PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

ANÁLISE DE PERFORMANCE COMPUTACIONAL EM DESIGN PATTERNS

GOIÂNIA
2023

GUSTAVO DA SILVA CARVALHO

ANÁLISE DE PERFORMANCE COMPUTACIONAL EM DESIGN PATTERNS

Trabalho de Conclusão de Curso apresentado à Escola Politécnica, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador(a): Prof. M.E.E. Rafael Leal Martins

GOIÂNIA

2023

GUSTAVO DA SILVA CARVALHO

ANÁLISE DE PERFORMANCE COMPUTACIONAL EM DESIGN PATTERNS

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Engenharia de Computação, em _____/_____/_____.

_____

Prof. Ma. Ludmilla Reis Pinheiro dos Santos

Coordenadora de Trabalho de Conclusão de Curso

Banca examinadora:

_____

Orientador: Prof. M.E.E. Rafael Leal Martins

_____

Prof. M.E.E. Fernando Gonçalves Abadia

_____

Prof. Dr. Nilson Cardoso Amaral

GOIÂNIA

2023

Dedico este trabalho ao meu pai, pois ele me ensinou a nunca desistir, mesmo quando tudo parecia impossível.

RESUMO

O trabalho a ser apresentado é o estudo e implementação de padrões de projeto com utilização de testes, análises de complexidade de algoritmo em projetos com arquitetura orientada a objetos e a partir disso entender as suas implicações na qualidade, performance, escalabilidade e variabilidade de software.

Palavras-Chave: Padrões de Projeto; performance; qualidade; arquitetura de software.

GOIÂNIA

2023

ABSTRACT

The work to be presented involves the study and implementation of design patterns using tests, algorithm complexity analysis in projects with object-oriented architecture, and understanding their implications on software quality, performance, scalability, and variability.

Keywords: Design Patterns; performance; quality; software architecture.

LISTA DE FIGURAS

LISTA DE SIGLAS

POO Programação Orientada a Objetos

DPD Design Patterns Detection

OMT Object Modeling Technique

AST - Abstract Syntax Tree

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho consta a utilização de ferramentas benchmark e testes de software para analisar e verificar a complexidade de algoritmo bem como a melhor aplicabilidade dos padrões de projeto em diferentes cenários que tenham como base a arquitetura orientada a objetos bem como medir o impacto, qualidade, performance e escalabilidade. (GAMMA, 2007).

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Eles nos ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. A implementação deles em grande parte envolve muito mais do que apenas o cenário correto de utilização pois cada padrão pode implicar no escopo inteiro do projeto dependendo de quantas vezes é utilizado e de como está sendo utilizado. O uso de maneira consciente deles permite aos projetistas experientes um domínio muito maior sobre os problemas futuros. (GAMMA, 2007).

A qualidade do software é um dos principais aspectos dentro de um projeto bem estruturado, muitos projetistas experientes a entendem como um pilar para a escalabilidade e manutenibilidade de um software por longos anos. Durante as diversas etapas do ciclo de desenvolvimento esta é uma das que mais defini o nível de complexidade que a solução pode ter, projetos orientados a objetos tendem a seguir as boas práticas definidas pela comunidade de um determinado framework ou convenção de alguma linguagem de programação, mas no geral essas condutas são revisadas e testadas por diversas vezes até que se entenda a quão sólida ela é e o quanto benéfica ela pode ser para os programadores. (GAMMA, 2007).

O objetivo é implementar, analisar e testar diferentes padrões de projeto afim de entendermos melhor o seu comportamento, características e suas aplicações como algoritmos reutilizáveis. Determinar fatores que possam influenciar positivamente ou negativamente na escolha de utilização de cada um e o quanto isso pode pesar no desenvolvimento de um projeto. Fornecer dados que demonstrem melhor os impactos causados. (GAMMA, 2007).

Diante deste contexto, este projeto visa responder à seguinte questão de pesquisa: **- Como será possível utilizar ferramentas de benchmark para definir as melhores escolhas de padrões em um projeto orientado a objetos?**

1.1 Objetivos

Os objetivos se dividem em objetivos gerais e objetivos específicos.

Objetivos gerais:

- Implementar os padrões e utilizar ferramentas de teste que possam analisar o comportamento e tempo de execução do código.
- Verificar por meio de testes se a utilização do padrão implicou em algum desgaste para o software ou o programador.
- Apurar os testes e entender se eles fazem sentido para o objetivo geral daquele projeto.

Objetivos específicos:

- Estudar melhor e entender a importância dos padrões de projeto.
- Estudar a análise e complexidade de algoritmos em padrões de projeto.
- Definir melhor os critérios para uso de cada padrão bem como auxiliar no uso deles em projetos maduros que sejam orientados a objeto.

1.2 Procedimentos metodológicos

Este trabalho por natureza é um resumo do assunto sobre padrões de projeto, pois busca analisar e compreender melhor, indicando sua evolução, importância e desafios na área de engenharia de software. (GAMMA, 2007).
Ao observamos a consistência do presente trabalho, pode-se entender que esta pesquisa se traduz em exploratória e experimental.

O pesquisador Wazlawick (2014) afirma, “Na pesquisa exploratória, o autor vai examinar um conjunto de fenômenos, buscando anomalias que não sejam ainda

conhecidas e que possam ser, então, a base para uma pesquisa mais elaborada" (WAZLAWICK, 2014, p. 22). Com as análises que serão feitas podemos então buscar pelas anomalias e fenômenos na qual o autor defini como importantes para a pesquisa exploratória.

Para Gil (2008), uma pesquisa experimental precisa necessariamente de um objeto de estudo, variáveis que podem manipulá-lo ou que o influenciam de forma indireta, formas de controle das variáveis. Neste trabalho seguiremos com esses objetivos visando atender os resultados esperados a partir de nossos experimentos.

Tendo isso em mente podemos definir que a pesquisa bibliográfica vai seguir os seguintes passos:

- Listar os periódicos e temas relevantes para a pesquisa apresentada
- Pesquisar artigos e estudos que sejam dos últimos 5 anos
- A análise dos dados será obtida pela coleta dos resultados gerados em testes de software e nos logs das ferramentas utilizadas.
- O trabalho será registrado em forma de monografia de TCC (Trabalho de conclusão de curso).

Podemos então definir que para a realização de uma pesquisa experimental é necessário seguir os seguintes passos:

A) Como implementar padrões de projeto de forma responsável e como utilizar uma ferramenta de detecção.

B) Inicialmente no plano experimental, abordaremos todos os campos de pesquisa com relação aos padrões de projeto criacionais e estruturais.

C) Quanto ao ambiente experimental iremos implementar através da ferramenta DPD.

D) A coleta de dados será feita de forma bibliográfica buscando artigos, livros e documentos mais recentes.

1.3 Resultados esperados

Espera-se que os resultados dessa pesquisa possam contribuir para:

- Ampliar o conhecimento sobre os padrões de projetos e seu ecossistema.
- Auxiliar desenvolvedores de software a implementarem padrões de projeto de maneira mais fácil.
- Definir a importância dos recursos e tempo de execução gastos em algoritmos.

1.4 O trabalho está sendo desenvolvido da seguinte forma:

O capítulo 1 foca em introduzir os conceitos básicos necessários sobre o tema proposto e aborda de maneira sucinta o que será apresentado nos próximos capítulos.

O Capítulo 2 trata das definições teóricas principais sobre o tema em questão.

O capítulo 3 se dedica a apresentar as propostas que deverão ser atendidas futuramente ao longo do desenvolvimento do TCC.

2 REFERENCIAL TEÓRICO

Antes de abordamos o estudo dos diferentes padrões de projeto é necessário entender brevemente alguns fatores básicos, mas muito importantes sobre o paradigma da programação orientada a objetos (POO). Dentro desse conceito podemos afirmar que um programa de computador vai utilizar classes e objetos criados a partir de modelos para representar e processar dados. (SANTOS, 2013).

Os dados pertencentes aos modelos serão representados por tipos nativos que são característicos das linguagens de programação e por fim serão encapsulados dentro de suas classes podendo ser apenas acessados por meio dos métodos especializados na manipulação destes atributos. Classes são escritas com os recursos e regras da linguagem de programação orientada a objetos para implementação dos modelos, mas em muitos dos casos as classes são somente moldes ou formas que representam os modelos abstratamente. A partir disso existe também a possibilidade de reutilizar essas classes através do conceito de herança onde atributos, métodos e restrições podem ser passados para outras diferentes classes filhas originadas de uma classe pai. (SANTOS, 2013).

Praticamente quase todas as arquiteturas orientadas a objetos bem estruturados estão cheias de padrões e com isso existem diversas maneiras de medir a qualidade de um sistema orientado a objetos, uma delas é verificar se os desenvolvedores tomaram bastante cuidado com as colaborações comuns entre seus objetos. Dedicar tempo nesse mecanismo durante o desenvolvimento de um sistema pode levar a uma arquitetura menor, mais simples e muito mais compreensível do que as que são produzidas sem o uso de padrões. (GAMMA, 2007).

Projetar software orientado a objetos é difícil, mas projetar software reutilizável orientado a objetos é ainda mais complicado. É necessário identificar objetos pertinentes, fatorar em classes no nível correto de granularidade, definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chave entre eles. O seu projeto deve ser específico para o problema a resolver, mas também genérico o suficiente para atender problemas e requisitos futuros. Também deseja evitar o reprojeto, ou pelo menos minimizá-lo. Os mais experientes projetistas de software orientado a objetos lhe dirão que um projeto reutilizável e flexível é difícil, senão impossível, de obter corretamente da primeira vez. Antes que um projeto esteja

terminado, eles normalmente tentam reutilizá-lo várias vezes, modificando-o a cada vez. (GAMMA, 2007).

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente. Em suma, ajudam um projetista a obter mais rapidamente um projeto adequado. (GAMMA, 2007).

2.1 O que é um padrão de projeto?

Podemos definir um padrão de projeto como algo que descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que você possa utilizar a solução diversas vezes sem nunca a fazer da mesma maneira. No geral os padrões possuem quatro características essenciais, sendo o nome do padrão, o problema, a solução e as consequências. O nome é uma referência que podemos usar para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras. Dar nome a um padrão aumenta imediatamente o nosso vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite-nos conversar sobre eles com nossos colegas, em nossa documentação e até com nós mesmos. O nome torna mais fácil pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas. Encontrar bons nomes foi uma das partes mais difíceis do desenvolvimento. O problema descreve em que situação aplicar o padrão. Ele explica o problema e seu contexto. Pode descrever problemas de projeto específicos, tais como representar algoritmos como objetos. Pode descrever estruturas de classe ou objeto sintomáticos de um projeto inflexível. Algumas vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão. A solução descreve os elementos que compõem o padrão de projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma

descrição abstrata de um problema de projeto e de como um arranjo geral de elementos (classes e objetos, no nosso caso) o resolve. As consequências são os resultados e análises das vantagens e desvantagens (trade-offs) da aplicação do padrão. Embora as consequências sejam raramente mencionadas quando descrevemos decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. (GAMMA, 2007).

As consequências para o software frequentemente envolvem balanceamento entre espaço e tempo. Elas também podem abordar aspectos sobre linguagens e implementação. Uma vez que a reutilização é frequentemente um fator no projeto orientado a objetos, as consequências de um padrão incluem o seu impacto sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema. Relacionar essas consequências explicitamente ajuda a compreendê-las e avaliá-las. (GAMMA, 2007).

Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização. (GAMMA, 2007).

2.2 Descrevendo os padrões de projeto

É necessário usar um formato consistente para descrever cada padrão por isso eles são divididos por seus nomes e classificações. Podemos então destacar da seguinte forma: (GAMMA, 2006).

- Nome e classificação do padrão: O nome do padrão expressa a sua própria essência de forma sucinta. Um bom nome é vital, porque ele se tornará parte do seu vocabulário de projeto.
- Intenção e objetivo: É uma curta declaração que responde às seguintes questões: o que faz o padrão de projeto? Quais os seus princípios e sua intenção? Que tópico ou problema particular de projeto ele trata?

- Também conhecido como: Quaisquer outros nomes bem conhecidos para o padrão caso existam.
- Motivação: Um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão soluciona o problema. O cenário ajudará a compreender as descrições mais abstratas do padrão que vêm a seguir.
- Aplicabilidade: Quais são as situações nas quais o padrão de projeto pode ser aplicado? Que exemplos de maus projetos ele pode tratar? Como você pode reconhecer essas situações?
- Estrutura: Uma representação gráfica das classes do padrão usando uma notação baseada na Object Modeling Technique (OMT) [RBP+ 91] *. Nós também usamos diagramas de interação [JCJO92, Boo94] para ilustrar sequências de solicitações e colaborações entre objetos.
- Participantes: As classes e/ou objetos que participam do padrão de projeto e suas responsabilidades.
- Colaborações: Como as classes se relacionam e colaboram para executar suas responsabilidades.
- Consequências: Como o padrão suporta a realização de seus objetivos? Quais são os seus custos e benefícios e os resultados da sua utilização? Que aspecto da estrutura de um sistema ele permite variar independentemente?
- Implementação: Que armadilhas, sugestões ou técnicas você precisa conhecer quando da implementação do padrão? Existem considerações específicas de linguagem?
- Exemplo de código: Fragmentos ou blocos de código que ilustram como você pode implementar o padrão em alguma linguagem de programação.
- Usos conhecidos: Exemplos do padrão encontrados em sistemas reais.
- Padrões relacionados: Que padrões de projeto estão intimamente relacionados com este? Quais são as diferenças importantes? Com quais outros padrões ele deveria ser usado? (GAMMA, 2007).

2.3 Padrões de criação

Os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar o software um sistema independente da forma como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto um padrão de criação de objeto delegará a instanciação para outro objeto. (GAMMA, 2007).

É essencial também entender que os padrões de criação se tornam importantes à medida que os sistemas evoluem no sentido de depender mais da composição de objetos do que da herança de classes. Quando isso acontece, a ênfase se desloca da codificação rígida de um conjunto fixo de comportamentos para a definição de um conjunto menor de comportamentos fundamentais, os quais podem ser compostos em qualquer número para definir comportamentos mais complexos. Assim, criar objetos com comportamentos particulares exige mais do que simplesmente instanciar uma classe. (GAMMA, 2007).

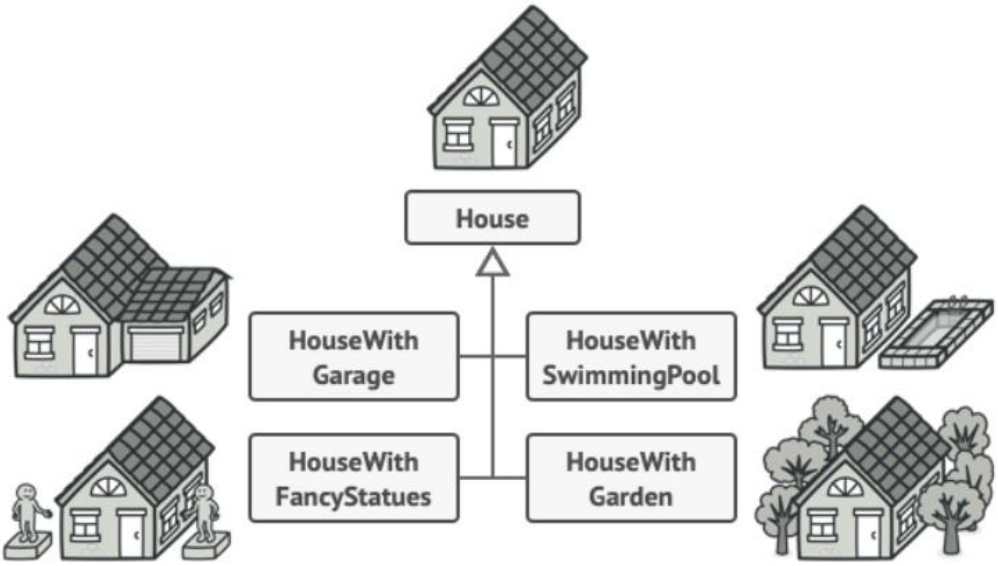
Existem dois pontos muito recorrentes neste tipo de padrão. Primeiro eles encapsulam conhecimento sobre as classes concretas definidas no sistema. Segundo ocultam o modo como as instâncias destas classes são criadas e compostas. Tudo o que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Consequentemente, os padrões de criação dão muita flexibilidade ao que, como e quando é criado e a quem cria. Eles permitem configurar um sistema com “objetos produto” que variam amplamente em estrutura e funcionalidade. A configuração pode ser estática (isto é, especificada em tempo de compilação) ou dinâmica (em tempo de execução). (GAMMA, 2007).

2.3.1 Builder

O padrão de projeto Builder tem como objetivo separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações. Imagine um objeto complexo que requer inicialização trabalhosa e passo a passo de muitos campos e objetos aninhados. Esse código de inicialização geralmente é enterrado dentro de um construtor monstruoso com muitos parâmetros. Ou pior ainda: espalhados por todo o código do cliente. (GAMMA, 2006).

Por exemplo, vamos pensar em como criar um House objeto. Para construir uma casa simples, você precisa construir quatro paredes e um piso, instalar uma porta, encaixar um par de janelas e construir um telhado. Mas e se você quiser uma casa maior e mais brilhante, com quintal e outras guloseimas (como sistema de aquecimento, encanamento e fiação elétrica)? a solução mais simples é estender a House classe base e criar um conjunto de subclasses para cobrir todas as combinações dos parâmetros. Mas eventualmente você acabará com um número considerável de subclasses. Qualquer novo parâmetro, como o estilo da varanda, exigirá ainda mais o crescimento dessa hierarquia. (GAMMA, 2007).

Figura 1 – Representação de Construções para um Objeto

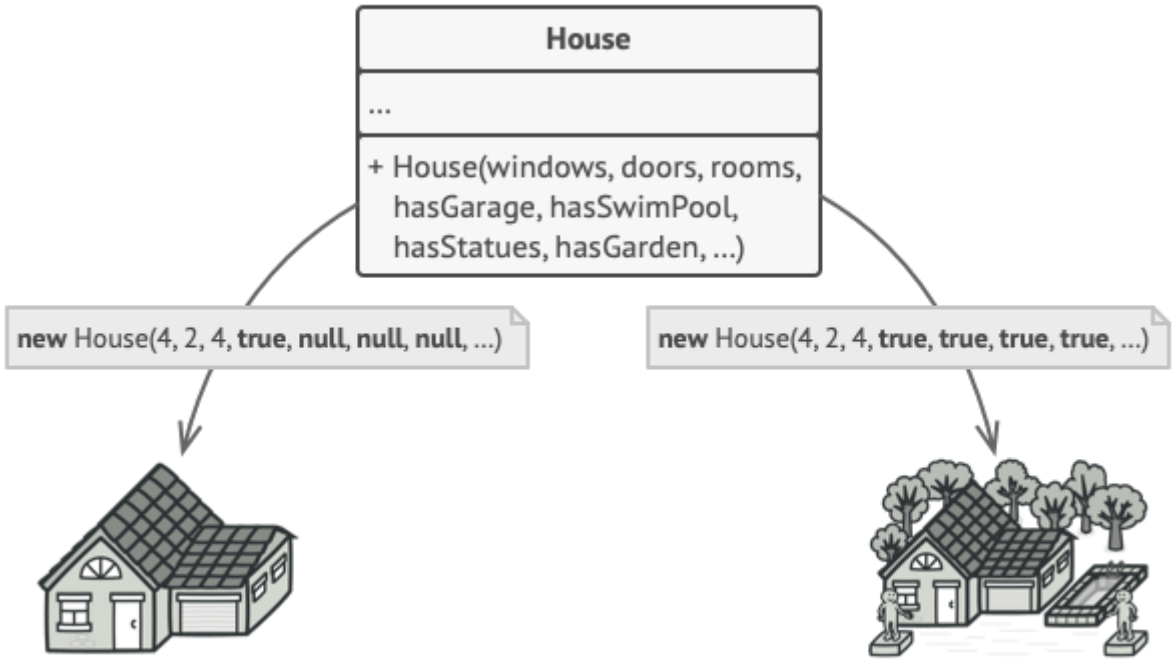


Fonte: Site Refactoring guru, 2022.

Na maioria dos casos, a maioria dos parâmetros não será usada, tornando as chamadas do construtor muito feias. Por exemplo, apenas uma fração das casas tem piscinas, então os parâmetros relacionados às piscinas serão inúteis nove em cada dez vezes. Devemos usar o padrão Builder quando:

- o algoritmo para criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas são montadas.
- o processo de construção deve permitir diferentes representações para o objeto que é construído. (GAMMA, 2007).

Figura 2 – A Maioria dos Parâmetros não será utilizada



Fonte: Site Refactoring guru, 2022.

Na figura 3 conseguimos compreender existem diversas vantagens e desvantagens ao utilizá-lo, das mais variadas formas este padrão pode implicar bastante na construção do software. Podemos destacar sendo:

- A complexidade geral do código aumenta, pois o padrão requer a criação de várias novas classes.
- Você pode construir objetos passo a passo, adiar etapas de construção ou executar etapas recursivamente.
- Você pode reutilizar o mesmo código de construção ao construir várias representações de produtos.
- Princípio da Responsabilidade Única. Você pode isolar o código de construção complexo da lógica de negócios do produto. (GAMMA, 2007).

Demonstraremos a seguir uma rápida implementação deste padrão em código Ruby:

main.rb: Exemplo conceitual

```
# A interface Builder específicas métodos para criar diferentes partes do Product objects.
class Builder
  # @abstract
  def produce_part_a
    raise NotImplementedError, "#{self.class} não implementou o método '#{__method__}'"
  end

  # @abstract
  def produce_part_b
    raise NotImplementedError, "#{self.class} não implementou o método '#{__method__}'"
  end

  # @abstract
  def produce_part_c
    raise NotImplementedError, "#{self.class não implementou o método '#{__method__}'"
  end
end

# As classes Concrete Builder seguem a interface Builder e fornecem implementações das etapas de construção. Seu programa pode ter várias variações de builders implementadas de forma diferente.
class ConcreteBuilder1 < Builder
  def initialize
    reset
  end

  def reset
    @product = Product1.new
  end

  # Concrete Builders devem fornecer seus próprios métodos para recuperar resultados. Isso porque vários tipos de builders podem criar produtos diferentes que não seguem a mesma interface portanto tal métodos podem ser declarados na interface base do Builder (pelo menos em uma linguagem de programação tipada estaticamente.)
  def product
    product = @product
    reset
```

```ruby
    product
  end

  def produce_part_a
    @product.add('PartA1')
  end

  def produce_part_b
    @product.add('PartB1')
  end

  def produce_part_c
    @product.add('PartC1')
  end
end

# Apenas faz sentido usar o padrão Builder quando seus Products são bastante complexos e exigem uma configuração extensa.

# Ao contrário de outros padrões de criação, diferentes Concrete Builders podem produzir produtos não relacionados.
  def initialize
    @parts = []
  end

  # @param [String] part
  def add(part)
    @parts << part
  end

  def list_parts
    print "Product parts: #{@parts.join(', ')}"
  end
end

class Director
  # @return [Builder]
  attr_accessor :builder

  def initialize
    @builder = nil
  end

  def builder=(builder)
    @builder = builder
  end

  def build_minimal_viable_product
    @builder.produce_part_a
  end
```

```
  def build_full_featured_product
    @builder.produce_part_a
    @builder.produce_part_b
    @builder.produce_part_c
  end
end

director = Director.new
builder = ConcreteBuilder1.new
director.builder = builder

puts Produto básico padrão'
director.build_minimal_viable_product
builder.product.list_parts

puts "\n\n"

puts ‘Produto padrão completo: '
director.build_full_featured_product
builder.product.list_parts

puts "\n\n"

puts 'Produto customizado: '
builder.produce_part_a
builder.produce_part_b
builder.product.list_parts
```

```
  print 'Singleton works, both variables contain the same instance.'
else
  print 'Singleton failed, variables contain different instances.'
end
```

2.4 Padrões Estruturais

Nas arquiteturas de projetos com orientação a objetos, temos como um dos fatores mais importantes a estrutura das classes e objetos e é nesse ponto que entra os padrões estruturais que são responsáveis por formar respectivamente classes e objetos com o objetivo de construir estruturas maiores. Os padrões estruturais usam a herança para definir interfaces e implementações, isto é, ao compor duas ou mais classes em outra no final teríamos uma classe que combina as propriedades das suas antecessoras. Essa principal característica faz com que este padrão seja muito útil em fazer bibliotecas desenvolvidas de maneira diferente conseguirem trabalharem juntas. (GAMMA, 2006).

A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes, tendo como exemplo o padrão Composite, nele temos como descrever a construção de uma hierarquia de classes compostas para dois tipos de objetos, sendo os primitivos e compostos. Os objetos compostos permitem compor objetos primitivos e outros objetos compostos em estruturas arbitrariamente complexas. Abordaremos mais a frente mais detalhes sobre esse último e sobre todos os outros padrões da família estrutural. (GAMMA, 2007).

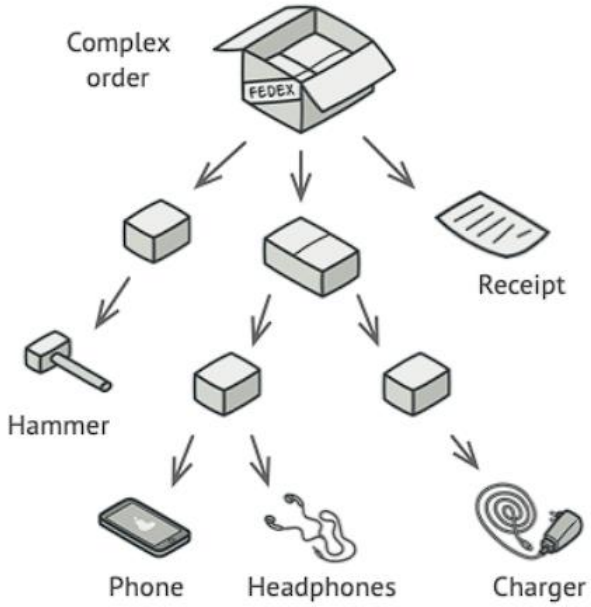
2.4.1 Composite

Quando falamos no Composite, podemos definir ele como um padrão de projeto estrutural que permite compor objetos em estruturas de árvore e, em seguida, trabalhar com essas estruturas como se fossem objetos individuais. Uma das motivações para a utilização deste padrão são as aplicações gráficas, sendo editores de desenhos digitais ou até mesmo sistemas de captura. O uso do padrão Composite só faz sentido quando o modelo principal do seu aplicativo pode ser representado como uma árvore. (GURU, 2022).

Um bom exemplo de problema para esse padrão é quando precisamos criar um sistema de pedidos que use essas classes, e nelas as encomendas podem conter produtos simples sem qualquer embalagem, assim como caixas recheadas de

produtos e com outras caixas. Como poderíamos fazer para determinar o preço total do pedido? A imagem abaixo descreve exatamente essa situação. (GURU, 2022).

Figura 3 – Exemplo de um problema com estrutura em árvore



Um pedido pode incluir vários produtos, embalados em caixas, que são embalados em caixas maiores e assim por diante. Toda a estrutura parece uma árvore de cabeça para baixo.

Fonte: Site Refactoring guru, 2022.

Na imagem referenciada temos um exemplo claro, você pode tentar a abordagem direta: desembrulhe todas as caixas, passe por cima de todos os produtos e depois calcule o total. Isso seria factível no mundo real, mas em um programa, não é tão simples quanto executar um loop. Você tem que conhecer as classes de Products e Boxes você está passando, o nível de aninhamento das caixas e outros detalhes desagradáveis de antemão. Tudo isso torna a abordagem direta muito estranha ou mesmo impossível. O padrão Composite nos sugere que você trabalhe com as classes Products e Boxes através de uma interface comum que declara um método para calcular o preço total. Como funcionaria esse método? Para um produto, ele

simplesmente retornaria o preço do produto. Para uma caixa, ele passaria por cada item que a caixa contém, perguntaria seu preço e retornaria um total para essa caixa. Se um desses itens fosse uma caixa menor, essa caixa também começaria a repassar seu conteúdo e assim por diante, até que os preços de todos os componentes internos fossem calculados. Uma caixa pode até adicionar algum custo extra ao preço final, como o custo da embalagem. (GURU, 2014).

O maior benefício dessa abordagem é que você não precisa se preocupar com as classes concretas de objetos que compõem a árvore. Você não precisa saber se um objeto é um produto simples ou uma caixa sofisticada. Você pode tratá-los da mesma forma através da interface comum. Quando você chama um método, os próprios objetos passam a solicitação pela árvore. (GURU, 2022).

A aplicabilidade deste padrão se faz nas seguintes condições:

- Use o padrão Composite quando precisar implementar uma estrutura de objeto semelhante a uma árvore.
- O padrão Composite fornece dois tipos básicos de elementos que compartilham uma interface comum: folhas simples e contêineres complexos. Um recipiente pode ser composto de folhas e outros recipientes. Isso permite construir uma estrutura de objeto recursivo aninhado que se assemelha a uma árvore.
- Use o padrão quando quiser que o código do cliente trate elementos simples e complexos de maneira uniforme.
- Todos os elementos definidos pelo padrão Composite compartilham uma interface comum. Usando esta interface, o cliente não precisa se preocupar com a classe concreta dos objetos com os quais trabalha. (GURU, 2022).

Também vale destacar as consequências da utilização deste padrão:

- Você pode trabalhar com estruturas de árvore complexas de forma mais conveniente: use polimorfismo e recursão a seu favor.
- Princípio aberto/fechado. Você pode introduzir novos tipos de elementos no aplicativo sem quebrar o código existente, que agora funciona com a árvore de objetos.
- Pode ser difícil fornecer uma interface comum para classes cuja funcionalidade seja muito diferente. Em certos cenários, você precisaria generalizar demais a interface do componente, dificultando a compreensão. (GURU, 2022).

A classe base Component declara operações comuns tanto para simples quanto para objetos complexos de uma composição. Opcionalmente, o Componente base pode declarar uma interface para configuração e acessando um pai do componente em uma estrutura de árvore. Ele também pode fornecer alguma implementação padrão para esses métodos. Em alguns casos, seria benéfico definir o controle das operações diretamente na classe Component base. Dessa forma, você não precisará expor quaisquer classes de componentes concretos ao código do cliente, mesmo durante a montagem da árvore de objetos. A desvantagem é que esses métodos estarão vazios para os componentes de nível folha. (GURU, 2022).

O componente base pode implementar algum comportamento padrão ou deixá-lo para classes concretas (declarando o método que contém o comportamento como "abstrato"). A classe Leaf representa os objetos finais de uma composição. Uma folha não pode ter quaisquer crianças. Normalmente, são os objetos Leaf que fazem o trabalho real, enquanto os objetos compostos apenas delegam para seus subcomponentes. A classe Composite representa os componentes complexos que podem ter filhos. Normalmente, os objetos Composite delegam o trabalho real para seus filhos e depois "soma" o resultado. Agora demonstraremos um exemplo conceitual de implementação deste padrão: (GURU, 2022).

main.rb:

```
class Component
  # @return [Component]
  def parent
    @parent
  end

  def parent=(parent)
    @parent = parent
  end

  def add(component)
    raise NotImplementedError, "#{self.class} este método não foi implementado
'#{__method__}'"
  end

  def remove(component)
```

```
    raise NotImplementedError, "#{self.class} este método não foi implementado
'#{__method__}'"
  end

  def composite?
    false
  end

  def operation
    raise NotImplementedError, "#{self.class} este método não foi implementado
'#{__method__}'"
  end
end

class Leaf < Component
  # return [String]
  def operation
    'Leaf'
  end
end


class Composite < Component
  def initialize
    @children = []
  end

  # @param [Component] component
  def add(component)
    @children.append(component)
    component.parent = self
  end

  # @param [Component] component
  def remove(component)
    @children.remove(component)
    component.parent = nil
  end

  # @return [Boolean]
  def composite?
    true
  end

  def operation
    results = []
    @children.each { |child| results.append(child.operation) }
    "Branch(#{results.join('+')})"
  end
end
```

```
  puts "RESULT: #{component.operation}"
end

def client_code2(component1, component2)
  component1.add(component2) if component1.composite?

  print "RESULT: #{component1.operation}"
end

simple = Leaf.new
puts 'Client: eu tenho um componente simples:'
client_code(simple)
puts "\n"

tree = Composite.new

branch1 = Composite.new
branch1.add(Leaf.new)
branch1.add(Leaf.new)

branch2 = Composite.new
branch2.add(Leaf.new)

tree.add(branch1)
tree.add(branch2)

puts 'Client: Agora eu tenho uma árvore composta:'
client_code(tree)
puts "\n"

puts 'Client: Não preciso checar as classes dos componentes mesmo gerenciando a árvore:'
client_code2(tree, simple)
```

2.4.2 Padrões Comportamentais

Os padrões comportamentais são um grupo de padrões de projeto que se concentram no comportamento e comunicação entre objetos. Eles lidam com a interação entre diferentes objetos e como eles se relacionam e colaboram para alcançar um determinado objetivo. Esses padrões ajudam a definir algoritmos, responsabilidades e interações entre objetos de forma flexível e eficiente. Existem muitos nessa categoria mais aqui daremos foco ao Interpreter pois ele será utilizado futuramente nos testes.

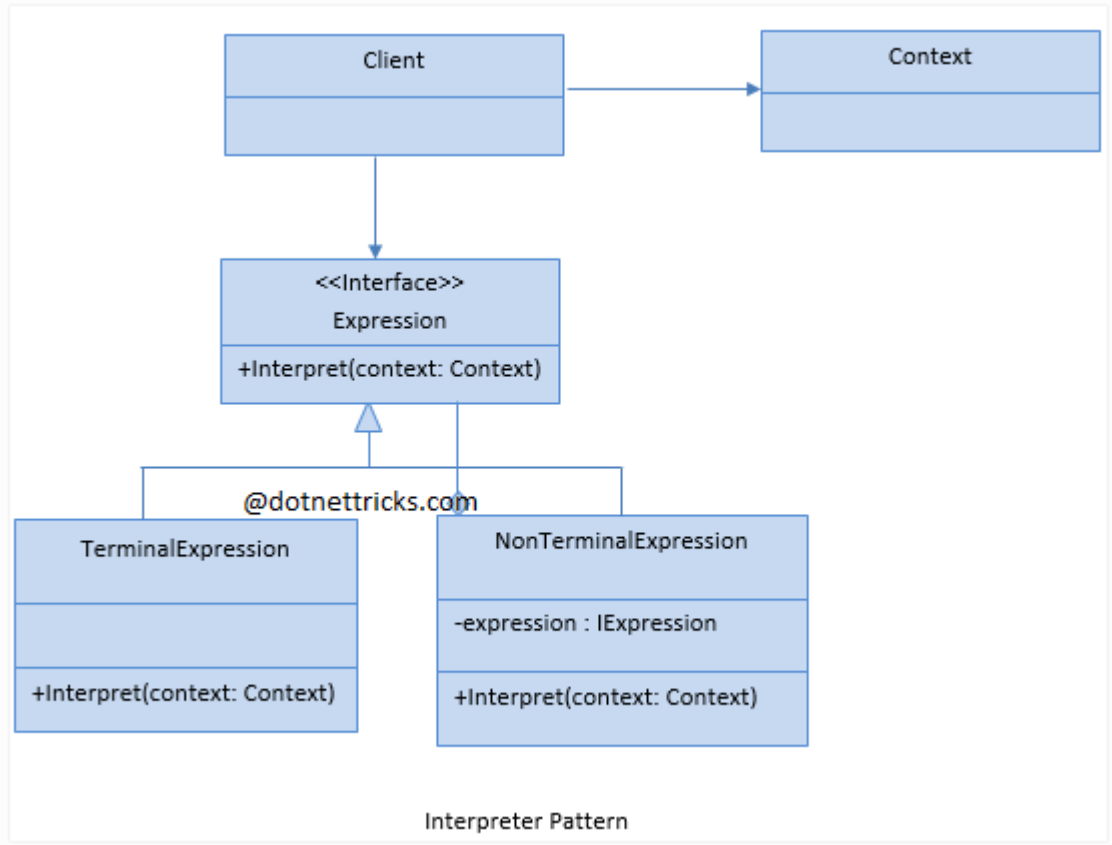
2.4.3 Interpreter

O padrão Interpreter, também conhecido como Padrão Interpretador, é um padrão de projeto de software que pertence à categoria dos padrões de projeto comportamentais. Ele permite a criação de uma linguagem de programação ou de uma linguagem de consulta personalizada, além de definir uma representação gramatical para essa linguagem e um interpretador que avalia as expressões da linguagem. O padrão Interpreter é usado para resolver problemas relacionados à interpretação de expressões ou comandos. Ele pode ser aplicado quando há necessidade de avaliar ou executar expressões complexas, como fórmulas matemáticas, consultas em bancos de dados ou expressões de consulta em linguagens de consulta personalizadas. O padrão Interpreter é composto por vários elementos principais, bem como:

- Abstract Expression (Expressão Abstrata): Define uma interface comum para todas as expressões da linguagem. Essa interface geralmente possui um método para avaliar a expressão.
- Terminal Expression (Expressão Terminal): Implementa as expressões terminais da linguagem, que não têm subexpressões. Essas expressões representam os blocos de construção básicos da linguagem.
- Nonterminal Expression (Expressão Não Terminal): Implementa as expressões não terminais, que têm subexpressões. Essas expressões combinam outras expressões para formar uma expressão maior.

- Context (Contexto): Contém informações globais que são compartilhadas entre as expressões durante a avaliação. Ele fornece um contexto para as expressões interpretarem.
- Interpreter (Interpretador): Define a lógica de interpretação da linguagem. Ele analisa a expressão gramaticalmente e avalia a expressão para obter um resultado.

Figura 4 – Diagrama UMl do padrão



Fonte: Site Refactoring guru, 2022.

2.4.4 Análise e complexidade de algoritmo

A análise de um algoritmo consiste em estimar os recursos necessários para sua execução. Embora às vezes a memória, a largura de banda ou o hardware sejam os principais fatores a serem considerados, é mais comum medirmos o tempo de execução. Analisando vários algoritmos candidatos para um problema, podemos facilmente identificar o mais eficiente. Embora a análise possa apontar mais de uma opção viável, geralmente podemos descartar algoritmos de qualidade inferior durante o processo. (CORMEN, 2012).

Para que seja possível realizar a análise de um algoritmo, é necessário primeiro ter um modelo da tecnologia de implementação que será utilizada, incluindo um modelo dos recursos dessa tecnologia e seus respectivos custos. Na maior parte deste livro, utilizaremos um modelo de computação genérico chamado de Máquina de Acesso Aleatório (Random-Access Machine, RAM), com um único processador como tecnologia de implementação. Consideraremos que nossos algoritmos serão implementados como programas de computador. No modelo de RAM, as instruções são executadas de forma sequencial, sem operações concorrentes. (CORMEN, 2012).

O tempo de execução de um algoritmo em uma entrada específica é medido pelo número de operações primitivas ou "passos" realizados. Para tornar a definição de passo o mais independente possível da máquina, é conveniente definir uma quantidade de tempo constante para cada linha do pseudocódigo. Embora diferentes linhas possam exigir diferentes quantidades de tempo, consideraremos que cada execução da linha i leva um tempo constante ci, onde ci é uma constante. Esse ponto de vista é coerente com o modelo de RAM e reflete o modo como o pseudocódigo seria implementado na maioria dos computadores reais. (CORMEN, 2012).

As notações para tempo de execução de algoritmos são utilizadas para expressar a complexidade temporal de um algoritmo em termos do tamanho da entrada. Essas notações ajudam a comparar e analisar a eficiência de diferentes algoritmos para o mesmo problema. As notações mais comuns são:

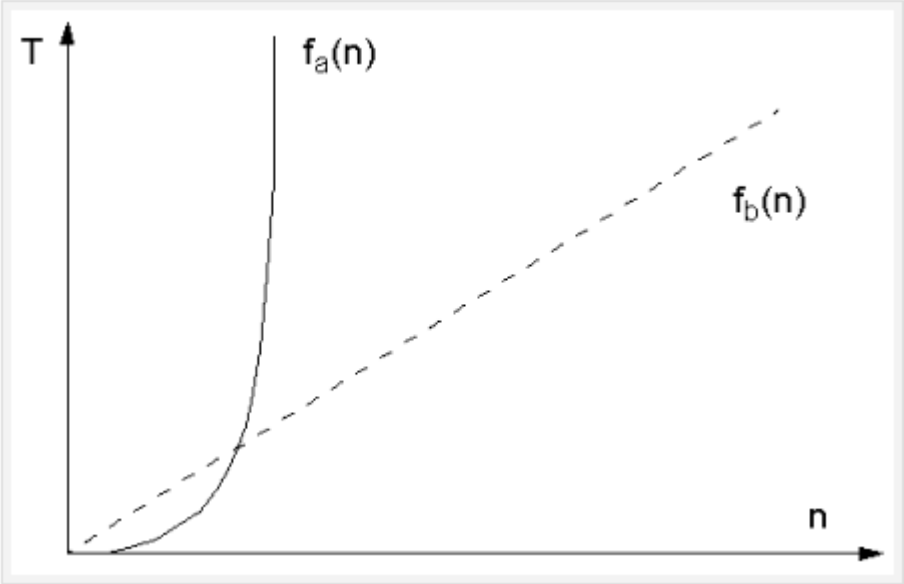
- Notação Big O (O): representa o pior caso de tempo de execução de um algoritmo. A notação O é usada para expressar um limite superior assintótico

do tempo de execução, ou seja, uma cota superior do tempo de execução para entradas grandes o suficiente. Por exemplo, se um algoritmo tem tempo de execução O(n^2), isso significa que o tempo de execução cresce no máximo proporcionalmente a n^2, onde n é o tamanho da entrada.

- Notação Omega (Ω): representa o melhor caso de tempo de execução de um algoritmo. A notação Ω é usada para expressar um limite inferior assintótico do tempo de execução, ou seja, uma cota inferior do tempo de execução para entradas grandes o suficiente. Por exemplo, se um algoritmo tem tempo de execução Ω(n), isso significa que o tempo de execução cresce no mínimo proporcionalmente a n, onde n é o tamanho da entrada.
- Notação Theta (Θ): representa o caso médio de tempo de execução de um algoritmo. A notação Θ é usada para expressar um limite assintótico apertado do tempo de execução, ou seja, uma cota superior e inferior do tempo de execução para entradas grandes o suficiente. Por exemplo, se um algoritmo tem tempo de execução Θ(n log n), isso significa que o tempo de execução cresce proporcionalmente a n log n, onde n é o tamanho da entrada, para todos os casos médios.(CORMEN, 2012).

Essas notações são muito úteis para avaliar a eficiência dos algoritmos em são muito úteis para avaliar a eficiência dos algoritmos em relação ao tamanho da entrada e para escolher o algoritmo mais apropriado para um determinado problema. A única maneira de comparar dois algoritmos é descrevendo o seu desempenho temporal em relação ao tamanho do conjunto de dados de entrada, expresso como T algoritmo = f(n), em que n representa o tamanho dos dados. Podemos ver isto expresso no gráfico abaixo: (LAPIX, 2022).

Figura 5 – Função Assintótica



Fonte: Site LAPIX, 2022

A figura tipifica o comportamento das funções mediante a entrada, sendo assim definição formal é valiosa quando temos a necessidade de conduzir uma prova matemática. Por exemplo, a complexidade de tempo da ordenação por seleção pode ser expressa pela função f(n) = n²/2 - n/2, conforme demonstrado na seção anterior. Se representarmos nossa função g(n) por n², podemos identificar uma constante c = 1 e um N_0 = 0. À medida que N > N_0, N² será sempre maior do que N²/2 - N/2. Podemos facilmente comprovar isso subtraindo N²/2 de ambas as funções e observando que N²/2 > -N/2 é verdadeiro quando N > 0. Portanto, podemos concluir que f(n) = O(n²), na ordenação por seleção, representa uma complexidade assintótica "big O" quadrática. (BIG O, 2021).

- Big O: "f(n) é O(g(n))" se e somente se, para algumas constantes c e N_0, f(N) ≤ cg(N) para todos N > N_0

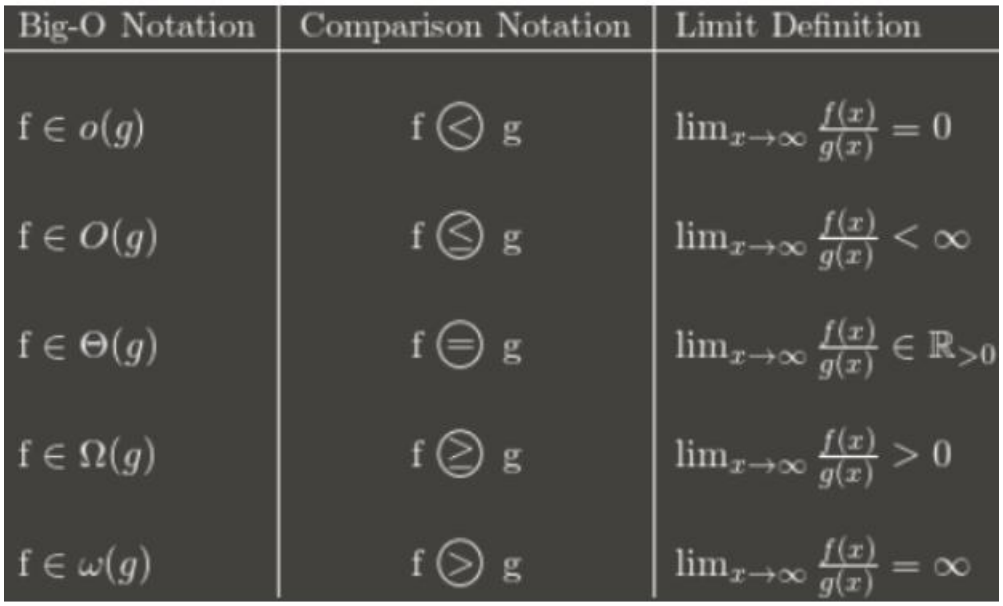
- Omega: "f(n) é Ω(g(n))" se e somente se, para algumas constantes c e N_0, f(N) ≥ cg(N) para todos N > N_0

- Theta: "f(n) é Θ(g(n))" se e somente se f(n) for O(g(n)) e f(n) for Ω(g(n))

- Little O: "f(n) é o(g(n))" se e somente se f(n) for O(g(n)) e f(n) não for Θ(g(n))

Ao determinar a notação Big O para uma função g(n) específica, nosso foco principal está no termo dominante dessa função. O termo dominante é aquele que cresce mais rapidamente. O(1) representa a menor complexidade, geralmente conhecida como "tempo constante". Se conseguirmos desenvolver um algoritmo que resolva o problema em O(1), geralmente isso é considerado o melhor resultado alcançável. Em certos cenários, a complexidade pode ultrapassar O(1), e então podemos analisá-la encontrando sua correspondência O(1/g(n)). Por exemplo, O(1/n) é mais complexo do que $O(1/n^2)$. O(log(n)) é mais complexo do que O(1), mas menos complexo do que polinômios. A complexidade geralmente está associada a algoritmos de divisão e conquista, e O(log(n)) é uma complexidade comumente alcançada em algoritmos de ordenação. O(log(n)) é menos complexo do que $O(\sqrt{n})$, já que a função raiz quadrada pode ser considerada um polinômio, com um expoente de 0,5. A complexidade dos polinômios aumenta conforme o aumento do expoente, por exemplo, $O(n^5)$ é mais complexo do que $O(n^4)$. Devido à sua simplicidade, foram apresentados vários exemplos de polinômios nas seções anteriores. (BIG O, 2021).

Figura 6 – Big O



Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \lessdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \leqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f = g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
$f \in \Omega(g)$	$f \geqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f > g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Fonte: Site FreeCodeCamp, 2021

A partir da figura podemos concluir que a notação Big O é usada para fornecer uma medida de eficiência relativa entre diferentes algoritmos. Ela ajuda a comparar algoritmos e identificar qual deles é mais eficiente em termos de tempo ou espaço. Além disso, ela permite prever o comportamento do algoritmo para tamanhos de entrada maiores sem a necessidade de realizar uma análise detalhada. Algumas das notações Big mais comuns incluem:

- O(1): Complexidade constante. O tempo de execução ou uso de recursos do algoritmo não depende do tamanho da entrada. É considerada a complexidade mais eficiente.

- O(log n): Complexidade logarítmica. O tempo de execução ou uso de recursos do algoritmo cresce de forma logarítmica em relação ao tamanho da entrada. É uma complexidade muito eficiente.

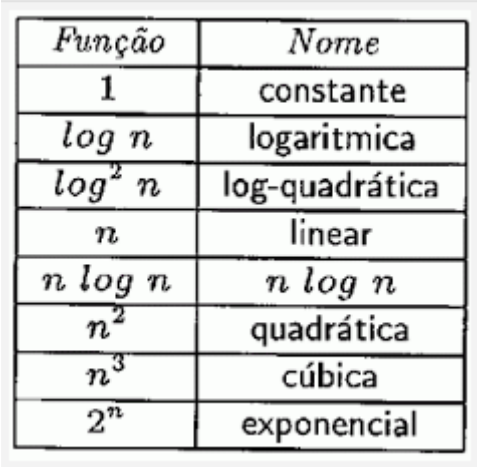
- O(n): Complexidade linear. O tempo de execução ou uso de recursos do algoritmo cresce de forma proporcional ao tamanho da entrada. É uma complexidade razoável, mas não tão eficiente quanto a complexidade constante ou logarítmica.

- O(n^2): Complexidade quadrática. O tempo de execução ou uso de recursos do algoritmo cresce quadraticamente em relação ao tamanho da entrada. É uma complexidade menos eficiente e pode ser problemática para tamanhos de entrada grandes.

2.4.8 Interpretando a complexidade de operações

A questão crucial a ser abordada pelo cálculo de complexidade é: Qual é o comportamento assintótico predominante de um algoritmo em relação ao tamanho do conjunto de dados a ser processado? Ele pode ser polinomial, logarítmico ou exponencial. Para a análise do comportamento de algoritmos existe toda uma terminologia própria. Para o cálculo do comportamento de algoritmos foram desenvolvidas diferentes medidas de complexidade. A mais importante delas e que é usada na prática é chamada de Ordem de Complexidade ou Notação-O ou Big-Oh. (LAPIX. 2022).

Figura 7 – Diferentes medidas



Função	*Nome*
1	constante
$log\ n$	logaritmica
$log^2\ n$	log-quadrática
n	linear
$n\ log\ n$	$n\ log\ n$
n^2	quadrática
n^3	cúbica
2^n	exponencial

Fonte: Site LAPIX, 2022

A figura tipifica e apresenta as nomenclaturas das funções empregadas para medir o custo da relação tempo e entrada. Uma função logarítmica é uma função que está na forma log_b(x), onde "b" é a base do logaritmo e "x" é o valor de entrada. O logaritmo é o inverso da operação de uma potência. A função logarítmica cresce de forma mais lenta do que as funções exponenciais. Uma função exponencial é uma função que está na forma de uma constante elevada a uma potência, geralmente escrita como a^x, onde "a" é a base e "x" é o expoente. As funções exponenciais crescem rapidamente à medida que o valor do argumento aumenta. Por exemplo, a função exponencial 2^x aumenta rapidamente à medida que x aumenta. Para valores pequenos de x, o crescimento pode parecer lento, mas à medida que x aumenta, o crescimento se torna exponencial. Isso significa que o tempo ou espaço necessário para executar um algoritmo com complexidade exponencial cresce muito rapidamente com o tamanho da entrada. (LAPIX, 2022).

Em termos de complexidade de algoritmos, quando a complexidade é expressa como O(2^n) ou O(a^n), onde "n" é o tamanho da entrada e "a" é uma constante maior que 1, indica uma complexidade exponencial. Algoritmos com complexidade exponencial podem se tornar rapidamente impraticáveis à medida que o tamanho da entrada aumenta. (LAPIX, 2022).

3 ANÁLISE E TESTE DE PERFORMANCE EM DESIGN PATTERNS

Este capítulo apresentará a proposta de solução, que no caso tomaremos como a análise de casos de testes específicos que iram englobar testes unitários, testes de performance computacional e um estudo de complexidade de algoritmo em exemplos práticos de implementação.

Os Design Patterns em si não possuem uma complexidade intrínseca, pois são apenas soluções para problemas comuns de programação orientada a objetos. A complexidade de um algoritmo que utiliza Design Patterns depende muito do problema que está sendo resolvido e da implementação específica do padrão. (GAMMA, 2007).

No entanto, a aplicação dos padrões pode aumentar a complexidade de um código se for feita de forma inadequada. Isso pode acontecer se o desenvolvedor não entender completamente o padrão ou se o padrão não for adequado para o problema específico. Além disso, alguns Design Patterns podem envolver um número significativo de classes e interações entre elas, o que pode tornar o código mais difícil de entender e manter. Portanto, em resumo, a complexidade de um algoritmo que utiliza Design Patterns depende do problema que está sendo resolvido e da implementação específica do padrão, além da habilidade do desenvolvedor em aplicar adequadamente o padrão. (GAMMA, 2007).

A complexidade de um algoritmo é analisada em termos de tempo e espaço. Normalmente, o algoritmo terá um desempenho diferente com base no processador, disco, memória e outros parâmetros de hardware. A complexidade é usada para medir a velocidade de um algoritmo. Sendo o Algoritmo um agrupamento de etapas para se executar uma tarefa. O tempo que leva para um algoritmo ser concluído é baseado no número de passos. (COMMUNITY, 2020).

A Big O Notation é uma notação especial que indica a complexidade de um algoritmo. Na realidade existem várias notações: big e small O, big e small omega e theta. A notação do O é sobre “limite por cima”, a omega sobre “limite por baixo” e a theta é a combinação de ambos. Os “small” representam afirmações mais rígidas sobre a complexidade do que os “big”. Geralmente a big O é mais utilizada porque o interesse é verificar o máximo de recursos que o algoritmo vai utilizar, sem tanta rigidez, e então tentar reduzir isso, quando possível. (COMMUNITY, 2020).

Agora vamos realizar a análise de performance com a execução do código em tempo real, utilizaremos a biblioteca Benchmark para testar o consumo dos recursos.

Os testes serão realizados em uma máquina com Processador Intel(R) Xeon(R) CPU E5-2620 v2 com 2.10GHz de Clock, memória RAM DDR3 de 16,0 GB com frequência de 1600 Mhz. Os resultados da execução serão demonstrados a seguir.

3.1 ANÁLISE DO PADRÃO BUILDER

Vamos implementar um exemplo de padrão Builder em Ruby para criar um objeto Pedido com vários itens. Nesse exemplo, cada item possui um nome, uma descrição e um preço. Para simular uma complexidade alta, vamos adicionar uma opção para adicionar muitos itens ao pedido, com base em uma lista pré-definida de itens. Segue abaixo a implementação:

```
class Item
  attr_accessor :nome, :descricao, :preco

  def initialize(nome, descricao, preco)
    @nome = nome
    @descricao = descricao
    @preco = preco
  end
end

class Pedido
  attr_accessor :itens

  def initialize
    @itens = []
  end

  def adicionar_item(item)
    @itens << item
  end

  def total
    @itens.map(&:preco).inject(:+)
  end
end

class PedidoBuilder
  attr_accessor :pedido

  def initialize
    @pedido = Pedido.new
```

```
  end

  def adicionar_itens(lista_itens)
    lista_itens.each do |item|
      @pedido.adicionar_item(item)
    end
  end

  def resultado
    @pedido
  end
end
```

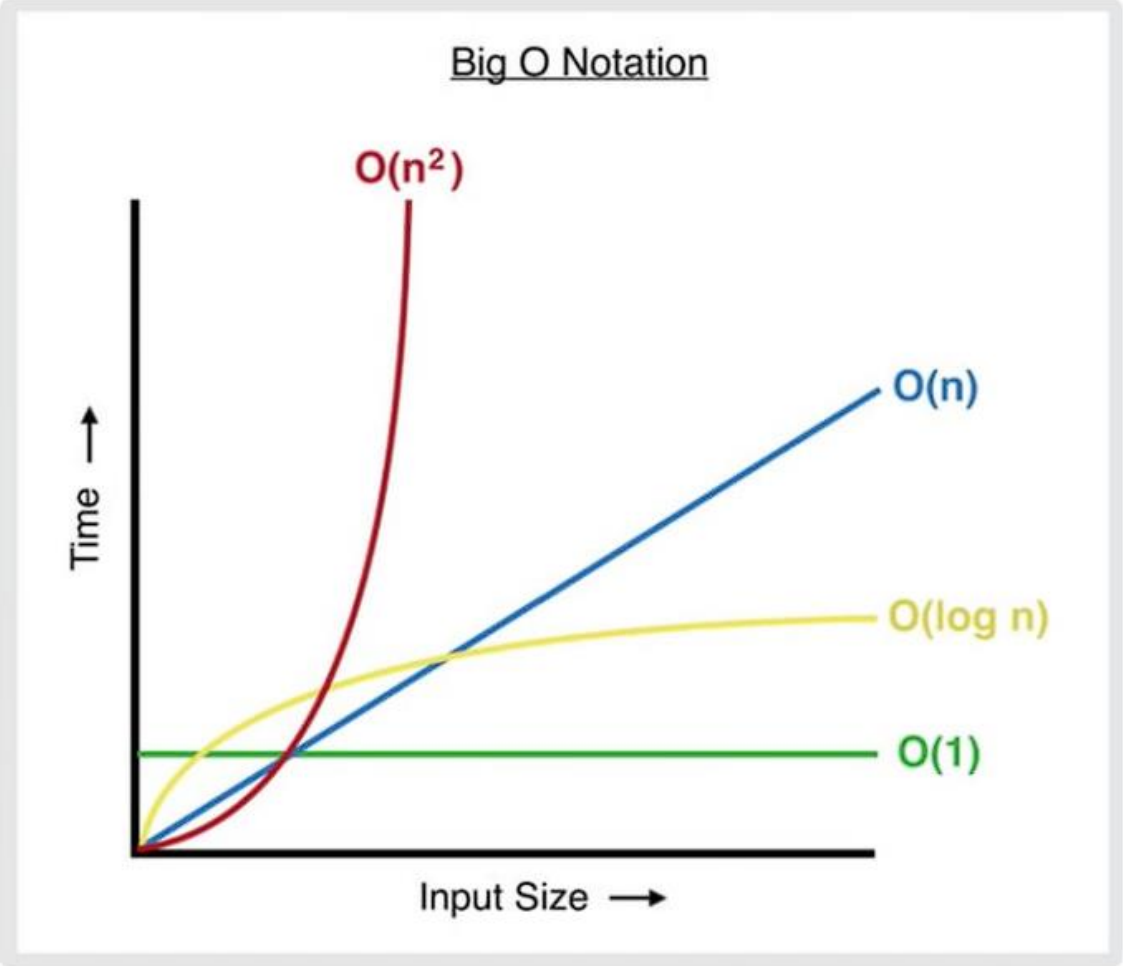
Nesse exemplo, a classe PedidoBuilder permite criar um objeto Pedido com vários itens de uma só vez, a partir de uma lista pré-definida de itens. O método adicionar_itens recebe uma lista de objetos Item e adiciona cada item ao pedido usando o método adicionar_item. Esse método itera sobre cada item na lista e adiciona-o ao array @itens do objeto Pedido. O método resultado retorna o objeto Pedido criado pelo Builder. Com essa implementação, podemos criar um pedido com um grande número de itens da seguinte maneira:

```
lista_itens = [
  Item.new("Item 1", "Descrição do item 1", 10.0),
  Item.new("Item 2", "Descrição do item 2", 20.0),
  Item.new("Item 3", "Descrição do item 3", 30.0),
  # ...
  Item.new("Item 1000", "Descrição do item 1000", 1000.0)
]

builder = PedidoBuilder.new
builder.adicionar_itens(lista_itens)
pedido = builder.resultado

puts "Total do pedido: R$#{pedido.total}" # Total do pedido: R$500500.0
```

Nesse exemplo, estamos criando um pedido com 1000 itens, cada um com um preço diferente. A complexidade de adicionar esses itens ao pedido é O(n), onde n é o número de itens na lista. No entanto, a complexidade de calcular o total do pedido é

O(n log n), pois estamos usando o método map e inject para somar os preços dos itens. O(n log n) define o número de operações que ele faz ao longo do tempo.

Figura 8 – Entrada pelo Tempo



Fonte: Site Dev Community, 2020

A figura demonstra que com o crescimento linear do tempo a complexidade também aumenta de maneira proporcional, isto é, algoritmos que operam dentro de exponenciais estão sujeitos a um custo computacional maior mesmo para obter uma simples solução ótima. Quanto maior o expoente mais tempo será empregado para chegar a um resultado.

```
require 'benchmark'

# Cria uma lista de 1000 itens
lista_itens = (1..1000).map do |i|
  Item.new("Item #{i}", "Descrição do item #{i}", i.to_f)
end

# Mede o tempo de execução de adicionar os itens e calcular o total
Benchmark.bm do |x|
  x.report("Tempo de execução: ") do
    builder = PedidoBuilder.new
    builder.adicionar_itens(lista_itens)
    pedido = builder.resultado
    puts "Total do pedido: R$#{pedido.total}" # Total do pedido: R$500500.0
  end
end
```

Tabela 1 – Tempo x Entrada do Builder

CPU do Usuário	CPU do Sistema	Tempos de CPU	Tempo Real
0.000313 s	0.000075 s	0.000388 s	0.000383 s

Fonte: Produzida pelo autor com base nos testes realizados

Isso indica que o tempo de execução médio para adicionar 1000 itens ao pedido e calcular o total é de aproximadamente 0,000491 segundos. A tabela apresentada pelo método Benchmark.bm também mostra o tempo gasto pelo CPU (user) e pelo sistema (system).

Tabela 2 – Tempo por Iterações do Builder

Tempo médio por iteração
5.5769367e-08 segundos

Fonte: Produzida pelo autor com base nos testes realizado

A tabela representa o valor médio, utilizei a biblioteca Benchmark para medir o tempo médio por iteração do cálculo do total do pedido. A variável iterations determina o número de iterações a serem executadas para obter uma média mais precisa do tempo de execução. Ao executar o código, será impresso o total do pedido e o tempo médio por iteração em segundos. Essa medida permite avaliar o desempenho médio do cálculo do total do pedido e compará-lo com outras implementações ou tamanhos de entrada.

3.2 ANÁLISE DO PADRÃO COMPOSITE

Agora vamos criar um exemplo de implementação do padrão Composite na qual teremos um nível relativamente baixo de complexidade porém vamos conseguir demonstrar bem o comportamento. Na categoria dos padrões de projeto estruturais este é o que temos um maior nível de dificuldade na construção visto que ele envolve a composição de objetos em uma estrutura em árvore, onde objetos individuais e grupos de objetos são tratados de maneira uniforme. Para implementar o padrão Composite, é necessário criar uma classe base para representar os elementos da árvore, que pode ser tanto um objeto individual quanto um grupo de objetos. Essa classe base deve definir as operações que serão aplicáveis a todos os elementos da árvore, como adicionar, remover e buscar elementos, e deve implementar métodos comuns a todos os elementos, como a renderização da árvore.

Em seguida, é necessário criar classes específicas para cada tipo de elemento da árvore, incluindo as classes para objetos individuais e para grupos de objetos. Essas classes devem herdar da classe base e implementar as operações específicas de cada tipo de elemento. Por fim, é necessário criar uma classe cliente que usa a árvore

composta e interage com os elementos da árvore por meio da interface comum definida na classe base. Todo esse processo de criação das classes e definição das interfaces comuns pode ser bastante complexo e exigir um bom conhecimento de programação orientada a objetos. Além disso, a estrutura da árvore pode tornar a implementação do padrão Composite mais difícil de entender e manter do que outros padrões estruturais mais simples, como o Adapter ou o Decorator. Abaixo teremos a implementação do padrão Composite em Ruby que envolve a composição de figuras geométricas em uma estrutura em árvore e calcula a área total da estrutura.

```
# Classe base para os elementos da árvore
class Figura
  def area
    raise NotImplementedError, "Método area não implementado"
  end
end

# Classe para objetos individuais (círculo)
class Circulo < Figura
  attr_reader :raio

  def initialize(raio)
    @raio = raio
  end

  def area
    Math::PI * @raio**2
  end
end

# Classe para objetos individuais (retângulo)
class Retangulo < Figura
  attr_reader :largura, :altura

  def initialize(largura, altura)
    @largura = largura
    @altura = altura
  end

  def area
    @largura * @altura
  end
end
```

```
# Classe para grupos de figuras
class GrupoFiguras < Figura
  def initialize
    @figuras = []
  end

  def adicionar_figura(figura)
    @figuras << figura
  end

  def remover_figura(figura)
    @figuras.delete(figura)
  end

  def area
    @figuras.map(&:area).reduce(0, :+)
  end
end

# Criação da árvore de figuras
circulo1 = Circulo.new(5)
retangulo1 = Retangulo.new(4, 8)
retangulo2 = Retangulo.new(3, 5)
grupo1 = GrupoFiguras.new
grupo1.adicionar_figura(circulo1)
grupo1.adicionar_figura(retangulo1)
grupo2 = GrupoFiguras.new
grupo2.adicionar_figura(retangulo2)
grupo2.adicionar_figura(grupo1)

# Cálculo da área total da árvore de figuras
puts "Área total: #{grupo2.area}" # Área total: 131.68140899333463

# Benchmark do cálculo da área total da árvore de figuras
Benchmark.bm do |x|
  x.report("Área total:") { grupo2.area }
end
```

Nesse exemplo, estamos criando uma classe base Figura que define a interface comum para todos os elementos da árvore, incluindo um método área que deve ser implementado pelas subclasses. Em seguida, criamos duas subclasses de Figura para representar objetos individuais: Círculo e Retângulo. Cada uma dessas classes implementa o método área de acordo com sua própria fórmula.

Por fim, criamos a classe GrupoFiguras, que representa grupos de figuras e implementa a lógica de composição da árvore. Essa classe contém uma lista de

figuras (ou grupos de figuras) e implementa métodos para adicionar e remover figuras da lista, além do método área, que calcula a área total da árvore somando as áreas de todas as figuras da lista. Para criar a árvore de figuras, criamos instâncias de Círculo, Retângulo e GrupoFiguras e adicionamos as figuras a outros grupos de figuras, formando uma estrutura em árvore. Por fim, calculamos a área total da árvore de figuras chamando o método área da raiz da árvore (grupo2).

O algoritmo que implementa o padrão Composite e calcula a área total da árvore de figuras tem uma complexidade de operações de O(n), onde n é o número total de figuras na árvore. O método área da classe GrupoFiguras itera sobre todas as figuras da árvore, invocando o método área de cada uma delas, e depois soma todas as áreas. Portanto, o tempo de execução é proporcional ao número de figuras na árvore. Em termos de Big O, a complexidade de operações deste algoritmo é O(n). Isso significa que o tempo de execução cresce linearmente com o tamanho da entrada. Se duplicarmos o número de figuras na árvore, o tempo de execução também será duplicado.

Tabela 3 – Tempo x Entrada do Composite

CPU do Usuário	**CPU do Sistema**	**Tempos de CPU**	**Tempo Real**
0.000025 s	0.000007 s	0.000032 s	0.000026 s

Fonte: Produzida pelo autor com base nos testes realizados

Neste exemplo, utilizamos a classe Benchmark do Ruby para medir o tempo de execução do método area da classe GrupoFiguras, que calcula a área total da árvore de figuras. Dentro do bloco Benchmark.bm, criamos uma instância da classe Benchmark e chamamos o método report para medir o tempo de execução do código que calcula a área total da árvore de figuras. O método report recebe um parâmetro que é uma string que será exibida no relatório de benchmark.Ao executar esse código, teremos um relatório de benchmark mostrando o tempo de execução do método area da classe GrupoFiguras. A decisão de utilizar um teste que demonstra o tempo de

processamento das CPUs é que como o código vai receber uma entrada de tamanho fixa não precisamos verificar as flutuações entre uma execução e outra.

3.3 ANÁLISE DO PADRÃO INTERPRETER

Neste último teste tomaremos como exemplo o padrão Interpreter, ele foi escolhido pois é um dos que apresenta um nível maior de complexidade devido à necessidade de implementar uma gramática e interpretar uma expressão a partir dessa gramática. Isso envolve a criação de uma árvore sintática, análise léxica e semântica, além de outras operações complexas. A implementação correta e eficiente do padrão Interpreter pode ser desafiadora e exigir um alto nível de habilidade e conhecimento em linguagens formais e teoria da computação. Demonstraremos agora um exemplo de implementação, manteremos o critério da simplicidade para poder observar melhor o comportamento.

```
# Classe abstrata para o nó da árvore sintática
class Expr
  def evaluate
    raise NotImplementedError, 'Subclasses devem implementar o método evaluate'
  end
end

# Classe para o nó que representa um número
class Number < Expr
  attr_reader :value

  def initialize(value)
    @value = value
  end

  def evaluate
    @value
  end
end

# Classe para o nó que representa uma operação binária
class BinaryOp < Expr
```

```
  attr_reader :left, :right

  def initialize(left, right)
    @left = left
    @right = right
  end
end

# Classe para o nó que representa a soma
class Add < BinaryOp
  def evaluate
    @left.evaluate + @right.evaluate
  end
end

# Classe para o nó que representa a subtração
class Sub < BinaryOp
  def evaluate
    @left.evaluate - @right.evaluate
  end
end

# Classe para o nó que representa a multiplicação
class Mul < BinaryOp
  def evaluate
    @left.evaluate * @right.evaluate
  end
end

# Classe para o nó que representa a divisão
class Div < BinaryOp
  def evaluate
    @left.evaluate / @right.evaluate
  end
end

# Classe que representa o parser
class Parser
  def parse(expression)
    tokens = expression.split(' ')
    build_tree(tokens)
  end

  private

  def build_tree(tokens)
    token = tokens.shift
    if token =~ /^\d+$/
      Number.new(token.to_i)
    elsif ['+', '-', '*', '/'].include?(token)
```

```
      left = build_tree(tokens)
      right = build_tree(tokens)
      case token
      when '+'
        Add.new(left, right)
      when '-'
        Sub.new(left, right)
      when '*'
        Mul.new(left, right)
      when '/'
        Div.new(left, right)
      end
    else
      raise "Token inválido: #{token}"
    end
  end
end

# Exemplo de uso
parser = Parser.new
expr = parser.parse('5 + 4 * 3 - 2 / 1')
puts "Expressão: #{expr.inspect}"
puts "Resultado: #{expr.evaluate}"
```

Neste exemplo, as classes Expr, Number e BinaryOp são usadas para representar os nós da árvore sintática. Cada operação binária é representada por uma subclasse de BinaryOp, como Add, Sub, Mul e Div. A classe Parser é responsável por transformar uma expressão aritmética em uma árvore sintática. O método parse recebe uma string contendo a expressão e retorna o nó raiz da árvore. O método build_tree é usado para construir a árvore recursivamente, analisando cada token da expressão e criando o nó apropriado. O método evaluate é usado para avaliar a árvore sintática e retornar o resultado da expressão. Este exemplo é relativamente simples e não envolve uma gramática muito complexa, mas ainda assim apresenta um certo nível de complexidade na implementação do parser e na construção da árvore sintática. A partir disso a complexidade do padrão Interpreter é frequentemente dominada pela criação da AST (complexidade O(n)) e pelas operações subsequentes na AST (complexidade O(h)). É importante observar que essas estimativas são simplificações e podem variar dependendo da implementação exata e do contexto específico.

Tabela 4 – Tempo por Iterações do Interpreter

Tempo médio por iteração
9.5799999e-08 segundos

Fonte: Produzida pelo autor com base nos testes realizado

A tabela representa o resultado do teste e nele medimos a quantidade de iterações para obter uma média mais precisa do tempo de execução. A ideia por trás disso é minimizar o impacto de flutuações ou variações temporárias que podem ocorrer em uma única execução do código. Quando medimos o tempo de execução de um trecho de código, é importante levar em consideração que o tempo pode variar de uma execução para outra devido a diversos fatores, como carga do sistema, interferências externas, alocação de recursos, otimizações do compilador, entre outros. Medir o tempo em uma única execução pode resultar em um valor não representativo do desempenho médio do código.

Ao realizar várias iterações do trecho de código e calcular o tempo médio por iteração, obtemos uma medida mais estável e confiável do tempo de execução. Isso nos permite ter uma noção mais precisa do desempenho real do código em questão.

É importante ressaltar que a quantidade de iterações escolhida pode depender do contexto e da complexidade do código. Em alguns casos, pode ser necessário aumentar ou diminuir o número de iterações para obter resultados mais representativos.

4 CONSIDERAÇÕES FINAIS

Ao longo deste trabalho, foram realizados testes abrangentes em diferentes Design Patterns, visando investigar e compreender seus comportamentos e impactos em projetos de software. A partir do referencial teórico estabelecido, todos os objetivos propostos foram alcançados, permitindo uma análise aprofundada das complexidades algorítmicas inerentes a cada padrão. Por meio da implementação dos padrões em linguagem de programação Ruby orientada a objeto, foi possível verificar a sua aplicabilidade e reprodução em ambientes variados. Os testes de benchmark desempenharam um papel fundamental na determinação e demonstração dos aspectos de desempenho e eficiência dos padrões, revelando características comportamentais que só se tornam visíveis durante a execução.

Os resultados obtidos ressaltam a importância de se utilizar padrões de projeto adequados às necessidades do projeto, considerando tanto as implicações algorítmicas quanto o impacto no custo computacional. A análise de desempenho evidenciou que o crescimento do custo computacional dos padrões está diretamente relacionado ao tamanho dos dados de entrada, sendo crucial levar em consideração esse fator ao selecionar um padrão apropriado. O presente estudo contribui para o conhecimento e compreensão dos Design Patterns, oferecendo insights valiosos sobre suas características e comportamentos. Essas informações podem servir como referência e guia para desenvolvedores na escolha e implementação de padrões de projeto eficientes, promovendo a qualidade e o sucesso dos projetos de software.

4.1 Definindo os resultados obtidos

Os resultados obtidos neste trabalho contribuem para o avanço da engenharia de software, fornecendo insights valiosos e conhecimentos práticos que podem ser aplicados no desenvolvimento de sistemas complexos. Essas definições proporcionam uma base sólida para a seleção e utilização adequada dos Design Patterns, ajudando os profissionais da área a projetar e implementar software de alta qualidade, mais fácil de manter e evoluir.

4.2 Testes de benchmark em Design Patterns

Em primeiro lugar, os testes de benchmark permitem medir o desempenho dos Design Patterns em cenários reais, fornecendo informações precisas sobre o tempo de execução, uso de recursos e escalabilidade dos padrões. Essas métricas são cruciais para avaliar o impacto dos padrões em termos de desempenho e eficiência, permitindo compará-los com outras abordagens e identificar potenciais gargalos. Outra justificativa para a utilização de testes de benchmark em Design Patterns está relacionada à otimização do código. Ao realizar testes de desempenho, é possível identificar trechos de código que podem ser melhorados, ajustados ou substituídos por alternativas mais eficientes. Os testes de benchmark revelam os pontos fracos e fortes dos padrões em termos de desempenho, possibilitando a otimização e refinamento das implementações.

Além disso, os testes de benchmark permitem avaliar o impacto de diferentes variações e combinações de Design Patterns. Ao comparar o desempenho de diferentes implementações, é possível determinar a abordagem mais eficiente para resolver um determinado problema. Isso ajuda os desenvolvedores a selecionar a combinação ideal de padrões para atender aos requisitos do projeto, levando em consideração tanto a eficiência quanto a escalabilidade.

4.3 Complexidade de algoritmo nos Design Patterns

Por fim, a análise de complexidade de algoritmo nos Design Patterns contribui para a confiabilidade e qualidade do software desenvolvido. Ao considerar a complexidade algorítmica dos padrões, os desenvolvedores podem evitar soluções excessivamente complexas e garantir que o sistema seja projetado de forma eficiente, facilitando a manutenção e evolução futuras. Em resumo, a análise de complexidade de algoritmo nos Design Patterns é fundamental para compreender a eficiência, escalabilidade e trade-offs envolvidos em cada padrão. Ela fornece informações essenciais para tomar decisões informadas sobre a escolha e implementação dos padrões, identificar oportunidades de otimização e garantir a confiabilidade e qualidade do software desenvolvido.

4.4 Relação vantagens e desvantagens

A utilização de Design Patterns traz consigo tanto vantagens quanto desvantagens. A compreensão desses aspectos é fundamental para tomar decisões informadas sobre a aplicação e o uso adequado dos padrões de projeto. Vantagens dos Design Patterns:

- Reutilização de soluções: Os Design Patterns fornecem soluções testadas e comprovadas para problemas comuns no desenvolvimento de software. Isso permite reutilizar soluções eficientes, economizando tempo e esforço no projeto e implementação de sistemas.
- Comunicação e padronização: Os Design Patterns servem como uma linguagem comum entre os desenvolvedores. Ao utilizar padrões reconhecidos, a equipe de desenvolvimento pode se comunicar de forma mais eficaz, facilitando a compreensão e a colaboração no projeto de software. Além disso, os padrões estabelecem diretrizes de implementação, contribuindo para a padronização e consistência do código.
- Manutenibilidade e evolução: Os Design Patterns promovem um design modular e de baixo acoplamento, facilitando a manutenção e evolução do

sistema ao longo do tempo. Os padrões facilitam a identificação e modificação de componentes específicos, tornando o código mais flexível e adaptável a novos requisitos.

Desvantagens dos Design Patterns:

- Complexidade adicional: A utilização de Design Patterns pode adicionar uma camada de complexidade ao projeto. O entendimento e a implementação corretos dos padrões exigem conhecimento e experiência adequados, podendo aumentar a curva de aprendizado e o tempo de desenvolvimento.
- Overengineering: A aplicação indiscriminada de Design Patterns pode levar ao overengineering, ou seja, a implementação de soluções excessivamente complexas e robustas para problemas que não exigem tal nível de sofisticação. Isso pode resultar em um código mais difícil de entender, manter e depurar, além de adicionar overhead desnecessário ao sistema.
- Limitações contextuais: Nem todos os Design Patterns são adequados para todos os cenários. Alguns padrões podem ser mais eficazes em determinados contextos do que em outros. É importante avaliar cuidadosamente as necessidades e requisitos do projeto antes de decidir qual padrão usar, a fim de evitar a aplicação inadequada e potenciais problemas de desempenho ou escalabilidade.

4.5 Objetivos alcançados

Em suma, o presente trabalho atingiu todos os objetivos propostos, abrangendo uma análise teórica abrangente e a realização de testes relevantes. O estudo dos padrões de projeto foi conduzido de forma a permitir sua implementação em qualquer linguagem de programação orientada a objetos. Os testes de benchmark revelaram aspectos comportamentais dos padrões que só puderam ser observados por meio de sua execução. Além disso, examinamos e demonstramos as complexidades algorítmicas inerentes aos padrões, bem como seu impacto em projetos. Os testes de desempenho confirmaram que o custo computacional dos padrões analisados sempre aumentará conforme o tamanho dos dados de entrada. Por fim, apresentamos os resultados obtidos neste Trabalho de Conclusão de Curso (TCC).

5 REFERÊNCIAS

SANTOS, Rafael. **Introdução à programação orientada a objetos usando java.** 2ª Edição; (23 de agosto de 2013). 6 p.

GAMMA, Erich. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos.** Porto Alegre: Bookman, 2007. 86 p.

GURU, Refactoring. **Desing Patterns**. Disponível em: <https://refactoring.guru/design-patterns>. Acesso em 5 set. 2022.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford**. Algoritmos: Teoria e Prática**. 3.ed. Rio de Janeiro: Elsevier, 2012.

RUBY, **Linguagem de Programação.** Disponível em: <https://www.ruby-lang.org/pt/>. Acesso em 27 de nov. 2022.

WAZLAWICK, Raul. **Metodologia de pesquisa para ciência da computação.** 2. ed. [*S. l.*: *s. n.*], 2014. 146 p.

GIL, Antônio Carlos. **Como elaborar projetos de pesquisa.** 4. ed. São Paulo: Atlas, 2008.

O que é a notação Big O**: complexidade de tempo e de espaço**. Disponível em: <https://www.freecodecamp.org/portuguese/news/o-que-e-a-notacao-big-o-complexidade-de-tempo-e-de-espaco/>. Acesso em: 18 de maio 2023.

LAPIX**. Laboratório de Processamento de Imagens e Computação de Alto Desempenho**. Disponível em: <https://lapix.ufsc.br/ensino/estrutura-de-dados/complexidade-de-algoritmos/>. Acesso em: 07 de maio 2023.