Estudo Dirigido 5 - Desempenho e Otimização de Código e Padrões de Projeto

Lucas Apolonio de Amorim

9 de janeiro de 2025

Sumário

1 Introdução

Durante o desenvolvimento de um Sistema de Inventário de Produtos, vemos que o algoritmo de busca ideal depende tanto da escala quanto das demandas do nosso projeto.

Nesses casos em que diferentes contextos que exigem diferentes estratégias surge a necessidade de adaptar as estratégias aplicadas às mudanças de contexto, o que pode ser feito de forma simples e organizada pelo Padrão de Projeto *Strategy*.

2 Sobre Padrões de Projeto

2.1 O que são?

Padrões de Projeto são soluções comuns para problemas recorrentes no Desenvolvimento de Software, essas soluções são empregadas visando a legibilidade e organização do código, minimizando custos de manutenção e facilitando eventuais expansões da codebase. Esses Padrões costumam ser divididos em 3 eixos principais: Padrões *Criacionais*, *Estruturais* e *Comportatentais*.

2.2 Padrão Strategy

O Padrão Strategy é um exemplo de Padrão Comportamental que pode ser aplicado quando há a necessidade do uso de diferentes estratégias/algoritmos a depender do contexto atual do programa.

No Padrão Strategy é definida uma *Interface* que contém os métodos em comum entre todas as estratégias, a partir dela podemos criar diferentes implementações desses métodos, essas implementações da nossa Interface são chamadas de Estratégias Concretas.

Agora podemos guardar uma referência à nossa Interface e utilizar uma das estratégias concretas por meio delas, para respeitar o princípio da responsabilidade única criamos uma classe de contexto, que guarda essa referência e invoca a execução da estratégia referenciada. A classe de contexto também é útil para realizar o gerenciamento de memória em linguagens que não tem coletor de lixo, assim evitando repassar essa responsabilidade para o usuário.

Nesse Estudo Dirigido, usaremos o Padrão Strategy para implementar diferentes estratégias de busca por um produto no Catálogo.

3 Implementação

As implementações das classes referentes ao Padrão Strategy foram feitas em C++ e inseridas como capturas de tela feitas usando o Visual Studio Code com a extensão CodeSnap

3.1 Classe SearchStrategy

Interface que define somente um método *search()*, que é o algoritmo de busca a ser implementado pelas Estratégias Concretas.

```cpp
class SearchStrategy {
  public:
    virtual ~SearchStrategy() = default;
    virtual ProdIterator search(const std::vector<Produto> &produtos, const int &id_buscado) = 0;
};
```

Figura 1: Definição em C++ da Interface SearchStrategy

3.2 Classe LinearSearchStrategy

Essa estratégia Implementa uma busca linear no vetor e retorna:

- **Caso seja encontrado:** Uma referência para o Produto procurado.
- **Caso não seja encontrado:** Uma referência para a Posição após o final do array.

```cpp
class LinearSearchStrategy : public SearchStrategy {
  public:
    ProdIterator search(const std::vector<Produto> &produtos, const int &id_buscado) override;
};
```

Figura 2: Definição em C++ da classe LinearSearchStrategy

```cpp
ProdIterator LinearSearchStrategy::search(const std::vector<Produto> &produtos, const int &id_buscado) {
  for (auto it = produtos.begin(); it != produtos.end(); ++it) {
    if (it->getId() == id_buscado) return it;
  }
  return produtos.end();
}

```

Figura 3: implementação em C++ da função *search()* na classe LinearSearchStrategy

3.3 Classe BinarySearchStrategy

Essa estratégia Implementa uma busca binária no vetor (assumindo que ele está ordenado) e retorna:

- **Caso seja encontrado:** Uma referência para o Produto procurado.
- **Caso não seja encontrado:** Uma referência para a Posição após o final do array.

```cpp
class BinarySearchStrategy : public SearchStrategy {
  public:
    ProdIterator search(const std::vector<Produto> &produtos, const int &id_buscado) override;
};
```

Figura 4: Implementação da Classe BinarySearchStrategy em C++

```cpp
ProdIterator BinarySearchStrategy::search(const std::vector<Produto> &produtos, const int &id_buscado) {
  auto IdComp = [] (const Produto &prod, const int &id) -> bool { return prod.getId() < id; };
  auto begin = produtos.begin(), end = produtos.end();

  while(end - begin > 1) {
    auto middle = begin+((end-begin)/2);
    if (IdComp(*middle, id_buscado)) begin = middle+1;
    else end =middle;
  }
  return begin->getId() == id_buscado ? begin : produtos.end();
}

```

Figura 5: Implementação em C++ do método *search()* da BinarySearchStrategy

3.4 Classe SearchContext

Guarda uma estratégia e permite sua execução.

```cpp
class SearchContext {
  SearchStrategy *strategy;

  public:
    SearchContext(SearchStrategy *strategy) : strategy(strategy) {}
    SearchContext() : strategy(new LinearSearchStrategy()) {}

    ProdIterator search(const std::vector<Produto> &produtos, const int &id_buscado) {
      return strategy->search(produtos, id_buscado);
    }

    void setStrategy(SearchStrategy *strategy) {
      delete this->strategy;
      this->strategy = strategy;
    }
};
```

Figura 6: Implementação da Classe SearchContext em C++

OBS: Apesar do nome, a decisão de mudar de estratégia deve ser tomada pelo cliente, a classe deve somente ser uma ponte para o uso da estratégia escolhida no momento.

4 Desempenho

4.1 Complexidade Assintótica

Busca Linear: A Busca Linear consiste em Interagir com os elementos do array um por um, do início até o final, logo no pior caso teremos de visitar todos os elementos do array, assim tendo uma complexidade de $O(N)$.
Busca Binária: A Busca Binária requer que o array esteja ordenado pois utiliza comparações para encontrar o valor procurado, o algoritmo consiste em determinar um intervalo que o elemento procurado pode estar e dividir esse intervalo sucessivamente através de comparações entre o elemento do meio desse intervalo e o elemento procurado. Seu pior caso occore quando procuramos um valor que não esteja no array, o que leva $\lceil \log_2 N \rceil$ operações, logo sua complexidade é de $O(\log N)$.

Um problema da Busca Binária é que, como o array deve estar ordenado, no caso de um array, temos que ou reordenar ele, como somente o último elemento não está ordenado, podemos utilizar o insertion sort, que reordena o array em O(N).

Logo, para sistemas que tenham muitas consultas e poucas inserções, a Busca Binária tende a ser vantajosa, mas no caso de inserções mais frequentes e buscas mais esparsas as Busca Linear tende a ser mais eficiente, já no caso de quantidades similares de buscas e inserções é recomendável o uso da análise empírica, verificando cenários similares ao que o sistema será submetido e coletando os dados para comparação.

4.2 Complexidade Empírica

Os tempos de execução (média de 100 buscas) foram testados usando cada uma das estratégias com diferentes tamanhos de array, os dados foram medidos utilizando a biblioteca *chrono* do C++ e armazenados em um arquivo .csv, os gráficos foram plotados usando com Python utilizando as bibliotecas seaborn e matplotlib.

Foram medidos os tempos para cenários partindo de 10.000 elementos até 1.000.000 de elementos em incrementos de 10.000 para ambos os algoritmos, para cada cenário os produtos tem id's únicos numerados de 1 até o tamanho do cenário atual. As 100 consultas foram geradas utilizando uma Distribuição Uniforme de valores entre 1 e 1,5 vezes o tamanho do cenário, assim sendo esperado que 66,6% das consultas sejam id's presentes no Catálogo e outros 33,3% das consultas sejam de id's fora do Catálogo.

Foram gerados 3 gráficos: um comparando a busca linear com a busca binária, e outros dois contendo somente uma das buscas com sua respectiva curva de fitting.

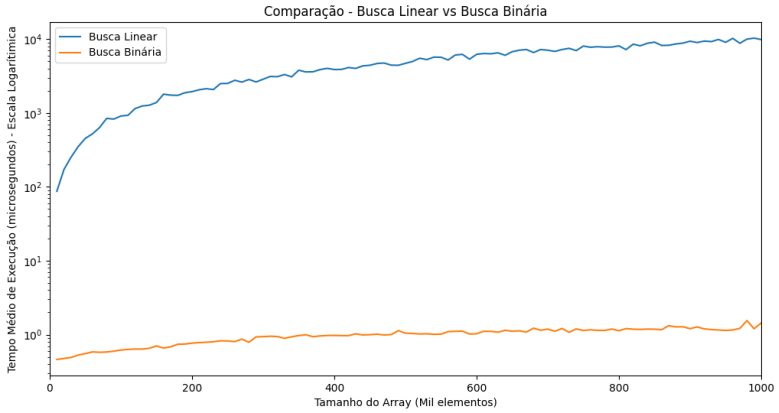

Figura 7: Comparação Entre os tempos de execução da Busca Linear e Busca Binária

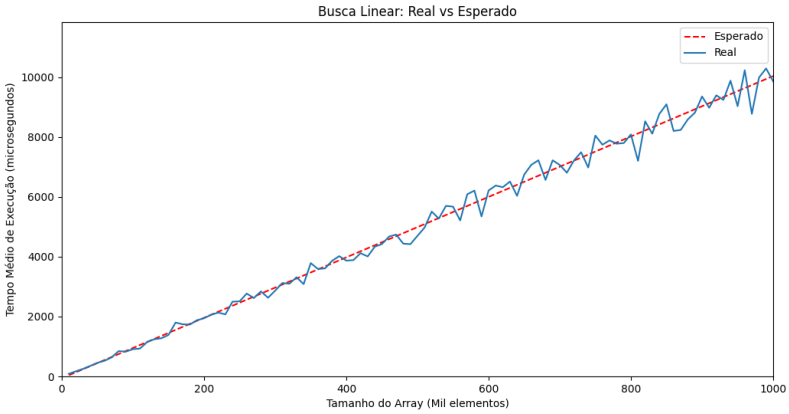

Figura 8: Comparação entre os tempos de execução reais e esperados da Busca Linear

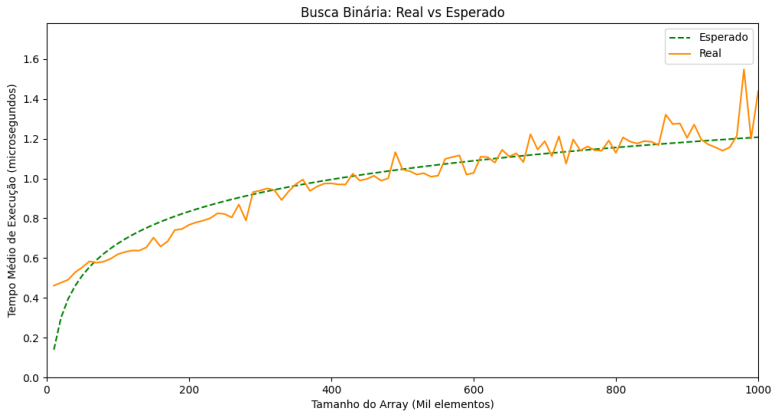

Figura 9: Comparação entre os tempos de execução reais e esperados da Busca Binária

Em quantidades a partir de 10.000 elementos, a Busca Binária performa várias ordens de magnitude mais rapidamente, como já era esperado. Mas vale conferir para catálogos com pouco produtos (< 1.000 produtos) o quão diferente é a performance de ambos os algoritmos.

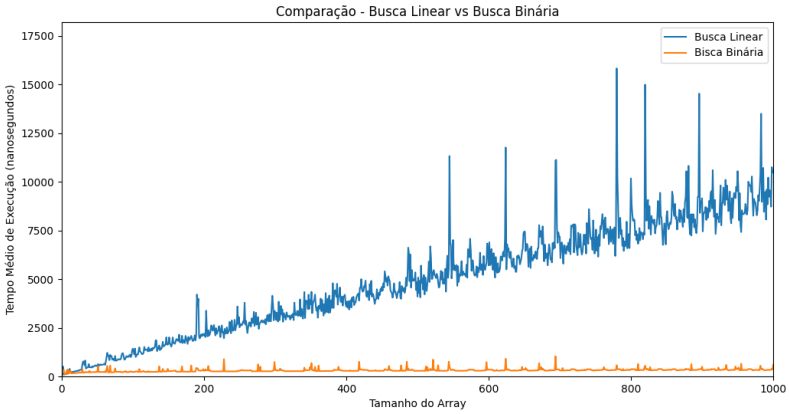

Figura 10: Comparação entre Busca Binária e Busca Linear para pequenas quantidade de elementos

Apesar do tamanho pequeno, a Busca Binária continua a performar consideravelmente melhor que a busca Linear, com a Linear sendo comparável com a Busca Binária somente em casos extremamente pequenos (< 12 produtos).

Uma Ineficiência que não foi contabilizada foi a realização do Insertion Sort após cada inserção, essa decisão foi intencional visto que, pela forma que foi introduzida o Padrão Strategy no sistema, sua responsabilidade é somente realizar a busca e não gerenciar os elementos no Catálogo, alternativamente, ao invés do Insertion Sort após cada inserção, poderia ser realizada uma ordenação em $\mathrm{O}(N \log N)$ antes de cada busca binária.

5 Discussão Sobre Estruturas de Dados

Como dito na seção anterior, um dos problemas na busca binária é a necessidade de que o array esteja ordenado, que no caso de arrays dinâmicos requer uma reordenação após cada inserção. Porém há outras estruturas de dados que permitem buscas rápidas e inserções rápidas, dentre elas, duas alternativas se destacam:

- **Tabelas Hash:** As Tabelas Hash Permitem a inserção e consulta por ID em $O(1)$ (Complexidade Esperada - Caso médio quando a função de hash é bem escolhida), mas algumas desvantagens são: complexidade de implementação, constantes razoavelmente altas nas operações e risco de colisões.
- **Árvores Binárias de Busca Balanceadas (BST's):** As BST's contém uma família de árvores que permitem a inserção em $O(\log N)$ e buscas em $O(\log N)$, a complexidade de implementação varia de acordo com a árvore

escolhida, cada uma atendendo diferentes necessidades. Alguns exemplos de BST's são: Árvores AVL, Árvores Rubro-Negras, Árvores B, Treaps e Árvores Splay.

- **Abordagem Híbrida:** Para sistemas mais complexos que necessitam de consultas eficientes em múltiplos parâmetros e grandes volumes de dados, é possível combinar diferentes estruturas. Um exemplo seria o uso conjunto de Tabelas Hash para busca direta por ID com Índices Invertidos (como os utilizados em sistemas de busca textual) para permitir consultas eficientes por diferentes atributos dos produtos. Por exemplo, uma Tabela Hash poderia mapear IDs para produtos, enquanto Índices Invertidos permitiriam buscar produtos por categoria, preço, ou outras características.

Apesar de Existirem essas opções, o array é uma solução completamente viável e bem menos complexa de implementar do que as mencionadas acima.

6 Conclusão

6.1 Resumo

O Padrão Strategy proporciona versatilidade e escalabilidade enquanto minimiza a dívida técnica.

6.2 Próximos Passos

Algumas melhorias que podem ser feitas seria a adição de novas Estratégias, usando diferentes Algoritmos e Estruturas de Dados, também podem ser introduzidos outros padrões de Projeto como:

- **Factory Method:** Para criar diferentes tipos de produtos de forma flexível e encapsulada, abstraindo a lógica de instanciação.
- **Observer:** Para notificar outros componentes do sistema quando mudanças ocorrem no catálogo de produtos, útil para manter consistência.
- **Decorator:** Para adicionar funcionalidades extras aos produtos de forma dinâmica, como descontos ou características especiais.
- **Command:** Para encapsular operações de modificação do catálogo como objetos, permitindo desfazer/refazer operações.
- **Singleton:** Para garantir uma única instância do catálogo de produtos em toda a aplicação.

A implementação desses padrões permitiria maior flexibilidade e extensibilidade do sistema, além de facilitar a manutenção e evolução do código.