar os registros

#### 9.1.1 Cabeçalho

[int: último ID usado - 4 bytes][long: ponteiro lista de excluídos - 8 bytes]

#### 9.1.2 Exclusão lógica

[byte: lápide - 1 byte][short: tamanho - 2 bytes][byte[]: dados - n bytes]

### 9.1.3 Cliente

int id → 4 bytes  
String cpf → 2 + n bytes (tamanho UTF + caracteres)  
String nome → 2 + n bytes  
String email → 2 + n bytes  
String[] telefones → 1 byte (quantidade) + 2 + n bytes por telefone

### 9.1.4 Pet

int id → 4 bytes  
String nome → 2 + n bytes  
String especie → 2 + n bytes  
String raca → 2 + n bytes  
float peso → 4 bytes  
String cpfDono → 2 + n bytes (chave estrangeira para Cliente)

### 9.1.5 Serviço

int id → 4 bytes  
String nome → 2 + n bytes  
int valor → 4 bytes (valor em reais, armazenado como inteiro)

### 9.1.6 Agendar

int id → 4 bytes  
String data → 2 + n bytes (LocalDate convertido para String: "2025-11-05")  
int idPet → 4 bytes (chave estrangeira para Pet)  
int idServico → 4 bytes (chave estrangeira para Serviço)

### 9.1.7 Interface comum

```
1 public interface Registro {  
2     void setId(int i);  
3     int getId();  
4     byte[] toByteArray() throws IOException;  
5     void fromByteArray(byte[] b) throws IOException;  
6 }
```

## 9.2 Tratamento das Strings multivaloradas

### 9.2.1 Atributo multivalorado

```
1 private String[] telefones; // Array de Strings
```

### 9.2.2 Serialização

```
1 // Passo 1: Escrever a quantidade de elementos
2 dos.writeByte(getTelefones().length); // 1 byte
3
4 // Passo 2: Escrever cada telefone individualmente
5 for (String telefone : this.telefones) {
6     dos.writeUTF(telefone); // 2 bytes (tamanho) + n bytes (conteudo)
7 }
```

### 9.2.3 Desserialização

```
1 // Passo 1: Ler a quantidade de elementos
2 int telefonesLength = dis.readByte(); // Le 1 byte
3
4 // Passo 2: Criar array com o tamanho correto
5 String[] telefones = new String[telefonesLength];
6
7 // Passo 3: Ler cada telefone em sequencia
8 for (int i = 0; i < telefonesLength; i++) {
9     telefones[i] = dis.readUTF(); // Le cada string
10 }
11
12 // Passo 4: Atribuir ao objeto
13 setTelefones(telefones);
```

### 9.2.4 Vantagens

- **Tamanho Variável**
  - Suporte de 0 a 127 telefones por cliente
  - Não há desperdício de espaço com posições vazias
- **Flexibilidade**
  - Cada telefone pode ter tamanhos diferentes
  - UTF permite caracteres especiais (parênteses, hífen)
- **Ordem Preservada**
  - Os telefones são lidos na mesma ordem em que foram escritos
- **Eficiência de Espaço**
  - Apenas 1 byte para o contador de quantidade
  - Strings UTF já incluem seu próprio tamanho (2 bytes)

## 9.3 Exclusão lógica

A exclusão lógica utiliza um marcador de 1 byte chamado lápide para indicar se o registro está ativo (') ou excluído (\*'), sem remover fisicamente os dados do arquivo.

### 9.3.1 Criação de registro

```
1 public int create(T obj) throws Exception {
2     // ... gerar ID ...
3     byte[] dados = obj.toByteArray();
4
5     long endereco = getDeleted(dados.length); // Tenta reusar espaco
6     if (endereco == -1) {
7         // Nenhum espaco excluido disponivel - adicionar no fim
8         arquivo.seek(arquivo.length());
9         endereco = arquivo.getFilePointer();
10        arquivo.writeByte(' '); // LAPIDE ATIVA
11        arquivo.writeShort(dados.length);
12        arquivo.write(dados);
13    } else {
14        // Reusar espaco de registro excluido
15        arquivo.seek(endereco);
16        arquivo.writeByte(' '); // REATIVAR LAPIDE
17        arquivo.skipBytes(2);
18        arquivo.write(dados);
19    }
20
21    indice.inserir(obj.getId(), endereco);
22    return obj.getId();
23 }
```

### 9.3.2 Leitura de registro

```
1 public T read(int id) throws Exception {
2     long endereco = indice.buscar(id);
3     if (endereco == -1) {
4         return null; // Nao encontrado no indice
5     }
6
7     arquivo.seek(endereco);
8     byte lapide = arquivo.readByte(); // LER LAPIDE
9
10    if (lapide != ' ') {
11        return null; // REGISTRO EXCLUIDO LOGICAMENTE
12    }
13
14    // Registro ativo - continuar leitura
15    short tamanho = arquivo.readShort();
16    byte[] dados = new byte[tamanho];
17    arquivo.read(dados);
18
19    T obj = construtor.newInstance();
20    obj.fromByteArray(dados);
21    return obj;
22 }
```

### 9.3.3 Exclusão

```
1 public boolean delete(int id) throws Exception {
2     long endereco = indice.buscar(id);
3     if (endereco == -1) {
4         return false; // Nao encontrado
5     }
6 }
```

```

7      arquivo.seek(endereco);
8      byte lapide = arquivo.readByte();
9
10     if (lapide != ' ') {
11         return false; // Ja foi excluido
12     }
13
14     short tamanho = arquivo.readShort();
15
16     // MARCAR COMO EXCLUIDO
17     arquivo.seek(endereco);
18     arquivo.writeByte('*'); // EXCLUSAO LOGICA!
19
20     // Adicionar a lista de espacos reutilizaveis
21     addDeleted(tamanho, endereco);
22
23     // Remover do indice
24     indice.remove(id);
25
26     return true;
27 }

```

## 9.4 Chaves utilizadas

### 9.4.1 Índice sequencial

Todas as entidades do sistema (Cliente, Pet, Serviço e Agendamento) possuem uma chave primária do tipo inteiro (ID) que é gerenciada por um índice sequencial. Este índice é mantido automaticamente pela classe Arquivo e armazena pares de ID e endereço físico no arquivo, permitindo acesso direto aos registros.

```

1  public class Arquivo<T extends Registro> {
2      private IndiceSequencial indice;
3
4      public Arquivo(...) throws Exception {
5          ...
6          this.indice = new IndiceSequencial(nomeArquivo);
7          ...
8      }
9
10     public T read(int id) throws Exception {
11         long endereco = indice.buscar(id);
12         if (endereco == -1) {
13             return null;
14         }
15         ...
16     }
17 }

```

### 9.4.2 Chaves candidatas: CPF, e-mail

O CPF é utilizado como chave candidata alternativa para a entidade Cliente, sendo garantida sua unicidade em todo o sistema. Embora não possua um índice dedicado, o CPF é uma chave natural importante que permite identificar clientes de forma única sem necessidade do ID. Similar ao CPF, o email é tratado como uma chave candidata única na entidade Cliente. A unicidade do email é validada tanto na inclusão quanto na alteração de clientes, impedindo que dois clientes compartilhem o mesmo endereço de email.

```

1 public class ClienteDAO {
2     private Arquivo<Cliente> arqClientes;
3
4     public Cliente buscarClientePorCPF(String cpf) throws Exception {
5         return arqClientes.findBy(cliente -> cliente.getCpf().equals(cpf));
6     }
7
8     public boolean incluirCliente(Cliente cliente) throws Exception {
9         // Validar se CPF ja existe
10        if (buscarClientePorCPF(cliente.getCpf()) != null) {
11            throw new IllegalArgumentException(
12                "Ja existe cadastro com o CPF: " + cliente.getCpf()
13            );
14        }
15
16        return arqClientes.create(cliente) > 0;
17    }
18
19    public boolean alterarCliente(Cliente cliente) throws Exception {
20        Cliente clienteExistente = arqClientes.read(cliente.getId());
21
22        // Validar se CPF mudou e se ja existe outro cliente com o novo CPF
23        if (!clienteExistente.getCpf().equals(cliente.getCpf())) {
24            Cliente clienteComMesmoCPF = buscarClientePorCPF(cliente.getCpf());
25            if (clienteComMesmoCPF != null &&
26                clienteComMesmoCPF.getId() != cliente.getId()) {
27                throw new IllegalArgumentException(
28                    "Ja existe cadastro com o CPF: " + cliente.getCpf()
29                );
30            }
31        }
32
33        return arqClientes.update(cliente);
34    }
35
36    public Cliente buscarClientePorEmail(String email) throws Exception {
37        return arqClientes.findBy(cliente -> cliente.getEmail().equals(email));
38    }
39
40    public boolean incluirCliente(Cliente cliente) throws Exception {
41        // Validar se email ja existe
42        if (cliente.getEmail() != null && !cliente.getEmail().trim().isEmpty()) {
43            if (buscarClientePorEmail(cliente.getEmail()) != null) {
44                throw new IllegalArgumentException(
45                    "Ja existe cadastro com o email: " + cliente.getEmail()
46                );
47            }
48        }
49
50        return arqClientes.create(cliente) > 0;
51    }
52 }

```

### 9.4.3 CPF do Dono - Chave Estrangeira com Hash Extensível (Pet)

O relacionamento 1:N entre Cliente (dono) e Pet é implementado através de uma chave estrangeira (CPF do dono) indexada por um Hash Extensível. Esta estrutura permite que um cliente possua múltiplos pets, enquanto cada pet pertence a apenas um cliente.

```

1 public class RelacionamentoPetDono implements ... {
2     private String cpfDono; // Chave de busca (CPF do cliente/dono)

```

```

3     private int idPet;           // ID do pet associado ao dono
4
5     public static final short TAMANHO_FIXO = 15; // 11 bytes CPF + 4 bytes ID
6
7     @Override
8     public int hashCode() {
9         // Combina CPF e ID do Pet para gerar uma chave unica
10        String cpfLimpo = cpfDono.replaceAll("[^0-9]", "");
11        String chaveUnica = cpfLimpo + "#" + idPet;
12        return chaveUnica.hashCode();
13    }
14    ...
15 }

```

```

1 public class IndiceHashExtensivel {
2     private HashExtensivel<RelacionamentoPetDono> hashExtensivel;
3     private static final int REGISTROS_POR_CESTO = 5;
4
5     public boolean inserir(String cpfDono, int idPet) throws Exception {
6         RelacionamentoPetDono rel = new RelacionamentoPetDono(cpfDono, idPet);
7         return hashExtensivel.create(rel);
8     }
9     ...
10 }

```

#### 9.4.4 Chave Composta (idPet, idServico) - Árvore B+ para Relacionamento N:N

O relacionamento N:N entre Pet e Serviço, materializado através da entidade Agendamento, é implementado usando uma Árvore B+ com chave composta. Esta estrutura garante que um mesmo par (Pet, Serviço) não possa ter múltiplos agendamentos simultâneos, enquanto permite que um pet tenha agendamentos com diferentes serviços e um serviço seja agendado para diferentes pets.

```

1 public class RegistroAgendamento implements ... {
2     private int idPet;           // Primeira chave (ordenacao primaria)
3     private int idServico;       // Segunda chave (ordenacao secundaria)
4     private int idAgendamento; // Valor associado
5
6     @Override
7     public short size() {
8         return 12; // 3 inteiros (4 bytes cada)
9     }
10
11    @Override
12    public byte[] toByteArray() throws IOException {
13        ByteArrayOutputStream baos = new ByteArrayOutputStream();
14        DataOutputStream dos = new DataOutputStream(baos);
15
16        dos.writeInt(idPet);
17        dos.writeInt(idServico);
18        dos.writeInt(idAgendamento);
19
20        return baos.toByteArray();
21    }
22
23    @Override
24    public int compareTo(RegistroAgendamento obj) {
25        // Comparacao por chave composta (idPet, idServico)
26        // Primeiro compara por idPet
27        if (this.idPet != obj.idPet) {
28            return Integer.compare(this.idPet, obj.idPet);
29        }

```

```

30         // Se idPet for igual, compara por idServico
31         return Integer.compare(this.idServico, obj.idServico);
32     }
33 }

```

## 9.5 Implementação do 1:N

O relacionamento 1:N entre Cliente (dono) e Pet é implementado através de Hash Extensível, permitindo que um cliente tenha múltiplos pets

### 9.5.1 Fluxo de busca

```

1  // 1. Recebe CPF do cliente
2  String cpf = "15974074610";
3
4  // 2. Busca IDs na Hash Extensível
5  List<Integer> idsPets = indiceHash.buscarIdsPetsPorCpf(cpf);
6  // Retorna: [1, 5, 12]
7
8  // 3. Para cada ID, busca no arquivo principal via indice sequencial
9  for (Integer idPet : idsPets) {
10     Pet pet = arqPets.read(idPet); // Usa indice sequencial interno
11     pets.add(pet);
12 }

```

### 9.5.2 Chave composta única

```

1  @Override
2  public int hashCode() {
3      String cpfLimpo = cpfDono.replaceAll("[^0-9]", "");
4      String chaveUnica = cpfLimpo + "#" + idPet; // Ex: "15974074610#5"
5      return chaveUnica.hashCode();
6  }

```

## 9.6 Navegação

```

1  public List<Integer> buscarIdsPetsPorCpf(String cpfDono) throws Exception {
2      List<Integer> idsPets = new ArrayList<>();
3      String cpfLimpo = cpfDono.replaceAll("[^0-9]", "");
4
5      // Varre TODOS os relacionamentos na hash
6      List<RelacionamentoPetDono> todosRelacionamentos = hashExtensivel.listAll();
7
8      // Filtra por CPF
9      for (RelacionamentoPetDono rel : todosRelacionamentos) {
10         String cpfRelLimpo = rel.getCpfDono().replaceAll("[^0-9]", "");
11         if (cpfRelLimpo.equals(cpfLimpo)) {
12             idsPets.add(rel.getIdPet()); // Acumula IDs
13         }
14     }
15
16     return idsPets; // Retorna: [1, 5, 12]
17 }

```

### 9.6.1 Integridade referencial

- Inserção de Pet
  - Pet só existe se relacionamento for criado na hash
  - Validação de duplicatas (mesmo nome + mesmo dono)

```
1 public boolean incluirPet(Pet pet) throws Exception {
2     // 1. Cria pet no arquivo principal
3     int idGerado = arqPets.create(pet); // Gera ID sequencial
4
5     if (idGerado > 0) {
6         // 2. Cria relacionamento na hash
7         String cpfDono = pet.getDono().getCpf();
8         indiceHash.inserir(cpfDono, idGerado); // Hash: CPF -> idPet
9     }
10
11     return true;
12 }
```

- Exclusão em cascata
  - Nenhum pet fica órfão (sem dono)
  - Hash mantém consistência (remove todos os relacionamentos)
  - Agendamentos dos pets são excluídos antes

```
1 public boolean excluirCliente(int id) throws Exception {
2     Cliente cliente = arqClientes.read(id);
3
4     // 1. Buscar todos os pets do cliente via hash
5     List<Integer> idsPets = indiceHash.buscarIdsPetsPorCpf(cliente.getCpf());
6
7     // 2. Excluir cada pet (cascata: Pet -> Agendamentos)
8     PetDAO petDAO = new PetDAO();
9     for (Integer idPet : idsPets) {
10         petDAO.excluirPet(idPet); // Remove pet + agendamentos
11     }
12
13     // 3. Excluir cliente do arquivo
14     boolean excluido = arqClientes.delete(id);
15
16     if (excluido) {
17         // 4. Limpar relacionamentos da hash
18         indiceHash.removeTodosPorCpf(cliente.getCpf());
19     }
20
21     return excluido;
22 }
```

- Alteração de pet
  - CPF do dono não muda, logo o relacionamento na hash permanece válido

```
1 public boolean alterarPet(Pet pet) throws Exception {
2     // Nao altera o relacionamento CPF -> idPet (imutavel neste contexto)
3     // Apenas atualiza dados do pet no arquivo principal
4     return arqPets.update(pet);
5 }
```

## 9.7 Implementação do N:N

O relacionamento N:N entre Pet e Serviço é implementado através da Árvore B+ (ordem 5) com chave composta (idPet, idServiço)

### 9.7.1 Fluxo de busca por chave composta

```
1 // 1. Criar registro com chave composta
2 RegistroAgendamento chave = new RegistroAgendamento(idPet=5, idServiço=3, 0);
3
4 // 2. Buscar na Arvore B+ (O(log n))
5 ArrayList<RegistroAgendamento> resultados = indiceBMais.read(chave);
6
7 // 3. Obter ID do agendamento
8 int idAgendamento = resultados.get(0).getIdAgendamento();
9
10 // 4. Buscar agendamento completo no arquivo principal
11 Agendar agendamento = arqAgendamentos.read(idAgendamento);
```

### 9.7.2 Chave composta lexicográfica

```
1 @Override
2 public int compareTo(RegistroAgendamento obj) {
3     // 1o criterio: idPet
4     if (this.idPet != obj.idPet) {
5         return Integer.compare(this.idPet, obj.idPet);
6     }
7     // 2o criterio: idServiço (se idPet for igual)
8     return Integer.compare(this.idServiço, obj.idServiço);
9 }
```

### 9.7.3 Navegação bidirecional

- Pet → Serviços

```
1 public List<Agendar> buscarAgendamentosPorPet(int idPet) throws Exception {
2     // Usa varredura no arquivo principal
3     return arqAgendamentos.findAll(a -> a.getIdPet() == idPet);
4 }
```

- Serviço → Pets

```
1 public List<Agendar> buscarAgendamentosPorServiço(int idServiço) throws Exception {
2     // Varredura no arquivo principal
3     return arqAgendamentos.findAll(a -> a.getIdServiço() == idServiço);
4 }
```

- Par (Pet, Serviço) → Agendamento

```

1 public Agendar buscarAgendamento(int idPet, int idServico) throws Exception {
2     // Usa arvore B+ para busca eficiente
3     RegistroAgendamento chave = new RegistroAgendamento(idPet, idServico, 0);
4     ArrayList<RegistroAgendamento> resultados = indiceBMais.read(chave);
5
6     if (resultados.isEmpty()) return null;
7
8     int idAgendamento = resultados.get(0).getIdAgendamento();
9     return arqAgendamentos.read(idAgendamento);
10 }

```

#### 9.7.4 Integridade referencial

- Inserção de Agendamento
  - Unicidade: cada par (Pet, Serviço) aparece apenas 1 vez na árvore
  - Consistência: registro só existe se estiver em ambos (arquivo + árvore B+)

```

1 public boolean incluirAgendamento(Agendar agendamento) throws Exception {
2     // 1. Validar duplicata
3     if (existeAgendamento(agendamento.getIdPet(), agendamento.getIdServico())) {
4         throw new IllegalArgumentException(...);
5     }
6
7     // 2. Criar agendamento no arquivo principal
8     int idGerado = arqAgendamentos.create(agendamento);
9
10    if (idGerado > 0) {
11        // 3. Inserir chave composta na Arvore B+
12        RegistroAgendamento registro = new RegistroAgendamento(
13            agendamento.getIdPet(),
14            agendamento.getIdServico(),
15            idGerado // Ponteiro para o arquivo principal
16        );
17        indiceBMais.create(registro); // Insere na B+
18    }
19
20    return true;
21 }

```

- Exclusão em cascata
  - Nenhum agendamento fica órfão (sem pet ou sem serviço)
  - Árvore B+ sempre sincronizada com arquivo principal
  - Exclusão atômica (remove de ambos ou de nenhum)

```

1 public int excluirAgendamentosPorPet(int idPet) throws Exception {
2     // 1. Buscar todos os agendamentos do pet
3     List<Agendar> agendamentos = buscarAgendamentosPorPet(idPet);
4
5     int count = 0;
6     for (Agendar agendamento : agendamentos) {
7         // 2. Para cada agendamento, excluir via chave composta
8         if (excluirAgendamento(agendamento.getIdPet(), agendamento.getIdServico())) {
9             count++; // Remove do arquivo + remove da B+
10        }
11    }
12    return count;
13 }

```

```

10     }
11 }
12
13     return count;
14 }
15
16 public int excluirAgendamentosPorServico(int idServico) throws Exception {
17     List<Agendar> agendamentos = buscarAgendamentosPorServico(idServico);
18
19     int count = 0;
20     for (Agendar agendamento : agendamentos) {
21         if (excluirAgendamento(agendamento.getIdPet(), agendamento.getIdServico())) {
22             count++;
23         }
24     }
25
26     return count;
27 }
28
29 public boolean excluirAgendamento(int idPet, int idServico) throws Exception {
30     // 1. Buscar ID do agendamento na B+
31     RegistroAgendamento chave = new RegistroAgendamento(idPet, idServico, 0);
32     ArrayList<RegistroAgendamento> resultados = indiceBMais.read(chave);
33
34     if (resultados.isEmpty()) return false;
35
36     int idAgendamento = resultados.get(0).getIdAgendamento();
37
38     // 2. Remover do arquivo principal
39     boolean removido = arqAgendamentos.delete(idAgendamento);
40
41     if (removido) {
42         // 3. Remover da Arvore B+
43         indiceBMais.delete(resultados.get(0));
44     }
45
46     return removido;
47 }

```

- Alteração de agendamento
  - Unicidade mantida (não cria duplicatas)
  - Índice sempre sincronizado (remove antiga + insere nova)

```

1 public boolean alterarAgendamento(Agendar agendamento) throws Exception {
2     Agendar agendamentoExistente = arqAgendamentos.read(agendamento.getId());
3
4     // Se mudou Pet OU Servico, atualizar chave na arvore B+
5     if (agendamentoExistente.getIdPet() != agendamento.getIdPet() ||
6         agendamentoExistente.getIdServico() != agendamento.getIdServico()) {
7
8         // 1. Validar nova chave
9         if (existeAgendamento(agendamento.getIdPet(), agendamento.getIdServico())) {
10             throw new IllegalArgumentException(...);
11         }
12
13         // 2. Remover chave antiga da B+
14         RegistroAgendamento registroAntigo = new RegistroAgendamento(
15             agendamentoExistente.getIdPet(),
16             agendamentoExistente.getIdServico(),
17             agendamento.getId()

```

```

18     );
19     indiceBMais.delete(registroAntigo);
20
21     // 3. Inserir nova chave na B+
22     RegistroAgendamento registroNovo = new RegistroAgendamento(
23         agendamento.getIdPet(),
24         agendamento.getIdServico(),
25         agendamento.getId()
26     );
27     indiceBMais.create(registroNovo);
28 }
29
30 // 4. Atualizar arquivo principal
31 return arqAgendamentos.update(agendamento);
32 }

```

## 9.8 Persistência dos índices em disco

O sistema implementa três tipos de índices persistentes, cada um com formato específico de armazenamento, estratégias de atualização e mecanismos de sincronização com os dados principais. A seguir, são detalhados os aspectos de persistência de cada estrutura de indexação.

### 9.8.1 Índice Sequencial

Cada entrada no índice ocupa exatamente 12 bytes (4 bytes para o ID inteiro + 8 bytes para o endereço long). Os registros são mantidos ordenados por ID, permitindo busca binária. O índice sequencial utiliza uma estratégia de cache completo em memória com sincronização imediata após cada modificação. A sincronização é automática e acoplada através da classe Arquivo.

```

1  public void inserir(int id, long endereco) throws Exception {
2      // 1. Atualizar em memoria
3      RegistroIndice novoRegistro = new RegistroIndice(id, endereco);
4      int posicao = buscaBinariaInsercao(id);
5      indices.add(posicao, novoRegistro); // Insere ordenadamente
6
7      // 2. Sincronizar com disco imediatamente
8      salvarIndices();
9  }
10
11 public boolean remover(int id) throws Exception {
12     int posicao = buscaBinaria(id);
13     if (posicao >= 0) {
14         // 1. Remover da memoria
15         indices.remove(posicao);
16
17         // 2. Sincronizar com disco imediatamente
18         salvarIndices();
19         return true;
20     }
21     return false;
22 }
23
24 public boolean atualizar(int id, long novoEndereco) throws Exception {
25     int posicao = buscaBinaria(id);
26     if (posicao >= 0) {
27         // 1. Atualizar em memoria
28         indices.get(posicao).endereco = novoEndereco;
29
30         // 2. Sincronizar com disco imediatamente
31         salvarIndices();

```

```

32         return true;
33     }
34     return false;
35 }
36
37 private void salvarIndices() throws Exception {
38     // Reescreve TODO o arquivo
39     arquivo.setLength(0); // Limpa o arquivo
40     arquivo.seek(0);
41
42     for (RegistroIndice registro : indices) {
43         arquivo.writeInt(registro.id);
44         arquivo.writeLong(registro.endereco);
45     }
46 }

```

### 9.8.2 Índice Hash Extensível

O hash extensível é utilizado para o relacionamento 1:N entre Cliente e Pet, armazenando pares (CPF do dono, ID do pet). Para o relacionamento Pet-Dono, cada elemento tem 15 bytes (11 bytes para CPF + 4 bytes para ID do pet). O hash extensível utiliza carregamento sob demanda com sincronização seletiva.

```

1 public boolean create(T elem) throws Exception {
2     // 1. Carregar diretório completo para memória
3     byte[] bd = new byte[(int) arqDiretorio.length()];
4     arqDiretorio.seek(0);
5     arqDiretorio.read(bd);
6     diretorio = new Diretorio();
7     diretorio.fromByteArray(bd);
8
9     // 2. Calcular hash e localizar cesto
10    int i = diretorio.hash(elem.hashCode());
11    long enderecoCesto = diretorio.endereco(i);
12
13    // 3. Carregar apenas o cesto necessario
14    Cesto c = new Cesto(construtor, quantidadeDadosPorCesto);
15    byte[] ba = new byte[c.size()];
16    arqCestos.seek(enderecoCesto);
17    arqCestos.read(ba);
18    c.fromByteArray(ba);
19
20    // 4. Se cesto tem espaço, inserir e gravar
21    if (!c.full()) {
22        c.create(elem);
23        arqCestos.seek(enderecoCesto);
24        arqCestos.write(c.toByteArray()); // Grava apenas este cesto
25        return true;
26    }
27
28    // 5. Se cesto cheio, dividir
29    if (c.profundidadeLocal >= diretorio.profundidadeGlobal) {
30        diretorio.duplica(); // Duplica diretório
31    }
32
33    // 6. Criar dois novos cestos
34    Cesto c1 = new Cesto(construtor, quantidadeDadosPorCesto,
35                          c.profundidadeLocal + 1);
36    arqCestos.seek(enderecoCesto);
37    arqCestos.write(c1.toByteArray());
38
39    Cesto c2 = new Cesto(construtor, quantidadeDadosPorCesto,

```

```

40         c.profundidadeLocal + 1);
41     long novoEndereco = arqCestos.length();
42     arqCestos.seek(novoEndereco);
43     arqCestos.write(c2.toByteArray());
44
45     // 7. Atualizar diretório
46     // ... atualizar ponteiros ...
47
48     // 8. Gravar diretório atualizado
49     bd = diretorio.toByteArray();
50     arqDiretorio.seek(0);
51     arqDiretorio.write(bd);
52
53     // 9. Redistribuir elementos (chamadas recursivas)
54     for (int j = 0; j < c.quantidade; j++) {
55         create(c.elementos.get(j));
56     }
57     create(elem);
58
59     return true;
60 }
61
62 public T read(int chave) throws Exception {
63     // 1. Carregar diretório
64     byte[] bd = new byte[(int) arqDiretorio.length()];
65     arqDiretorio.seek(0);
66     arqDiretorio.read(bd);
67     diretorio = new Diretorio();
68     diretorio.fromByteArray(bd);
69
70     // 2. Calcular hash e localizar cesto
71     int i = diretorio.hash(chave);
72     long enderecoCesto = diretorio.endereco(i);
73
74     // 3. Carregar apenas o cesto necessario
75     Cesto c = new Cesto(construtor, quantidadeDadosPorCesto);
76     byte[] ba = new byte[c.size()];
77     arqCestos.seek(enderecoCesto);
78     arqCestos.read(ba);
79     c.fromByteArray(ba);
80
81     // 4. Buscar no cesto (em memória)
82     return c.read(chave);
83 }

```

### 9.8.3 Índice Árvore B+

A Árvore B+ é utilizada para o relacionamento N:N entre Pet e Serviço através dos agendamentos, com chave composta (idPet, idServiço). Para a chave composta (idPet, idServiço), cada elemento tem 12 bytes (4 bytes para idPet + 4 bytes para idServiço + 4 bytes para idAgendamento), o tamanho de cada página é fixo e calculado com base na ordem da árvore. A Árvore B+ utiliza acesso direto por posição com modificações in-place.

```

1 public boolean create(T elem) throws Exception {
2     // 1. Recuperar raiz
3     arquivo.seek(0);
4     long raiz = arquivo.readLong();
5
6     if (raiz == -1) {
7         // Árvore vazia - criar primeira página
8         Pagina pagina = new Pagina(construtor, ordem);

```

```

9      pagina.elementos.add(elem);
10
11      // Gravar no final do arquivo
12      long novaPosicao = arquivo.length();
13      arquivo.seek(novaPosicao);
14      arquivo.write(pagina.toByteArray());
15
16      // Atualizar raiz
17      arquivo.seek(0);
18      arquivo.writeLong(novaPosicao);
19
20      return true;
21  }
22
23  // 2. Insercao recursiva
24  cresceu = false;
25  elemAux = null;
26  paginaAux = -1;
27
28  create1(elem, raiz);
29
30  // 3. Se houve crescimento, criar nova raiz
31  if (cresceu) {
32      Pagina novaPagina = new Pagina(construtor, ordem);
33      novaPagina.elementos.add(elemAux);
34      novaPagina.filhos.add(raiz);
35      novaPagina.filhos.add(paginaAux);
36
37      long novaPosicao = arquivo.length();
38      arquivo.seek(novaPosicao);
39      arquivo.write(novaPagina.toByteArray());
40
41      // Atualizar ponteiro da raiz
42      arquivo.seek(0);
43      arquivo.writeLong(novaPosicao);
44  }
45
46  return true;
47  }
48
49  private void create1(T elem, long enderecoPagina) throws Exception {
50      // Carregar pagina
51      Pagina pagina = new Pagina(construtor, ordem);
52      byte[] buffer = new byte[pagina.TAMANHO_PAGINA];
53      arquivo.seek(enderecoPagina);
54      arquivo.read(buffer);
55      pagina.fromByteArray(buffer);
56
57      // ... logica de insercao recursiva ...
58
59      // Se pagina foi modificada, gravar de volta
60      arquivo.seek(enderecoPagina);
61      arquivo.write(pagina.toByteArray());
62  }

```

## 9.9 Estrutura do projeto no GitHub

O projeto **tp-aeds3** está organizado seguindo uma **arquitetura em camadas** (MVC adaptado para JavaFX) com foco em persistência de dados e estruturas de indexação.

### 9.9.1 Raiz

```
tp-aeds3/
├── docs/
│   ├── dcu.png
│   ├── der.png
│   └── docs-tp.pdf
├── src/
├── .gitignore
├── README.md
├── pom.xml
└── run.sh
```

### 9.9.2 Pasta src (todas as implementações em código)

```
src/
├── app/
│   ├── Main.java
│   └── BateriaTestes.java
├── controller/
│   ├── MainController.java
│   ├── ClienteController.java
│   ├── PetController.java
│   ├── ServicoController.java
│   ├── AgendarController.java
│   └── AlterarAgendamentoDialogController.java
├── model/
│   ├── Cliente.java
│   ├── Pet.java
│   ├── Servico.java
│   └── Agendar.java
├── dao/
│   ├── Arquivo.java
│   ├── Registro.java
│   ├── ClienteDAO.java
│   ├── PetDAO.java
│   ├── ServicoDAO.java
│   ├── AgendarDAO.java
│   ├── IndiceSequencial.java
│   ├── HashExtensivel.java
│   ├── IndiceHashExtensivel.java
│   ├── ArvoreBMais.java
│   ├── RelacionamentoPetDono.java
│   ├── RegistroHashExtensivel.java
│   └── RegistroArvoreBMais.java
└── main/resources/
```

```
├── css/
│   └── Style.css
├── images/
├── view/
│   ├── MainView.fxml
│   ├── ClienteView.fxml
│   ├── PetView.fxml
│   ├── ServicoView.fxml
│   ├── AgendarView.fxml
│   └── AlterarAgendamentoDialog.fxml
```

### 9.9.3 Pasta de armazenamento de dados (gerada na execução)

```
dados/
├── clientes/
│   ├── clientes.db
│   └── clientes.idx
├── pets/
│   ├── pets.db
│   ├── pets.idx
│   ├── pets_hash.dir
│   └── pets_hash.db
├── servicos/
│   ├── servicos.db
│   └── servicos.idx
└── agendamentos/
    ├── agendamentos.db
    ├── agendamentos.idx
    └── agendamentos_bmais.db
```