Trabalho Prático - Fase II

Bernardo Bicalho, Bernardo Pires, Eduardo Luttembarck AEDs III

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

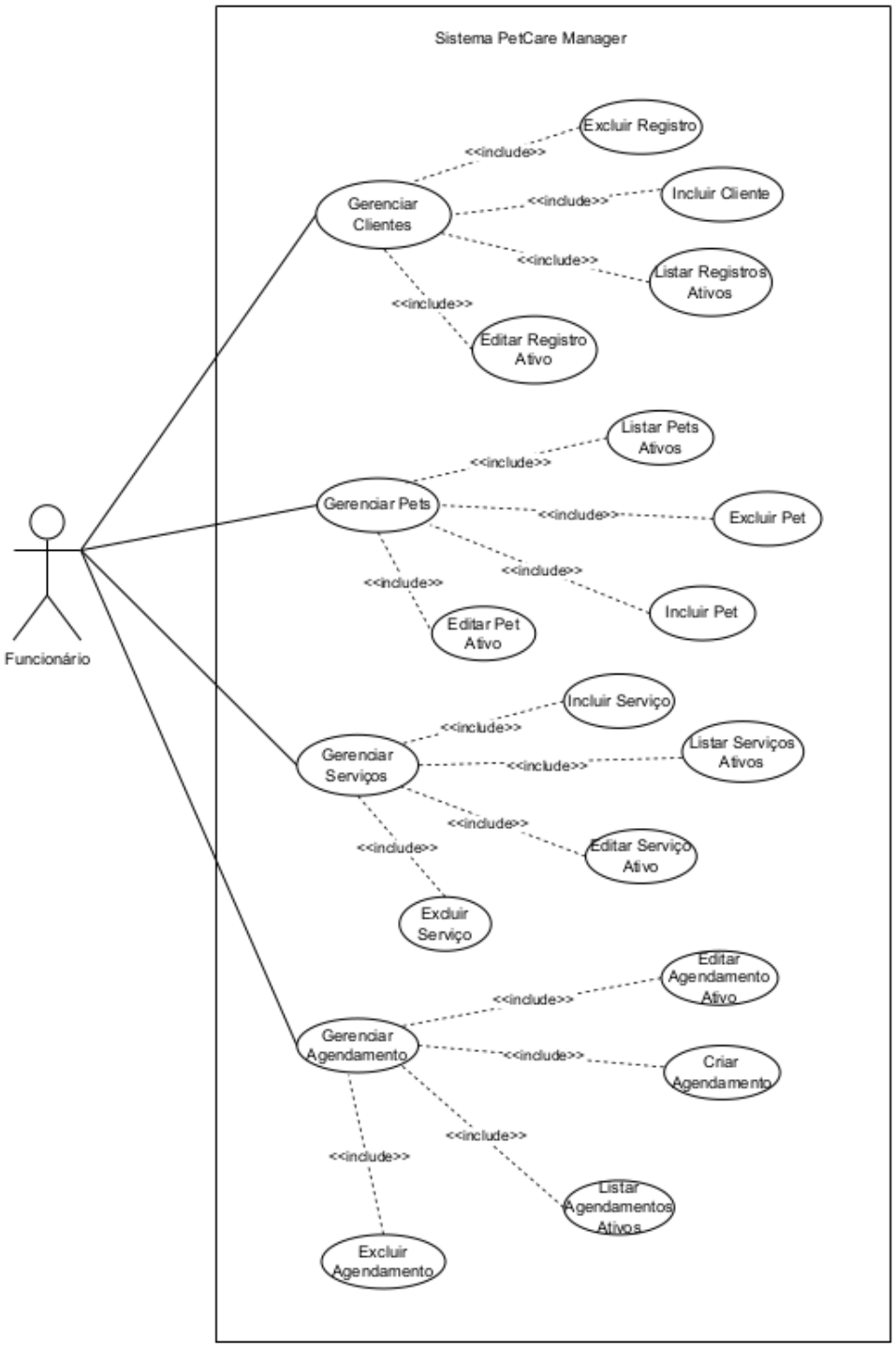

Figura 1: DCU: PCM

7 Diagrama Entidade-Relacionamento

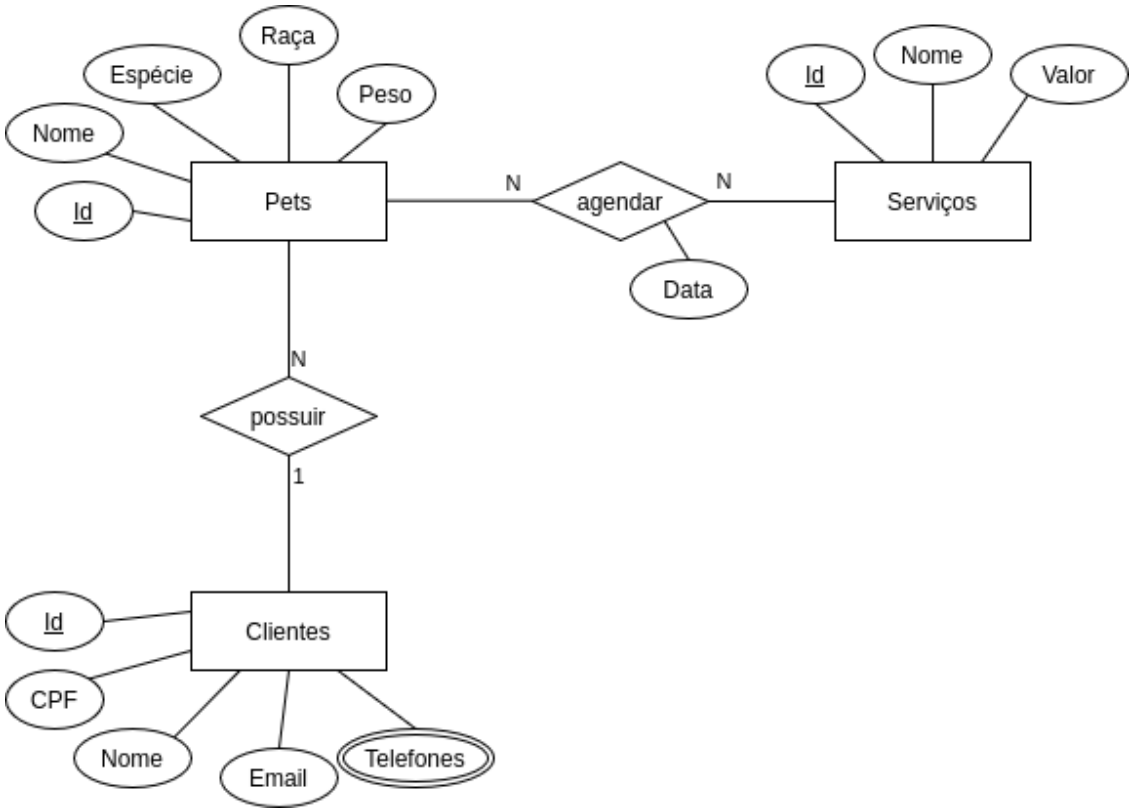

Figura 2: DER: PCM

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

A estrutura dos registros no sistema foi padronizada através da interface `Registro.java`. Ela funciona como um contrato que obriga todas as classes de modelo (como `Cliente`, `Pet`, etc.) a implementarem métodos essenciais para a sua manipulação e persistência em arquivos binários.

O contrato definido pela interface `Registro` exige os seguintes métodos:

- `getId()` e `setId(int id)`: Fornecem uma maneira padrão de acessar e definir a **chave primária** de qualquer registro, fundamental para a indexação.
- `toByteArray()`: Método responsável pela **serialização**. Ele converte o estado de um objeto (seus atributos) em um array de bytes (`byte[]`), formato necessário para ser gravado em disco.
- `fromByteArray(byte[] b)`: Método responsável pela **desserialização**. Ele reconstrói um objeto a partir de um array de bytes lido do arquivo.

A classe `Cliente.java` serve como um exemplo concreto dessa implementação. O método `toByteArray()` define a estrutura exata do registro de cliente no arquivo binário, gravando os campos em uma ordem fixa e com tipos de dados bem definidos, utilizando `DataOutputStream`.

Listing 1: Estrutura do registro Cliente no método `toByteArray()`

```
public byte[] toByteArray() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);

    // Estrutura do registro no arquivo:
    dos.writeInt(getId());          // 4 bytes para o ID
    dos.writeUTF(getCpf());         // String de tamanho vari vel
    dos.writeUTF(getNome());        // String de tamanho vari vel
    dos.writeUTF(getEmail());       // String de tamanho vari vel

    dos.writeInt(getTelefones().length); // 4 bytes para a qtd. de telefones
    for (String telefone : this.telefones) {
        dos.writeUTF(telefone);     // Strings para cada telefone
    }

    return baos.toByteArray();
}
```

Essa abordagem garante que a manipulação de registros seja genérica (através da classe `Arquivo.java`, que opera sobre qualquer `Registro`) e consistente, pois a responsabilidade de como ser escrito e lido do disco pertence à própria classe de modelo.

9.2 Tratamento das Strings multivaloradas

O tratamento de atributos multivalorados, como a lista de telefones do `Cliente`, foi implementado utilizando a técnica de **prefixo de tamanho**. Esta abordagem permite armazenar um número variável de strings de forma eficiente dentro de um único registro.

O processo de serialização (escrita) e desserialização (leitura) ocorre da seguinte forma:

- **Na escrita (`toByteArray`)** Primeiramente, é gravado um número inteiro (`int`) que representa a quantidade total de strings no array de telefones. Em seguida, o sistema entra em um laço e grava cada string individualmente.
- **Na leitura (`fromByteArray`)** O processo inverso é executado: o sistema primeiro lê o inteiro que indica a quantidade de telefones, cria um array de strings com esse tamanho exato e, então, executa um laço para ler cada uma das strings do arquivo.

O código abaixo, extraído da classe `Cliente.java`, ilustra a implementação dessa técnica.

Listing 2: Serialização e Desserialização do campo multivalorado `telefones`

```
// Trecho do metodo toByteArray()
dos.writeInt(getTelefones().length);
for (String telefone : this.telefones) {
    dos.writeUTF(telefone);
}

// Trecho do metodo fromByteArray()
int telefonesLength = dis.readInt();
String[] telefones = new String[telefonesLength];
for (int i = 0; i < telefonesLength; i++) {
    telefones[i] = dis.readUTF();
}
setTelefones(telefones);
```

Essa abordagem é robusta e otimiza o uso do disco, pois o espaço alocado para os telefones é exatamente o necessário para cada registro, sem desperdício com espaços fixos pré-alocados.

9.3 Exclusão lógica

A exclusão de registros no sistema é puramente **lógica**, o que significa que os dados não são fisicamente removidos do arquivo. Em vez disso, eles são marcados como inválidos através da técnica da **lápide** (tombstone), e o espaço que ocupavam é gerenciado para futuro reaproveitamento.

A estrutura de cada registro no arquivo de dados é precedida por um `byte` de controle (a lápide). Um registro válido é identificado por um caractere de espaço (`' '`), enquanto um registro logicamente excluído é marcado com um asterisco (`'*'`).

O processo de exclusão, implementado no método `delete(int id)` da classe `Arquivo.java`, segue os seguintes passos:

1. O endereço do registro a ser excluído é localizado através do índice da chave primária.
2. O sistema verifica se o registro já não está marcado como excluído.
3. O `byte` da lápide, no início do registro, é sobrescrito com o caractere `'*'`.
4. O espaço liberado é adicionado a uma lista encadeada de espaços livres para reaproveitamento (gerenciado pelo método `addDeleted`).
5. A entrada correspondente ao `id` do registro é removida do índice, garantindo que o dado não seja mais encontrado em buscas futuras.

O código abaixo ilustra a implementação do método de exclusão.

Listing 3: Implementação da exclusão lógica em `Arquivo.java`

```
public boolean delete(int id) throws Exception {
    // Buscar endereco no indice
    long endereco = indice.buscar(id);
    if (endereco == -1) {
        return false; // Nao encontrado
    }
```

```
    // Verificar se o registro ainda existe
    arquivo.seek(endereco);
    byte lapide = arquivo.readByte();
    if (lapide != ' ') {
        return false; // Ja foi excluido
    }

    short tamanho = arquivo.readShort();

    // Marcar como excluido no arquivo de dados
    arquivo.seek(endereco);
    arquivo.writeByte('*');
    addDeleted(tamanho, endereco);

    // Remover do indice
    indice.remover(id);

    return true;
}
```

Além da lápide, o sistema gerencia ativamente os espaços vagos. Quando um novo registro é criado através do método `create()`, o sistema primeiro consulta a lista de espaços livres (através do método `getDeleted()`) para verificar se existe algum espaço previamente excluído que seja grande o suficiente para o novo dado. Apenas se nenhum espaço for encontrado é que o novo registro é inserido no final do arquivo.

Esta estratégia combinada de lápide e gerenciamento de espaços livres garante que o sistema seja eficiente tanto na performance das operações de exclusão quanto na otimização do uso do espaço em disco.

9.4 Chaves utilizadas

O sistema utiliza chaves primárias para a identificação única de cada registro e uma chave de negócio (CPF) para estabelecer os relacionamentos entre as entidades.

Chaves Primárias

As chaves primárias são responsáveis por garantir a unicidade de cada registro dentro de seu respectivo arquivo de dados.

- **Cliente:** A entidade `Cliente` utiliza o campo `id` (inteiro, autoincremental) como sua **chave primária**.
- **Pet:** A entidade `Pet` também utiliza um campo `id` (inteiro, autoincremental) como sua **chave primária**.

Chaves de Relacionamento (Chave Estrangeira)

O relacionamento 1:N, onde um Cliente pode possuir múltiplos Pets, é estabelecido e indexado utilizando o CPF do Cliente como a chave de ligação.

- **Armazenamento Físico:** Dentro do registro de cada `Pet` no arquivo `pets.db`, o **CPF** do dono é gravado diretamente. Neste contexto, o CPF atua como uma chave estrangeira, ligando o registro do pet ao seu respectivo cliente.
- **Indexação do Relacionamento:** Para a busca otimizada, o relacionamento é formalmente modelado pela classe `RelacionamentoPetDono.java`. Esta classe encapsula a chave de busca principal, `cpfDono` (String), com o identificador do registro relacionado, `idPet` (inteiro). Objetos desta classe são efetivamente as entradas utilizadas no índice de Hash Extensível para permitir a recuperação rápida de todos os pets de um cliente a partir de seu CPF.

9.5 Estruturas utilizadas para cada chave

Para cada tipo de chave definida na seção anterior, foi escolhida uma estrutura de dados de indexação apropriada para otimizar as operações de busca, conforme detalhado abaixo.

Estrutura para Chaves Primárias: Índice Sequencial

As chaves primárias de todas as entidades (ex: `Cliente.id`, `Pet.id`) são gerenciadas por um **Índice Sequencial**, implementado na classe `IndiceSequencial.java`.

A estratégia utilizada é a seguinte: na inicialização, todos os pares de [ID, Endereço] são carregados do arquivo de índice (`.idx`) para um `ArrayList` em memória. Esta lista é mantida constantemente ordenada pelo ID. As buscas por um ID específico são, então, realizadas através de um método de **busca binária**, o que garante uma performance de O(log n), sendo extremamente eficiente.

O trecho de código abaixo demonstra a implementação da busca binária sobre a lista de índices em memória.

Listing 4: Implementação da busca binária no `IndiceSequencial.java`

```
private int buscaBinaria(int id) {
    int inicio = 0;
    int fim = indices.size() - 1;

    while (inicio <= fim) {
        int meio = (inicio + fim) / 2;
        int idMeio = indices.get(meio).id;

        if (idMeio == id) {
            return meio;
        } else if (idMeio < id) {
            inicio = meio + 1;
        } else {
            fim = meio - 1;
        }
    }

    return -1; // Nao encontrado
}
```

Estrutura para Chave de Relacionamento: Hash Extensível

Para indexar o relacionamento 1:N entre Cliente e Pet, foi utilizada uma estrutura de **Hash Extensível**, cuja lógica é gerenciada pela classe `IndiceHashExtensivel.java`.

Este índice armazena objetos do tipo `RelacionamentoPetDono`, que encapsulam o CPF do dono e o ID de um de seus pets. A chave utilizada para o *hashing* é um `hashCode()` gerado a partir da combinação única do CPF do dono e do ID do pet, garantindo que cada par (Dono, Pet) seja um registro único na estrutura de dados.

Quando um novo pet é inserido, o `PetDAO` invoca o método `inserir` do índice de hash para criar o vínculo, como mostra o código abaixo.

Listing 5: Inserção do relacionamento no Hash Extensível via `PetDAO.java`

```
public boolean incluirPet(Pet pet) throws Exception {
    // Criar o pet no arquivo principal
    int idGerado = arqPets.create(pet);

    if (idGerado > 0) {
        // Inserir relacionamento na Hash Extensivel
        String cpfDono = pet.getDono() != null ? pet.getDono().getCpf() : null;
        if (cpfDono != null && !cpfDono.isEmpty()) {
            indiceHash.inserir(cpfDono, idGerado);
        }
        return true;
    }

    return false;
}
```

Uma particularidade desta implementação é que, para buscar todos os pets de um mesmo dono (uma busca pelo CPF), é necessário percorrer todos os registros na estrutura de hash e filtrar aqueles que correspondem ao CPF desejado. Isso ocorre porque a chave de hashing foi projetada para garantir a unicidade do par, e não para agrupar múltiplos valores sob uma mesma chave de busca (CPF).

9.6 Implementação do 1:N

A implementação do relacionamento 1:N (um `Cliente` possui N `Pets`) foi realizada através da combinação de três componentes principais: o armazenamento da chave estrangeira no registro de dados, a manutenção de um índice secundário de Hash Extensível e a orquestração das operações pelos DAOs.

1. Modelagem e Armazenamento Físico

No nível do arquivo de dados (`pets.db`), a ligação entre um pet e seu dono é feita armazenando-se diretamente o CPF do `Cliente` dentro do registro do `Pet`. O método `toByteArray()` da classe `Pet` é responsável por serializar essa chave estrangeira junto com os demais dados do pet.

Listing 6: Armazenamento do CPF do dono no registro do Pet

```
// Trecho do metodo Pet.toByteArray()
dos.writeUTF(getDono() != null ? getDono().getCpf() : "");
```

2. Manutenção do Relacionamento e do Índice

A consistência entre os dados e o índice de relacionamento é mantida pelos DAOs durante as operações de escrita (inserção e exclusão).

Inserção: Quando um novo pet é cadastrado, o método `PetDAO.incluirPet()` executa duas ações cruciais:

1. O registro completo do `Pet` é criado no arquivo `pets.db`, e seu novo ID (chave primária) é gerado.
2. Em seguida, o método invoca o `IndiceHashExtensivel` para criar uma nova entrada de relacionamento, associando o CPF do dono ao ID recém-gerado do pet.

Listing 7: Atualização do índice Hash na inserção de um Pet

```
// Trecho do metodo PetDAO.incluirPet()
int idGerado = arqPets.create(pet);

if (idGerado > 0) {
    // Inserir relacionamento na Hash Extensivel
    String cpfDono = pet.getDono() != null ? pet.getDono().getCpf() : null;
    if (cpfDono != null && !cpfDono.isEmpty()) {
        indiceHash.inserir(cpfDono, idGerado);
    }
    return true;
}
```

Exclusão: A integridade é mantida na exclusão de qualquer uma das pontas do relacionamento.

- **Ao excluir um Pet:** O método `PetDAO.excluirPet()` remove o registro de `pets.db` e, em seguida, remove a entrada correspondente (`cpfDono`, `idPet`) do índice de hash.
- **Ao excluir um Cliente:** O método `ClienteDAO.excluirCliente()` remove o registro de `clientes.db` e invoca um método no índice de hash (`removerTodosPorCpf`) para apagar todas as entradas de relacionamento associadas àquele CPF, garantindo que não fiquem "pets órfãos"no índice.

3. Recuperação dos Dados (A Consulta 1:N)

A recuperação de todos os pets de um cliente é o principal objetivo desta implementação e é realizada de forma otimizada pelo método `buscarPetsPorCpfDono()` no `PetDAO`. O fluxo ocorre em duas etapas:

1. O método recebe o CPF do cliente e o utiliza para consultar o `IndiceHashExtensivel`, que retorna uma lista com todos os IDs de pets vinculados àquele CPF.
2. Com a lista de IDs em mãos, o sistema itera sobre ela. Para cada ID de pet, ele realiza uma busca rápida pelo índice primário (o `IndiceSequencial` da classe `Arquivo`) para recuperar o registro completo do pet no arquivo `pets.db`.

Essa estratégia de "consultar índice secundário para obter chaves, depois usar chaves para buscar dados"é fundamental para a eficiência do sistema.

Listing 8: Fluxo de busca otimizada do relacionamento 1:N

```
public java.util.List<Pet> buscarPetsPorCpfDono(String cpfDono) throws Exception {
    java.util.List<Pet> pets = new java.util.ArrayList<>();

    // 1. Buscar IDs dos pets usando a Hash Extensivel
    java.util.List<Integer> idsPets = indiceHash.buscarIdsPetsPorCpf(cpfDono);

    // 2. Buscar cada pet pelo ID usando o indice sequencial
    for (Integer idPet : idsPets) {
        Pet pet = arqPets.read(idPet);
        if (pet != null) {
            pets.add(pet);
        }
    }

    return pets;
}
```

9.7 Persistência dos índices em disco

A persistência dos índices em disco foi implementada com estratégias distintas para cada tipo de estrutura, visando equilibrar simplicidade, consistência e performance.

Persistência do Índice Sequencial

A estratégia adotada para o `IndiceSequencial.java` é a de **carregamento total em memória** e **reescrita completa** a cada modificação.

- **Carregamento:** Ao ser instanciado no início do programa, o índice lê sequencialmente todo o conteúdo do seu arquivo de persistência (`.idx`). Cada par de [ID (int), Endereço (long)] é lido e armazenado em um `ArrayList` em memória, que é então ordenado. Este processo é realizado pelo método `carregarIndices()`.
- **Salvamento:** Qualquer operação que altere a estrutura do índice – como `inserir()`, `remover()` ou `atualizar()` – invoca, ao final, o método privado `salvarIndices()`. Este método apaga completamente o conteúdo do arquivo `.idx` e o reescreve do zero com a versão mais recente e ordenada da lista de índices que reside na memória.

O código abaixo demonstra a estratégia de reescrita total do arquivo de índice.

Listing 9: Método de salvamento completo do `IndiceSequencial`

```
private void salvarIndices() throws Exception {
    arquivo.setLength(0); // Limpar arquivo
    arquivo.seek(0);

    for (RegistroIndice registro : indices) {
        arquivo.writeInt(registro.id);
        arquivo.writeLong(registro.endereco);
```

```
    }
}
```

Embora simples de implementar e garanta consistência, essa abordagem pode se tornar menos performática para um número muito grande de registros, pois cada alteração incorre no custo de reescrever todo o arquivo de índice.

Persistência do Índice de Hash Extensível

Diferentemente do índice sequencial, a **Hash Extensível** utiliza uma abordagem de **acesso direto em disco**, onde a maior parte da estrutura de dados permanece no arquivo e é manipulada sob demanda. A persistência é dividida em dois arquivos distintos:

- **Arquivo de Diretório (`_hash.dir`):** Armazena o diretório da hash, que é um array de ponteiros (endereços) para os cestos. Esta estrutura é relativamente pequena e é carregada em memória na inicialização do índice para agilizar a localização do cesto correto para uma dada chave. O diretório é salvo de volta no disco sempre que sua estrutura é alterada (por exemplo, ao duplicar de tamanho).
- **Arquivo de Cestos (`_hash.db`):** Armazena os cestos (buckets), que contêm os registros de índice propriamente ditos. Este arquivo pode crescer consideravelmente e seus dados **não são** carregados inteiramente em memória.

As operações de escrita (inserção, alteração, exclusão) utilizam `RandomAccessFile` para ler e escrever em porções específicas do arquivo de cestos. Apenas o cesto que está sendo modificado é lido do disco para a memória, alterado e, em seguida, escrito de volta na mesma posição. Esta técnica é significativamente mais escalável, pois o custo de uma operação de escrita não depende do tamanho total do índice, mas apenas do tamanho de um único cesto.

9.8 Estrutura do projeto no GitHub

O projeto foi organizado em uma estrutura de pastas que visa separar as diferentes responsabilidades da aplicação, inspirando-se em padrões de arquitetura como o *Model-View-Controller* (MVC) e o *Data Access Object* (DAO). A estrutura principal no repositório é a seguinte:

- `/src/`: Diretório principal que contém todo o código-fonte Java (`.java`) e também os arquivos de dados gerados pela aplicação. Ele é subdividido nos seguintes pacotes:
  - `/app/`: Contém as classes responsáveis pela camada de apresentação e interação com o usuário. Isso inclui a classe principal (`Main.java`), as classes de menu de console (`MenuCliente.java`, `MenuPet.java`) e as rotinas de teste (`BateriaTestes.java`).
  - `/dao/`: Contém as classes da camada de acesso a dados (Data Access Object). Estas classes formam a ponte entre a aplicação e o armazenamento físico, encapsulando toda a lógica de manipulação de arquivos binários (`Arquivo.java`) e a utilização das estruturas de índice (`IndiceSequencial.java`, `IndiceHashExtensivel.java`).
  - `/model/`: Contém as classes de modelo (ou domínio), que representam as entidades do sistema (`Cliente.java`, `Pet.java`, etc.). Estas classes implementam a interface `Registro` para permitir sua serialização.

  - `/dados/`: Diretório onde todos os arquivos de dados (`.db`) e índices (`.idx`, `_hash.dir`, etc.) são criados e armazenados pelo sistema em tempo de execução. Para este projeto, optou-se por aninhar esta pasta dentro do diretório de código-fonte `/src`.

- `/docs/`: Pasta destinada a armazenar a documentação do projeto, como diagramas, o relatório do trabalho e outros artefatos relevantes.

- `/.gitignore`: Arquivo de configuração do Git que especifica arquivos e pastas a serem ignorados pelo controle de versão. É utilizado para evitar o envio de arquivos gerados automaticamente, como os arquivos compilados (`/bin`) e os arquivos de banco de dados da pasta `/src/dados`.