**Aluno(a): Hugo Rodrigues Torquato**
**Orientador(a): Heinrich Da Solidade Santos**

**Curso:** MBA em Engenharia de Software

# Analisador de software C++ baseados em programação orientado a objetos

## Introdução

No contexto de programação orientada a objetos, as abstrações são fundamentais para organização e atribuição de responsabilidades. A compreensão da lógica é simples quando os assuntos são comumente conhecidos e/ou com poucos níveis de abstrações. Entretanto, o código se torna mais complexo e, por consequência, demanda mais esforço e tempo para compreendê-lo quando se trata de uma base de código extensa, desenvolvida durante vários anos, por vários profissionais e com foco em um conhecimento específico. Em uma ótica empresarial, este cenário implica em maiores investimentos para que novos engenheiros sejam incorporados à equipe por causa da complexidade, e adiciona um nível de abstração extra nas tomadas de decisão relacionadas ao desenvolvimento do *software* por não ter uma visão geral do todo. Sob essa perspectiva, visualizar a organização de uma base de código clara e intuitivamente agrega valor, principalmente, na rotina das empresas desenvolvedoras de *softwares*.

A visualização e organização de informações de um *software* auxilia no seu gerenciamento, o que, de forma secundária, reflete na maior mantenabilidade do produto. Martin (2009) ilustra essa dinâmica ao comparar produtividade pelo tempo de existência do *software*: ao passo que a extensão e complexidade do código aumentam, especialmente sem planejamento, as chances de se transformar em um emaranhado de soluções improvisadas também aumentam. Nesse cenário, o autor argumenta que a produtividade para desenvolver novas funcionalidades e a manutenção do código declina drasticamente a médio e longo prazo, quando não gerenciado corretamente, tornando-se quase impraticável e economicamente inviável.

O ponto de partida para projetos dessa natureza contempla a análise de código fonte, tema de investigação consolidado para diversas áreas na computação: a otimização de compilador, abordado por Dean (1995), que expõe os benefícios ao realizar análises de hierarquia de classes como método para resolver ambiguidades em invocações de funções virtuais; análises de qualidade de *software*, abordado por Ceconello (2016), apresenta alguns dos problemas recorrentes na programação orientada a objetos; e estudos até mesmo no ramo da avaliação do quão eficiente estão os modelos de inteligência artificial baseados em linguagens (LLMs) quando comparado programação funcional com orientada a objetos, abordado por Wang (2024), que evidencia a necessidade de melhorias.

Inúmeras aplicabilidades, soluções e produtos foram propostos com objetivo de aprimorar a experiência durante e após o desenvolvimento de *software*. Silva (2023) detalha os benefícios gerais das ferramentas que podem ser integradas em diversos momentos no ciclo de vida do *software*, desde a etapa de codificação até a validação do produto final. Como exemplos temos: o SonarQube, uma plataforma de análise de código com foco em qualidade e cobertura de testes; o *Clang Static Analyzer,* uma ferramenta de análise nativa do LLVM aplicada ao C e C++; e *AddressSanitizer,* como ferramenta para identificar erros de memória em tempo de execução.

Frente ao exposto, ferramentas de visualização e organização de uma base de código são bastante requisitadas em projetos de *software*. Por consequência, é tema de pesquisas tanto no âmbito acadêmico quanto no desenvolvimento de novos produtos e soluções no setor privado. A literatura relacionada a este projeto de pesquisa está associada com a investigação feita em nível de compiladores e nos ambientes de desenvolvimento integrado (IDEs), com o intuito de auxiliar o programador na escrita e otimização da compilação de código. Entretanto, o mapeamento estruturado para facilitar discussões e a tomada de decisão, não tem sido amplamente abordado e/ou mencionados em relevantes estudos e produtos. Sendo assim, se faz relevante arquitetar uma maneira simples e direta de obter essas informações sobre o código de forma estruturada.

**Objetivo**

Este trabalho tem como objetivo desenvolver uma API capaz de realizar uma análise estática em códigos baseados em programação orientada a objetos, extrair informação sobre a organização das classes e exportar os resultados de forma estruturada.

**Material e Métodos**

O presente trabalho fundamenta-se na análise de código que, pela literatura, pode ser feito tanto por uma abordagem estática ou dinâmica. Um comparativo é apresentado por Silva (2023), que ressalta os aspectos positivos e os reveses de cada estratégia. O autor enfatiza o potencial positivo alcançado quando utilizadas em conjunto. No contexto do projeto de pesquisa proposto, a análise estática revela-se como opção promissora por depender unicamente do código, sem a necessidade de compilá-lo. O que permite uma expansão para demais linguagens de forma mais concisa e avalia a intenção original do desenvolvedor ao examinar o código antes de quaisquer otimizações do compilador.

Tal análise segue um fluxo de desenvolvimento alinhado a implementação de compiladores. O processo inicia-se com o mapeamento do código fonte, evoluindo para a organização das informações em forma de tokens. Esse grupamento constitui a entrada para o processo de *parsing*, que agrupa os tokens em expressões sintáticas mais abrangentes que viabilizam a construção das árvores de sintaxe. Essas, por sua vez, realizam o processo de análise do código de acordo com a especificidade de cada linguagem de programação. As informações necessárias para o proposto projeto de pesquisa serão coletadas nesta etapa e tange, principalmente, a árvore de sintaxe.

Embora existam outras etapas em um projeto de compilador, o resultado produzido por elas não agrega informações úteis a este projeto. Ademais, Nystrom (2020) apresenta uma visão detalhada de todo o processo, explicando a criação de um interpretador. Incluindo as etapas: otimização do código fonte; tradução para linguagem baixo nível mais próxima de máquina e compilação conforme a escolha do ambiente de execução do *software*.

Os autores Nystrom (2020), Aho (2007), e Cooper (2012) serão as referências principais para embasar o tema compiladores. Explorando, em detalhe, as etapas que são importantes para o presente trabalho, temos o mapeamento dos caracteres de um código como a primeira das três etapas. Nesta fase inicial, a performance é crucial, sendo o momento de maior demanda computacional por requerer uma análise sequencial de todos os caracteres dos arquivos de entrada do sistema. Todas as informações têm de ser categorizadas para possibilitar o agrupamento dos caracteres em *tokens*, que constitui uma abstração de informações úteis para o interpretador. Em termos simplificados, é o menor agrupamento de caracteres com representabilidade em um código, sendo tratadas as ambiguidades. A identificação dos *tokens* é realizada, dentre outras maneiras, por meio das expressões regulares, capazes de identificar padrões definidos na gramática léxica da linguagem de programação utilizada no código analisado.

A segunda etapa é o *parser*, que determina se a sequência de *tokens*, gerado pelo mapeamento do código, constitui uma sentença válida e o insere em uma representação, com estrutura aprimorada, que emprega árvores de sintaxe para organizar as expressões lógicas. Obtém, como resultado intermediário, a tradução do código para uma linguagem livre de contextos e simplifica a contextualização do token utilizando de modelos de algoritmos já validados como: *top-down* e *bottom-up*. O *top-down* para correlacionar as entradas dos *tokens* com as produções gramaticais, visando a eficiência ao prever a próxima palavra. O bottom-up opera com os detalhes, a precisa sequência de palavras é mantida em memória até que identificado o respectivo contexto.

Por fim, a etapa de análise de contexto, no âmbito dos compiladores, é responsável por realizar um processamento adicional ao *parser,* identificando erros funcionais uma vez

que o código está organizado de forma estruturada na árvore de sintaxe. No contexto do projeto de pesquisa proposto, essa etapa será explorada de maneira limitada, restringindo na identificação da relação entre classes em um código.

Frente ao contexto apresentado e o panorama funcional esperado para o *software*, é pertinente discutir como será implementado. Em uma visão de alto nivel, considerando o *software* uma caixa preta, tem de receber uma base de código como entrada e retornar uma estrutura de dados no formato .json para utilização posterior. Já em uma visão detalhada do processo, é possível suportar várias linguagens de programação, mapear diversos tipos de *tokens* para cada uma das linguagens e performar diversos tipos de operações dependendo do *token* e da linguagem selecionada. Embora o escopo do projeto de pesquisa seja restringido a entradas com linguagem C++ e somente a estruturas de classes, a escalabilidade de tipos e operações é uma questão sensível caso o objetivo seja expandir o projeto.

Na discussão da implementação da pesquisa, o padrão de projeto *visitor* é um exemplo que aborda a discussão entre inserção de tipos e operações. Iglberger (2022) discorre sobre o tema com análises sobre a escolha do paradigma de programação frente a necessidade de adicionar tipos ou operações. Ao passo que em uma programação procedural seja simples adicionar operações novas, adicionar tipos tem por consequência alterar componentes base do código e propagar a mudança para as demais camadas. Em contrapartida, a programação orientada a objetos simplifica a adição de novos tipos mas complica a inserção de novas operações. O visitor atua, junto a orientação a objetos, para simplificar a inserção de operações ao delegar a implementação da operação para uma classe externa que será, em termos simples, aceita pela classe principal.

O *visitor* convencional transforma o estilo de programação orientado a objetos de simples inserções de tipos para difícil e de difícil inserção de operação para simples, mas existe uma alternativa que simplifica ambas inclusões, o *std::variant* do C++. Tal alternativa reserva em memória o espaço necessário para o maior tipo que o *visitor* pode operar e utiliza desse bloco de memória para definir a ação e a classe que será executada. A viabilidade da utilização do *std::variant* será avaliada e, se promissor, incorporado para o projeto final.

O padrão *visitor* também é abordado por Nystrom (2020) no contexto do *parser,* ressaltando a simplicidade de adicionar novas operações em um conjunto de tipos específicos sem alterar a implementação da classe em questão. Outros padrões de projeto podem e devem ser utilizados, embora a identificação de todos os contextos e detalhes de implementação que justifiquem sua escolha ainda estarem em análise. À medida que

validado os benefícios do padrão para o projeto e com base nas explicações fornecidas por Iglberger (2022), serão implementados.

Das estratégias de implementação para o ferramental a ser utilizado, a linguagem C++ foi escolhida pela eficiência. A estrutura do *software* em uma API RESTful segue a tendência em proporcionar praticidade na utilização, sem a necessidade de transferir e instalar localmente, simplesmente interagir com o serviço e obter a informação desejada. Resultado disponibilizado em formato .json, amplamente adotado em APIs e de simples conversão para diversas aplicações. A biblioteca em C++ utilizada para implementar a API ainda está em análise dos benefícios e reveses de cada uma, dentre as opções temos: *Crow, Pistache, Restbed* e *Cpp-httplib.*

Frente ao projeto proposto, é fundamental associar a implementação de testes ao desenvolvimento do projeto para assegurar a qualidade e confiabilidade do *software*. Atrelado ao projeto de pesquisa é pertinente desenvolver dois tipos de testes: unitários e de integração. Os testes unitários serão utilizados para validação de componentes da implementação, de forma isolada, explorando as particularidades da implementação. O *framework* escolhido para os testes unitários foi o *Google Tests*, amplamente adotado na indústria e com fácil acesso a documentação. A metodologia de implementação dos testes segue o padrão do *Test-Driven Development (TDD)* abordado por Langr (2013), que ilustra os conceitos utilizando o *framework* com C++. Embora fundamentais no desenvolvimento, os testes unitários não abrangem a completude do algoritmo, se faz necessário utilizar dos testes de integração por, em uma visão de alto nível, avaliar se o *software* está comportando da maneira esperada. Este modelo de teste consiste em prover entradas já conhecidas e avaliar o retorno obtido com o esperado para aquela determinada entrada. Essa combinação estratégica de testes explora a granularidade dos testes unitários com a abrangência dos testes de integração para garantir um *software* de melhor qualidade.

Frente a amplitude dos temas abordados, a criação de um repositório para orquestrar todos o desenvolvimento do projeto se faz necessário. Vislumbra-se também a criação de uma rotina de integração constante (CI), para compilar e executar testes sempre que alguma alteração é inserida no repositório. Em um horizonte futuro, também vislumbra a implementação do despacho contínuo (CD) de atualização sempre que tiver uma versão estável. Contudo, este foi um assunto pouco explorado e que requer ser investigado com mais detalhes para definição de como será implementado. Ainda existem questões sem definições precisas mas que já estão mapeadas para implementação como a infraestrutura da hospedagem da API, automatização da compilação em cada mudança no repositório e acionamento da execução dos testes (unitários e de integração) após cada compilação.

## Resultados Esperados

É esperado obter, com a conclusão da pesquisa, uma API capaz de realizar uma análise estática em códigos C++ e retornar um conjunto de dados estruturados em formato .json.

O *software* tem foco na análise estática de hierarquia entre as classes em uma base de código que segue os princípios da programação orientada a objetos. O processo consiste em estruturar informações e relações entre as classes em um grafo para facilitar operações e identificação de padrões.

Ademais, será abordado uma discussão acerca da arquitetura definida para o desenvolvimento do *software*, com foco na explicação dos benefícios oferecidos pelos padrões selecionados frente às necessidades específicas do presente projeto.

## Cronograma de Atividades

| **Atividades planejadas** | **Mês** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **03** | **04** | **05** | **06** | **07** | **08** | **09** | **10** | **11** | **12** |
| Escrita do Projeto de pesquisa | x | | | | | | | | | |
| Entrega do Projeto de Pesquisa | | x | | | | | | | | |
| Organização do repositório GIT | | x | | | | | | | | |
| Implementação esqueleto API | | x | x | | | | | | | |
| Implementação do mapeamento do código fonte | | | x | x | | | | | | |
| Implementação do *Parser* | | | x | x | x | | | | | |
| Implementação da análise | | | | x | x | x | | | | |
| Escrita do resultados preliminares | | | | x | | | | | | |
| Entrega do resultados preliminares | | | | x | | | | | | |
| Planejamento da Arquitetura de *software*. | | x | x | | | | | | | |
| Estudo do referencial teórico | x | x | x | x | x | | | | | |
| Implementação dos testes | | x | x | x | x | | | | | |
| Escrita do TCC | | | | | x | x | x | | | |
| Preparação da apresentação | | | | | | | | x | | |
| Defesa | | | | | | | | | x | |
| Sugestões da banca | | | | | | | | | x | x |

**Referências:**

Aho,A.V; Lam,M.S; Sethi, R; Ullman, J.D. 2007. Compilers: Principles, Techniques, & Tools. Segunda Edição. Pearson Education, Inc, Boston, Massachusetts, Estados Unidos.

Ceconello, J. 2016. Ferramenta de auxílio à análise de anomalias em códigos Orientados a Objeto. Monografia em Ciência da Computação. Universidade Federal do Rio Grande do Sul, Porto Alegre, Rio Grande do Sul, Brasil.

Cooper, K.D; Torczon,L. 2012. Engineering a Compiler. Segunda Edição. ELSEVIER, Huston, Texas, Estados Unidos.

Dean,J; Grove,D; Chambers,C. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. 9th European Conference: 77-101.

Iglberger, K. 2022. C++ Software Design: Design Principles and Patterns for High-Quality Software. O'Reilly Media, Inc, Sebastopol, Califórnia, Estados Unidos.

Langr, J. 2013. Modern C++ Programming with Test-Driven Development. The Pragmatic Programmers, LLC. Colorado Springs, Colorado, Estados Unidos.

Martin, R.C. 2009. Clean Code A Handbook of Agile Software Craftsmanship. Pearson Education, Inc, Boston, Massachusetts, Estados Unidos.

Nystrom, R. 2015-2020. Crafting Interpreters. Genever Benning. Seattle, Washington, Estados Unidos.

Silva, D; Samarasekara, H.M.P.P.K.H; Hettiarachchi, R.T; Senanayake, W.A.B.M; Sanjaya, A.M.T; Abeysekara, M.P. 2023. A Comparative Analysis of Static and Dynamic Code Analysis Techniques. TechRxiv: DOI: 10.36227/techrxiv.22810664.v1.

Wang, S; Ding, L; Shen, L; Luo, Y; Du, B; Tao, D. 2024. Título do trabalho. Findings of the Association for Computational Linguistics: 13619–13639.