Ciência de Dados e I.A.
Escola de Matemática Aplicada
Fundação Getúlio Vargas

Engenharia de Requisitos

Implementação da AST

LLM para Engenharia de Requisitos

Aluno: Isabela Yabe
Orientador: Rafael de Pinho André
Escola de Matemática Aplicada, FGV/EMAp
Rio de Janeiro - RJ.

Rio de Janeiro, 2025

Sumário

0.1 API de geração de embeddings

A API de geração de embeddings constitui a camada final do pipeline de análise estática implementado neste trabalho. Seu objetivo é transformar o código-fonte Python, já representado por meio dos nós sintáticos enriquecidos (`TNodes`), em representações vetoriais densas adequadas para tarefas de mineração de software, recuperação semântica e modelos de apoio à Engenharia de Requisitos. Essa camada integra informações estruturais, textuais e relacionais, produzindo embeddings em três níveis hierárquicos: nó, arquivo e repositório.

0.1.1 Entradas da API: nós enriquecidos e grafo da AST

Após a execução do microkernel de passes, cada arquivo analisado é representado por um conjunto de `TNodes` que descreve, de forma unificada, a estrutura sintática do programa e os metadados extraídos durante o enriquecimento. Cada `TNode` contém informações como nome qualificado, visibilidade, assinatura, tipo de classe ou método, docstrings, profundidade sintática, posição no arquivo e métricas locais do grafo de chamadas. Esses nós constituem a entrada para a etapa de codificação vetorial.

0.1.2 Codificação estrutural dos nós

A primeira etapa da API consiste na extração de *features* estruturais, implementada no módulo `struct_features.py`. Esse módulo define vocabulários categóricos que capturam a natureza sintática do nó (por exemplo, `ClassDef`, `FunctionDef`, `Return`, `Assign`), bem como indicadores derivados dos passes, como visibilidade, tipo de classe, tipo de método e papel lógico do nó na AST reduzida (`core_kind`).

Além disso, são extraídos atributos numéricos, tais como profundidade do nó na AST, número de parâmetros, número de exceções levantadas e diversas flags booleanas que identificam padrões estruturais (como `is_class`, `is_method`, `is_core`). Essas informações são transformadas em um vetor contínuo de dimensão fixa (`STRUCT_DIM`), que sintetiza o papel estrutural do nó no código-fonte.

0.1.3 Codificação textual via SBERT

Embora as propriedades estruturais sejam essenciais, elas não capturam inteiramente o significado semântico dos elementos do código. Por essa razão, a API integra um codificador textual baseado em SBERT (Sentence-BERT), conforme implementado em `tnode_encoder.py`. Nesse módulo, cada `TNode` é convertido em uma descrição textual consolidada que inclui, entre outros elementos: nome qualificado, tokens do identificador, decoradores, classes base, lista de parâmetros, docstring e blocos de comentários associados.

Essa descrição é então processada pelo modelo SBERT, que produz um embedding textual de dimensão `TEXT_EMB_DIM`. Assim, cada nó passa a possuir duas representações complementares: uma estrutural e outra semântica.

0.1.4 Combinação estrutural–semântica

Os vetores estruturais e textuais são concatenados e projetados por um codificador multicamadas (`TNodeMLPEncoder`), resultando em um embedding final de dimensão reduzida (típicamente 128). Essa etapa atua como um *fusion encoder*, combinando informações sintáticas e semânticas de forma robusta e apropriada para processamento gráfico posterior.

0.1.5 Construção do grafo da AST com pesos gaussianos

A etapa seguinte consiste na construção do grafo de dependências sintáticas, conforme implementado no módulo `ast_graph.py`. A função `build_ast_edge_index` gera uma estrutura `edge_index` no formato utilizado por bibliotecas como PyTorch Geometric, conectando nós pais e filhos da AST.

Além disso, o módulo implementa funções de ponderação de arestas baseadas em núcleos gaussianos, que atribuem maior influência a pares de nós sintaticamente próximos (por profundidade ou posição no arquivo). Esse mecanismo introduz um viés posicional útil para modelos gráficos, permitindo que a convolução considere a proximidade estrutural como um fator contínuo.

0.1.6 Rede Neural de Grafos (GNN)

O módulo `gnn.py` implementa a arquitetura utilizada para propagar informações pelo grafo e produzir embeddings globais. A rede consiste em duas camadas de convolução gráfica (GCN), combinadas com normalização simétrica, aplicação dos pesos gaussianos nas arestas e um mecanismo de *global attention pooling*. O fluxo é o seguinte:

1. cada nó recebe seu embedding combinado (estrutura + semântica);
2. as camadas GCN propagam informações entre nós conectados;
3. o mecanismo de atenção atribui pesos relativos aos nós;
4. obtém-se um único embedding representativo de cada arquivo.

Esse embedding global sintetiza características estruturais, semânticas e relacionais do arquivo, capturando padrões de uso de classes e funções, estruturas de controle, interações de chamadas e estilo geral do código.

0.1.7 API de alto nível

Com todos os componentes integrados, a API expõe funções de alto nível que ocultam os detalhes internos e permitem aplicar a GNN de forma simples:

- `encode_file_with_gnn(path)`: retorna os embeddings dos nós, o embedding global do arquivo e os `TNode`s enriquecidos correspondentes;

- `encode_repository_with_gnn(root)`: processa todos os arquivos de um repositório, gerando embeddings por nó, por arquivo e um embedding global do repositório;
- `run_gnn_repository(...)`: função de fachada que integra análise estática, construção da AST, aplicação dos passes e execução da GNN.

Cada chamada retorna estruturas serializáveis que podem ser reutilizadas em tarefas posteriores, como clusterização, busca semântica ou extração automatizada de artefatos de requisitos.

0.1.8 Níveis de embeddings produzidos

A API produz embeddings em três níveis, cada um adequado a diferentes tarefas:

- **Nível de nó**: um embedding por `TNode`, útil para identificar padrões sintáticos e semânticos finos;
- **Nível de arquivo**: um embedding global por arquivo, adequado para agrupamento de componentes, identificação de papéis arquiteturais e análise modular;
- **Nível de repositório**: um embedding único que resume o estilo global, o design e as convenções de um sistema como um todo.

0.1.9 Resumo da API no contexto do TCC

A API de embeddings permite consolidar, em um único pipeline, a análise estrutural, textual e relacional do código-fonte Python. Essa abordagem, fundamentada na combinação de ASTs enriquecidas, SBERT e Redes Neurais de Grafos, torna possível representar código de maneira vetorial, preservando propriedades sintáticas e semânticas que são essenciais para tarefas de Engenharia de Software baseadas em aprendizado de máquina. Esses embeddings constituem a base empírica para as aplicações desenvolvidas neste trabalho, incluindo identificação de comportamentos arquiteturais, agrupamento funcional e suporte à extração automatizada de requisitos.

0.1.10 Grafo da AST e pesos gaussianos nas arestas

A estrutura do arquivo-fonte é modelada como um grafo não direcionado, em que os vértices são os `TNode`s e as arestas representam relações pai–filho da AST. A função `build_ast_edge_index` produz a matriz de adjacência esparsa no formato do PyTorch Geometric, `edge_index` $\in \mathbb{N}^{2\times E}$, contendo arestas bidirecionais entre nós sintaticamente adjacentes.

Além da topologia do grafo, este trabalho introduz uma heurística para atribuir pesos às arestas, com o objetivo de incorporar informação de proximidade estrutural diretamente no mecanismo de *message passing* da GNN. Em vez de tratar todas as arestas com peso unitário, cada aresta (i, j) recebe um peso escalar w_{ij} calculado a partir de uma função gaussiana da distância entre os nós:

$$w_{ij} \;=\; \exp\!\left(-\frac{d_{ij}^2}{2\sigma^2}\right),$$

em que d_{ij} é uma distância escalar derivada de atributos dos nós (por exemplo, diferença de profundidade na AST ou diferença de número de linha no arquivo) e $\sigma > 0$ é um hiperparâmetro de escala. Na prática, a implementação considera duas formas de distância: $d_{ij}^{(\text{depth})}$, baseada na diferença entre os campos `depth` dos nós, e $d_{ij}^{(\text{lineno})}$, baseada na diferença entre os campos `lineno`. Uma variante combinada (`gaussian_edge_weights_combined`) agrega ambas as componentes em um único peso contínuo.

Intuitivamente, pesos gaussianos próximos de 1 são atribuídos a pares de nós estruturalmente próximos (mesma região da AST ou linhas próximas no arquivo), enquanto arestas que conectam nós muito distantes recebem pesos próximos de zero. Dessa forma, a GNN é incentivada a dar mais importância a mensagens trocadas entre elementos que, do ponto de vista sintático e posicional, pertencem ao mesmo "bloco lógico" de código, atenuando a influência de vizinhos mais distantes.

Essa estratégia está alinhada com a intuição de abordagens baseadas em cadeias de Markov sobre ASTs e grafos de fluxo de controle, que modelam explicitamente a probabilidade de transição entre tipos de nós. Aqui, no entanto, essa noção de proximidade é incorporada de forma contínua nos pesos das arestas e utilizada diretamente nas camadas de convolução gráfica. Do ponto de vista metodológico, os pesos gaussianos são tratados como uma heurística: o capítulo de experimentos compara explicitamente o desempenho da GNN em duas configurações — grafo não ponderado ($w_{ij} = 1$ para todas as arestas) e grafo ponderado por `gaussian_edge_weights` — avaliando se a introdução desse viés posicional resulta em melhora na qualidade dos embeddings para as tarefas de interesse deste trabalho.

0.1.11 GNN para representação do grafo de código

Após a construção do grafo da AST e do cálculo dos vetores de features por nó (seção anterior), a ferramenta aplica uma *Graph Neural Network* (GNN) para combinar informações locais e globais do código–fonte em embeddings vetoriais. Essa etapa é implementada pela classe `CodeGNN`, que consiste em duas camadas de convolução em grafos (`SimpleGCNLayer`) seguidas de uma operação de *global attention pooling* para obter um embedding em nível de arquivo.

A classe `SimpleGCNLayer` implementa uma camada do tipo GCN com normalização simétrica e suporte a pesos por aresta. Dado um conjunto de embeddings de nós

$$X \in \mathbb{R}^{N \times d_{\text{in}}}$$

e um conjunto de arestas direcionadas representadas por `edge_index` $\in \mathbb{N}^{2\times E}$, a camada executa os seguintes passos:

1. Adiciona *self-loops* para todos os nós, garantindo que cada nó também considere suas próprias features na agregação. Na prática, isso equivale a trabalhar com a matriz de adjacência $A' = A + I$.

2. Calcula o grau de cada nó, a partir dos destinos das arestas:

$$d_i = \sum_j A'_{ji},$$

e a respectiva normalização simétrica

$$\hat{A}_{ij} = \frac{A'_{ij}}{\sqrt{d_i d_j}}.$$

3. Aplica uma transformação linear aos embeddings de entrada,

$$H = XW, \quad W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}},$$

seguida de uma etapa de agregação por vizinhança:

$$H'_i = \sigma\left(\sum_j \hat{A}_{ji} H_j\right),$$

onde σ é a função de ativação ReLU.

O módulo suporta ainda a incorporação de um peso escalar por aresta (`edge_weight`), derivado de funções gaussianas que dependem da profundidade na AST e/ou da linha de código (seção anterior). Esses pesos atuam como um viés multiplicativo sobre a normalização estrutural, permitindo reforçar ou atenuar contribuições de determinadas arestas:

$$\tilde{A}_{ij} = \hat{A}_{ij} \cdot w_{ij},$$

onde w_{ij} é o peso gaussiano atribuído à aresta (i, j).

A classe `CodeGNN` instancia duas camadas `SimpleGCNLayer` em sequência. A primeira camada projeta os embeddings de nós da dimensão de entrada d_{in} para uma dimensão intermediária d_{hidden}, e a segunda camada produz os embeddings finais com dimensão d_{out}. Essa pilha de duas camadas permite que cada nó agregue informação de vizinhos de primeira e segunda ordem no grafo (por exemplo, chamadas encadeadas ou relações indiretas na AST).

Após as convoluções em grafos, a `CodeGNN` aplica um mecanismo de *global attention pooling* para obter um único vetor representando o grafo (isto é, o arquivo). Seja $H \in \mathbb{R}^{N \times d_{\text{out}}}$ o conjunto de embeddings finais dos nós. A camada de atenção aprende um escalar de importância para cada nó:

$$s_i = \text{MLP}(H_i) \in \mathbb{R},$$

onde MLP é uma pequena rede totalmente conectada. Esses escores são normalizados via *softmax*,

$$\alpha_i = \frac{\exp(s_i)}{\sum_j \exp(s_j)},$$

e o embedding global do grafo é calculado como uma média ponderada:

$$\mathbf{g} = \sum_i \alpha_i H_i.$$

Dessa forma, a `CodeGNN` produz simultaneamente:

- embeddings por nó (H), utilizados para análises mais locais (por exemplo, agrupar métodos ou classes semelhantes); e
- um embedding global $\mathbf{g}$ para cada arquivo, utilizado como representação compacta do código a nível de unidade de compilação.

Na API de embeddings, esse modelo é aplicado sobre o grafo de TNodes gerado pela análise estática. Os vetores de features por nó (combinação de features estruturais e textuais) alimentam a `CodeGNN`, que retorna tanto os embeddings de nós quanto o embedding do arquivo. Em um nível superior, os embeddings de arquivos podem ainda ser agregados (por exemplo, via média) para produzir um embedding em nível de repositório.

0.1.12 Modelagem em grafos: arquivo como unidade de embedding

A etapa de geração de embeddings estruturais do código foi projetada para operar em nível de arquivo, isto é, cada arquivo Python (`.py`) é tratado como um módulo independente e convertido em um grafo AST individual. Essa decisão respeita tanto a semântica da linguagem quanto princípios de engenharia de software que favorecem modularidade, isolamento de responsabilidade e escalabilidade do processamento em grafos.

Em Python, cada arquivo corresponde, por definição, a um módulo. Esse módulo tende a encapsular um conjunto coeso de classes, funções e definições fortemente relacionadas entre si. Ao converter cada arquivo em um grafo separado, o modelo preserva essa unidade natural do código-fonte e a organização lógica definida pelo desenvolvedor. O encoder segue exatamente essa estrutura: (i) cada arquivo gera sua própria AST; (ii) cada AST é convertida em um grafo (`edge_index` e vetores de features por nó); (iii) a GNN produz um embedding global para o módulo; e (iv) o embedding do repositório é obtido pela agregação (por exemplo, média) dos embeddings dos módulos. Assim, obtém-se uma arquitetura hierárquica de representação consistente com a subseção anterior: embeddings em nível de nó, de arquivo e de repositório.

Uma alternativa conceitualmente atraente seria construir um único grafo global contendo todos os nós de todos os arquivos, adicionando arestas adicionais para chamadas entre módulos, relações de importação, herança cruzada e outras dependências estruturais. Entretanto, essa abordagem traz desafios significativos. Em primeiro lugar, há uma explosão de complexidade e custo computacional: repositórios reais podem conter dezenas de milhares de nós de AST e um número ainda maior de arestas, o que aumenta consideravelmente o custo de *message passing* na GNN em termos de memória e tempo de processamento, dificultando a aplicação em projetos de médio e grande porte.

Em segundo lugar, a fusão de todos os arquivos em um único grafo reduz a interpretabilidade do modelo, tornando mais difícil identificar limites naturais entre componentes, compreender o significado do embedding resultante e associar mudanças no embedding a trechos específicos do código. O modelo deixa de refletir, de maneira explícita, a modularidade do projeto. Em terceiro lugar, a abordagem global é menos robusta: um erro em um único arquivo (por exemplo, uma AST inválida ou uma

falha pontual na etapa de encoding) pode comprometer o processamento de todo o grafo. Na abordagem modular adotada neste trabalho, um arquivo com problemas pode ser descartado ou apenas registrado em log, sem impedir a análise do restante do repositório.

Do ponto de vista dos objetivos deste trabalho, a construção de um grafo global também não se mostrou necessária. A extração de estruturas funcionais (como candidatos a casos de uso), a clusterização de métodos públicos e o cálculo de métricas de dependência são viabilizados de forma adequada pelo modelo hierárquico atual, no qual embeddings por arquivo já oferecem uma visão suficientemente rica das responsabilidades de cada módulo.

Em contrapartida, a decisão de gerar embeddings por arquivo proporciona vantagens práticas claras: (i) escalabilidade, pois cada grafo é pequeno o suficiente para ser processado de forma eficiente; (ii) paralelização natural, uma vez que arquivos distintos podem ser analisados em paralelo; (iii) simplicidade de implementação, com uma arquitetura clara, estável e compatível com padrões amplamente usados em GNNs, como *global pooling* por grafo; e (iv) aderência às unidades de significado já presentes no código, alinhando a modelagem matemática com a forma como desenvolvedores organizam seus projetos.

Apesar de a abordagem por módulo ser suficiente para os experimentos conduzidos neste trabalho, a construção de um grafo unificado do repositório surge como uma direção natural de trabalhos futuros. Nesse cenário, o grafo poderia incorporar explicitamente arestas de importação, chamadas entre arquivos, herança cruzada e outras dependências intermódulo, permitindo que o modelo propagasse informação entre módulos de forma mais direta. Essa linha é particularmente promissora para arquiteturas recentes de *Software Graph Transformers* e variantes de GNN projetadas para lidar com grafos gigantes por meio de mini-batches estruturados, e representa um caminho interessante para aprofundar a representação arquitetural de sistemas de software em futuras extensões deste trabalho.

1 cluster

2 Extração e seleção automática de casos de uso

A ferramenta proposta constrói diagramas de casos de uso diretamente a partir do código-fonte da aplicação. A ideia central consiste em tratar determinados métodos de classes como candidatos a casos de uso e, a partir das relações de chamada entre esses métodos, derivar um grafo de dependências que é posteriormente projetado em um diagrama UML de casos de uso.

A abordagem é inspirada em trabalhos como Pereira et al. (**Pereira2011**), que propõem a recuperação de casos de uso a partir de métodos públicos de classes e de suas relações de chamada, mas adapta essas ideias ao contexto de análise estática, uso de embeddings de código e técnicas de clusterização.

2.1 Identificação dos candidatos a casos de uso

O primeiro passo consiste em identificar, na AST (*Abstract Syntax Tree*) dos arquivos analisados, quais nós representam potenciais casos de uso. Para isso, a ferramenta aplica uma regra mínima baseada em duas condições:

- o nó deve representar um método de classe;
- o método deve possuir visibilidade pública.

Em outras palavras, cada método público de uma classe é tratado como um candidato a caso de uso. Para cada candidato, a ferramenta armazena:

- um identificador canônico (`uc_id`), normalmente o nome qualificado do método (por exemplo, `OrderController.handle_request`);
- o nó da AST correspondente (`TNode`), contendo metadados sobre a classe proprietária, visibilidade, posição no arquivo e demais atributos derivados pelos passes de análise;
- informações sobre a classe dona (por exemplo, se é concreta, abstrata ou um protocolo e quais são suas superclasses);
- a lista de outros métodos chamados por esse método, obtida a partir dos passes responsáveis pela análise de chamadas.

Esse conjunto de candidatos corresponde à "visão bruta" dos casos de uso do sistema: todo ponto público de interação exposto pelas classes é considerado um possível caso de uso.

2.2 Embeddings e clusterização de casos de uso

Na etapa seguinte, cada candidato a caso de uso é codificado em um vetor denso (*embedding*). Essa representação combina:

- *features* estruturais do nó da AST, tais como tipo de nó, profundidade, visibilidade e métricas simples de código;
- informação textual associada ao nó, incluindo nome do método, nome da classe, comentários e docstrings.

Essas informações são reunidas em uma representação vetorial e refinadas por meio de uma *Graph Neural Network* (GNN) aplicada sobre o grafo da AST do arquivo, conforme descrito na Seção **??**. O resultado é um embedding para cada método público identificado como candidato a caso de uso.

A partir desses vetores, a ferramenta aplica o algoritmo *k-means* globalmente sobre todos os candidatos, agrupando métodos estrutural e semanticamente semelhantes em até K clusters. Cada cluster é interpretado como um conjunto de casos de uso relacionados, por exemplo, operações pertencentes a um mesmo domínio funcional ou a um mesmo componente controlador.

2.3 Construção das relações entre casos de uso

Com o conjunto de candidatos estabelecido, a ferramenta constrói um grafo de relações entre casos de uso. São considerados três tipos principais de relações: dependência (*dependency/include/extend*), generalização e métricas estruturais derivadas.

2.3.1 Relações de dependência, include e extend

A ferramenta percorre o corpo de cada método público e identifica chamadas a outros métodos públicos. Cada chamada gera uma aresta dirigida entre casos de uso: se o método A chama o método B, cria-se uma aresta $A \rightarrow B$ no grafo de casos de uso. Em um segundo momento, essas arestas são classificadas em:

- **dependency**: caso genérico, quando não há evidências suficientes para caracterizar a chamada como *include* ou *extend*;
- **include**: quando o método de destino é chamado de forma não guardada (isto é, fora de condicionais) e apresenta alto grau de reutilização (fan-in elevado), indicando que seu comportamento faz parte do fluxo principal do caso de uso de origem;
- **extend**: quando o método de destino é chamado apenas em contextos guardados (por exemplo, dentro de condicionais), sugerindo que ele representa um fluxo alternativo acionado sob certas condições.

Essas regras constituem uma heurística que aproxima o grafo de chamadas do código da semântica de *include/extend* da UML, permitindo interpretar métodos reutilizados e fluxos condicionais como passos reutilizáveis ou extensões de casos de uso.

2.3.2 Generalização entre casos de uso

A ferramenta também explora a hierarquia de classes. Quando uma classe filha sobrescreve um método público de sua classe base com o mesmo nome, a ferramenta cria uma relação de generalização entre os casos de uso correspondentes: o método na subclasse é tratado como uma especialização do método na superclasse. No diagrama, essa relação é representada por uma aresta de generalização ($\mathrm{UC}_{\mathrm{filha}} \; - - | > \; \mathrm{UC}_{\mathrm{pai}}$), análoga à generalização entre casos de uso na UML.

2.3.3 Métricas estruturais e papéis dos casos de uso

A partir do grafo de dependências, são calculadas métricas simples para cada caso de uso:

- **fan_in**: número de outros casos de uso que invocam aquele caso de uso;
- **fan_out**: número de casos de uso invocados por ele;
- **include_score**: heurística baseada em `fan_in` e `fan_out` que indica o quão provável é que um caso de uso seja um candidato a *include*.

Com base nessas métricas, cada caso de uso é classificado em um dos seguintes papéis:

- **entry** (entrada): `fan_in` $= 0$ e `fan_out` > 0, sugerindo um fluxo de entrada no sistema;
- **helper** (auxiliar): `fan_in` > 0 e `fan_out` $= 0$, sugerindo um caso de uso reutilizado, mas que não delega para outros;
- **bridge** (ponte): `fan_in` > 0 e `fan_out` > 0, intermediando fluxos entre casos de uso;
- **isolated** (isolado): `fan_in` $= 0$ e `fan_out` $= 0$, sem conexões com outros casos de uso.

2.4 Seleção de casos de uso primários

Nem todos os casos de uso identificados são exibidos no diagrama final. Para evitar diagramas excessivamente densos, a ferramenta seleciona um subconjunto de casos de uso primários (ou focos), combinando informações de clusterização e métricas estruturais.

2.4.1 Ordenação de clusters e escolha de representantes

Inicialmente, os clusters obtidos pelo *k-means* são ordenados pelo número de membros, priorizando grupos maiores. Em seguida, para cada cluster, a ferramenta seleciona um único caso de uso como foco. Os candidatos dentro do cluster são ordenados de acordo com dois critérios:

1. **papel estrutural**: casos de uso classificados como `entry` têm prioridade, seguidos por `bridge`, e, por último, `helper` e `isolated`;
2. **restrição de inclusão/extensão**: casos de uso que são alvo de relações *include* ou *extend* não são considerados primários. A ferramenta assume que, à luz de Pereira et al. (**Pereira2011**) e da própria UML, casos de uso incluídos ou estendidos por outros representam passos reutilizados ou fluxos alternativos e, portanto, não devem ser exibidos como fluxos principais.

O primeiro candidato que satisfaz esses critérios é escolhido como representante do cluster. Em um caso extremo, em que todos os candidatos do cluster sejam "proibidos" (por exemplo, todos são alvo de *include* ou *extend*), a ferramenta recorre ao representante original definido pelo próprio algoritmo de clusterização.

2.4.2 Limite de casos de uso visíveis

O processo descrito é repetido até que se atinja um limite máximo de casos de uso primários (`max_visible_ucs`), definido como parâmetro da ferramenta. O resultado é um conjunto de identificadores de foco (`focus_uc_ids`), que determina quais casos de uso serão efetivamente destacados no diagrama final.

2.5 Construção do diagrama final de casos de uso

A etapa final consiste na construção do diagrama de casos de uso a partir de um especificador de diagrama (`DiagramSpec`), que reúne:

- os casos de uso primários (focos);
- casos de uso secundários diretamente relacionados aos focos por *include*, *extend* ou generalização;
- as arestas de dependência e generalização entre esses nós;
- as métricas e papéis (`entry`, `helper`, `bridge`, `isolated`) de cada caso de uso;
- informações adicionais, como o pacote de origem de cada caso de uso.

A partir desse especificador, o diagrama é exportado em PlantUML, utilizando estereótipos e cores para diferenciar:

- casos de uso de entrada, auxiliares, de ponte e isolados;
- casos de uso primários (representantes de cluster), que são destacados visualmente;
- relações de *include*, *extend*, *dependency* e *generalization*.

Dessa forma, a ferramenta produz automaticamente um diagrama de casos de uso que sintetiza o comportamento exposto pelo código-fonte em termos de fluxos principais (casos de uso primários) e passos reutilizados ou variantes (casos de uso secundários), mantendo coerência com a semântica de *include*/*extend* discutida na literatura e oferecendo uma visão de alto nível da funcionalidade do sistema a partir do código.