

1 Descrição do problema

O sistema PetCare Manager deve permitir o cadastro e gerenciamento de **Clientes, 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

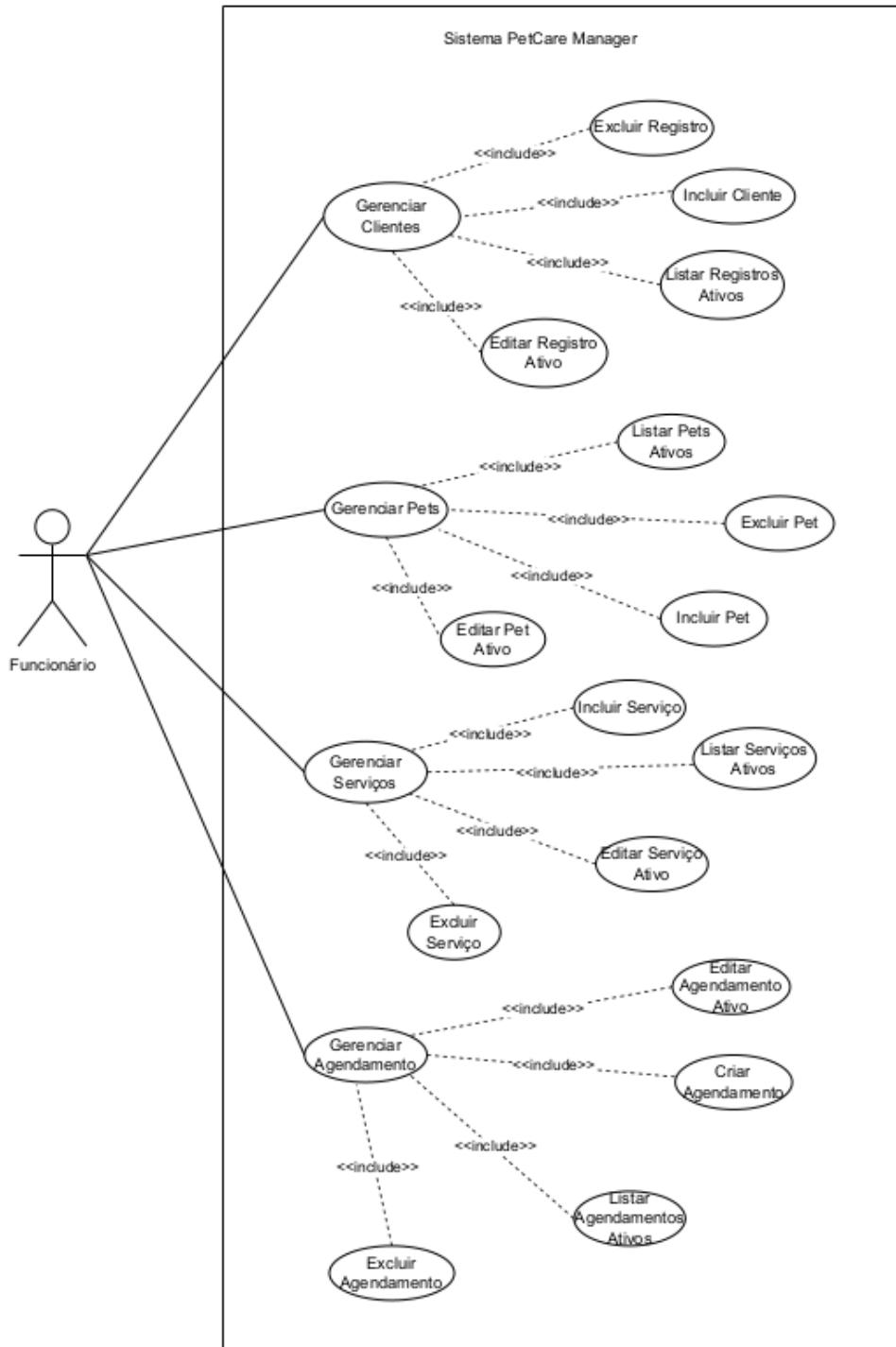


Figura 1: DCU: PCM

7 Diagrama Entidade-Relacionamento

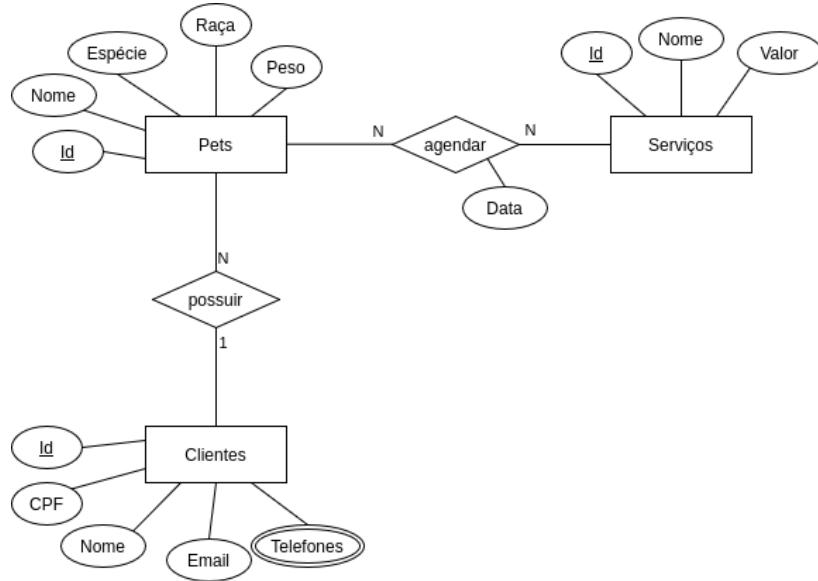


Figura 2: DER: PCM

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 idServiço → 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 (conteúdo)
7 }
```

9.2.3 Dessorialização

```
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 sequência
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, hífens)

- **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 espaço
6     if (endereco == -1) {
7         // Nenhum espaço excluído disponível - 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 espaço de registro excluído
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; // Não encontrado no índice
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; // Não 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.remover(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, idServiço) - 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 idServiço
31     return Integer.compare(this.idServiço, obj.idServiço);
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 índice sequencial
9 for (Integer idPet : idsPets) {
10    Pet pet = arqPets.read(idPet); // Usa índice 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.removerTodosPorCPF(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     // Não altera o relacionamento CPF -> idPet (imutável 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 Árvore 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 idServiço) throws Exception {
2     // Usa arvore B+ para busca eficiente
3     RegistroAgendamento chave = new RegistroAgendamento(idPet, idServiço, 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.getIdServiço())) {
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.getIdServiço(),
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.getIdServiço())) {
9             count++; // Remove do arquivo + remove da B+

```

```

10         }
11     }
12
13     return count;
14 }
15
16 public int excluirAgendamentosPorServico(int idServiço) throws Exception {
17     List<Agendar> agendamentos = buscarAgendamentosPorServiço(idServiço);
18
19     int count = 0;
20     for (Agendar agendamento : agendamentos) {
21         if (excluirAgendamento(agendamento.getIdPet(), agendamento.getIdServiço())) {
22             count++;
23         }
24     }
25
26     return count;
27 }
28
29 public boolean excluirAgendamento(int idPet, int idServiço) throws Exception {
30     // 1. Buscar ID do agendamento na B+
31     RegistroAgendamento chave = new RegistroAgendamento(idPet, idServiço, 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 Árvore 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 Serviço, atualizar chave na árvore B+
5     if (agendamentoExistente.getIdPet() != agendamento.getIdPet() ||
6         agendamentoExistente.getIdServiço() != agendamento.getIdServiço()) {
7
8         // 1. Validar nova chave
9         if (existeAgendamento(agendamento.getIdPet(), agendamento.getIdServiço())) {
10             throw new IllegalArgumentException(...);
11         }
12
13         // 2. Remover chave antiga da B+
14         RegistroAgendamento registroAntigo = new RegistroAgendamento(
15             agendamentoExistente.getIdPet(),
16             agendamentoExistente.getIdServiço(),
17             agendamento.getId())
18 }
```

```

18 );
19 indiceBMais.delete(registroAntigo);
20
21     // 3. Inserir nova chave na B+
22 RegistroAgendamento registroNovo = new RegistroAgendamento(
23     agendamento.getIdPet(),
24     agendamento.getIdServiço(),
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     diretório = new Diretório();
7     diretório.fromByteArray(bd);
8
9     // 2. Calcular hash e localizar cesto
10    int i = diretório.hash(elem.hashCode());
11    long endereçoCesto = diretório.endereco(i);
12
13    // 3. Carregar apenas o cesto necessário
14    Cesto c = new Cesto(construtor, quantidadeDadosPorCesto);
15    byte[] ba = new byte[c.size()];
16    arqCestos.seek(endereçoCesto);
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(endereçoCesto);
24        arqCestos.write(c.toByteArray()); // Grava apenas este cesto
25        return true;
26    }
27
28    // 5. Se cesto cheio, dividir
29    if (c.profundidadeLocal >= diretório.profundidadeGlobal) {
30        diretório.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(endereçoCesto);
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 = diretório.toByteArray();
50     arqDiretório.seek(0);
51     arqDiretório.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) arqDiretório.length()];
65     arqDiretório.seek(0);
66     arqDiretório.read(bd);
67     diretório = new Diretório();
68     diretório.fromByteArray(bd);
69
70     // 2. Calcular hash e localizar cesto
71     int i = diretório.hash(chave);
72     long endereçoCesto = diretório.endereço(i);
73
74     // 3. Carregar apenas o cesto necessário
75     Cesto c = new Cesto(construtor, quantidadeDadosPorCesto);
76     byte[] ba = new byte[c.size()];
77     arqCestos.seek(endereçoCesto);
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          Página pagina = new Página(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 Compressão de Dados

Os testes de compressão dos dados foram realizados após a implementação e otimização da criptografia **RSA**, na qual, por criptografar o campo e-mail, acaba aumentando moderadamente o consumo de espaço, que pode variar a cada execução.

Além disso, foram utilizados os dados pré-carregados da **bateria de testes** disponível na

aplicação. Na qual apresenta: **11** Clientes, **15** Pets, **15** Serviços e **26** Agendamentos, com um total de **6.397** bytes.

9.9.1 Compressão obtida com Huffman

- Tamanho do arquivo original: **6.397** bytes
- Tamanho do arquivo comprimido: **6.044** bytes
- Cálculo da taxa de compressão

Taxa = Tamanho Comprimido / Tamanho Original

$$\text{Taxa} = 6.044 / 6.397$$

Taxa = 0.9448

Percentual de redução: **5.52%**

Espaço economizado: **353** bytes

- Interpretação do resultado:

BAIXA: Compressão moderada.

O algoritmo HUFFMAN teve eficiência limitada.

9.9.2 Compressão obtida com LZW

- Tamanho do arquivo original: **6.397** bytes
- Tamanho do arquivo comprimido: **4.589** bytes
- Cálculo da taxa de compressão

$$\text{Taxa} = 4.584 / 6.397$$

Taxa = 0.7166

Percentual de redução: **28.34%**

Espaço economizado: **1.813** bytes

- Interpretação do resultado:

BOM: Compressão eficiente.

O algoritmo LZW conseguiu uma redução significativa.

9.9.3 Dificuldades na Implementação e Soluções

- Huffman

1º Problema: A tabela de códigos Huffman precisava ser armazenada junto com os dados comprimidos. Inicialmente, usando writeUTF() para strings, cada entrada consumia 10 bytes de overhead, resultando em expansão ao invés de compressão para arquivos pequenos.

Solução: Otimização da serialização:

- **writeShort()** para tamanho da tabela (2 bytes ao invés de 4)

- `writeByte()` para comprimento de cada código (1 byte)
- Representação em bits compacta dos códigos (ao invés de strings UTF)
- Redução de 10 bytes para 4 bytes por caractere na tabela

2º Problema: Huffman trabalha com caracteres, mas os dados são bytes (-128 a 127). A conversão entre `byte[]` e String poderia corromper dados binários.

Solução: Uso consistente da codificação ISO-8859-1 (1 byte = 1 caractere), preservando valores binários durante conversões.

3º Problema: Huffman comprime baseado em frequência de caracteres, mas arquivos .db têm distribuição uniforme de bytes.

Solução: Mantido no sistema com aviso de que LZW é mais adequado para este tipo de dado.

- LZW

1º Problema: Índices do dicionário precisavam ser armazenados de forma compacta (não como inteiros de 4 bytes).

Solução: Uso de `VetorDeBits` com `BITS_POR_INDICE = 12` (4096 entradas possíveis), permitindo armazenar cada índice em apenas 12 bits ao invés de 32.

2º Problema: Dicionário crescendo indefinidamente causaria problemas de memória e performance.

Solução: Limite de $2^{12} - 1 = 4095$ entradas, parando de adicionar novas sequências quando cheio.

3º Problema: Java não tem `unsigned byte`, valores variam de -128 a 127.

Solução: Inicialização do dicionário com loop de -128 a 127, e uso de `& 0xFF` ao ler tamanhos como `unsigned`.

9.9.4 Escolha das estruturas de dados

- Huffman
 - **HashMap<Character, Integer> hmapWC** (Contagem de Frequências)
 - Por quê:** Acesso O(1) para incrementar contadores durante varredura do texto
 - Alternativa rejeitada:** Array de 256 posições seria mais rápido, mas menos flexível para Unicode
 - **PriorityQueue<node>** (Min-Heap)
 - Por quê:** Construção da árvore requer sempre os dois nós de menor peso. `PriorityQueue` garante O(log n) para inserção/remoção
 - Alternativa rejeitada:** Ordenar `ArrayList` a cada iteração seria O(n log n) repetidamente
 - **HashMap<Character, String> hmapCode** (Tabela de Codificação)
 - Por quê:** Durante codificação, precisamos buscar o código de cada caractere em O(1)
 - Alternativa rejeitada:** Array seria limitado ao range de caracteres

- **HashMap<String, Character> hmapCodeR** (Tabela de Decodificação Reversa)

Por quê: Durante decodificação, lemos bits e precisamos encontrar rapidamente qual caractere corresponde

Alternativa rejeitada: Busca linear seria $O(n)$ por caractere decodificado
- Classe interna **node** (Nós da Árvore)

Por quê: Encapsulamento - nós são específicos do Huffman, com campos weight, ch, left, right

Design: Comparator customizado permite PriorityQueue ordenar por peso
- LZW
 - **ArrayList<ArrayList<Byte>> dicionario** (Dicionário de Sequências)

Por quê:

 - * Índices numéricos sequenciais mapeiam diretamente para posições no ArrayList (acesso $O(1)$)
 - * indexOf() permite busca de sequências existentes
 - * Crescimento dinâmico até o limite

Alternativa rejeitada: **HashMap<ArrayList<Byte>, Integer>** teria overhead de hashing
 - **ArrayList<Integer> saida** (Índices Comprimidos)

Por quê: Tamanho desconhecido durante compressão, ArrayList cresce dinamicamente

Conversão: Posteriormente convertido para VetorDeBits para compactação
 - **VetorDeBits** (Armazenamento Compacto)

Por quê: Baseado em BitSet, permite manipulação bit-a-bit eficiente

Vantagem: 12 bits por índice ao invés de 32 (economia de 62.5)

Métodos: set(), clear(), get() abstraem manipulação de bits
 - **ArrayList<Byte>** como Elemento do Dicionário

Por quê: Sequências têm tamanho variável, ArrayList é flexível

Clone importante: clone() evita que modificações afetem entradas do dicionário

9.10 Criptografia

9.10.1 Campo escolhido

O campo escolhido para ser criptografado foi o **e-mail** do **Cliente**, por se tratar de uma informação consideravelmente sensível, já que o PetCare Manager não possui campos que exigem segurança extrema como senhas ou cartões de débito/crédito.

9.10.2 Implementação do RSA

- Estrutura das chaves pública e privada Chave pública

Classe: RSA.ChavePublica (em RSA.java)

Campos: BigInteger e (expoente público) e BigInteger n (módulo).

Representa (e, n) .

Chave privada

Classe: RSA.ChavePrivada (em RSA.java)

Campos: BigInteger d (expoente privado) e BigInteger n (mesmo módulo).

Representa (d, n) .

Observação importante sobre geração em RSA.gerarChaves():

A implementação gera dois primos p e q (usando BigInteger.probablePrime), calcula $n = p * q$ e $z = (p-1)*(q-1)$. Diferente do fluxo clássico, o código escolhe primeiro um d tal que $\text{gcd}(d, z) = 1$ e depois calcula $e = d.\text{modInverse}(z)$ (funcionalmente correto — e é o inverso modular de d módulo z)

- Como e onde as chaves foram armazenadas

Arquivos no repositório de dados:

Chave pública: rsa_public.key

Chave privada: rsa_private.key

Formato de gravação (RSAKeyManager.salvarChaves()):

Abre DataOutputStream e grava os comprimentos e os bytes das componentes:

Para a pública: escreve int com tamanho de e.toByteArray() seguido dos bytes de e, depois int com tamanho de n.toByteArray() e os bytes de n.

Para a privada: análogo com d e n.

Ou seja: arquivos binários com [tamanho][bytes] para cada peça (não texto legível).

- Como foram carregadas pelo sistema

RSAKeyManager.getInstance() (singleton em RSAKeyManager.java) é o ponto de carregamento:

No construtor privado é chamado carregarOuGerarChaves():

Se os arquivos rsa_public.key e rsa_private.key existem, chama carregarChaves() que usa DataInputStream para ler os comprimentos e reconstruir BigInteger e/n e BigInteger d/n. Se não existirem, chama gerarNovasChaves() que instancia new RSA(TAMANHO_CHAVE) e depois salvarChaves().

Uso por outras partes do sistema:

Qualquer código que chame RSAKeyManager.getInstance() obtém as chaves já carregadas (ou geradas na primeira invocação). Cliente.setEmail(...) e Cliente.fromByteArray(...) chamam esse singleton para criptografar/descriptografar

- Tamanho das chaves escolhidas e justificativa Valor em RSAKeyManager: TAMANHO_CHAVE = 512 bits (atualizado para reduzir uso de espaço).

Contexto e justificativa:

Inicialmente foi usado 1024 bits, depois reduzido para 512 bits para diminuir o espaço ocupado pelo cifrado (trade-off entre segurança e armazenamento). O código exibe a capacidade por

bloco calculada: $((n.\text{bitLength}() / 8) - 11)$ → por exemplo, com 512 bits o bloco útil ficou em 53 bytes.

- Em qual momento a criptografia do(s) campo(s) ocorre (no CRUD). Ponto de execução: ao atribuir o email no objeto Cliente em memória:

Cliente.setEmail(String email) (arquivo Cliente.java) — ao definir o email, o setter chama:
this.email = email;

```
this.emailCriptografado = RSAKeyManager.getInstance().criptografar(email);
```

Resumo: a criptografia é feita imediatamente no momento da atribuição/edição do campo email no objeto Cliente, antes de qualquer operação de persistência.

Integração com operações CRUD:

ClienteDAO.incluirCliente(...) chama arqClientes.create(cliente). O objeto cliente já tem emailCriptografado preenchido pelo setter; no toByteArray() o campo salvo é o emailCriptografado.

alterarCliente(...) funciona com a instância em memória (que contém o email em texto claro e o emailCriptografado atualizado se setEmail foi chamado).

Portanto: a criptografia é realizada no nível do modelo (no setter) — antes da gravação no arquivo.

- Em qual momento ocorre a descriptografia.

Ponto de execução: na leitura do registro do disco para memória:

Cliente.fromByteArray(byte[] b) (em Cliente.java) lê do streaming o campo de email armazenado (o emailCriptografado) e então chama: this.email = RSAKeyManager.getInstance().descriptografar(emailCriptografado);

Assim, assim que o objeto Cliente é criado a partir dos bytes do arquivo, o email armazenado (criptografado) é traduzido de volta para o texto claro e mantido em this.email para uso pelo sistema.

Fallbacks:

Se a descriptografia falhar, o código faz fallback retornando o texto lido (para compatibilidade com dados antigos) e loga erro.

- Conversões realizadas (ex.: string → bytes → blocos).

Conversão geral (fluxo de criptografia em RSAKeyManager.criptografar):

```
String do email → bytes via texto.getBytes("UTF-8").
```

Determina tamanhoMaximoBloco = (n.bitLength() / 8) - 11 (capacidade por bloco, cálculo conservador considerando padding).

Se bytesOriginais.length <= tamanhoMaximoBloco:

Cria BigInteger m = new BigInteger(1, bytes) e cifra com modPow(e, n) (através de RSA.cifrarComChavePublica ou via modPow direto).

Resultado: BigInteger c. Converte c.toByteArray() → representa os bytes cifrados. Esses bytes cifrados são então codificados em hexadecimal (bytesToHex(bytesCifrados)) e retornados como string.

Se maior que o bloco: Divide bytesOriginais em blocos de até tamanhoMaximoBloco. Para cada bloco:

BigInteger blocoCifrado = new BigInteger(1, bloco).modPow(e, n);

Converte blocoCifrado.toByteArray() → bytes cifrados → bytesToHex(...).

Concatena blocos como strings hex separadas por | (barra vertical) para formar o valor final a ser persistido (ex.: hexBloco1|hexBloco2|hexBloco3).

Conversão geral (fluxo de descriptografia em RSAKeyManager.descriptografar):

Recebe string armazenada (hex ou |-separada).

Se contém |:

split("

|") → para cada bloco hex:

hexToBytes(blocoHex) → bytes cifrados.

BigInteger blocoCifrado = new BigInteger(bytesCifrados) → blocoDecifrado = blocoCifrado.modPow(d, n).

blocoDecifrado.toByteArray() → bytes do bloco original → new String(bytes, "UTF-8") → append ao texto final.

Se não contém |:

Trata como bloco único: hexToBytes(...), new BigInteger(...) e usa RSA.decifrarParaString ComChavePrivada.

Em caso de falha no formato hex, há tentativa de fallback para o formato Base64 antigo (compatibilidade).

9.11 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.

9.11.1 Pasta raiz

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

9.11.2 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
```

9.11.3 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
└── servicios/
    ├── servicios.db
    └── servicios.idx
└── agendamentos/
    ├── agendamentos.db
    ├── agendamentos.idx
    └── agendamentos_bmais.db
```