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

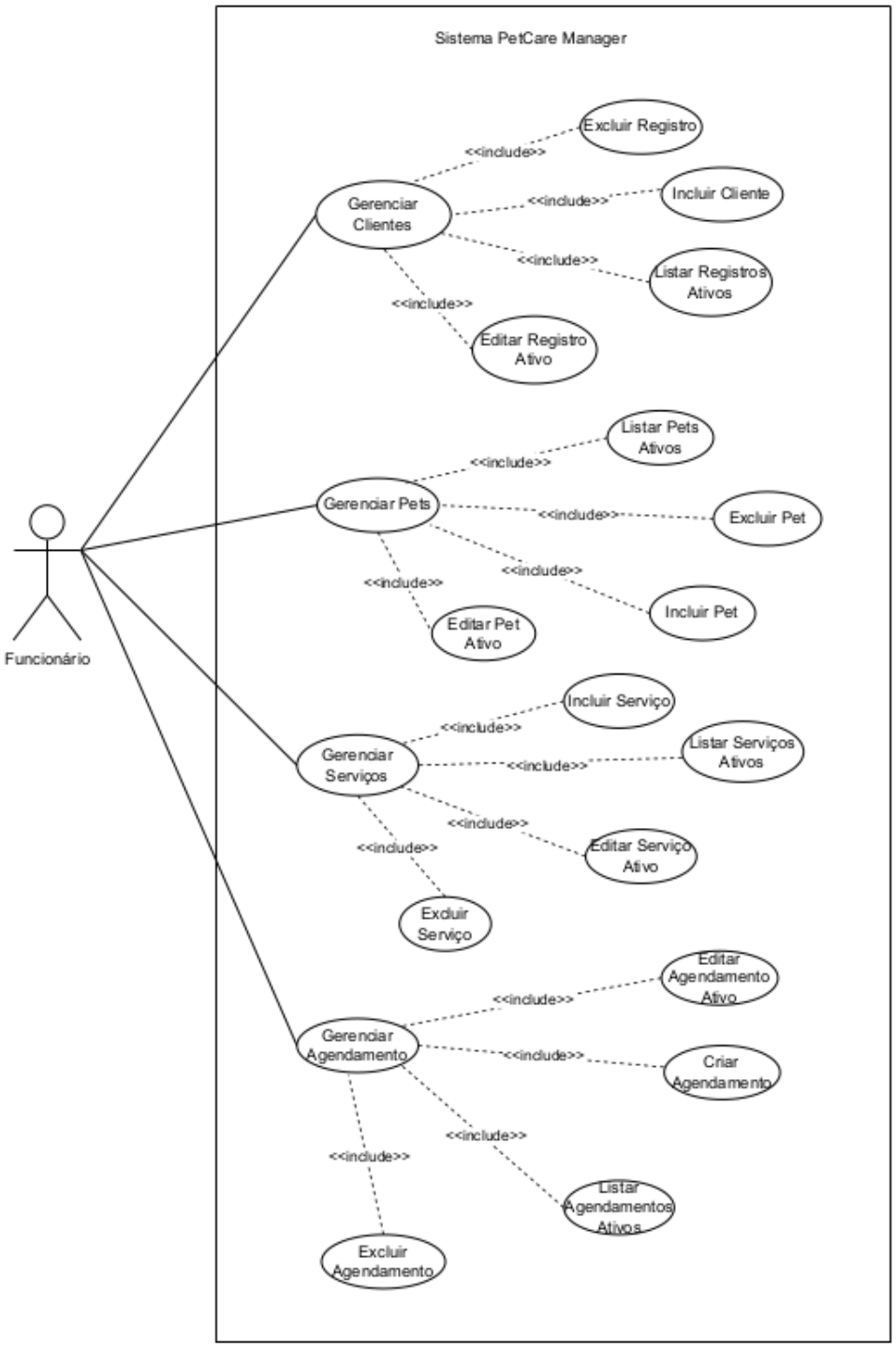

Figura 1: DCU: PCM

7 Diagrama Entidade-Relacionamento

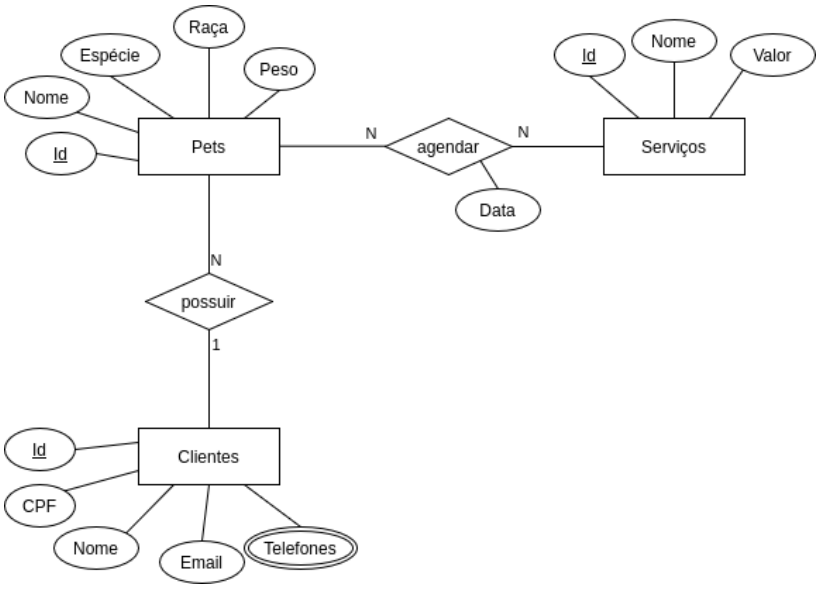

Figura 2: DER: PCM

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 Estrutura para representar os registros

9.1 Cabeçalho

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

9.2 Exclusão lógica

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

9.3 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

9.4 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)

9.5 Serviço

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

9.6 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)

9.7 Interface comum

```
public interface Registro {
    void setId(int i);
    int getId();
    byte[] toByteArray() throws IOException;
    void fromByteArray(byte[] b) throws IOException;
}
```

10 Tratamento das Strings multivaloradas

O atributo multivalorado **telefones** é tratado como um array de Strings na classe **Cliente**. Durante a serialização para persistência no arquivo, o método **toByteArray()** primeiro grava o tamanho do array como 1 byte, seguido pela gravação sequencial de cada número de telefone usando **writeUTF()** que armazena strings com seu comprimento prefixado. Na deserialização através do método **fromByteArray()**, o processo inverso ocorre: primeiro é lido o byte que indica quantos telefones existem, então um array de strings é alocado com esse tamanho e cada posição é preenchida lendo sequencialmente as strings UTF do *stream* de bytes. Essa abordagem permite armazenar um número variável de telefones por cliente de forma eficiente, mantendo a estrutura compacta no arquivo binário e facilitando a manipulação dos telefones como uma coleção em memória. Nos formulários JavaFx, os telefones são concatenados com vírgula para exibição em tabelas e podem

ser editados através de campos de texto separados que são convertidos de/para o array durante as operações CRUD.

11 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.

12 Chaves utilizadas

12.1 Indíce 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.

12.2 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.

12.3 CPF do Dono - Chave Estrangeira com Hash Extensível

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.

12.4 Chave Composta (idPet, idServico) com Árvore B+

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.

13 Implementação do 1:N

O relacionamento 1:N entre **Cliente** e **Pet** foi implementado usando o **CPF** do cliente como chave estrangeira no registro de pet, permitindo que um cliente possua múltiplos pets. A navegação entre registros é gerenciada através de uma estrutura de **Hash Extensível** (IndiceHashExtensivel) que mapeia cada CPF para uma lista de **IDs** de pets, possibilitando busca eficiente dos pets de um determinado **dono** através do método **buscarIdsPetsPorCpf()**. A integridade referencial

é garantida através de validações no **ClienteDAO** e **PetDAO**: ao cadastrar um pet, o sistema verifica se o CPF do dono existe na base de clientes. Ao excluir um cliente, o sistema implementa exclusão em cascata, primeiro buscando todos os IDs de pets associados ao CPF através do **índice hash**, excluindo cada pet individualmente via **PetDAO.excluirPet()**, e depois removendo todos os relacionamentos do índice hash antes de finalmente excluir o cliente. Essa abordagem mantém a consistência dos dados impedindo pets órfãos (sem dono cadastrado) e garantindo que a remoção de um cliente limpe automaticamente todos os seus pets e relacionamentos associados, preservando a integridade do banco de dados em todas as operações CRUD.

13.1 Implementação do N:N

O relacionamento N:N entre **Pet** e **Serviço** foi implementado através da entidade intermediária **Agendar**, que armazena os IDs de ambas as entidades relacionadas junto com informações adicionais como data, horário e status do agendamento. A navegação entre registros é facilitada pelo **AgendarDAO** que oferece métodos para buscar agendamentos por ID de pet, por ID de serviço, ou listar todos os agendamentos, permitindo navegar bidireccionalmente entre pets e serviços através dos agendamentos intermediários. A integridade referencial é mantida através de validações rigorosas: ao criar um agendamento, o sistema verifica se tanto o pet quanto o serviço existem na base de dados através de consultas aos respectivos DAOs, impedindo a criação de relacionamentos com entidades inexistentes. Ao excluir um pet, o sistema primeiro remove todos os agendamentos associados a ele através de **excluirAgendamentosPorPet()** antes de excluir o registro do pet, e similarmente ao excluir um serviço, todos os agendamentos vinculados são removidos primeiro. A tabela intermediária se integra com o CRUD das entidades principais através do AgendarController que gerencia a criação de agendamentos validando a existência de pets e serviços, permite alterações de data/horário/status mantendo os IDs de relacionamento intactos, e implementa exclusão em cascata garantindo que a remoção de qualquer entidade principal (pet ou serviço) não deixe registros órfãos na tabela de agendamentos, preservando assim a consistência do banco de dados em todas as operações.

14 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.

14.1 Índice Sequencial

O índice sequencial é implementado como um arquivo binário simples (.idx) que armazena pares ordenados de ID e endereço físico no arquivo de dados, mantido em memória através de um **ArrayList<RegistroIndice>** ordenado. A persistência ocorre através de **RandomAccessFile** onde cada entrada é gravada sequencialmente como um **int** de 4 bytes (ID) seguido por um **long** de 8 bytes (endereço), totalizando 12 bytes por registro de índice. A atualização é realizada através dos métodos **inserir(), atualizar()** e **remover()** da classe **IndiceSequencial**, que modificam tanto a lista em memória (mantendo-a ordenada via **Collections.sort()**) quanto regrava todo o arquivo de índice do início ao fim para refletir as mudanças. A sincronização com os dados principais ocorre automaticamente através da classe Arquivo que encapsula todas as operações CRUD: ao

criar um registro, o método **create()** primeiro grava os dados no arquivo principal e então chama **indice.inserir** (id, endereco) passando o ID e a posição física onde o registro foi gravado. Ao atualizar, verifica se o tamanho mudou e, caso positivo, marca o registro antigo como excluído e cria um novo, atualizando o índice com o novo endereço. Ao excluir, marca o registro como **lápide** no arquivo de dados e remove a entrada correspondente do índice. Este índice sequencial é usado por todas as entidades principais, através da classe **Arquivo**, permitindo busca binária eficiente por ID ao invés de varredura sequencial completa do arquivo de dados.

14.2 Índice Hash Extensível

O índice direto é implementado através de **Hash Extensível** persistido em dois arquivos distintos: um arquivo de diretório (_hash.dir) que armazena os ponteiros para os *buckets* e a profundidade global do hash, e múltiplos arquivos de buckets que contêm os pares chave-valor reais. Cada bucket é serializado através da interface **RegistroHashExtensivel** que converte os dados para formato binário usando **DataOutputStream**, armazenando a profundidade local, o número de elementos e os registros sequencialmente. A atualização é gerenciada pela classe **HashExtensivel** que, ao detectar *overflow* de um bucket durante inserção, realiza operações de *split* aumentando a profundidade local ou global conforme necessário e redistribuindo os elementos entre buckets antigos e novos, com todas as mudanças sendo imediatamente gravadas nos arquivos através de **RandomAccessFile**. A sincronização com os dados principais ocorre através do **IndiceHashExtensivel** usado no relacionamento **Pet-Dono**, onde cada operação no **PetDAO** dispara atualizações correspondentes no hash extensível, inserindo, modificando ou removendo entradas que mapeiam CPFs de donos para listas de IDs de pets, mantendo a consistência entre o arquivo de dados e o índice hash através de operações transacionais que garantem atomicidade.

14.3 Índice Árvore B+

A **Árvore B+** é utilizada especificamente para indexação secundária na entidade **Agendamento**, permitindo buscas eficientes por chaves compostas. A estrutura é persistida em arquivo binário onde o cabeçalho armazena metadados como ordem da árvore e ponteiro para a raiz, seguido pela serialização de cada nó através da interface **RegistroArvoreBMais** que define **toByteArray()** e **fromByteArray()**. A atualização ocorre através de operações de inserção, busca e remoção que navegam pela árvore desde a raiz até as folhas, executando splits quando um nó excede sua capacidade ou *merges* quando fica abaixo do mínimo, com cada modificação estrutural sendo imediatamente persistida através de **RandomAccessFile** que reescreve apenas os nós afetados em suas posições calculadas no arquivo. A sincronização com os dados principais acontece no **AgendarDAO** que mantém uma instância de **ArvoreBMais<RegistroAgendamento>**, interceptando todas as operações CRUD de agendamentos: ao criar um agendamento, após gravar no arquivo de dados principal através do **Arquivo**, o DAO insere a entrada correspondente na árvore B+ com a chave composta e o endereço do registro. Ao excluir ou alterar, remove ou atualiza as entradas na árvore antes de modificar os dados, garantindo que os índices sempre reflitam o estado atual e permitam consultas complexas sem necessidade de varredura completa do arquivo.

15 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.

15.1 Compressão obtida com Huffman

- Tamanho do arquivo original: **6.397** bytes
- Tamanho do arquivo comprimido: **6.044** bytes

  Taxa = Tamanho Comprimido / Tamanho Original

  Taxa = 6.044 / 6.397

  Taxa = **0.9448**

  Percentual de redução: **5.52%**

  Espaço economizado: **353** bytes
- O algoritmo **HUFFMAN** teve eficiência limitada.

15.2 Compressão obtida com LZW

- Tamanho do arquivo original: **6.397** bytes
- Tamanho do arquivo comprimido: **4.589** bytes

  Taxa = 4.584 / 6.397

  Taxa = **0.7166**

  Percentual de redução: **28.34%**

  Espaço economizado: **1.813** bytes
- O algoritmo **LZW** conseguiu uma redução significativa.

15.3 Dificuldades na implementação, soluções e estruturas de dados

15.3.1 Huffman

A principal dificuldade na implementação do Huffman foi o *overhead* significativo da tabela de códigos que precisava ser armazenada junto com os dados comprimidos, resultando inicialmente em expansão ao invés de compressão para arquivos pequenos ou com distribuição uniforme de caracteres. Isso foi resolvido através de múltiplas otimizações na serialização: substituindo **writeUTF()** por formatos mais compactos usando **writeShort()** para o tamanho da tabela, **writeByte()** para o comprimento de cada código, e representação *bit-packed* dos códigos ao invés de strings UTF, reduzindo o overhead de aproximadamente 10 bytes para 4 bytes por caractere na tabela. Outra dificuldade foi a conversão entre bytes e caracteres sem corrupção de dados binários, resolvida

usando consistentemente a codificação **ISO-8859-1** que garante mapeamento 1:1 entre bytes e caracteres. Para as estruturas de dados, foi escolhido **HashMap<Character, Integer>** para contagem de frequências devido ao acesso O(1) para incrementar contadores durante a varredura do texto. **PriorityQueue<node>** como *min-heap* para construção eficiente da árvore, garantindo acesso O(log n) aos dois nós de menor peso necessários a cada iteração. **HashMap<Character, String>** para a tabela de codificação permitindo busca O(1) do código de cada caractere durante a compressão. Por fim, **HashMap<String, Character>** para decodificação reversa com busca O(1) do caractere correspondente a cada sequência de bits. A árvore de Huffman propriamente dita foi implementada usando uma classe interna **node** com campos para peso, caractere, e ponteiros left/right, encapsulando a estrutura específica do algoritmo e permitindo travessia recursiva para geração dos códigos.

15.3.2 LZW

A maior dificuldade no LZW foi representar eficientemente os índices do dicionário sem desperdiçar espaço, já que armazená-los como inteiros de 32 bits resultaria em expansão ao invés de compressão. Isso foi resolvido utilizando a classe **VetorDeBits** baseada em **BitSet** para armazenar cada índice com exatamente 12 bits, permitindo até 4096 entradas no dicionário e economizando **62.5%** de espaço comparado a inteiros completos. Outra dificuldade foi o gerenciamento do crescimento do dicionário, resolvido implementando um limite de 2^{12-1} entradas onde o dicionário para de crescer quando cheio, evitando problemas de memória e degradação de performance. O tratamento de bytes com sinal exigiu cuidados especiais, resolvidos inicializando o dicionário com um loop de -128 a 127 e usando máscaras **&0xFF** ao ler tamanhos como *unsigned*. Para as estruturas de dados, foi escolhido **ArrayList<ArrayList<Byte»** para o dicionário principal porque os índices numéricos sequenciais mapeiam diretamente para posições no **ArrayList** com acesso O(1), e o método **indexOf()** permite busca de sequências existentes. **ArrayList<Integer>** para armazenar a saída de índices durante a compressão devido ao tamanho desconhecido e crescimento dinâmico eficiente. e **ArrayList<Byte>** como elemento individual do dicionário para representar sequências de tamanho variável, com uso cuidadoso de **clone()** ao manipular entradas para evitar que modificações afetem acidentalmente o dicionário. O uso de **ArrayList** indexado ao invés de **HashMap** foi crucial para LZW porque os índices são atribuídos sequencialmente, tornando o acesso por posição muito mais eficiente que hashing.

16 Criptografia

16.1 Campo escolhido

O campo escolhido para ser criptografado foi o **email** 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 informações de cartões de débito/crédito.

16.2 Implementação do RSA

O RSA foi implementado no projeto através da classe **RSA.java** que define duas classes internas: **ChavePublica** encapsulando o par (e, n) onde 'e' é o expoente público e 'n' é o módulo,

e **ChavePrivada** encapsulando o par (d, n) onde 'd' é o expoente privado e 'n' é o mesmo módulo, ambos usando **BigInteger** para suportar números grandes. As chaves são armazenadas em arquivos binários no diretório dados como **rsa_public.key** e **rsa_private.key**, onde cada componente é serializada gravando primeiro o tamanho em bytes seguido dos bytes do **BigInteger**, usando **DataOutputStream** para escrita e **DataInputStream** para leitura. O carregamento é gerenciado pela classe *singleton* **RSAKeyManager** que, ao ser instanciado pela primeira vez através de **getInstance()**, verifica se os arquivos de chave existem: caso existam, carrega-os reconstruindo os **BigInteger** a partir dos bytes lidos. Caso contrário, gera um novo par de chaves chamando **new RSA(TAMANHO_CHAVE)** e persiste-as nos arquivos. O tamanho das chaves foi definido como 512 bits (reduzido de 1024 bits) para minimizar o impacto no espaço de armazenamento dos dados criptografados, resultando em blocos de aproximadamente 53 bytes úteis por operação, sendo esta uma escolha de compromisso entre segurança e eficiência de espaço. A criptografia ocorre no momento da atribuição do valor ao campo **email** através do método **setEmail()** da classe **Cliente**, onde imediatamente após armazenar o **email** em texto claro na variável **this.email**, o setter chama **RSAKeyManager.getInstance().criptografar(email)** e armazena o resultado criptografado em **this.emailCriptografado**, garantindo que o valor já esteja cifrado antes de qualquer operação de persistência através do CRUD. A descriptografia acontece durante a leitura do registro do arquivo no método **fromByteArray()** da classe **Cliente**, onde após ler o campo **emailCriptografado** do stream de bytes, o sistema imediatamente chama **RSAKeyManager.getInstance().descriptografar(emailCriptografado)** para recuperar o texto claro e armazená-lo em **this.email**, disponibilizando-o para uso imediato pela aplicação. As conversões seguem o fluxo: o email em string é convertido para bytes UTF-8, esses bytes são divididos em blocos de até 53 bytes, cada bloco é convertido para **BigInteger**, cifrado usando exponenciação modular m^e mod n, o resultado é convertido para array de bytes, esses bytes são codificados em hexadecimal (2 caracteres por byte para compactação), e múltiplos blocos são concatenados com separador |, sendo esse formato hexadecimal final o que é persistido no arquivo através de **writeUTF()** no método **toByteArray()**, com o processo inverso ocorrendo na descriptografia onde o hexadecimal é decodificado para bytes, convertido para **BigInteger**, decifrado usando c^d mod n, e os bytes resultantes são convertidos de volta para string UTF-8.

17 Casamento de Padrões

17.1 Campo escolhido

O campo escolhido foi **nome** do **Cliente**, pois em relação aos outros atributos do sistema, a busca por nomes de pessoas é o mais comum, por exemplo para saber quantos registros possuem o nome Ana ou Pedro ou pesquisar rapidamente um nome sem ter que digitá-lo por completo.

17.2 KMP (Knuth-Morris-Pratt)

O algoritmo KMP foi implementado na classe **Kmp.java** através de dois métodos principais: **computeLPSArray()** que constrói a tabela de prefixos mais longos (LPS) armazenando em um **ArrayList<Integer>** os valores de *fallback* para cada posição do padrão, permitindo evitar comparações redundantes ao detectar desalinhamentos. E **contains()** que executa a busca propriamente dita mantendo dois índices (i para o texto e j para o padrão) e, ao encontrar um *mismatch*, consulta a tabela LPS para determinar quantas posições retroceder no padrão sem retroceder no

texto, resultando em complexidade O(n+m). A principal dificuldade na implementação foi garantir o correto cálculo da tabela LPS, especialmente nos casos onde o prefixo atual não casa e é necessário "cair de volta"para um prefixo menor através de **len = lps.get(len - 1)**, exigindo entendimento profundo da estrutura de prefixos-sufixos do padrão. Outra dificuldade foi o gerenciamento cuidadoso dos índices para evitar acessos fora dos limites do array e loops infinitos, particularmente no tratamento do caso onde **len == 0** e é necessário simplesmente avançar para o próximo caractere. A integração ao sistema foi feita através do **BuscaPadraoController** que chama **Kmp.contains(nomeLower, padraoLower)** para cada cliente, implementando busca *case insensitive* ao converter tanto o nome quanto o padrão para *lowercase* antes da comparação, permitindo que "maria", "Maria"e "MARIA"retornem os mesmos resultados sem modificar o algoritmo base.

17.3 Boyer-Moore (Bad Character)

O algoritmo Boyer-Moore foi implementado na classe **BoyerM.java** utilizando apenas a heurística *bad character*, através dos métodos **badCharHeuristic()** que pré-processa o padrão construindo um array de 256 posições armazenando a última ocorrência de cada caractere no padrão, e **contains()** que executa a busca da direita para a esquerda comparando o padrão com o texto de trás para frente, e ao encontrar um mismatch, usa a tabela bad character para calcular quanto pode "pular"no texto através da fórmula **s += max(1, j - badchar[txt[s + j]])**, evitando comparações desnecessárias e potencialmente alcançando complexidade sublinear em textos favoráveis. A principal dificuldade foi compreender e implementar corretamente o cálculo do *shift*, especialmente garantir que o deslocamento seja sempre positivo através da função **max(1, ...)** para evitar que shifts negativos causem loops infinitos ou retrocessos indevidos. Outra dificuldade foi o tratamento do limite do array ao acessar **texto[s + j]** para consultar a tabela bad character, exigindo verificações cuidadosas para não ultrapassar o tamanho do texto. A integração seguiu o mesmo padrão do KMP através do **BuscaPadraoController**, porém convertendo as strings para arrays de caracteres pois o algoritmo opera sobre chars, e chamando **BoyerM.contains(txtArray, patArray)** para cada cliente após converter nome e padrão para lowercase, mantendo a busca case insensitive. Uma dificuldade adicional na integração foi evitar reimplementar os algoritmos no **Controller**, sendo resolvido adicionando métodos **contains()** públicos nas classes de algoritmo que encapsulam toda a lógica de busca e retornam **boolean**, permitindo que o controller simplesmente invoque esses métodos prontos ao invés de duplicar código, resultando em implementação limpa onde o controller apenas itera sobre os clientes e delega a verificação de padrão para os algoritmos especializados.

18 Estrutura do projeto no GitHub

O projeto segue a estrutura padrão **Maven** com o diretório **src** contendo todo o código-fonte e recursos, **target** (está no **.gitignore**) armazenando os arquivos compilados **.class** e build artifacts gerados durante a compilação, **docs** guardando documentação do projeto incluindo diagramas de entidade-relacionamento, instruções contendo especificações e requisitos do trabalho prático, além de arquivos de configuração na raiz como **pom.xml**, **README.md** (instruções de execução) e scripts de execução como **run.sh** para facilitar o *start* da aplicação.