Qt Clang-UML: um Plugin de Geração e Visualização de Documentação UML na IDE Qt Creator

Trabalho de Conclusão de Curso

Luan Pinheiro da Silva

Sandro Santos Andrade
Orientador

Instituto Federal da Bahia - IFBA
Curso de Análise e Desenvolvimento de Sistemas
Campus Salvador

Salvador, Bahia, Brasil
05 de fevereiro de 2026

Sumário

1 Visão Geral

1.1 Declaração do problema

Documentar artefatos computacionais é um desafio recorrente na vida de qualquer desenvolvedor de *software*, independentemente da subárea em que atue. Diante desse desafio, surge a questão: como produzir essa documentação?

Uma das abordagens mais utilizadas ao longo dos anos é o uso de diagramas *UML* (*Unified Modeling Language*). Eles podem ser gerados pelas equipes de desenvolvimento de diversas formas, seja por meio de *softwares* com interface gráfica (*GUI — Graphical User Interface*) ou por linguagens próprias de modelagem [1] [2].

A linguagem de programação C++ é atualmente a segunda mais popular do mundo [3]. Entre suas principais ferramentas, destaca-se o *framework* Qt, amplamente utilizado para o desenvolvimento de aplicações *cross-platform*. Esse *framework* é comumente empregado em conjunto com a IDE (*Integrated Development Environment*) *FOSS* (*Free and Open Source*) Qt *Creator* [4]. Este *software* adota uma arquitetura que possibilita a criação de novas funcionalidades por meio de *plugins*. Apesar disso, ela ainda não dispõe de um módulo capaz de gerar ou exibir diagramas *UML* [5].

Nesse contexto, existe a carência de uma ferramenta simplificada e abrangente para a documentação de código no formato de diagramas *UML* no Qt *Creator*, necessitando de uma forma que facilite a criação e visualização de modelagem *UML* em projetos desenvolvidos nessa IDE.

Nesta pesquisa serão analisadas ferramentas existentes de geração e exibição de diagramas UML a partir de código fonte, para que sejam identificadas as principais carências delas, todas elas sendo código *open source*, para que seja possível analisar soluções de escopo similar, além da possibilidade de maior detalhamento da análise das mesmas.

1.2 Proposta de solução de software

O objetivo da solução é resolver a problemática da documentação através de um *plugin* para a IDE Qt *Creator*, em que um usuário desenvolvedor possa ser capaz de gerar e visualizar diagramas UML exclusivamente de projetos C++ que utilizem o compilador Clang, sendo esses com ou sem o uso do *framework* Qt, e com o código fonte do mesmo sendo traduzido para diagramas UML, em formato de imagem, com o propósito de ser visualizada lado a lado do código fonte. O *plugin* terá o nome de Qt Clang-UML.

Em sua estrutura, o Qt Clang-UML utilizará um binário compilado do clang-uml para realizar as funcionalidades de geração dos diagramas, e PlantUML para converter o código em imagem, portanto o *plugin* é uma *interface* gráfica que auxilia o desenvolvedor a utilizar essas duas ferramentas.

1.3 Tecnologias adotadas

1.3.1 C++

C++ é uma linguagem de programação de propósito geral que foi desenvolvida por Bjarne Stroustrup como um aprimoramento da linguagem C para adicionar o paradigma orientado a objetos. Atualmente, é a segunda linguagem mais popular do mundo [3].

Ela é considerada uma linguagem de nível médio [6], pois combina características de linguagens de alto nível, como abstrações robustas, por exemplo seu modelo de *I/O* baseado em *streams*, e de baixo nível como controle de *hardware* em baixo nível e manipulação direta de memória. Sua flexibilidade permite desenvolver *software* das mais diversas demandas, incluindo alto desempenho e concorrência, sem comprometer requisitos essenciais como portabilidade e manutenibilidade. [7].

1.3.2 Qt

Qt é um *framework* de desenvolvimento de aplicativos multiplataforma para sistemas operacionais *desktop*, embarcados e *mobile*. As plataformas suportadas incluem Linux, OS X, Windows, Android, iOS, entre outros.

Qt não é uma linguagem de programação por conta própria, ele é um *framework* escrito em C++. Um pré-processador, o MOC (*Meta-Object Compiler*), é usado para estender o C++ com recursos como *signals* e *slots*. Antes da etapa de compilação, o MOC analisa o código fonte em C++ Qt e gera código objeto C++ padrão. Assim, o próprio framework e as aplicações / bibliotecas que o utilizam podem ser compiladas por qualquer compilador C++ padrão como Clang, GCC e MinGW [8].

1.3.3 Qt Creator

Qt Creator é uma IDE *open source* multiplataforma projetada para as necessidades dos desenvolvedores Qt, sendo a oficial do Qt *Group* [4]. Dentre suas características se destaca a arquitetura modular, em que todos os elementos são módulos (ou *plugins*), podendo ser adicionados e removidos livremente pelo usuário. Além disso, a galeria de *plugins* dessa ferramenta é extensa, com contribuições podendo ser feitas para o projeto, sendo possível criar novos módulos, que podem ser submetidos oficialmente ao projeto.

1.3.4 Clang-uml

O clang-uml foi escolhido para o desenvolvimento do *plugin* pois possui as principais propriedades necessárias: abrange uma maior gama de diagramas UML, compatível com C++ e 100% *open source*.

Por se tratar de um *software* que não possui *interface* gráfica, a proposta desta pesquisa busca suprir essa limitação por meio de uma *interface* integrada ao Qt *Creator*. Essa dependência pode ser considerada tanto uma vantagem, por aproveitar a infraestrutura e os recursos de extensibilidade da IDE, quanto uma desvantagem, por limitar o uso do *plugin* a

esse ecossistema específico.

1.3.5 PlantUML

O clang-uml, apesar de ser responsável pela geração dos diagramas, não possui capacidade nativa de exportá-los como imagens. Sua saída padrão consiste em arquivos .puml, os quais utilizam a linguagem de diagramação da ferramenta PlantUML, permitindo a conversão desses arquivos para representações gráficas, como PNG e SVG. Portanto, o PlantUML torna-se uma dependência necessária para que o *plugin* possa apresentar visualmente os diagramas gerados.

1.4 Trabalhos relacionados

As ferramentas aqui analisadas são todas *open source* e de geração/visualização de diagramas UML a partir de código fonte.

1.4.1 App Map

AppMap é uma inteligência artificial arquiteta de software, que possui extensões para o editor de texto VSCode e IDEs da JetBrains que realiza uma gama de funcionalidades relacionada a análise de código estática e dinâmica [9]. Este *software* pode exibir diagramas de sequência, com base no código fonte de um projeto, e inclui a funcionalidade *jump to code*, mas não possui suporte para C++. Possui um plano gratuito e dois planos pagos, para profissionais e empresas. O foco dessa ferramenta é a gama de ferramentas de análise de código, além disso ela é *open source* apenas nas aplicações *client*, como o *plugin* do VSCode, com licença MIT, dificultando a extensão da ferramenta, já que o código com as funcionalidades em si são proprietárias.

1.4.2 Doxygen

Doxygen é uma ferramenta totalmente *open source*, licença GPL-2.0 que gera a artefatos de documentação, com suporte a diferentes linguagens de programação, criada inicialmente apenas para C++, e que tem como principal objetivo gerar documentação completa, e não apenas diagramas, para utilização como documentação final. Esse programa realiza a geração para um projeto através de comentários em seções no código fonte, podendo exportar para HTML, PDF, Man page, dentre outros, sendo bastante flexível e com foco em um artefato final da documentação. O Doxygen possui uma *GUI* oficial[10]. Apesar de ter suporte para C++ o Doxygen possui suporte limitado a UML, tendo apenas o diagrama de classes com personalização limitada.

1.4.3 Clang-uml

Clang-uml é uma ferramenta totalmente *open-source*, licença Apache 2.0, de geração de diagramas UML no formato de *diagrams as code* para projetos compilados com o compilador das linguagens C/C++ Clang, possuindo execução apenas via *CLI (Command Line Interface)*. Esse software gera diagramas de classe, sequência e pacotes, com alta personalização dos

mesmos. Os formatos que os diagramas podem ser exportados são PlantUML, MermaidJS, GraphML e JSON, e com auxílio dessas ferramentas pode gerá-las em diversos formatos de imagem, como JPG, PNG e SVG [**??**]. Apesar da flexibilidade e abrangência de tipos e formatos dos diagramas, o clang-uml sofre de uma curva de aprendizado mais íngreme, devido a sua alta capacidade de personalização dos diagramas, além da falta de *interface* gráfica ou sequer um *plugin* para editores de texto e IDEs.

1.4.4 Comparação das ferramentas

Funcionalidades / Ferramentas	100% *Open Source*	Compatibilidade com C++	Existe *plugin* em editor de texto ou IDE	*Interface* gráfica	Variedade de diagramas UML
AppMap			X	X	
Doxygen	X	X	X	X	
Clang-uml	X	X			X

Tabela 1: Comparação das ferramentas descritas nos trabalhos relacionados 1.4

2 Requisitos

2.1 Requisitos funcionais

Nº	História de Desenvolvedor
RF01	Como desenvolvedor, desejo criar diagramas de classe referentes a um código fonte dentro de uma pasta de minha escolha para documentá-la
RF02	Como desenvolvedor, desejo visualizar diagramas de classe de uma pasta para entender seus relacionamentos em um sistema
RF03	Como desenvolvedor, desejo acessar o código fonte das classes visualizadas em um diagrama de classes para editar seus relacionamentos

Tabela 2: Requisitos funcionais no formato de histórias de usuário

2.2 Requisitos não funcionais

Nº	Requisito
RNF01	*Interface* deve seguir os padrões de UI do Qt Creator para consistência visual
RNF02	Processamento deve ser feito em múltiplas *threads* para não bloquear a UI, especialmente durante análises longas
RNF03	Compatibilidade deve abranger projetos em C++ e C++/Qt
RNF04	O tempo de busca por uma classe da RF03 não deve ser maior que 1 segundo
RNF05	Projetos que utilizarem essa ferramenta devem ser compilados com Clang

Tabela 3: Requisitos não funcionais do Qt Clang-UML

3 Design

3.1 Visão arquitetural

A IDE Qt Creator é desenvolvida com o padrão arquitetural *Microkernel*, na qual seu núcleo fornece um *framework* básico que carrega seus *plugins* em tempo de execução, sendo esses quem definem as funcionalidades disponíveis na IDE.

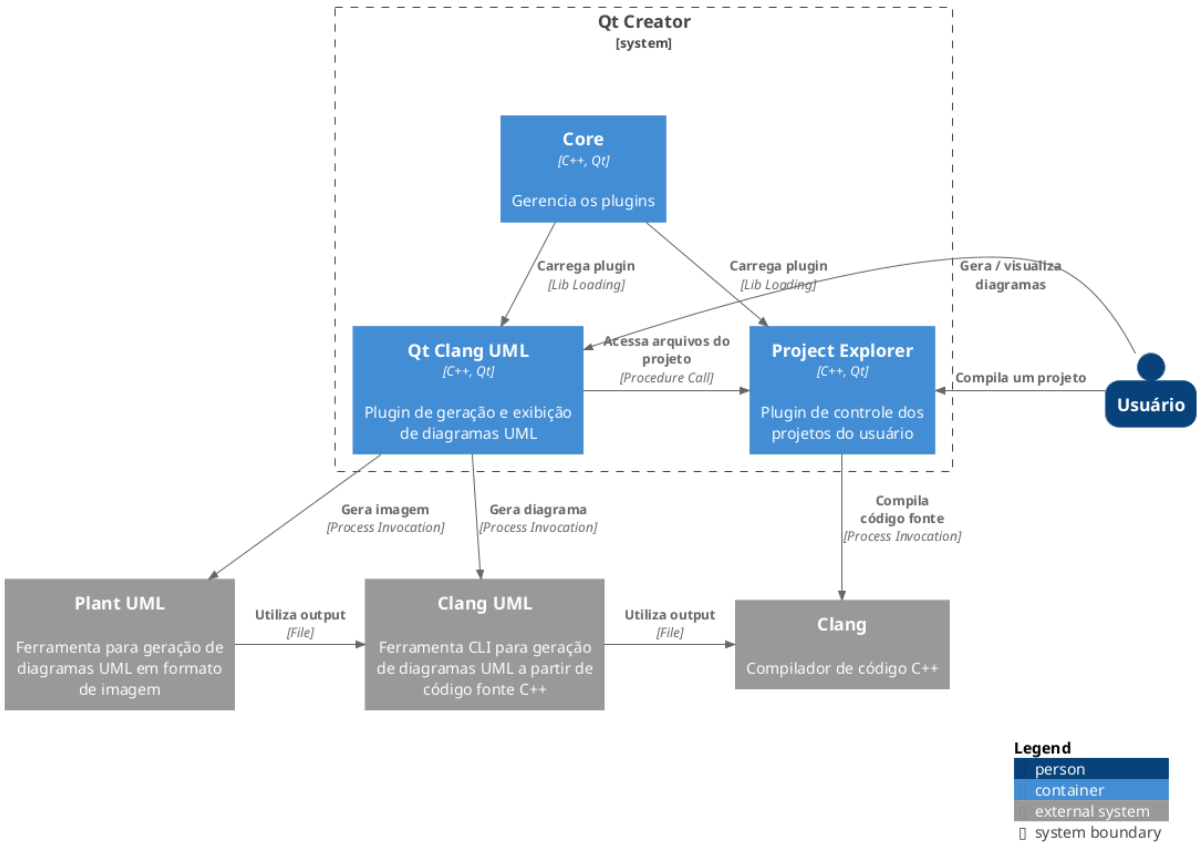

Figura 1: Diagrama C4 de *container* do Qt Creator com o *plugin* Qt Clang UML

O Qt Clang UML é um *plugin* desenvolvido em C++ e Qt, sendo desenvolvido para a IDE Qt Creator baseado na *interface* disponibilizada pela IDE para que outros usuários possam adicionar funcionalidades de forma desacoplada. O módulo em questão realiza a geração de diagramas; para isso, utiliza duas aplicações externas, executadas como processos independentes, que realizam o processo de geração propriamente dito. São elas:

- **clang-uml:** sistema que gera diagramas no paradigma *diagram as code*, utilizando como entrada os dados de um projeto compilado com Clang.
- **PlantUML:** sistema que gera imagens dos diagramas, utilizando o diagrama gerado anteriormente pelo clang-uml. Os artefatos finais resultantes são exibidos ao usuário

Para identificar o projeto cujo diagrama será gerado, o Qt Clang UML utiliza um *plugin* do

Qt Creator:

- ***ProjectManager***: responsável pelo controle dos projetos do usuário, como o gerenciamento dos ambientes de *build* e a navegação dos projetos por meio de uma árvore de arquivos. Este *plugin* fornece diversas APIs públicas, sendo essencial para identificar o projeto atual e seus arquivos que contêm as classes utilizadas nos diagramas.

3.2 Projeto UML

3.2.1 Diagrama de classes

Neste diagrama [Link para diagrama de classes do Qt Clang UML] estão definidas as classes que compõem o Qt Clang-UML, bem como suas relações com classes do *framework* Qt.

- ***QtClangUMLPlugin:*** Ponto de entrada da aplicação, no qual são definidas a *interface* e as dependências com o Qt Creator.
- ***ClangUmlEnv:*** Armazena os dados de configuração do projeto atual em tempo de execução, incluindo caminhos das pastas *build* e de *compile commands* do *kit* selecionado.
- ***ClangUmlController:*** Responsável por construir e atualizar o estado de barras de carregamento para as diversas ações do *plugin*, como inicialização e geração de diagramas, sendo também a classe que inicia a cadeia de execução dessas regras de negócio. Estas barras de carregamento são parte do *Core* do Qt Creator.
- ***ClangUmlWorker:*** Executa as regras de negócio do clang-uml e plant-uml, invocando os processos quando necessário. É executado em uma *thread* separada pois contém os processos de maior custo computacional.
- ***SvgWidget:*** Responsável pela definição visual dos diagramas. Nessa classe estão implementados o dimensionamento dos diagramas e o comportamento de elementos que respondem a clique.
- ***DiagramWidget:*** *Widget* que é exibida ao lado do código, exibindo mensagens ou diagramas ao usuário.
- ***DiagramGenerator:*** Classe responsável por interagir com o arquivo .clang-uml, adicionando a estrutura dos diagramas que serão gerados posteriormente.
- ***ClangUmlOptionsPage:*** Responsável por construir a página de edição do *plugin* na seção *preferences* do Qt Creator. Atualiza o caminho para os binários do clang-uml e PlantUML.
- ***SearchTask:*** Realiza a busca de uma classe dado seu nome, utilizada para realizar *jump to code*, que ocorre quando o usuário seleciona um componente de um diagrama de classes. Seu processamento ocorre em uma *thread* diferente da principal, pois a busca pode ser demorada em projetos com grande quantidade de classes.

3.2.2 Diagrama de Componentes

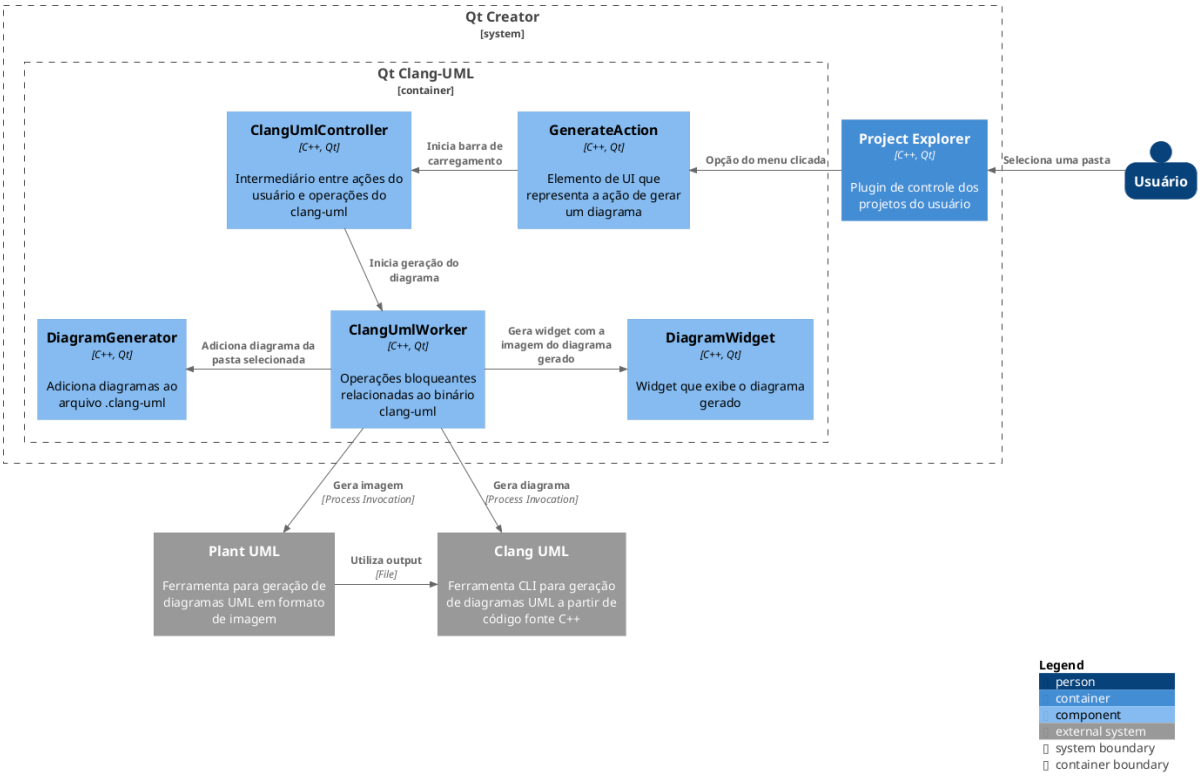

Figura 2: Diagrama C4 de componentes demonstrando o fluxo de geração de diagramas com o *plugin* Qt Clang UML

O diagrama C4 de componentes ilustra o processo de criação da *widget* do diagrama no contexto do plugin Qt Clang-UML.

A interação inicia-se com o usuário, que seleciona uma pasta no *plugin Project Explorer*. Logo após é disparada a ação *GenerateAction*, que é a opção de geração de diagramas na interface gráfica. O *ClangUmlController* atua como intermediário entre a interface do usuário e as operações internas do sistema, iniciando a geração do diagrama e gerenciando barras de carregamento, relativas à conclusão das tarefas executadas.

As tarefas de execução bloqueante e integração com ferramentas externas são delegadas ao *ClangUmlWorker*, que invoca o clang-uml para análise do código-fonte C++ e o PlantUML para geração da imagem do diagrama. Antes da execução dos processos externos o componente *DiagramGenerator* gerencia a inclusão do diagrama selecionado pelo usuário no arquivo .clang-uml. Por fim, o *DiagramWidget* renderiza e exibe a imagem final do diagrama ao usuário.

3.3 Modelo de dados

Trecho de código 1: Arquivo .clang-uml gerado pelo Qt Clang UML

```
compilation_database_dir: .../build/Desktop-Debug/.qtc_clangd/compile_commands.
    json
output_directory: docs/diagrams
diagrams:
  src:
    type: class
    title: "Folder: src"
    glob:
      - src/*.cpp
      - src/*.cc
    include:
      paths:
        - src
    exclude:
      paths:
        - thirdparty
```

A ferramenta clang-uml opera com base em um arquivo de configuração nomeado .clang-uml, que deve ser posicionado no diretório raiz do projeto. Esse arquivo define os parâmetros essenciais para a geração dos diagramas (trecho 1), são essas:

- **compilation_database_dir:** caminho para o arquivo compile_commands.json, resultante da compilação do projeto com Clang.
- **output_directory:** O diretório de saída onde os diagramas serão armazenados localmente.
- **diagrams:** Os tipos de diagrama a serem criados (como diagramas de classe, sequência ou pacotes).
- **glob/include/exclude:** Arquivos que devem ou não estar contidos na geração de um diagrama específico.

Todo o processo de geração dos diagramas ocorre na máquina do usuário, através da execução do clang-uml, que interpreta o código-fonte com base na configuração fornecida e do PlantUML que gera as imagens dos diagramas em formato SVG.

O Qt Clang-UML é uma extensão que aproveita essa infraestrutura, automatizando e facilitando o processo de configuração e gerenciamento dos diagramas. Por meio de uma interface gráfica, o usuário precisa apenas indicar uma pasta com código fonte para que seja visualizado seu diagrama, integrando de forma transparente a funcionalidade do clang-uml a um ambiente de desenvolvimento mais amigável e produtivo.

4 Implementação

Esta seção descreve alguns dos principais elementos implementados no *plugin* Qt Clang UML, explicando as decisões mais relevantes.

4.1 Geração de diagramas

Todo o processo de geração ocorre na máquina do usuário, necessitando apenas configurar os caminhos dos binários (clang-uml e plant-uml), selecionar uma pasta através do *plugin* Project Explorer, então selecionar a opção "*Generate diagram for this folder*, para então visualizar o diagrama da pasta desejada. O Qt Clang UML irá gerenciar o processo de geração sem interferência do usuário a partir desse ponto, invocando os processos externos necessários para gerar o diagrama da pasta escolhida, não incluindo suas subpastas, apenas as classes utilizadas no código fonte da pasta em questão, e então exibe o SVG resultante ao lado do código fonte.

O *plugin* realiza todo o processo de geração sempre que uma pasta é escolhida, sempre a partir do *input* do usuário, como descrito no processo anterior.

4.2 Qt Creator APIs

O Qt Clang UML utilizou bastante das diretrizes e APIs oficiais disponibilizadas pelo Qt Creator para desenvolvimento de *plugins*, que especificam desde organização de código até estrutura de metadados. Apesar de ser possível desenvolver o *plugin* em Lua, foi decidido usar C++ devido a sua maior estabilidade e suporte a funcionalidades da IDE [11].

Seguindo as principais diretrizes de projeto, foi desenvolvido um código fortemente orientado a objetos, priorizando a coesão, o baixo acoplamento e a reutilização de componentes. Além disso, foi adotado o uso consistente das APIs oficiais da IDE, o que contribui diretamente para a estabilidade, a manutenibilidade e a compatibilidade do sistema com o ambiente de desenvolvimento.

Trecho de código 2: Classe ClangUmlWorker utilização o *plugin* Project Explorer para recuperar o nome de uma pasta selecionada

```
#include <projectexplorer/projectnodes.h>
#include <projectexplorer/projecttree.h>

//...

void ClangUmlWorker::initializeFolderDiagram() {

    // ...

    if (auto *node = ProjectExplorer::ProjectTree::currentNode()) {
        auto *folderNode = dynamic_cast<ProjectExplorer::FolderNode *>(node);

        // ...

        QString folderName = folderNode->displayName();

        // ...

        saveClassDiagram(folderName, ..., ..., ...);

        emit initializeFolderSuccess(...);
```

```
    } else {
        emit initializeFolderError();
    }
}
```

Trecho de código 3: Classe QtClangUMLPlugin utilizando o *plugin* Project Explorer e *interface* Core

```
    void initialize() final {
        Core::IOptionsPage::registerCategory(
            "clang-uml", "Qt Clang UML",
            Utils::FilePath());

        new ClangUmlOptionsPage(m_clangumlenv);

        // ...

        connect(
            ProjectExplorer::ProjectManager::instance(),
            &ProjectExplorer::ProjectManager::activeBuildConfigurationChanged,
            m_clangumlenv, &ClangUmlEnv::updateEnv);
        connect(
            ProjectExplorer::ProjectManager::instance(),
            &ProjectExplorer::ProjectManager::activeBuildConfigurationChanged,
            m_diagramwidget, &DiagramWidget::setDefaultWidget);

        // ...
    }
```

Nos trechos 2 e 3, observa-se a integração com o *plugin Project Explorer*, responsável por fornecer informações contextuais relacionadas à navegação de projetos na IDE, tais como a árvore de arquivos do código-fonte organizada em pastas e as interações realizadas pelo usuário sobre esses elementos. Além disso, é utilizada a interface *Core*, que expõe funcionalidades gerais da aplicação, incluindo mecanismos de configuração, gerenciamento do ciclo de vida do *plugin* e suporte à organização e execução de *threads*.

Por meio da classe *ProjectTree*, torna-se possível acessar o elemento atualmente selecionado pelo usuário na árvore de projetos e tratá-lo como um nó do tipo pasta, permitindo a extração de informações relevantes, como o seu nome. Tal informação é essencial para o fluxo de geração de diagramas, uma vez que define o contexto no qual a análise do código-fonte será realizada.

Adicionalmente, foi utilizada uma instância da classe *ProjectManager* para monitorar alterações na configuração de *build* ativa do projeto. Esse monitoramento é realizado por meio do sistema de *signals* e *slots* do framework Qt, possibilitando que mudanças realizadas pelo usuário sejam imediatamente refletidas no estado interno da aplicação, por meio da atualização das variáveis de ambiente representadas pela classe *ClangUmlEnv*, bem como nos componentes visuais, como o *DiagramWidget*.

4.3 Threads

Trecho de código 4: Classe ClangUmlController movendo a execução de uma instância de ClangUmlWorker para uma *thread* dedicada

```
ClangUmlController::ClangUmlController(ClangUmlWorker *worker, QObject *parent)
    : QObject(parent), m_worker(worker) {

    m_worker->moveToThread(&m_thread);

    connect(this, &ClangUmlController::initializeClangUml, m_worker,
            &ClangUmlWorker::initialize);
    connect(this, &ClangUmlController::initializeClangUml, this,
            &ClangUmlController::initializeProgress);

    \\ ...

    m_thread.start();
}
```

Para remover travamentos na *thread* principal da IDE foi necessário utilizar técnicas de *multithreading*, com foco nas tarefas de maior custo computacional, como as de uso de processos externos e buscas dentro do projeto.

No trecho 4, observa-se o uso da classe *QThread*, abstração de *thread* fornecida pelo *framework* Qt, dedicada à execução de uma instância da classe *ClangUmlWorker*. Em função da natureza assíncrona de diversas funcionalidades implementadas nessa classe, tais como escrita em arquivos e execução de processos externos, foi essencial delegar essas operações a uma *thread* separada, evitando o bloqueio da interface principal da IDE.

A combinação das abstrações *QThread* e *QObject* permite o gerenciamento do ciclo de vida de objetos de forma estruturada e em alto nível. Nesse contexto, a instância de *ClangUmlWorker* é movida para uma *thread* dedicada, possibilitando que seu processamento seja realizado de maneira independente da *thread* principal.

A classe *ClangUmlController* é responsável por configurar e atualizar barras de progresso com base no estado das operações executadas pelo *ClangUmlWorker*. Para a sincronização dos eventos e a comunicação entre as *threads*, foi utilizado o mecanismo de *signals* e *slots* do framework Qt, o que permite a execução paralela de tarefas computacionalmente mais custosas, mantendo a responsividade e a estabilidade da interface da IDE.

Trecho de código 5: Classe DiagramWidget delegando o processamento de busca da classe SearchTask a uma nova *thread*

```
void DiagramWidget::jumpToFile(const QString &className) {
    const QString projectPath = m_clangumlenv.getEnvironment().at(
        ClangUmlEnv::EnvVariables::ProjectPath);

    auto task = new SearchTask(className, projectPath);

    connect(task, &SearchTask::fileFound, this, &DiagramWidget::onFileFound);
    connect(task, &SearchTask::searchFailed, this,
            &DiagramWidget::onSearchFailed);

    QThreadPool::globalInstance()->start(task);
```

```
12 }
```

Além do uso citado no trecho 4, empregou-se uma *QThreadPool* na implementação da funcionalidade *jump to code*, mais especificamente durante o processo de busca, em todo o código-fonte do projeto, por uma classe selecionada pelo usuário, conforme o trecho 5. Essa estratégia foi adotada em razão da natureza *fire-and-forget* dessas tarefas, que podem ocorrer de forma recorrente, tornando o uso de uma *thread pool* mais simples e eficiente do que a instanciação e o gerenciamento manual de *threads* individuais.

Durante os testes iniciais, realizados com projetos de baixa complexidade, não foi observada redução significativa de desempenho, o que permitiu uma implementação inicial executada diretamente na *thread* principal. Entretanto, após o teste do projeto de maior complexidade (seção 5), foi identificado um atraso perceptível durante o processo de busca, notado através da interrupção momentânea da interface gráfica durante o processamento. Diante desse comportamento, tornou-se necessário delegar a execução da busca a uma nova *thread*. Essa abordagem simplificou o gerenciamento de concorrência, melhorando a responsividade da *interface* e assegurando uma melhor experiência de uso, mesmo em projetos de maior complexidade.

5 Testes de Software

5.1 Projeto de testes

Para validar as funcionalidades do Qt Clang-UML, foi utilizada uma suíte de projetos exemplo em C++ e Qt, abrangendo diferentes cenários. Esses projetos servem como casos de teste reais para verificar a capacidade de geração de diagramas com diferentes complexidades de código.

5.1.1 Estratégia de testes

Os testes foram realizados com 3 projetos, variando em presença do *framework* Qt, quantidade de linhas e complexidade de código, com o objetivo de encontrar *edge cases* de cada caso, corrigindo *bugs* e gargalos de desempenho durante o processo.

Foram gerados alguns diagramas de acordo com a complexidade dos projetos e pastas disponíveis. Para cada um dos projetos estão disponibilizadas suas respectivas configurações e resultados encontrados, tnato no processo de geração quanto de busca.

Para medir o desempenho foi utilizado uma máquina com sistema operacional Arch Linux e ambiente Hyprland, especificações detalhadas estão na figura 3. Quanto a métricas de desempenho, foi medido o tempo de execução dos processos externos(clang-uml e plantuml) e o tempo de busca por uma classe ao clicar em um componente do diagrama, ambos utilizaram a classe QElapsedTimer, além dos mecanismos de *signals* e *slots* do Qt, garantindo precisão na obtenção dos dados (trecho 6).

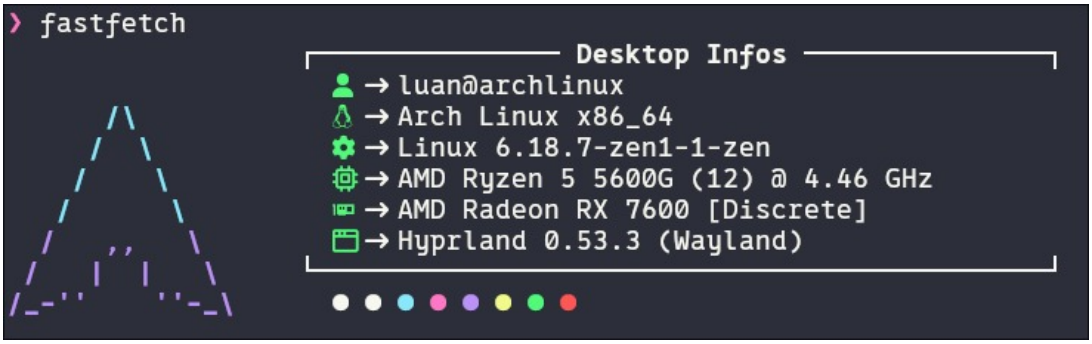

Figura 3: Especificações da máquina usada nos testes, *output* da ferramenta fastfetch

Trecho de código 6: *Benchmarking* da geração dos diagramas do projeto

```
void ClangUmlWorker::generateFolderDiagram(const QString &relativeFolderName) {
    // ...
    auto elapsedTimer = std::make_shared<QElapsedTimer>();
    auto clangUmlElapsed = std::make_shared<qint64>();
    auto plantUmlElapsed = std::make_shared<qint64>();

    QProcess *clangProcess = new QProcess();
    QProcess *plantumlProcess = new QProcess();

    connect(clangProcess, &QProcess::finished, this, [/*...*/ plantumlProcess,
        relativeFolderName, clangUmlElapsed, elapsedTimer]() {
                *clangUmlElapsed = elapsedTimer->elapsed();

                qDebug() << "Project:" << environment.at(ClangUmlEnv::
                    EnvVariables::ProjectName);
                qDebug() << "Folder:" << relativeFolderName;
                qDebug() << "Time elapsed (clang-uml):" << (*clangUmlElapsed /
                    1000.0) << "seconds";

                elapsedTimer->restart();

                // ...

                plantumlProcess->start();
            });

    connect(plantumlProcess, &QProcess::finished, this, [/*...*/
        plantUmlElapsed, clangUmlElapsed, elapsedTimer](/*...*/) {
                *plantUmlElapsed = elapsedTimer->elapsed();

                qDebug() << "Time elapsed (plantuml):" << (*plantUmlElapsed /
                    1000.0) << "seconds";
                qDebug() << "Time elapsed (total):" << (*clangUmlElapsed /
                    1000.0) + (*plantUmlElapsed / 1000.0) << "seconds";

                // ...
            });

    // ...

    elapsedTimer->start();
    clangProcess->start();
}
```

5.1.2 Projeto básico

- **Tecnologias:** C++
- **Quantidade de classes:** 4
- **Complexidade:** baixa

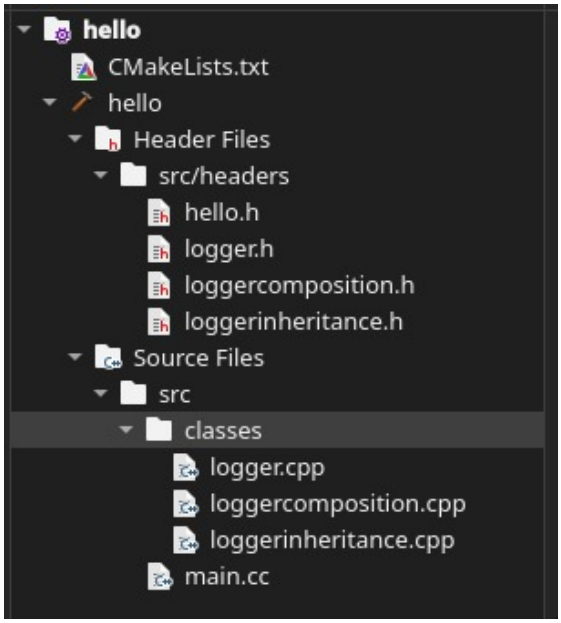

Figura 4: Estrutura de pastas do projeto básico

Este projeto constitui um estudo de caso inicial, com o propósito de validar as funcionalidades fundamentais do *plugin* em um cenário controlado e de baixa complexidade. A estrutura do código é composta por um conjunto de classes que implementam uma mesma interface comportamental para operações de *logging*, diferenciadas por seus detalhes de implementação interna. O ponto de entrada do sistema (main.cc) instancia e utiliza um objeto da classe *Logger*.

O objetivo principal deste teste é verificar o correto funcionamento básico da ferramenta e avaliar capacidades mais específicas, tais como: a geração de diagramas a partir de código distribuído em múltiplas pastas, a representação precisa de relações de herança, composição, além de modificadores de visibilidade (público, protegido e privado) para atributos e métodos.

5.1.3 Resultados

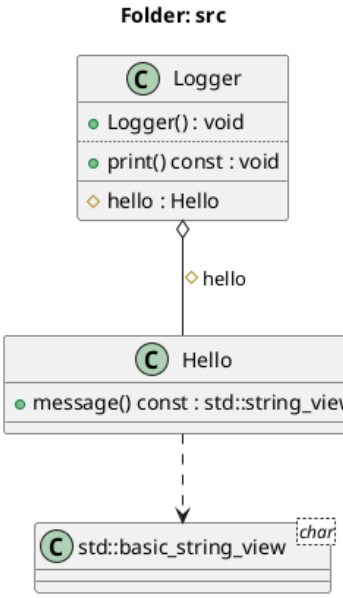

Figura 5: Diagrama UML de classes gerado pelo Qt Clang-UML do projeto básico na pasta src/

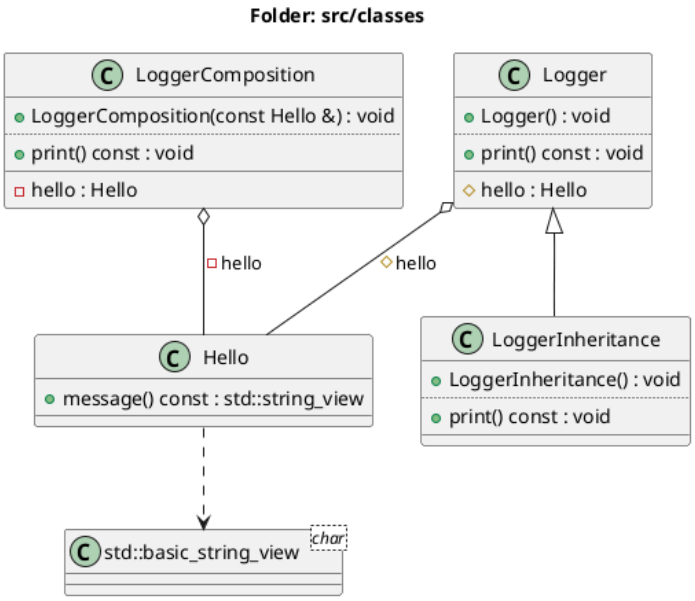

Figura 6: Diagrama UML de classes gerado pelo Qt Clang-UML do projeto básico na pasta src/classes

Trecho de código 7: *Output* no *terminal* dos *logs* de desempenho na geração dos diagramas do projeto básico

```
Project: "basic"
Folder: "src"
Time elapsed (clang-uml): 1.122 seconds
Time elapsed (plantuml): 0.552 seconds
```

```
Time elapsed (total): 1.674 seconds
Project: "basic"
Folder: "src/classes"
Time elapsed (clang-uml): 1.888 seconds
Time elapsed (plantuml): 0.547 seconds
Time elapsed (total): 2.435 seconds
```

Trecho de código 8: *Output* no *terminal* dos *logs* de desempenho na busca das classes ”Hello”e ”LoggerComposition”no projeto básico

```
Finding class: "Hello"
Elapsed time (ms): 1
Finding class: "LoggerComposition"
Elapsed time (ms): 0
```

Através deste projeto, foi possível validar as funcionalidades fundamentais do *plugin*. A análise permitiu observar diversos tipos de relacionamento entre classes, incluindo herança simples e composição, bem como exibição correta de classes da biblioteca padrão do C++. Por fim, foi confirmada a exibição precisa dos diferentes níveis de visibilidade (private e protected, nesse caso) do atributo *hello*, bem como a geração contextual de diagramas, cujo escopo é dinamicamente definido pela seleção do diretório de código-fonte pelo usuário.

Ao realizar o *benchmarking* do projeto, verificou-se que o tempo necessário para geração dos diagramas está dentro de limites aceitáveis. A etapa mais demorada corresponde à geração dos arquivos `.puml` pelo clang-uml, enquanto a renderização dos diagramas é concluída em menos de 1 segundo, conforme referenciado no trecho 7. Quanto ao tempo de resposta das buscas, os resultados obtidos atendem ao especificado no RNF04, garantindo a usabilidade e o funcionamento adequado do *plugin*, como visto no trecho 8.

5.1.4 Projeto Qt básico

- **Tecnologias:** C++/Qt
- **Quantidade de classes:** 3
- **Complexidade:** baixa

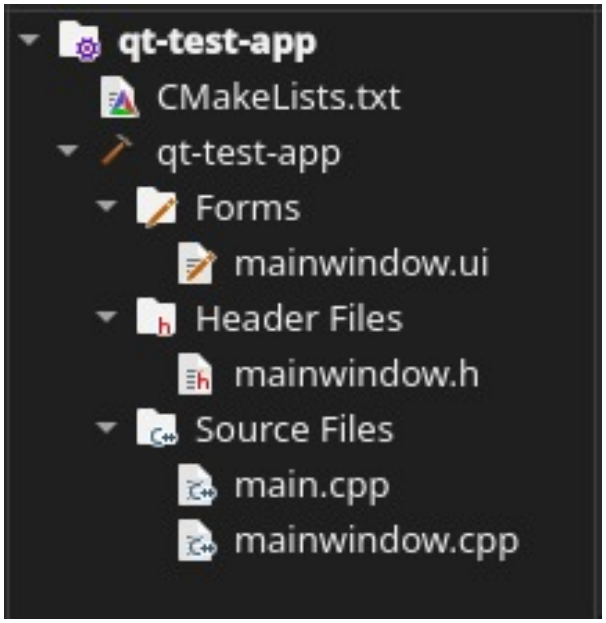

Figura 7: Estrutura de pastas do projeto Qt básico

Este projeto constitui um estudo de caso de um projeto C++/Qt de baixa complexidade, com o propósito de validar o comportamento do *plugin* em projetos que utilizem o *framework* especializado da IDE. A estrutura do código é composta por uma única classe MainWindow que possui um arquivo de formulário (extensão .ui), sem nenhuma regra de negócio implementada. No arquivo main.cpp é criada uma instância de MainWindow, torna-a visível e inicia o *loop* de eventos da aplicação Qt.

5.1.5 Resultados

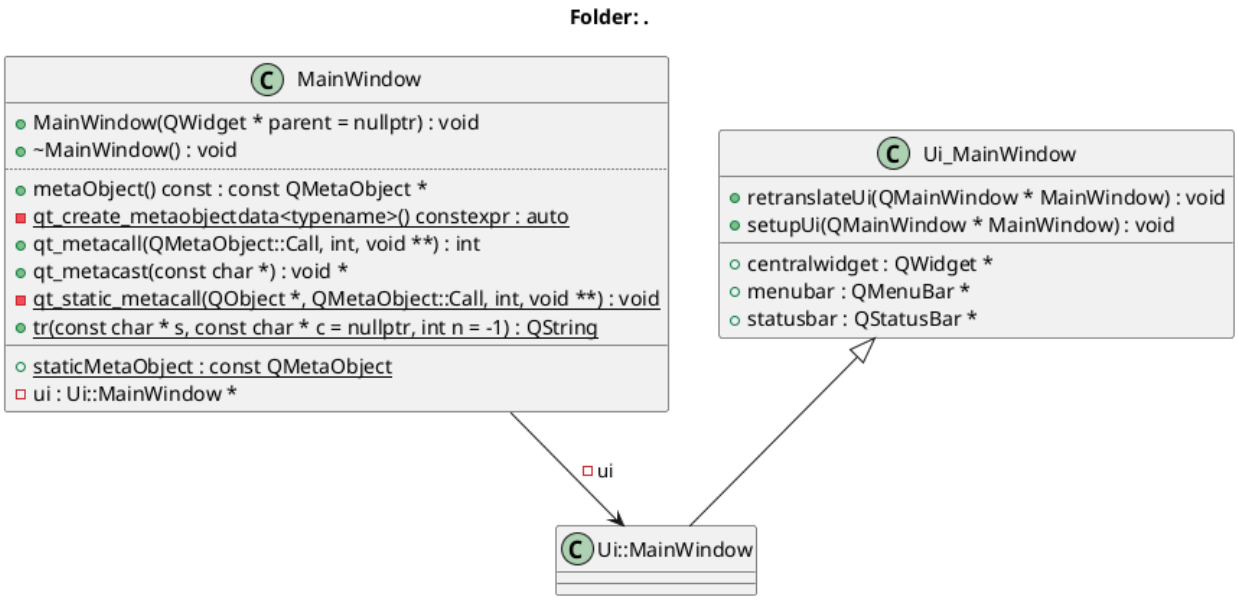

Figura 8: Diagrama UML de classes gerado pelo Qt Clang-UML do projeto Qt básico em sua pasta raiz

Trecho de código 9: *Output* no *terminal* dos *logs* de desempenho na geração dos diagramas do projeto Qt básico

```
Project: "qt-basic"
Folder: "."
Time elapsed (clang-uml): 4.773 seconds
Time elapsed (plantuml): 0.545 seconds
Time elapsed (total): 5.318 seconds
```

Trecho de código 10: *Output* no *terminal* dos *logs* de desempenho na busca das classes ”MainWindow”, ”Ui_MainWindow”e ”Ui::MainWindow”no projeto Qt básico

```
Finding class: "MainWindow"
Elapsed time (ms): 0
Finding class: "Ui_MainWindow"
Elapsed time (ms): 0
Finding class: "Ui::MainWindow"
Elapsed time (ms): 1
```

Através deste projeto, foi possível validar a geração de diagramas em projetos C++/Qt. A análise permitiu observar a correta inclusão de dependências e métodos ocultos que são adicionados através de herança e macros do *framework*. Através deste projeto foi possível identificar parâmetros necessários para incluir corretamente bibliotecas e *frameworks*, como o próprio Qt, no processo de geração de diagramas.

O *benchmark* deste projeto se manteve equivalente ao do projeto anterior, apresentando variação significativa apenas no tempo de execução do `clang-uml`, que aumentou de aproxi-

madamente 2 segundos para cerca de 5 segundos, conforme demonstrado no trecho 9. Este acréscimo, contudo, não impactou de forma relevante a experiência do usuário. O tempo de resposta das buscas manteve-se inalterado, como evidenciado no trecho 10.

5.1.6 Projeto avançado

- **Tecnologias:** C++
- **Quantidade de classes:** 1693
- **Complexidade:** alta

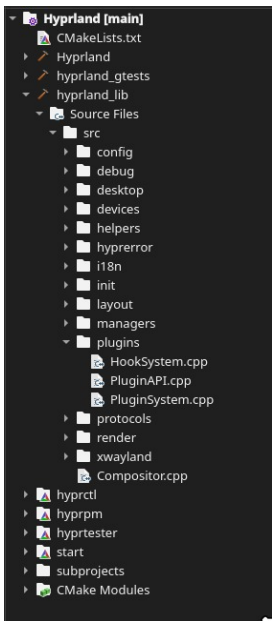

Figura 9: Estrutura de pastas do projeto Hyprland

Hyprland é um gerenciador de janelas com disposição dinâmica, que providencia um sistema próprio de *plugins*, *Inter-Process-Communication* (IPC), animações dentre outros [12]. A estrutura do código e suas regras de negócio são bastante complexas, portanto para limitar o escopo do teste serão analisadas apenas as sessões de *internationalization* (i18n) e *decorations*. Esse projeto foi selecionado para teste devido ao seu amplo escopo, o foco então será no desempenho.

5.1.7 Resultados

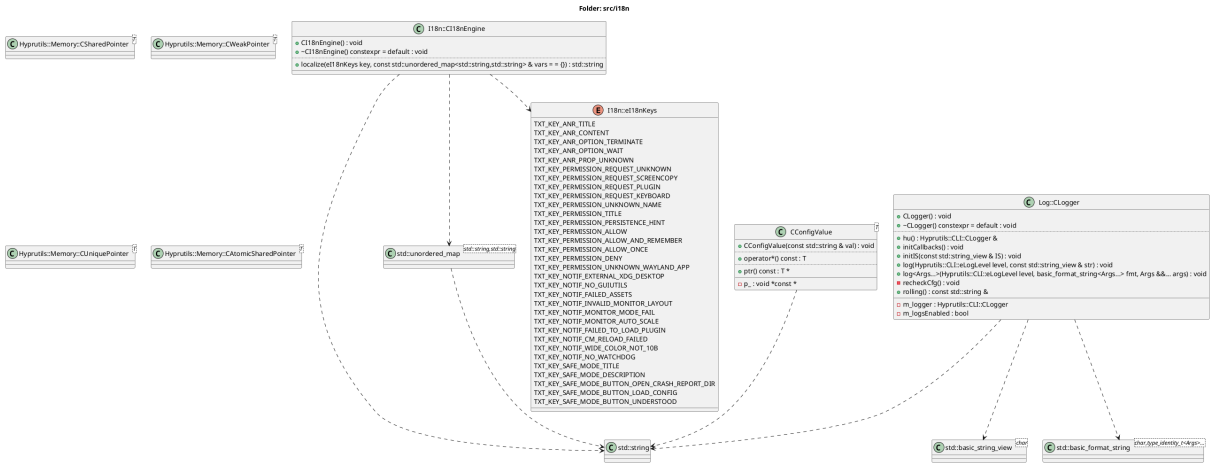

Figura 10: Diagrama UML de classes gerado pelo Qt Clang-UML do Hyprland na pasta src/i18n

Trecho de código 11: *Output* no *terminal* dos *logs* de desempenho na geração dos diagramas do projeto avançado

```
Project: "Hyprland"
Folder: "src/i18n"
Time elapsed (clang-uml): 3.541 seconds
Time elapsed (plantuml): 756.242 seconds
Time elapsed (total): 759.783 seconds
```

Trecho de código 12: *Output* no *terminal* dos *logs* de desempenho na geração dos diagramas do projeto avançado

```
Finding class: "I18n::eI18nKeys"
Elapsed time (ms): 403
Finding class: "CConfigValue"
Elapsed time (ms): 352
```

Por meio deste projeto, foi possível identificar limitações no algoritmo de busca da classe SearchTask, evidenciadas por casos em que determinadas classes não eram localizadas corretamente no código fonte. Esse problema demandou refatorações do algoritmo de busca, resultando em uma implementação mais robusta mediante a incorporação de expressões regulares mais abrangentes, as quais passaram a contemplar *edge cases* não previstos nas primeiras implementações.

Através desses testes também foi possível identificar um problema, tamanho excessivo dos diagramas gerados em diretórios que apresentam alta densidade de relacionamentos (alguns alcançando centenas de conexões), que inclusive não são gerados por completo em formato PNG, conforme ilustrado na figura [Link para trecho de diagrama UML de classes gerado pelo Qt Clang-UML do Hyprland na pasta src/render/decorations]. Essa característica, além de demandar navegação extensiva entre componentes visuais devido ao pequeno espaço

disponível na interface, resultou em queda significativa de desempenho. Esta degradação é perceptível no *benchmarking* apresentado no trecho 11, no qual o tempo total de execução atingiu aproximadamente 12 minutos e 40 segundos, uma diferença considerável em relação aos testes anteriores. Adicionalmente, observou-se a ocorrência constante de *screen tearing* mesmo após a conclusão da geração, devido a escala ampliada do diagrama produzido, o que compromete sua usabilidade.

Em relação ao desempenho da busca, os resultados embora superiores aos dos testes anteriores ainda satisfazem o RNF05 (trecho 12), indicando que o algoritmo mantém desempenho aceitável mesmo em projetos complexos.

6 Manual do Usuário

Para que o usuário possa utilizar corretamente o *plugin* Qt Clang UML é necessário seguir os passos descritos abaixo.

1. Verificando instalação na IDE:

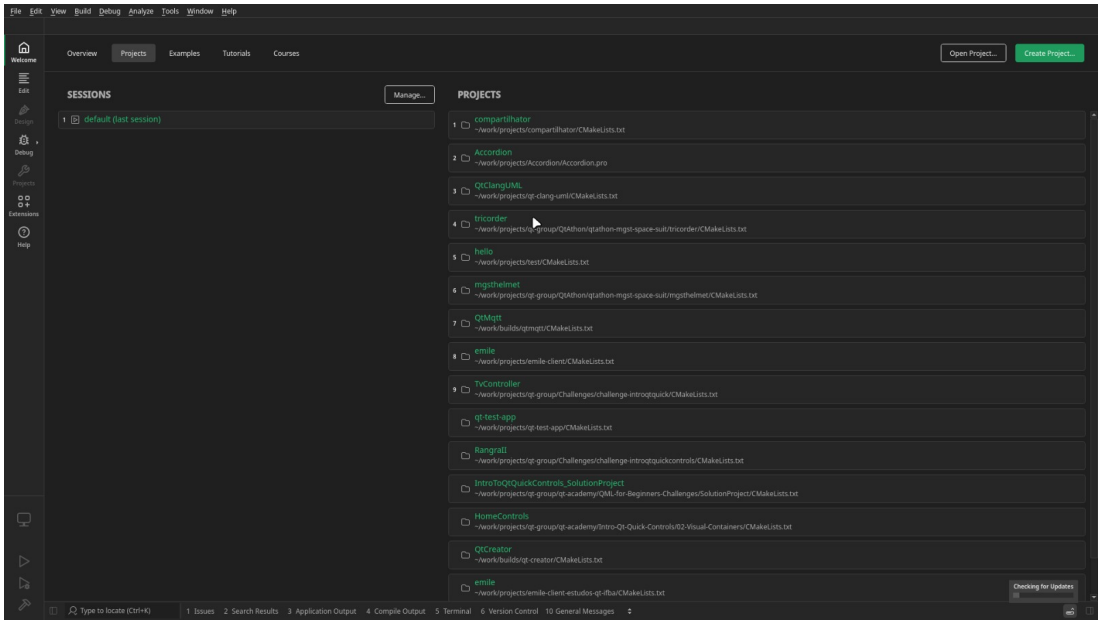

Figura 11: Tela inicial do Qt Creator

- Tela inicial que o usuário visualiza ao iniciar a IDE Qt Creator

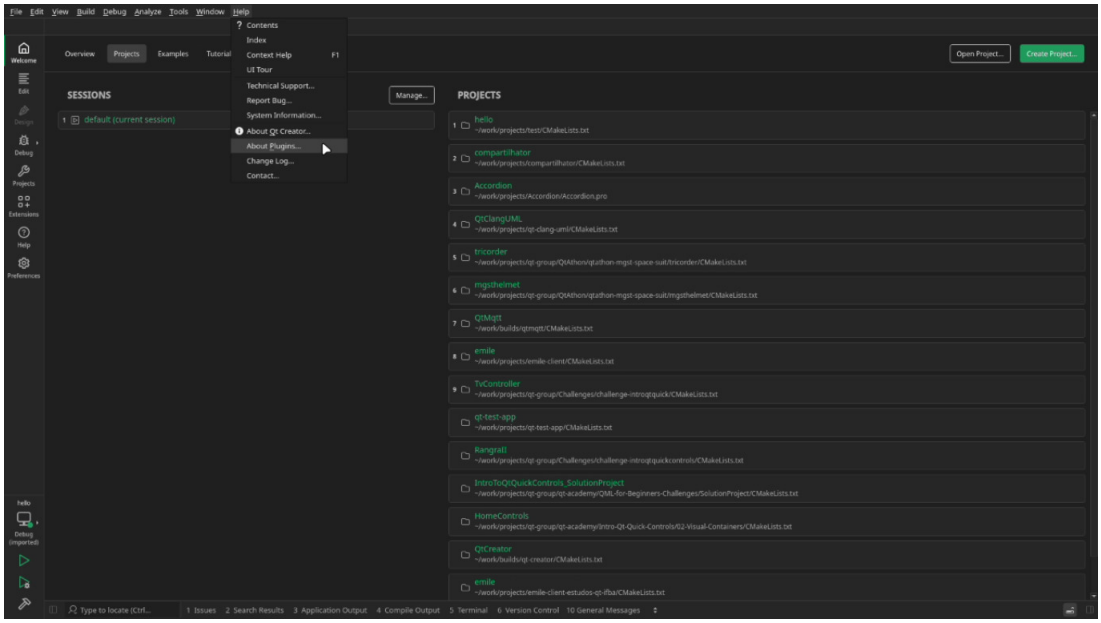

Figura 12: Acessando a aba de *plugins* do Qt Creator

- O usuário pode conferir se o *plugin* foi adicionado corretamente selecionando esta opção no menu superior

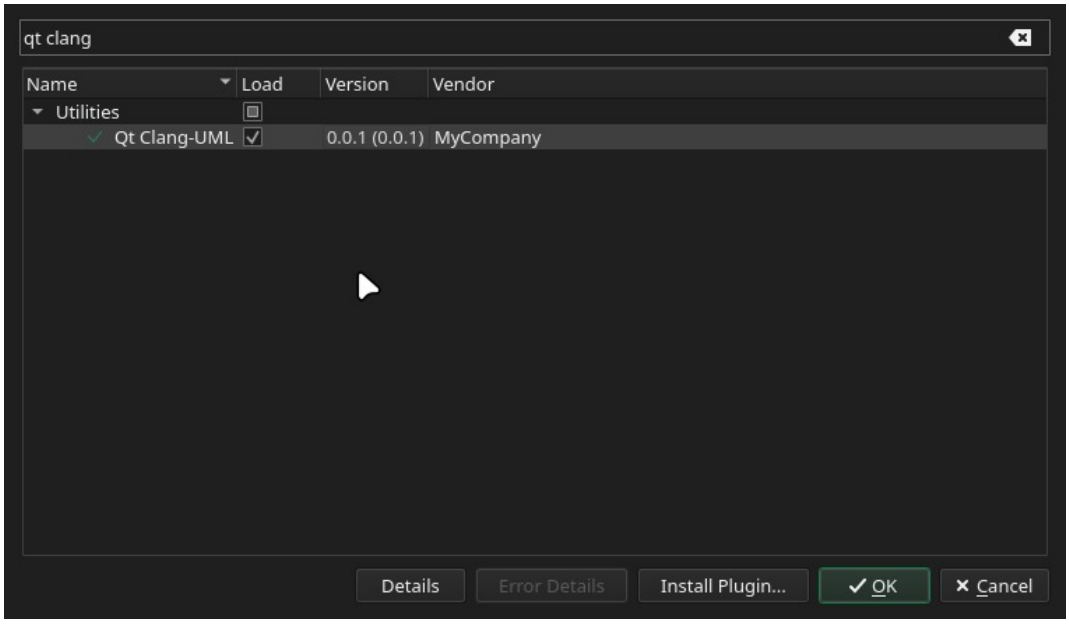

Figura 13: Verificando se o *plugin* está instalado corretamente na IDE

- Esta tela possui um campo de busca para conferir quais *plugins* estão carregados no momento atual. Busque por Qt Clang-UML e confira se está com a *checkbox* marcada.

2. Selecionando um projeto e gerando diagrama:

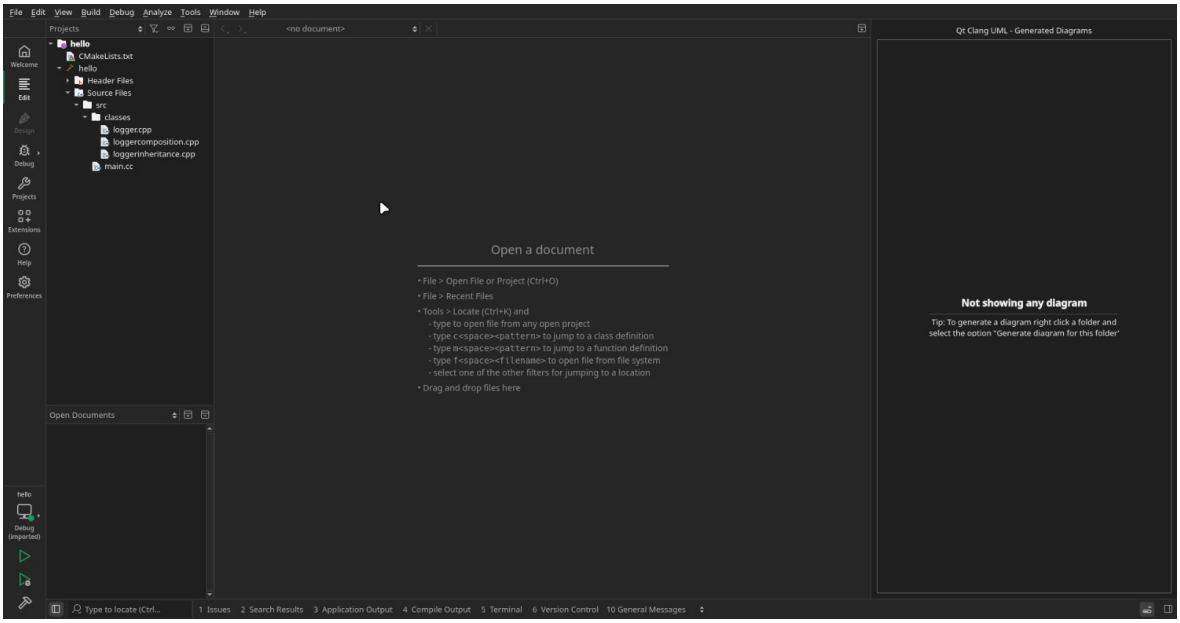

Figura 14: Projeto com sua estrutura de pastas e tela inicial do *plugin*

- Selecione um projeto no menu *Projects* na tela da figura 11. O projeto é exibido em formato de árvore, com seu respectivo código fonte e recursos. Para exibir/esconder a *interface* do *plugin* utilize o atalho Ctrl+Shift+N ou selecione a opção do menu superior *Tools* > *clang-uml* > *Show diagrams*.

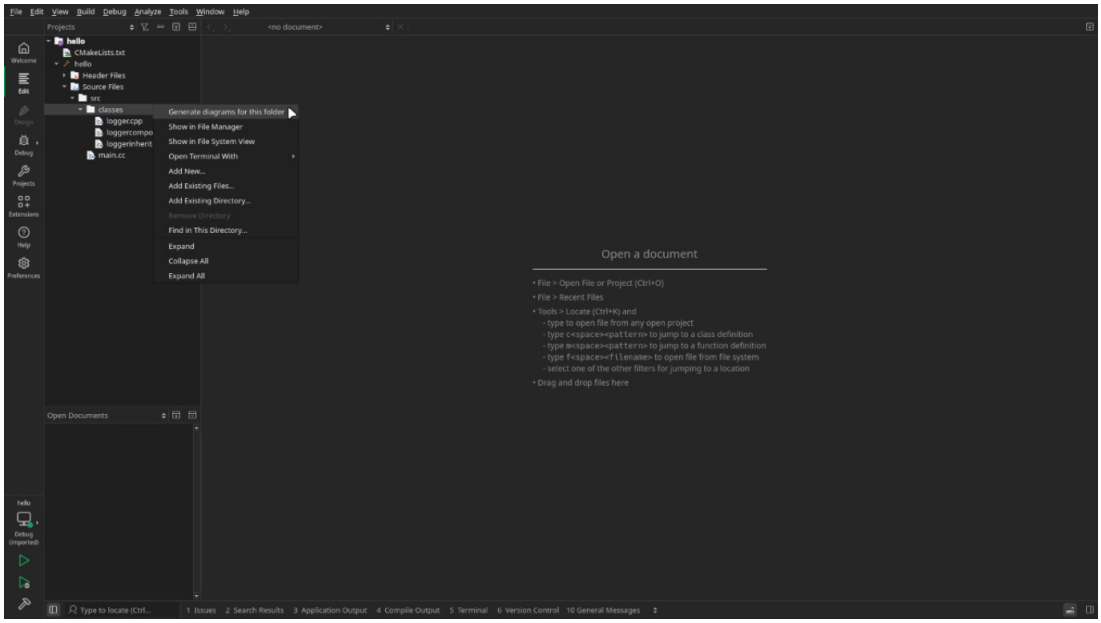

Figura 15: Selecionando a pasta src/classes para geração de diagramas

- Clique com o botão direito do *mouse* na pasta que deseja gerar o diagrama para abrir o menu de contexto, logo depois selecione a opção *"Generate diagrams for this folder"* para iniciar o processo de geração.

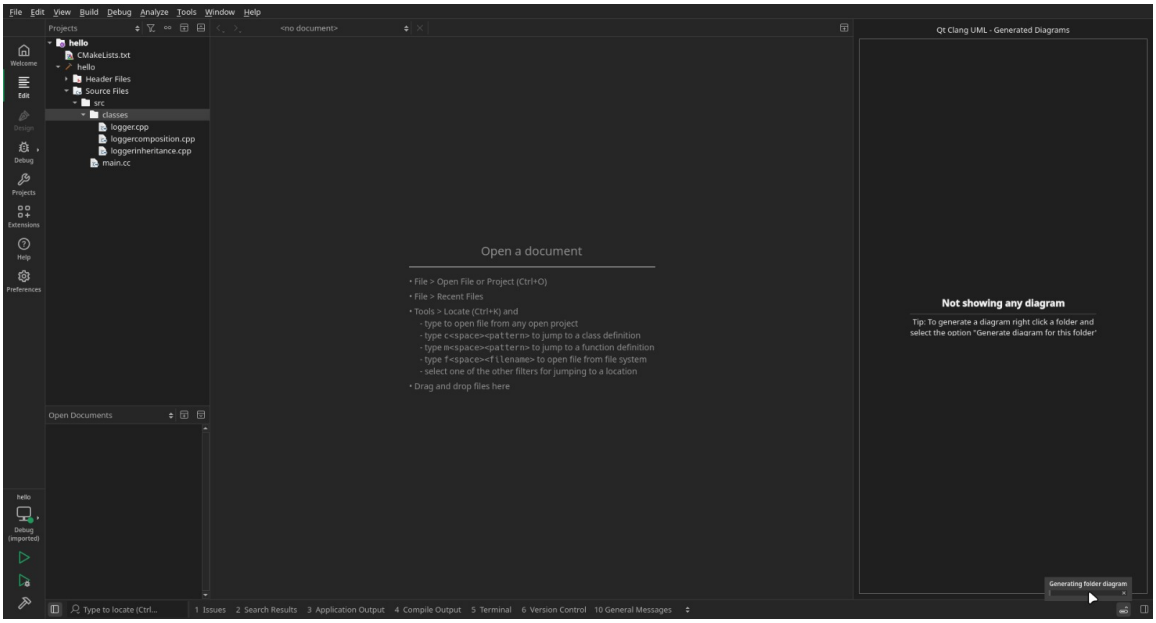

Figura 16: Gerando diagrama para pasta src/classes

- O processo foi iniciado, é possível acompanhar a barra de progresso no canto inferior direito da IDE.

3. Navegando por classes de um diagrama:

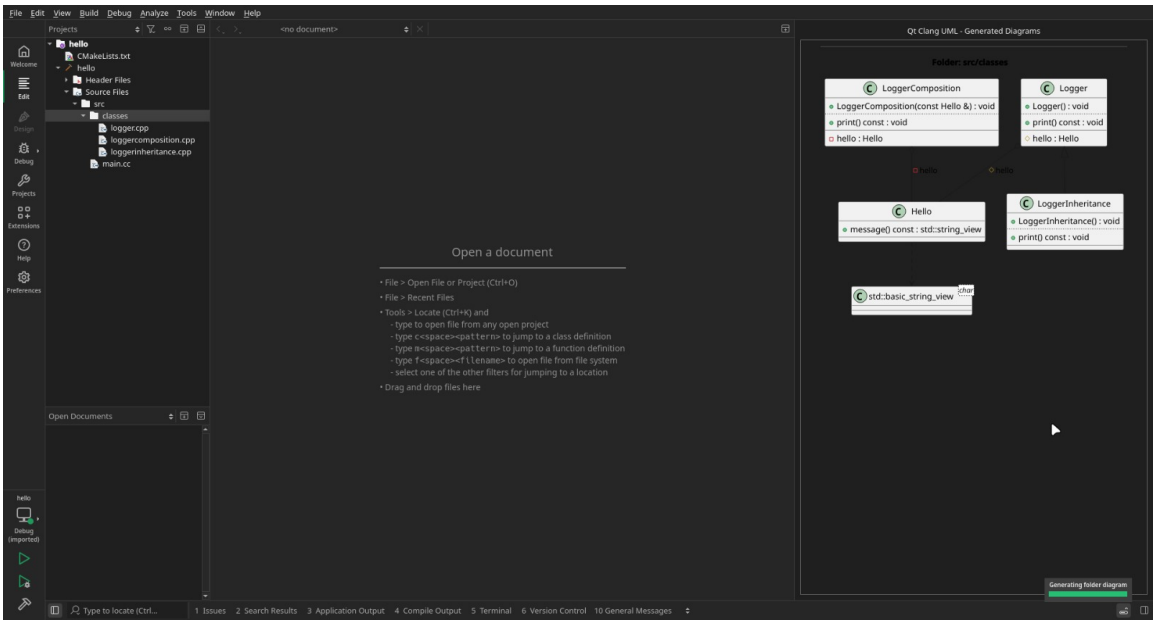

Figura 17: Exibindo diagrama gerado pelo *plugin*

- A barra de progresso é concluída e o diagrama é exibido na *interface* do *plugin*, no lado direito da IDE.

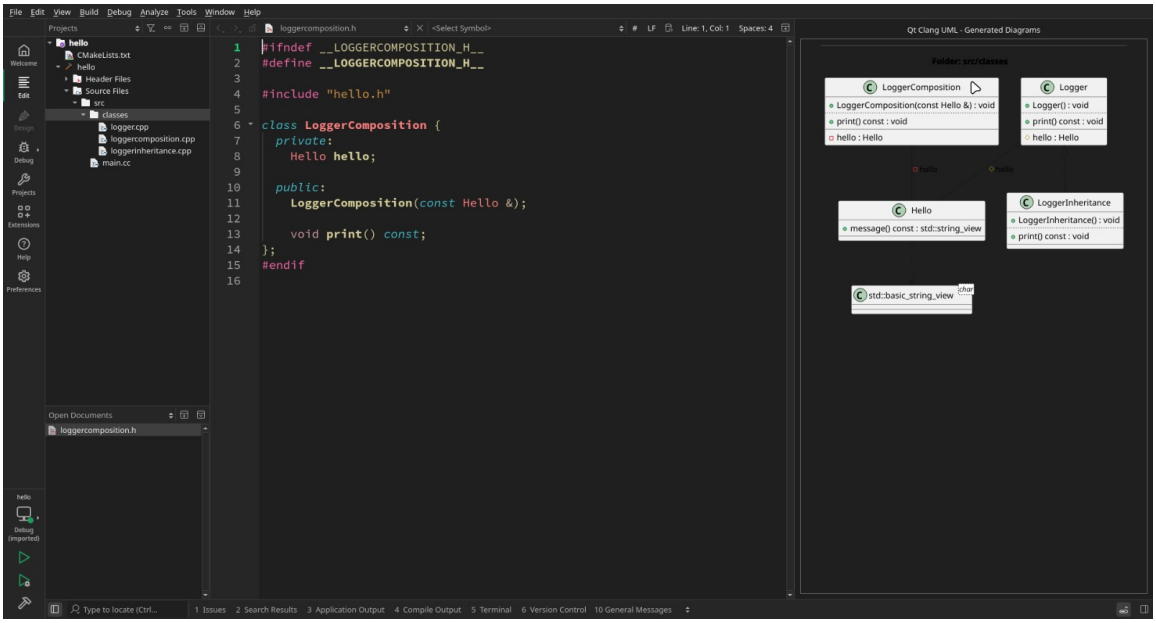

Figura 18: Código de uma classe selecionada pelo usuário

- Para acessar o código fonte das classes exibidas no diagrama basta clicar com o botão esquerdo do *mouse* em seu respectivo componente, por fim o código fonte é exibido no editor de texto do Qt Creator.

Agradecimentos

Muito obrigado a todas as pessoas que fizeram parte dessa jornada. Deixo aqui meus curtos mas sinceros agradecimentos a algumas das pessoas que foram essenciais em minha jornada.

Agradeço a meus amigos do curso, em especial os que fiz no GSORT bem no começo dessa jornada, Camila, Matheus e Yuri. Vocês são pessoas incríveis, sou muito grato pela amizade que criamos.

Agradeço a meu parceiro Gabriel, sem você essa conquista não seria possível. Muito obrigado por estar presente em todas as fases dessa trajetória, nos momentos bons e nos ruins também.

Agradeço a minha mãe, meu pai, minha irmã e toda minha família, que me apoiaram bastante nessa minha jornada. Obrigado tio Pedro, por ter me sugerido este curso quando eu ainda estava incerto do que estudar.

Agradeço aos professores do curso, muito obrigado pelas contribuições acadêmicas e profissionais. Destaco aqui dois professores, Renato por ter sido o verdadeiro começo da minha trajetória profissional e Sandro por ter apresentado a mim e a outros alunos o *framework* deste TCC, o Qt, além de ser o orientador desse projeto.

Por fim, mas não menos importante, agradeço a todos que, de alguma forma, direta ou indireta, contribuíram para essa conquista. Muito obrigado!

Referências

1 OZKAYA, M. Analysing uml-based software modelling languages. In: **Journal of Aeronautics and Space Technologies**. [S.l.: s.n.], 2018. v. 11, p. 119–134.

2 DESHMUKH, R.; KUMAR, S. Unified modeling language (uml) and its contribution to software maintenance efficiency. **Advances in Software Engineering**, p. 1–15, 2023.

3 TIOBE. **TIOBE Index — TIOBE - The Software Quality Company**. 2025. Acesso em: 20 mai. 2025. Disponível em: ⟨https://www.tiobe.com/tiobe-index⟩.

4 Qt Group. **Qt Creator - Qt Wiki**. 2025. Acesso em: 19 mai. 2025. Disponível em: ⟨https://wiki.qt.io/Qt_Creator⟩.

5 ____. **Qt Creator Plug-in Gallery - Qt Wiki**. 2021. Acesso em: 20 mai. 2025. Disponível em: ⟨https://wiki.qt.io/Qt_Creator_Plug-in_Gallery⟩.

6 GEEKSFORGEEKS. **Introduction to C++ Programming Language**. 2019. Acesso em: 02 ago. 2025. Disponível em: ⟨https://www.geeksforgeeks.org/cpp/cpp-programming-intro/⟩.

7 KALB, G. A. J. **C++ Today: The Beast is Back**. [S.l.]: O'Reilly Media, 2015.

8 Qt Group. **Qt - Qt Wiki**. 2025. Acesso em: 02 ago. 2025. Disponível em: ⟨https://wiki.qt.io/About_Qt⟩.

9 GetAppMap. **AppMap**. 2025. Acesso em: 14 jun. 2025. Disponível em: ⟨https://appmap.io⟩.

10 Doxygen. **Doxygen: Main Page**. 2025. Acesso em: 14 jun. 2025. Disponível em: ⟨https://www.doxygen.nl/index.html⟩.

11 Qt Group. **Creating Extensions and Plugins — Extending Qt Creator Manual**. 2025. Acesso em: 13 jan. 2026. Disponível em: ⟨https://doc.qt.io/qtcreator-extending/creating-plugins.html⟩.

12 VAXSERSKI. **Hyprland - Dynamic tiling wayland compositor**. 2022. Acesso em: 10 jan. 2026. Disponível em: ⟨https://github.com/hyprwm/Hyprland⟩.

Apêndices

Glossário, Siglas e Abreviações

[Forneça as definições de todos os termos, siglas e abreviações necessárias à compreensão deste documento.] [Opcional]