



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

Engenharia de Requisitos

Proposta de TCC

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

1	Resumo	1
2	Resumo	1
3	Introdução	1
3.1	Problematização	2
3.2	Questão e hipótese	3
3.3	Objetivos	3
3.4	Relevância	4
4	Revisão literária	4
4.1	Análise sistemática da literatura	5
5	Implicações para este trabalho.	9
6	Implementação	11
6.0.1	Pipeline de análise	13
6.1	Geração de embeddings a partir dos TNodes	15
6.1.1	Embedding por nó: estrutura e texto	15
6.1.2	Grafo da AST e pesos gaussianos nas arestas	15
6.1.3	Entrada para a GNN	16
7	Escolha dos repositórios	16
7.1	Colossal Cave Adventure	17
7.2	Descrição do jogo	17
7.3	17

1 Resumo

2 Resumo

Este trabalho investiga se é possível recuperar artefatos de requisitos, em particular diagramas de casos de uso, diretamente a partir de sistemas implementados em Python, combinando análise estática de código e técnicas recentes de representação semântica com *Large Language Models* (LLMs).

A proposta parte da *Abstract Syntax Tree* (AST) do código-fonte, tratada como modelo intermediário em um fluxo de *Model-Driven Reverse Engineering* (MDRE). Sobre essa estrutura, são extraídas *features* estruturais (tipo de nó, escopo, relações de chamada) e textuais (nomes, docstrings, comentários), que alimentam um encoder de nós e uma GNN responsável por produzir embeddings semântico-estruturais do arquivo. A partir desses embeddings, métodos públicos são identificados como candidatos a casos de uso e agrupados por similaridade semântica, enquanto o grafo de chamadas fornece relações de dependência entre casos.

O resultado esperado é um processo de redocumentação capaz de gerar diagramas de casos de uso em *PlantUML* a partir de código Python, preservando a semântica observada e oferecendo uma visão de alto nível do sistema. A principal contribuição é aproximar engenharia de requisitos e engenharia reversa ao mostrar como LLMs e modelos de código orientados por AST podem apoiar a recuperação de requisitos em cenários em que a documentação está ausente ou desatualizada.

3 Introdução

A engenharia de software estuda e avalia métodos capazes de aproximar o código-fonte da linguagem natural. Essa busca se manifesta em duas vertentes complementares: a interação com o usuário final e a comunicação entre os próprios desenvolvedores.

Este estudo fundamenta-se em autores que defendem o desenvolvimento estruturado e orientado ao usuário, projetado a partir da visão e das necessidades de quem utiliza o sistema, e não apenas da estrutura interna ou das preferências de quem o desenvolve. Essa perspectiva deu origem a princípios de design centrados na função e no comportamento observável do sistema, enfatizando que a organização do código deve refletir a experiência do usuário e os fluxos de interação previstos.

Yourdon e Constantine (1979) descrevem o processo tradicional de desenvolvimento de software como uma cadeia de tradução sucessiva: o diálogo entre o proprietário do produto, o usuário e o analista é continuamente reinterpretado pelo engenheiro de requisitos, pelo designer e pelo programador, conforme ilustrado na Figura 1.

Cada etapa dessa cadeia implica a perda ou distorção de parte do significado original do usuário, o que pode resultar em comportamentos apenas próximos ao desejado. Diante disso, os autores propõem o projeto estruturado, cujo ponto inicial é a clareza e a visibilidade das decisões e atividades envolvidas, promovendo uma compreensão compartilhada e garantindo que o design reflita as intenções originais do sistema.



Figura 1: cadeia de tradução de requisitos segundo Constantine 1979.

3.1 Problematização

Com o mesmo intuito de tornar o comportamento do sistema visível e compreensível, surge a modelagem de casos de uso como um instrumento de unificação entre requisitos, design e usabilidade. Segundo Booch, Rumbaugh e Jacobson (1999), nenhum sistema existe isoladamente: todo sistema relevante interage com atores, humanos ou automáticos, que esperam comportamentos previsíveis. O diagrama de casos de uso permite que analistas e desenvolvedores discutam o comportamento do sistema sem se prender aos detalhes da implementação, oferecendo uma linguagem comum e verificável para representar comportamentos.

Autores posteriores ampliaram essa discussão para o nível do código, enfatizando a necessidade de que o código não seja apenas executável, mas também compreensível. Como sintetiza Fowler (2018), “qualquer tolo escreve um código que um computador possa entender; bons programadores escrevem código que seres humanos possam entender”.

Entretanto, a legibilidade do código, por si só, não substitui a documentação de requisitos. Enquanto o código explica como o sistema se comporta, a documentação torna explícito por que ele deve se comportar assim. Segundo Sommerville e Sawyer (1997), a documentação de requisitos atua como um contrato conceitual entre usuários, analistas e desenvolvedores, garantindo o alinhamento entre o comportamento implementado e as expectativas de negócio. Quando essa documentação falta ou envelhece, a legibilidade do código torna-se o principal ponto de apoio para reconstruir as intenções originais, o que representa um desafio na manutenção e evolução de sistemas legados.

3.2 Questão e hipótese

Se o código é um texto escrito para ser lido por humanos, então suas palavras, nomes e estruturas carregam pistas úteis sobre o que o sistema faz e para quem. Partindo dessa premissa, pergunta-se: é possível reconstruir casos de uso a partir do código-fonte, combinando análise estrutural e interpretação semântica automatizada?

A hipótese deste trabalho é que técnicas de representação semântica, como embeddings e *Large Language Models* (LLMs), quando aplicadas sobre estruturas abstratas do código, como a *Abstract Syntax Tree* (AST), podem viabilizar a reconstrução de artefatos de alto nível, como diagramas de casos de uso, mesmo na ausência de documentação formal.

3.3 Objetivos

O objetivo geral deste trabalho é propor um processo de redocumentação automatizada capaz de gerar diagramas de casos de uso a partir do código-fonte, preservando a semântica do sistema original. Para isso, o método combina:

Brunelière *et al.* (2010) o MoDisco, um framework genérico para engenharia reversa orientada por modelos (Model-Driven Reverse Engineering — MDRE). Ele sugere resumirmos os sistemas em modelos, uma estrutura mais homogênea. A principal ideia é recuperar modelos existentes no sistema. O processo é dividido em duas fases, descoberta do modelo e compreensão do modelo. Na fase de descoberta, um componente chamado discoverer extrai informações do código-fonte, dados brutos, documentações e artefatos disponíveis. Passando estas informações para uma representação da estrutura do sistema. Já na fase de compreensão, o conteúdo desse modelo é analisado e transformado em representações de alto nível, diagramas, métricas ou relatórios, que podem servir à redocumentação, à modernização de sistemas ou à análise de qualidade.

A partir dessa arquitetura, adotaremos a mesma lógica de abstração proposta por Tonella e Potrich (2007), utilizando uma representação sintática reduzida do código-fonte que preserva apenas os elementos essenciais ao fluxo de objetos, criações, atribuições e chamadas, e ignora instruções de controle. Essa simplificação torna possível construir a Abstract Syntax Tree (AST) como modelo intermediário, permitindo representar a estrutura e os diagramas de casos de uso.

Esse tipo de investigação é definido por Chikofsky e Cross (1990) como *Re-documentation* em *Reverse engineering*, ou seja, engenharia reversa com foco em redocumentação, no sentido de criar representações de abstração do sistema existente, destinadas à leitura humana, sem alterar o comportamento do software.

Além da linguagem abstrata, este trabalho incorpora informações semânticas extraídas diretamente das *docstrings*, comentários e nomenclaturas do código. Esses elementos textuais são tratados como extensões dos objetos, pois também comunicam intenções, objetivos e relações entre entidades. Com o apoio de *Large Language Models* (LLMs), essas evidências são analisadas de forma contextual, permitindo inferir papéis, objetivos e interações que não estão explicitamente representados nas chamadas ou estruturas do código.

Dessa forma, o processo de redocumentação combina a análise estrutural, que descreve como os objetos estão correlacionados, e a análise semântica, que interpreta

o vocabulário interno do sistema revelando as intenções dos desenvolvedores.

3.4 Relevância

Este trabalho contribui para auxiliar desenvolvedores durante a codificação e também na compreensão de sistemas sem documentação. Ao gerar visões de alto nível do sistema, especificamente casos de uso, a proposta facilita a compreensão e as interações entre componentes.

Segundo Larman (2002), os casos de uso não apenas documentam funcionalidades, mas representam um instrumento de convergência entre analistas, projetistas e programadores. Em contextos dinâmicos, casos de uso bem definidos apoiam a priorização de requisitos, a validação de comportamentos e a manutenção de uma visão compartilhada do sistema, mesmo diante de mudanças constantes.

Embora a maioria dos estudos sobre Model-Driven Reverse Engineering (MDRE) e redocumentação concentre-se em linguagens como Java, este trabalho propõe uma abordagem direcionada à linguagem Python, que, segundo o TIOBE Index (2025), mantém-se como a linguagem mais popular globalmente.

Por fim, além de oferecer uma nova aplicação prática de Large Language Models na engenharia de software, o estudo propõe uma ponte entre engenharia de requisitos e engenharia reversa, reforçando a ideia de que compreender um sistema começa por compreender seu código, não apenas como sequência de instruções, mas como expressão das intenções humanas que lhe deram origem.

4 Revisão literária

A revisão tem o objetivo de compreender o estado da arte das abordagens de engenharia reversa que partem de código-fonte e produzem artefatos de alto nível, como diagramas UML. Para garantir uma análise sistemática e comparável entre diferentes propostas, foram definidas perguntas de pesquisa (*Research Questions — RQs*) que orientam a coleta e síntese dos dados extraídos dos estudos selecionados.

- **RQ1.** Em quais linguagens e domínios as abordagens que partem de código-fonte foram aplicadas?
- **RQ2.** Quais modelos/artefatos de alto nível são gerados?
- **RQ3.** Qual aspecto é privilegiado (estático, dinâmico, híbrido) e com qual objetivo (compreensão, redocumentação, migração, qualidade)?
- **RQ4.** Quais técnicas e transformações viabilizam a passagem do código para o modelo de alto nível?
- **RQ5.** Quais ferramentas/frameworks são utilizados?
- **RQ6.** Como as abordagens são validadas e com que qualidade prática?

A coleta dos estudos seguiu uma estratégia sistemática de busca em bases reconhecidas, IEEE Xplore e ACM Digital Library no período de 2015 a 2025.

A query se estrutura na combinação de três blocos temáticos:

- ("Abstract": "MDRE"OR "reverse engineering"OR "model driven reverse engineering"OR "design recovery")
- ("Abstract": "UML"OR "UML class diagram"OR "UML activity diagram"OR "UML sequence diagram"OR "UML models"OR "Diagram")
- : ("Abstract": "static analysis"OR "source code analysis"OR "abstract syntax tree"OR "AST"OR "text-to-model"OR "T2M"OR "parser"OR "source code"OR "parsing")

Foram incluídos apenas os estudos que propõem uma abordagem de engenharia reversa aplicada à geração de modelos UML (classes, atividades ou sequência) diretamente a partir do código-fonte.

Foram excluídos os trabalhos que se enquadravam em uma ou mais das seguintes categorias:

- Foco em forward engineering ou geração de código.
- Estudos centrados em rastreabilidade ou anti-padrões.
- Trabalhos puramente empíricos ou teóricos sem proposta de transformação automatizada.
- Abordagens puramente dinâmicas.

Com base nos critérios de inclusão e exclusão, foram selecionados os seguintes estudos para análise detalhada:

- A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code (2019).
- Condensing Class Diagrams With Minimal Manual Labeling Cost (2016). (parte do diagrama e aperfeiçoamento)
- Enhancing Model-Driven Reverse Engineering Using Machine Learning (2024).
- Reverse Engineering of Source Code to Sequence Diagram Using Abstract Syntax Tree (2016).
- Towards a New Hybrid Approach of the Reverse Engineering of UML Sequence Diagram (2016).
- WIP: Generating Sequence Diagrams for Modern Fortran (2017).

4.1 Análise sistemática da literatura

A partir da síntese da Tabela 1, organizamos os achados por eixo (RQ1–RQ6).

Autores / Referência	Linguagem / Domínio	Modelo Gerado	Aspecto	Técnica / Tipo de Transformação	Ferramenta / Framework	Validação / Estudo de Caso
Zhang (2016)	Java; pequenos sistemas OO (eLib, Minesweeper, Blog, PayrollSys, myAlgsLib)	UML Classe; UML Sequência	Estático — compreensão, manutenção/redocumentação	Código→AST→J2X; mapeamentos (gen./impl./assoc./dep.); sentenças simplif.→OFC; CFG+OFC→Sequência	J2UML; JavaCC; Dom4j; J2X (DTD/XML)	5 casos pequenos; acurácia: classes 96,4-100%; relações 65,0-90,4%
Yang <i>et al.</i> (2016)	Java; sistemas OO	UML Classe (condensado)	Estático; COMPREENSÃO/REDOCUMENTAÇÃO; estrutural	Extração de métricas (SDMetrics) → Normalização (z-score) → k-means clustering → Random undersampling → Ensemble learning (Random Forest) → diagrama condensado	MagioDraw; SDMetrics; Random Forest; Windows 7	OSS (9 projetos, 2640 classes); AUC=0.73; custo de rótulo=10%; teste de Wilcoxon e Cliff's δ
Fauzi, Hendradjaya e Sumindyo (2016)	Java; sistemas orientados a objetos	UML Sequência (comportamental)	Estático; COMPREENSÃO/REDOCUMENTAÇÃO	Código → AST (JavaParser) → DFS pós-ordem → PlantUML (Seq)	REVUML; JavaParser; PlantUML	126 casos de teste (8 categorias; geração correta e consistente de diagramas)
Baidada e Jakimi (2016)	Java/Genérico; aplicações OO	UML Sequência (HLSD)	Híbrido (Estático + Dinâmico); Compreensão/Redocumentação (comportamento)	CFG→entradas; execuções+traços (filtro); traços→CPN; CPN→UML SD	Sem ferramenta nominal; UML 2; instrumentação/VM/debugger; CPN (IR)	Sem validação; futuro
Leatongkam, Nanthamornphong e Rouson (2017)	Fortran OO; computação científica e engenharia	UML Classe; UML Sequência; modelo intermediário XMI	Estático — compreensão e redocumentação	Regras de mapeamento código → UML (OMG); parsing estático; árvore sintática; geração XMI → importação ArgoUML	ForUML (extensão); ArgoUML; padrão OMG UML/XMI;	<i>Work in progress</i>
Sabir <i>et al.</i> (2019)	Java (sistemas legados orientados a objetos)	UML <i>Class Diagram</i> + <i>Activity Diagram</i> (em pacote UML)	Estático; objetivo: compreensão/redocumentação	T2M/M2M em duas fases: Parser→AST→IM(XML) e mapeamentos IM→UML (classe/atividade)	Eclipse + UML2/EMF; JavaParser (IMD); (Papyrus/StarUML/Rational Rose na validação manual)	Comparação especialista (modelos manuais vs. gerados), 5 estudos de caso; ATM e Amandeus descritos
Siala (2024)	Java; Python; sistemas legados	UML Classe; OCL	Estático — compreensão, redocumentação e migração	código → tokenização/simplificação → geração textual UML/OCL intermediária(LLM) → model repair → diagramas UML/OCL	Graphviz; PlantUML; Modelio; AgileUML; LLM	Comparação MDRE; dois estudos de caso; correção semântica, completude e compreensibilidade

Tabela 1: Síntese comparativa dos estudos selecionados.

RQ1 — Linguagens e domínios. Predomina o ecossistema **Java** em sistemas orientados a objetos, tanto em estudos estruturais quanto comportamentais (Zhang (2016), Yang *et al.* (2016), Fauzi, Hendradjaya e Sunindyo (2016) e Sabir *et al.* (2019)). Há ampliação pontual para **Fortran OO** em contexto de computação científica (Leatongkam, Nanthaamornphong e Rouson (2017)) e menção tanto a **Java** quanto a **Python** em proposta recente com LLMs (Siala (2024)). Em síntese, o corpus avaliado é fortemente dominado por Java, Python surge como alvo ainda subexplorado.

RQ2 — Modelos/artefatos gerados. A produção concentra-se em **UML Classe** e **UML Sequência**. Em Zhang (2016), ambos são gerados a partir de um pipeline *código* \rightarrow *J2X* \rightarrow *CFG/CFG* \rightarrow *UML* (Classe+Sequência). Leatongkam, Nanthaamornphong e Rouson (2017) propõe estender o *ForUML* para também extrair **Sequência** a partir de Fortran OO, exportando um **XMI** intermediário para visualização (e.g., ArgoUML). Fauzi, Hendradjaya e Sunindyo (2016) derivam **Sequência** diretamente da **AST**, com saída em *PlantUML* (Seq), abordagem estática focada em interações. Sabir *et al.* (2019) incluem **Activity** além de **Class**, gerando “modelos UML de alto nível” (classe + atividade) a partir de um modelo intermediário (UML2). Yang *et al.* (2016) não “geram” um novo tipo de diagrama, mas *condensam diagramas de classe* via métricas + *ensemble learning*, reduzindo a complexidade visual ao destacar “classes importantes” (AUC, testes de Wilcoxon/Cliff’s δ). Em contraste, **Casos de Uso** aparecem sobretudo como enquadramento conceitual para Sequência, mas não como artefato recuperado dos códigos analisados, sinalizando uma lacuna na redocumentação de requisitos a partir de código.

RQ3 — Qual aspecto é privilegiado (estático, dinâmico, híbrido) e com qual objetivo? No conjunto analisado, prevalece de forma nítida o aspecto **ESTÁTICO**, quase sempre orientado à **COMPREENSÃO/REDOCUMENTAÇÃO**. Os trabalhos clássicos da vertente estrutural e comportamental, como Zhang (2016) e Fauzi, Hendradjaya e Sunindyo (2016), operam integralmente sobre o código (sem execução), partindo de *parsing*/AST e passando por representações intermediárias (p.ex., J2X) ou travessias específicas (DFS pós-ordem) para derivar, respectivamente, diagramas de Classe e Sequência. Em Leatongkam, Nanthaamornphong e Rouson (2017), a mesma orientação estática se mantém ao estender o *ForUML* para Fortran OO via regras de mapeamento e exportação XMI; e em Sabir *et al.* (2019), trata-se de um processo estático T2M com fluxo *código* \rightarrow AST \rightarrow IM \rightarrow UML, com objetivo voltado à compreensão e redocumentação. Ainda sob a ótica estática, Yang *et al.* (2016) não cria um novo artefato, mas trata o *pós-processamento* do diagrama de classes via métricas e *machine learning*, reduzindo a complexidade visual sem recorrer a dados de execução. Em contraste com esse predomínio, Baidada e Jakimi (2016) introduzem um caminho **HÍBRIDO** (estático + dinâmico) para Sequência: gera-se um conjunto de entradas a partir do CFG, coletam-se traços por instrumentação/VM, sintetiza-se uma IR (*Intermediate Representation*) comportamental em *Colored Petri Nets* e, então, mapeia-se para UML. Por fim, Siala (2024) preserva o caráter **ESTÁTICO** ao integrar LLMs como camada semântica (texto intermediário UML/OCL seguido de *model repair*), ampliando o objetivo para **migração** em sistemas legados e apontando

uma inflexão do “estrutural puro” para um *estrutural + semântico*.

RQ4 — Técnicas e transformações. As abordagens convergem em um esqueleto MDRE que encadeia *Text-to-Model* e *Model-to-Model*: análise sintática do código (*parsing*) para **AST** ou **IR** textual, seguida de mapeamentos para o metamodelo UML. Ainda assim, diferem nos *intermediários*, nos operadores de fluxo usados para recuperar comportamento e no quanto incorporam *semântica* além da sintaxe. Em Zhang (2016), o núcleo é a **J2X** (DTD/XML), uma IR que padroniza elementos de linguagem; o diagrama de classes surge de metadados extraídos dessa IR, enquanto o diagrama de sequência resulta da **integração OFG+CFG** (rastros de objetos + fluxo de controle) para identificar *lifelines*, *messages* e *combined fragments* (**alt/opt/loop**) de modo inteiramente estático. Fauzi, Hendradjaya e Sunindyo (2016) elimina o XML e vai direto da **AST** (JavaParser) para *Sequence*, guiado por uma travessia **DFS pós-ordem** com registro de variáveis, resolução de herança/polimorfismo e marcação de estruturas condicionais/iterativas; a apresentação é automatizada via **PlantUML**. Já Sabir *et al.* (2019) formalizam o pipeline clássico **T2M/M2M** em duas fases: da AST para um **Intermediate Model (XML/EMF)** e, então, do IM (*Intermediary Model*) para **UML** (Classe + *Activity* por operação), com regras de transformação implementadas no ecossistema Eclipse/UML2. O **híbrido** de Baidada e Jakimi (2016) desloca a recuperação comportamental para uma IR executável: um **CFG** orienta a geração de entradas; execuções instrumentadas produzem *traces* filtrados; esses traços são sintetizados como **Colored Petri Nets (CPN)** e finalmente mapeados para *UML Sequence*, capturando paralelismo e operadores combinados. Em domínio não-Java, Leatongkam, Nanthaamornphong e Rouson (2017) mantém a análise estática por **regras formais de mapeamento** (Fortran OO → UML), uma *tree node structure* análoga à AST, e geração de **XMI** para importação/visualização no ArgoUML, expandindo o ForUML para *Sequence*. Duas linhas recentes aplicam *aprendizado*: Yang *et al.* (2016) não cria um novo artefato, mas **condensa** o diagrama de classes com um pipeline *métricas* → *normalização (z-score)* → *k-means* → *under-sampling* → *ensemble (Random Forest)*, priorizando classes “importantes” e reduzindo a complexidade visual; e Siala (2024) introduz **LLMs** como camada *semântica*: o código é tokenizado/simplificado, traduzido para uma **representação textual intermediária UML/OCL**, submetida a *model repair* e convertida em diagramas (PlantUML/Graphviz/Modelio).

RQ5 — Ferramentas/frameworks. O ecossistema técnico das abordagens analisadas agrega *parsers*/geradores UML, frameworks MDE e utilitários de visualização/mineração. Em Zhang (2016), a cadeia J2X apoia-se na **J2UML** (orquestração), no **JavaCC** (geração do parser/AST), em **DOM4J** (manipulação XML) e no próprio **J2X (DTD/XML)** como IR; os experimentos reportam ambiente Windows 32-bit (3 GB RAM; Core 2 Duo). Em Yang *et al.* (2016), para “condensar” diagramas de classes, utilizam-se **MagicDraw** (recuperação de Classe), **SDMetrics** (métricas), e **Random Forest** (classificador), com relatos do ambiente *Windows 7* (64-bit). A Fauzi, Hendradjaya e Sunindyo (2016) (REVUML) integra **JavaParser** (AST) e **PlantUML** (renderização do Sequência), dispensando IR XML intermediária. Em Baidada e Jakimi (2016), não há ferramenta nominal de ponta a ponta: a coleta

se dá por **instrumentação/JVM/debugger**, a IR comportamental usa **Colored Petri Nets** (sugerindo uso de *CPN Tools*), e o mapeamento segue **UML 2**. Em Le-atongkam, Nanthamornphong e Rouson (2017), a extensão da **ForUML** gera **XMI (OMG)** para importação no **ArgoUML** (visualização de Sequência). No framework Sabir *et al.* (2019) (Src2MoF), o gerador baseia-se em **Eclipse+UML2/EMF** e o **JavaParser** integra o IMD; ferramentas como **Papyrus/StarUML/Rational Rose** são citadas apenas para comparação manual, não no pipeline automático. Por fim, Siala (2024) combina **AgileUML/OMG MDA** com **LLMs** (camada semântica) e usa **Graphviz, PlantUML** e **Modelio** para materializar *UML/OCL* (fase M2V).

RQ6 — Validação e qualidade prática. A avaliação varia de **estudos de caso pequenos com acurácia estrutural** (classes 96,4–100%, relações 65,0–90,4) (Zhang, 2016), a **testes sistemáticos** de geração de sequência (Fauzi; Hendradjaya; Sunindyo, 2016), e **comparação especialista** sem métricas quantitativas (Sabir *et al.*, 2019). Yang *et al.* (2016) recorre a **AUC** e testes estatísticos (Wilcoxon, Cliff’s δ) para condensação de classes. Baidada e Jakimi (2016) não reporta validação empírica. Em suma, há carência de avaliações *comparativas* com ground truth e métricas padronizadas na maioria dos trabalhos.

5 Implicações para este trabalho.

Plano estático (objetivo: casos de uso a partir de código Python)

Código \rightarrow AST (IM/IR). Parser de Python para construir a AST como modelo intermediário.

Candidatos a casos de uso. Heurística estática: funções/métodos “públicos” viram use cases candidatos.

Clusterização para agrupar intenções. Embeddings de código e de AST (NFASRR-TRANS) para representar o código fonte; K-means/HDBSCAN agrupam métodos por intenção/semântica. – O centróide do cluster nomeia o caso de uso; demais métodos viram passos/serviços/extends/includes.

Relacionamentos estáticos.

include: subrotinas/serviços invocados por $\geq k$ casos de uso candidatos (reuso sistemático no grafo de chamadas).

extend: chamadas opcionais guardadas por condições (flags/feature toggles/validações) detectadas no AST/CFG; marcam “pontos de extensão” do caso de uso base.

generalização (atores e casos): herança/implementação e sobrecarga de nomes (métodos homônimos (Shape, TextShape, GeometryShape) em super/subclasses) indicam especializações; atores inferidos de camadas de borda (controllers, adapters, decorators de rota) - rever.

dependência entre casos: mapeada de chamadas entre métodos “públicos” \rightarrow dependência entre use cases.

Geração (PlantUML). Serialização dos nós (casos, atores) e arestas (include/extend/generalização/dependência) em PlantUML.

Extração de casos de uso a partir de métodos públicos. Pereira, Martínez e Favre (2011)

Generalização por herança/nomes homônimos. O mesmo trabalho mapeia herança estática para generalização. Fonte usada: regras de transformação (ATL) — adotamos a mesma leitura para Python (classes e MRO).

Dependências entre casos a partir de chamadas. Fonte usada: regra Dependency2Dependency — base para nossas arestas de dependência/include.

Embeddings estáticos guiados por AST. Liang e Huang (2024) descrevem um modelo com dois encoders (um para semântica lexical e outro para estrutura de AST) para sumarização de código. Assim aproveitamos os embeddings para melhorar a qualidade dos clusters.

Implicações para este trabalho. Propomos redocumentação semântica a partir de Python, combinando (i) AST como modelo intermediário (IM/IR) e (ii) embeddings sensíveis à estrutura (NFASRPR-TRANS) para inferir intenção e papel de métodos (docstrings, comentários e nomes). Casos de uso são inferidos por métodos de borda; generalizações vêm de herança/assinaturas homônimas; include decorre de subrotinas reutilizadas; extend de ramos opcionais. A saída é um Use Case Diagram em PlantUML. (Base: Pereira — extração estática por métodos públicos; embeddings orientados a AST.)

código \rightarrow AST \rightarrow embeddings+clusterização \rightarrow regras estáticas (include/extend/atores) \rightarrow PlantUML

De onde vem: mapeamento “método público \rightarrow caso de uso” e “chamada entre públicos \rightarrow dependência” (Pereira), robustez semântica dos vetores via AST-encoder.

Representação: nós = atores, casos; arestas = include, extend, generalização, dependência.

Sinais estáticos:

Atores: classes/módulos de interface (controllers/views/adapters; decorators de rota/CLI).

include: função/método chamado por $\geq k$ casos distintos; baixa complexidade ciclomática; sem guardas específicas.

extend: chamadas condicionais (predicados sobre entrada/estado/feature-flag) marcando “pontos de extensão”.

generalização: herança + homonímia de assinatura nos métodos.

$score = \alpha \cdot (\text{freq. de reuso}) + \beta \cdot (\text{centralidade no call graph}) - \gamma \cdot (\text{força da guarda condicional})$

Saída: .puml gerado a partir do grafo.

Elementos e regras de extração (todos estáticos):

Casos de uso (UC): cada método público vira um UC com o mesmo nome do método (no seu caso, prefixe com o nome da classe para evitar colisões). Trecho-base (“Recovering Use Case Diagrams from Object Oriented Code”): “Basic use cases are extracted analyzing public methods. Each public method of a class corresponds to a use case whose name is the same as the method name.”

Generalização (entre UCs): quando houver herança e mesmo nome/assinatura entre métodos em classes distintas, gere generalização entre os UCs correspondentes: “For each pair of same-name methods which are members of different classes with a generalization between them, a generalization between use cases... is generated.”

Dependência entre UCs (aresta genérica): para cada chamada $p.g()$ dentro de um método público m , crie dependência entre $UC(m)$ e $UC(g)$: “For each public method of a class, the method calls are analyzed to extract dependence relationships

among basic use cases. Each message sent to an object corresponds to a dependence between use cases.”

→ Decisão do seu trabalho (100% estático): manter apenas “Dependency” (sem classificar em «include»/«extend»), pois a própria autora afirma que o tipo (include/extend) “may be inferred by a dynamic analysis” — que você não quer usar.

Atores: não inferir automaticamente (manter passo manual/regras externas), pois “actors can not be automatically inferred”.

Passos do processo (estático): (1) gerar código abstrato/AST; (2) executar algoritmo de recuperação estática; (3) (dinâmico para classificar dependências) — omitido no seu trabalho; (4) definir UCs de alto nível.

Elevação de nível (agrupar UCs): aplicar heurísticas de clusterização para formar UCs mais abstratos: “use cases may be clustered in single more abstract use cases by applying simple heuristics”. (Heurísticas a/b do passo 4).

Observação prática: você pode manter todas as arestas como Dependency e, apenas na visualização, sinalizar candidatos a include (reuso alto/fan-in elevado) e candidatos a extend (dependências sob guarda condicional detectável no AST/CFG). Isso segue a letra do artigo (classificação formal requer dinâmica), mas entrega valor prático na redocumentação.

6 Implementação

Este capítulo descreve a implementação para representar código-fonte Python a partir de sua Árvore Sintática Abstrata (AST) e gerar embeddings semânticos para apoio a tarefas de análise de código. A solução foi organizada em três camadas principais: (i) um núcleo de análise estática que constrói a AST e produz a estrutura intermediária TNode; (ii) uma arquitetura de microkernel baseada em passes plugáveis, responsável por enriquecer cada nó com metadados estruturais e semânticos; e (iii) uma camada de representação vetorial, que combina features estruturais e textuais em um encoder de nós e, em seguida, aplica uma GNN sobre o grafo da AST para obter embeddings por nó e um embedding global do arquivo.

O TNode é a representação intermediária que agrega a AST bruta com as informações derivadas pelos passes (Figura 2). Sua definição é organizada em blocos de campos, cada um associado a um grupo de passes especializado: Informações de caminho (`path_info`); Informações de nomes e visibilidade (`names_visibility` e `naming`); Tipo de método e assinatura de I/O (`method_kind` e `io_signature`); Tipo de classe (`class_kind`); Documentação e comentários (`docs_comments`).

Nenhum desses campos é oferecido diretamente pela AST nativa do Python; todos são inferidos pelo conjunto de passes executados.

Módulos principais da implementação do ASTCore (Figura 4):

- **astcore.model**: define as classes centrais `Ctx` e `TNode`, utilizando `@dataclass` da biblioteca padrão do Python. Essa camada encapsula a representação intermediária e é independente dos passes de análise.
- **astcore.walker**: implementa as estratégias de travessia da AST nativa do Python (`recursive_pre`, `iterative_pre`, `bfs`), produzindo uma sequência

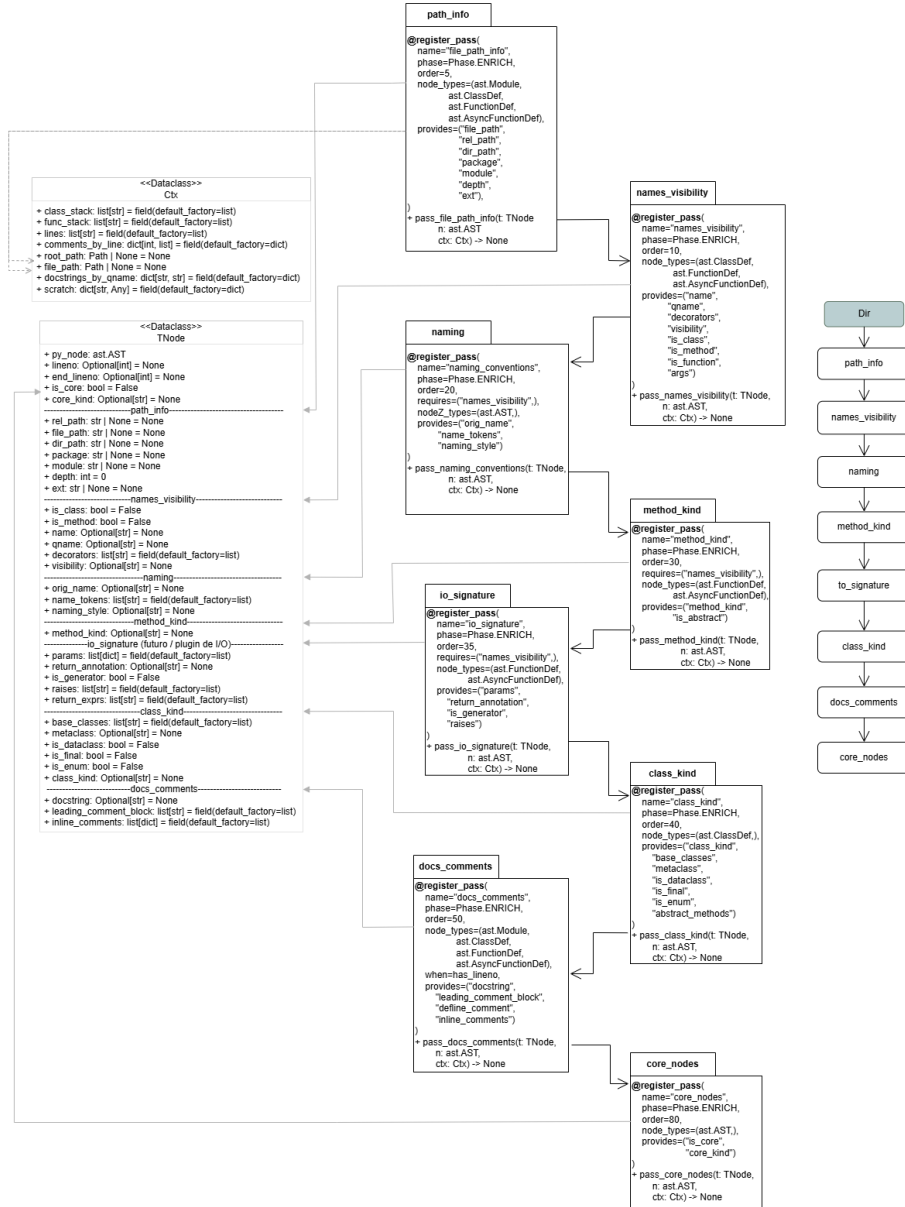


Figura 2: Plugins.

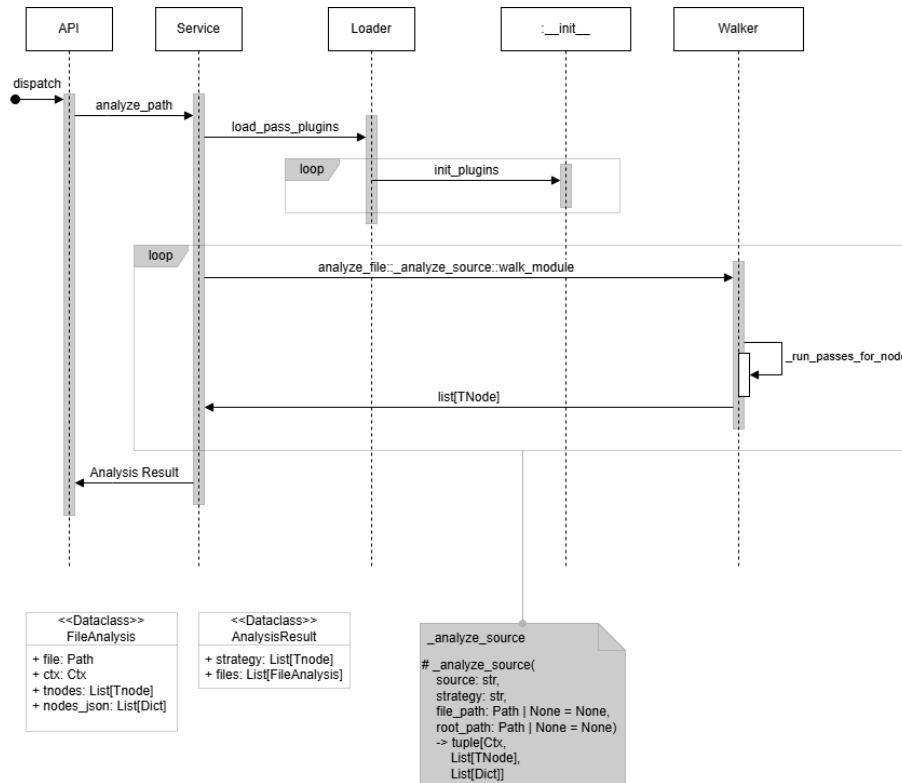


Figura 3: Diagrama de sequencia da construção do TNode.

de TNodes a partir de um `ast.Module`. Essa camada atua como um *visitor* especializado, mas desacoplado dos detalhes de cada análise.

- **pass_plugins**: pasta de plugins que contém os passes responsáveis por enriquecer os nós (`names_visibility`, `method_kind`, `class_kind`, `docs_comments`, `naming`, `io_signature`, `path_info`, etc.). Cada plugin conhece apenas a API de `Ctx` e `TNode`, e não interage diretamente com outros plugins.
- **pass_plugins.pass_registry**: módulo que registra os passes disponíveis, resolve dependências entre eles e determina a ordem de execução. Funciona como um *registry* e uma pequena camada de *inversão de controle*: o usuário informa quais passes carregar e o `PassRegistry` garante que sejam aplicados na ordem correta.
- **src.service**: módulo de alto nível que oferece operações como `analyze_path` e `export_json`. Ele atua como *fachada* do `ASTCore`, orquestrando carregamento de plugins, travessia da AST, enriquecimento dos nós e serialização dos resultados.

6.0.1 Pipeline de análise

Do ponto de vista operacional, o fluxo de implementação do `ASTCore` segue as seguintes etapas:

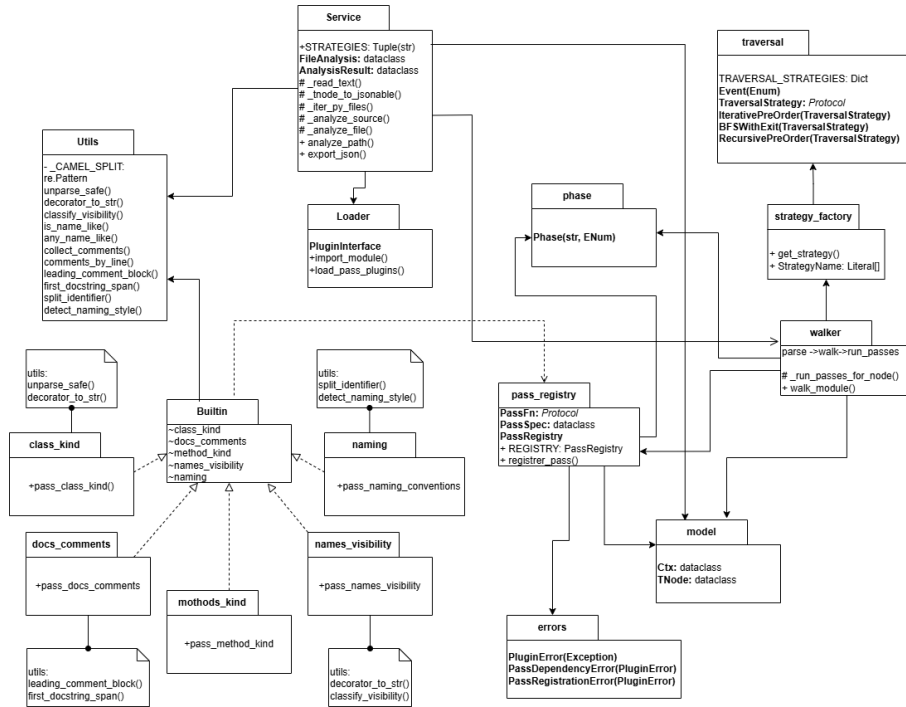


Figura 4: Arquitetura modular do ASTCore.

1. **Descoberta de arquivos:** a função `analyze_path` recebe um caminho raiz de projeto e itera sobre os arquivos `.py`. Para cada arquivo, o código-fonte é lido por uma função utilitária (*wrapper* de `Path.read_text`) que tenta múltiplas codificações e retorna o texto bruto.
2. **Construção da AST nativa:** o texto é convertido em uma AST do Python via `ast.parse`. Essa AST é a “fonte de verdade” sintática sobre a qual o ASTCore opera.
3. **Inicialização do contexto:** é criado um objeto `Ctx` com as linhas do arquivo (`lines`), o mapeamento de comentários por linha (`comments_by_line`) e os caminhos `root_path` e `file_path`. Esse contexto é compartilhado por todos os passes.
4. **Travessia da AST (`walk_module`):** o módulo `astcore.walker` percorre a AST segundo a estratégia selecionada (por exemplo, `recursive_pre`). A cada nó visitado, é criado um `TNode` contendo a referência ao nó original (`py_node`) e informações estruturais básicas (`lineno`, `end_lineno`, etc.).
5. **Aplicação dos passes:** para cada `TNode`, o `PassRegistry` aplica em sequência os plugins carregados (`load_pass_plugins`). Cada plugin lê e atualiza campos específicos do `TNode`, respeitando o princípio de separação de preocupações.
6. **Empacotamento dos resultados:** ao final da análise de um arquivo, o módulo `service` cria instâncias de `FileAnalysis` (arquivo, contexto, lista de `TNodes` e versão serializada de cada nó). Para uma análise completa, os resultados são

agrupados em um `AnalysisResult` contendo a estratégia utilizada e a lista de arquivos analisados.

7. **Serialização:** a função `export_json` converte cada `TNode` em um dicionário JSON-*friendly*, preservando tanto os metadados estruturais quanto os campos enriquecidos pelos passes. Esse JSON é o insumo direto para as etapas seguintes do TCC (tokenização semântica, geração de embeddings e clusterização de candidatos a casos de uso).

6.1 Geração de embeddings a partir dos TNodes

A representação vetorial dos nós da AST ocorre em duas etapas. Na primeira, cada `TNode` é codificado individualmente em um vetor d -dimensional que combina características estruturais e textuais. Na segunda etapa, o grafo da AST recebe pesos escalares em suas arestas, calculados por funções gaussianas que dependem da distância relativa entre os nós (em profundidade e/ou número de linha). Essa etapa é inspirada no uso de *Gaussian embedders* proposto por Liang e Huang (2024), que empregam núcleos gaussianos para codificar posições relativas em modelos *Transformer*; neste trabalho, a mesma ideia é adaptada ao contexto de grafos, sendo aplicada como viés multiplicativo nas arestas durante a agregação de mensagens da GNN.

6.1.1 Embedding por nó: estrutura e texto

A função `build_node_feature_matrix` transforma a lista de `TNodes` em uma matriz de atributos $X \in \mathbb{R}^{N \times d}$, em que cada linha corresponde ao embedding de um nó.

O vetor de entrada de cada nó é construído a partir de dois grupos de atributos estruturais e textuais.

Os vetores estrutural e textual são então concatenados e passados por uma MLP (`TNodeMLPEncoder`), definida em `tnode_encoder.py`, que projeta o resultado em um espaço de dimensão d (neste trabalho, $d = 128$):

$$\text{encode_tnode}(t) = \text{MLP}\left(\text{structural_features}(t) \parallel \text{text_features}(t)\right) \in \mathbb{R}^d.$$

A matriz final de atributos de nó é dada por

$$X = \text{stack}\left(\{\text{encode_tnode}(t) \mid t \in \text{tnodes}\}\right) \in \mathbb{R}^{N \times d},$$

combinando simultaneamente propriedades sintáticas (AST) e semânticas (texto).

6.1.2 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 `TNodes` 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.

Para enriquecer essas conexões, cada aresta (i, j) recebe um peso escalar w_{ij} calculado a partir de uma função gaussiana sobre a distância estrutural entre os nós.

Inspirado em esquemas de *relative position encoding* com kernels gaussianos para código-fonte, o peso é definido como

$$w_{ij} = \exp\left(-\frac{|p_i - p_j|^2}{2\sigma^2}\right),$$

em que p_i e p_j podem representar a profundidade na AST ou o número de linha no arquivo, e σ é um hiperparâmetro de escala. Essa lógica é implementada nas funções `gaussian_edge_weights` e `gaussian_edge_weights_combined`. A versão combinada utiliza duas dimensões (profundidade e linha), agregando os respectivos pesos (por exemplo, via produto ou média), de modo que a proximidade sintática e a proximidade no código-fonte contribuam conjuntamente para a força da aresta.

Dessa forma, o grafo deixa de ser puramente não ponderado e passa a carregar um campo contínuo de relacionamentos estruturais: nós próximos (mesma região da árvore ou do arquivo) recebem pesos mais altos, enquanto nós distantes são atenuados.

6.1.3 Entrada para a GNN

Ao final dessa etapa, o modelo dispõe de três elementos:

- $X \in \mathbb{R}^{N \times d}$: embeddings iniciais de cada `TNode`;
- `edge_index` $\in \mathbb{N}^{2 \times E}$: topologia do grafo da AST;
- `edge_weight` $\in \mathbb{R}^E$: pesos gaussianos associados a cada aresta.

Esses tensores são passados para o modelo `CodeGNN`:

```
node_embs, graph_emb = CodeGNN(X, edge_index, edge_weight),
```

onde `node_embs` representa as embeddings contextuais refinadas de cada nó após a propagação de mensagens, e `graph_emb` é obtido por um *pooling* global, produzindo uma representação vetorial compacta do arquivo como um todo.

7 Escolha dos repositórios

Para a análise foram escolhidos três repositórios independentes, dois de David Beazley e um de Brandon Rhodes, duas referências em linguagem Python. Os repositórios de David Beazley possuem documentação completa no próprio repositório, facilitando a compreensão do software construído. Já o repositório de Brandon Rhodes não contém documentação, mas seu conteúdo é a adaptação do jogo *Colossal Cave Adventure* de Fortran para Python.

7.1 Colossal Cave Adventure

Este trabalho utiliza como base uma reimplementação de Rhodes (2010–2015) em Python 3, que preserva o jogo original de Crowther e Don Woods, utilizando o arquivo de dados `advent.dat` Crowther e Woods (1977). O pacote permite jogar em dois modos, no *prompt* do Python e em terminal do sistema operacional. Além disso, disponibiliza *walkthroughs* automatizados na pasta de testes.

7.2 Descrição do jogo

Colossal Cave Adventure, também conhecido como *ADVENT* ou simplesmente *Adventure*, é amplamente reconhecido como o primeiro jogo de aventura baseado em texto da história, criado por Will Crowther em meados de 1975 e expandido por Don Woods em 1976.

Ambientado em uma caverna repleta de tesouros, criaturas e labirintos, o jogador interage por comandos de texto, como *"GO NORTH"* ou *"GET LAMP"*. O sistema responde com descrições que narram as consequências das ações.

Como observa Dibbell (1998), o jogo automatiza o papel do mestre (*Dungeon Master*) característico de campanhas de *Dungeons and Dragons*. Suas descrições textuais simulam a fala do mestre (*"YOU ARE IN A MAZE OF TWISTY LITTLE PASSAGES, ALL ALIKE"*).

“Como qualquer programa significativo, *Adventure* expressava a personalidade e o ambiente de seus autores.” Levy (2010)

Will Crowther e sua ex-esposa, Patricia Crowther, ambos programadores e espeleólogos, participaram do mapeamento do sistema de cavernas *Mammoth Cave*. No verão de 1974, enquanto jogava campanhas de *Dungeons and Dragons*, Will começou o desenvolvimento do seu jogo utilizando o Fortran. O mapa utilizado no jogo foi inspirado diretamente nos levantamentos realizados pelo casal durante as expedições à Mammoth Cave, construindo no código a estrutura real da caverna.

Como o próprio Will Crowther relata, a ideia do jogo surgiu da combinação entre suas experiências em espeleologia e seu interesse por *Dungeons and Dragons*: “Eu estava envolvido em um jogo de interpretação de papéis... e tive uma ideia que combinasse o meu interesse por exploração de cavernas com algo que também fosse um jogo para as crianças...” Peterson (1983).

Levy (2010) conta como inicia a colaboração de Donald Woods, um pesquisador da *Stanford Artificial Intelligence Laboratory* (SAIL), em 1976. Após ter contato com uma prévia do jogo, Woods entrou em contato com Crowther, obteve sua permissão e passou a expandir o código. Sua versão incorporou novos puzzles, criaturas e elementos de fantasia inspirados na obra de Tolkien, além de um sistema de pontuação que estabelecia um objetivo ao jogador. A versão combinada de Crowther e Woods é um marco na história da interação humano-computador.

7.3

Como o jogo não possui documentação original, utilizei o artigo de Jerz (2007) como referência para compreender a estrutura e o funcionamento do código. O autor recupera e examina o código-fonte escrito por Will Crowther, a partir de um backup

preservado no SAIL. Jerz descreve as seis tabelas centrais que organizam os dados do jogo: descrições longas, rótulos curtos das salas, dados de mapa, vocabulário agrupado, estados estáticos e eventos ou dicas.

Essa arquitetura de dados é mantida na reimplementação em Python, embora expandida para doze seções, resultado da integração da versão de Don Woods Rhodes (2010–2015). A leitura e o processamento dessas tabelas ocorrem por meio do arquivo `advent.dat`, que preserva a semântica e a estrutura do código original.

As seis tabelas descritas por Crowther estruturam o mundo do jogo e suas interações:

1. **Long Descriptions:** textos descritivos longos que definem os ambientes e estados narrativos;
2. **Short Room Labels:** nomes curtos usados internamente para identificar locais e facilitar a navegação;
3. **Map Data:** conexões topológicas entre os ambientes e as direções de movimento possíveis;
4. **Grouped Vocabulary Keywords:** agrupamento de palavras-chave e comandos interpretados pelo sistema;
5. **Static Game States:** variáveis e condições fixas que controlam a lógica do jogo;
6. **Hints and Events:** mensagens de ajuda, eventos dinâmicos e respostas a situações específicas.

As outras seis adicionadas na versão em colaboração com Woods são:

1. *Object locations* — localização dos objetos;
2. *Action defaults* — mensagens padrão ligadas a verbos de ação;
3. *Liquid assets / flags* — COND por sala (luz, líquidos, restrições do pirata, bits de dicas);
4. *Class messages* — faixas de pontuação e mensagens de classificação do jogador;
5. *Hints* — dicas (turnos necessários, penalidade, pergunta e resposta);
6. *Magic messages* — mensagens de inicialização e manutenção.

Tabela 1 – Long Descriptions. A Tabela 1 contém descrições extensas dos ambientes do jogo. Com entradas identificadas de 1 a 140, ela define os textos apresentados ao jogador em diferentes locais. Cada linha representa uma sala ou estado narrativo. Parte dessas descrições refere-se diretamente a locais da caverna, como o trecho “*YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK BUILDING*”, enquanto outras descrevem situações de falha ou eventos inesperados, como “*YOU ARE AT THE BOTTOM OF THE PIT WITH A BROKEN NECK*”.

Exemplos:

- 1 *AROUND YOU IS A FOREST. A SMALL STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.*
- 2 *YOU HAVE WALKED UP A HILL, STILL IN THE FOREST. THE ROAD SLOPES BACK DOWN THE OTHER SIDE OF THE HILL. THERE IS A BUILDING IN THE DISTANCE.*
- 3 *YOU ARE INSIDE A BUILDING, A WELL HOUSE FOR A LARGE SPRING.*

Tabela 2 – Short Room Labels. A Tabela 2 contém rótulos curtos correspondentes às localizações/ambientes do jogo. Com entradas numeradas de 1 a 130, nem todas as salas ou estados definidos em *Long Descriptions* possuem equivalentes resumidos.

Exemplos:

- 1 *YOU'RE AT END OF ROAD AGAIN.*
- 3 *YOU'RE INSIDE BUILDING.*
- 18 *YOU'RE IN NUGGET OF GOLD ROOM.*
- 19 *YOU'RE IN HALL OF MT KING.*

Tabela 3 – Map Data. A Tabela 3 codifica a topologia do mundo do jogo e as regras de navegação, funcionando como um grafo dirigido rotulado. A primeira coluna indica o ambiente em que o jogador se encontra, a segunda define o ambiente de destino, e as colunas subsequentes agrupam os vocabulários que podem ser utilizados para realizar a transição entre os dois pontos. O mapeamento dos vocabulários é definido na Tabela 4.

Em alguns casos, o valor do destino representa uma condição especial, e não uma simples sala. Se o número de destino for maior que 500, o jogo exibe uma mensagem da Tabela 6 e o jogador permanece no mesmo local; Se estiver entre 300 e 500, o valor indica um salto especial para um trecho de código do jogo.

Exemplos:

- 1 2 2 44 29: o jogador se desloca do ambiente 1 ao ambiente 2, se utilizados os comando 2, 44 ou 29.
- 3 1 3 11 32 44: o jogador se desloca do ambiente 2 ao ambiente 1 se utilizados os comando 3, 11, 32 ou 44.

Tabela 4 – Grouped Vocabulary Keywords. No código original em Fortran, toda entrada de texto era truncada nos cinco primeiros caracteres, de modo que o comando *“inventory”*, por exemplo, poderia ser digitado simplesmente como *“inven”*. A reimplementação em Python de Rhodes (2010–2015) preserva essa lógica.

Os dados da tabela 4 são divididos em 4 grupos: o primeiro com id's entre 1 e 100 para movimento no jogo; com ids entre 1000 e 2000, trata de objetos manipuláveis ou características de cenário; com ids entre 2000 e 3000 são verbos de ação, se entre 3000 e 4000 são para casos especiais.

- 1–100: verbos de movimento, utilizados para navegação no espaço do jogo;
- 1000–2000: objetos e elementos de cenário manipuláveis;
- 2000–3000: verbos de ação (*carry*, *attack*, *drop*, etc.);
- 3000–4000: verbos de casos especiais, geralmente associados a eventos ou mensagens específicas definidas na Tabela 6.

Além dos comandos clássicos de navegação por bússola, "*EAST*"/"*E*", "*WEST*"/"*W*", "*NORTH*"/"*N*", "*SOUTH*"/"*S*", parte dos verbos de movimentos são nomes de locais da caverna como "*BEDQU*"(truncamento de *Bedquilt*), "*HOUSE*", "*GATE*" e "*FORES*"(*forest*).

Exemplos:

- 2 *ROAD*
- 3 *ENTER*
- 3 *DOOR*
- 3 *GATE*
- 4 *UPSTR*
- 5 *DOWNS*
- 6 *FORES*

Palavras de mesmo sentido/sinônimos possuem mesmo id, como "*ENTER*", "*DOOR*" e "*GATE*".

Tabela 5 – Static Game States. A Tabela 5 armazena descrições curtas que representam estados do jogo, correspondendo às mudanças permanentes no ambiente. Cada linha contém um número e uma mensagem descritiva.

Quando o identificador está entre 1 e 100, a linha define a mensagem de inventário associada a um objeto, exemplo: "*SET OF KEYS*" se refere a "*KEYS*". Quando o identificador é um múltiplo de 100, a mensagem descreve uma propriedade do objeto.

Exemplos:

- 1 SET OF KEYS
- 000 THERE ARE SOME KEYS ON THE GROUND HERE.
- 2 BRASS LANTERN
- 000 THERE IS A SHINY BRASS LAMP NEARBY.
- 100 THERE IS A LAMP SHINING NEARBY.
- 3 *GRATE
- 000 THE GRATE IS LOCKED.
- 100 THE GRATE IS OPEN.

Tabela 6 – Hints and Events. A Tabela 6 reúne mensagens arbitrárias usadas como dicas e como descrições de eventos pontuais. Essas mensagens não estão relacionadas a um ambiente ou objeto específicos, elas são acionadas por outras estruturas do jogo, como as tabelas 3, 4, 8 e 11.

Exemplos:

1. 3 AXE AT YOU WHICH MISSED, CURSED, AND RAN AWAY.
2. 6 NONE OF THEM HIT YOU!
3. 13 I DON'T UNDERSTAND THAT!
4. 24 YOU ARE ALREADY CARRYING IT!
5. 33 I DON'T KNOW HOW TO LOCK OR UNLOCK SUCH A THING.

Tabela 7 – Object Locations. A Tabela 7 define onde cada objeto surge no mundo do jogo e se ele é móvel ou fixo. Cada linha possui o identificador do objeto, a sala inicial, e um campo opcional que indica imobilidade (-1) ou uma segunda sala quando o objeto existe simultaneamente em dois lugares

- Sala inicial = 0: o objeto não aparece no mundo no início e só será criado por algum evento ou ação do jogador.
- Terceiro campo = -1: o objeto está fixo naquela sala (não pode ser carregado).
- Terceiro campo = número de sala: o objeto está presente em duas salas ao mesmo tempo, objetos com duas localizações são tratados como imóveis.

Exemplos:

- 1 3: objeto 1 (1001 - KEY, KEYS) começam na sala 3 (INSIDE BUILDING).
- 2 3: objeto 2 (1002 - LAMP, HEADL, LANTE) começam na sala 3 (INSIDE BUILDING).
- 3 8 9: objeto 3 (1003 - grate) existe nas salas 8 e 9 simultaneamente (8 - YOU'RE OUTSIDE GRATE, 9 - YOU'RE BELOW THE GRATE.).
- (9 - DOOR) (94 - YOU ARE AT ONE END OF AN IMMENSE NORTH/SOUTH PASSAGE.)
- 9 94 -1: objeto 9 (1009 - DOOR) é fixo na sala 94 (94 - YOU ARE AT ONE END OF AN IMMENSE NORTH/SOUTH PASSAGE.).
- 15 0: objeto 15 (1015 - OYSTE) começa fora do mundo e aparece mais tarde.

Tabela 8 – Action Defaults. A Tabela 8 define o comportamento padrão dos verbos de ação, associando cada identificador de verbo ao índice da mensagem correspondente na Tabela 6. Cada linha contém dois valores: o primeiro é o número do verbo de ação, e o segundo é o identificador da mensagem padrão que deve ser exibida.

Exemplos:

- 1 24: o verbo de ação associado ao id 1 (2001 - CARRY, TAKE, KEEP, CATCH, STEAL, CAPTU, GET, TOTE) e a mensagem 24 da tabela 6 (YOU ARE ALREADY CARRYING IT!).
- 6 33: o verbo de ação associado ao id 6 (2006 - LOCK, CLOSE) e a mensagem 33 da tabela 6 (I DON'T KNOW HOW TO LOCK OR UNLOCK SUCH A THING.).
- 7 38: o verbo de ação associado ao id 7 (2007 - LIGHT, ON) e a mensagem 38 da tabela 6 (YOU HAVE NO SOURCE OF LIGHT.).

Tabela 9 – Liquid Assets, Etc. A Tabela 9 define os bits de condição associados a cada sala, controlando luz, líquidos, presença de inimigos e zonas de interesse para as rotinas de dicas. Cada linha contém um identificador de bit e uma lista de até vinte localizações nas quais esse bit é ativado. O jogo usa esses bits para determinar o comportamento dinâmico de cada ambiente.

- 0: indica que o ambiente está naturalmente iluminado.
- 1: tipo de líquido usado em conjunto com o bit 2. Quando o bit 2 está ativo, este bit diferencia óleo (1) de água (0).
- 2: marca as salas que contêm água ou óleo.
- 3: impede que o pirata apareça ali, exceto quando persegue o jogador.
- 4: jogador tentando entrar na caverna.
- 5: tentativa de capturar o pássaro.
- 6: interação com a cobra.
- 7: perdido no labirinto.
- 8: refletindo no quarto escuro.
- 9: na área final Witt's End.

Exemplos:

- 0 1 2 3 4 5 6 7 8 9 10 100 115 116 126: salas naturalmente iluminadas próximas à entrada.
- 2 1 3 4 7 38 95 113 24: presença de líquido (água ou óleo) nessas salas.
- 9 108: marca a área final do jogo, Witt's End.

Tabela 10 – Class Messages. A Tabela 10 contém as mensagens de classificação do jogador de acordo com a pontuação total atingida ao final da partida. Cada linha associa um limite superior de pontuação a uma mensagem que descreve o título ou o nível de habilidade alcançado.

Exemplos:

- 35: YOU ARE OBVIOUSLY A RANK AMATEUR. BETTER LUCK NEXT TIME.
- 100: YOUR SCORE QUALIFIES YOU AS A NOVICE CLASS ADVENTURER.
- 130: YOU HAVE ACHIEVED THE RATING: ‘EXPERIENCED ADVENTURER’.
- 200: YOU MAY NOW CONSIDER YOURSELF A ‘SEASONED ADVENTURER’.
- 250: YOU HAVE REACHED ‘JUNIOR MASTER’ STATUS.
- 300: MASTER ADVENTURER CLASSES C.
- 330: MASTER ADVENTURER CLASSES B.
- 349: MASTER ADVENTURER CLASSES A.
- 9999: ALL OF ADVENTUREDOM GIVES TRIBUTE TO YOU, ADVENTURER GRANDMASTER!

Tabela 11 – Hints. A Tabela 11 associa dicas contextuais a condições determinadas de jogo. Cada linha contém cinco valores:

- O primeiro valor vincula a dica a uma condição definidos na Tabela 9.
- O segundo valor define quantos turnos o jogador deve gastar no mesmo estado antes da dica ser oferecida.
- O terceiro valor representa a penalidade subtraída da pontuação total ao aceitar a ajuda.
- Os dois últimos valores apontam para mensagens da Tabela 6: a pergunta inicial e a resposta.

Exemplos:

- 4 4 2 62 63 — Bit 4 (entrada da caverna): após 4 turnos no local, o jogo exhibe a pergunta 62 (Do you need help getting inside?) e, se aceita, mostra a resposta 63 (Perhaps you should explore the grate.), descontando 2 pontos.
- 6 8 2 20 21 — Bit 6 (cobra): depois de 8 turnos, o jogador recebe uma dica para resolver o enigma da serpente.

- 7 75 4 176 177 — Bit 7 (labirinto): após 75 turnos perdido, é oferecida uma dica de saída, com penalidade de 4 pontos.
- 8 25 5 178 179 — Bit 8 (quarto escuro): a dica surge depois de 25 turnos, custando 5 pontos.

Tabela 12 – Magic Messages. A Tabela 12 contém as chamadas *Magic Messages*, um conjunto de mensagens reservadas utilizadas pelos modos de inicialização, manutenção e administração do jogo. Embora seu formato seja idêntico ao da Tabela 6, elas são separadas para facilitar o acesso e o controle das rotinas especiais do sistema. Cada linha contém um identificador e um texto associado.agens internas do sistema.

Exemplos

- 1 *A LARGE CLOUD OF GREEN SMOKE APPEARS IN FRONT OF YOU... HE MAKES A SINGLE PASS OVER YOU WITH HIS HANDS, AND EVERYTHING FADES AWAY INTO A GREY NOTHINGNESS.*
- 2 *EVEN WIZARDS HAVE TO WAIT LONGER THAN THAT!*
- 3 *I'M TERRIBLY SORRY, BUT COLOSSAL CAVE IS CLOSED. OUR HOURS ARE:*
- 4 *ONLY WIZARDS ARE PERMITTED WITHIN THE CAVE RIGHT NOW.*

Referências

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River: Prentice Hall PTR, 2002.

YOURDON, E.; CONSTANTINE, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs: Prentice-Hall, 1979.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. Reading: Addison-Wesley, 1999. (Addison-Wesley Object Technology Series).

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. 2. ed. Boston: Pearson Education, 2018. (Addison-Wesley Signature Series (Fowler)).

SOMMERVILLE, I.; SAWYER, P. *Requirements Engineering: A Good Practice Guide*. Chichester: Wiley, 1997.

BRUNELIÈRE, H. *et al.* MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. *In: PROCEEDINGS of the IEEE International Conference on Software Maintenance (ICSM)*. Timișoara: IEEE, 2010. p. 173–182.

TONELLA, P.; POTRICH, A. *Reverse Engineering of Object-Oriented Code*. New York: Springer, 2007. (Monographs in Computer Science).

CHIKOFFSKY, E. J.; CROSS, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, IEEE, v. 7, n. 1, p. 13–17, 1990.

ZHANG, H. An Approach for Extracting UML Diagram from Object-Oriented Program Based on J2X. Versão inglesa. *In: INTERNATIONAL Forum on Mechanical, Control and Automation (IFMCA 2016)*. Changchun, China: Atlantis Press, 2016. p. 266–276. Published in *Advances in Engineering Research*, vol. 113.

YANG, X. *et al.* Condensing Class Diagrams With Minimal Manual Labeling Cost. Versão inglesa. *In: PROCEEDINGS of the 40th IEEE Annual International Computers, Software and Applications Conference (COMPSAC 2016)*. Atlanta, Georgia, USA: IEEE, 2016. p. 22–31. Published in *COMPSAC 2016: Proceedings of the 40th IEEE Annual International Computers, Software and Applications Conference*.

FAUZI, E.; HENDRADJAYA, B.; SUNINDYO, W. D. Reverse Engineering of Source Code to Sequence Diagram Using Abstract Syntax Tree. *In: 2016 International Conference on Data and Software Engineering (ICoDSE)*. Denpasar, Indonesia: IEEE, 2016. p. 1–6.

BAIDADA, C.; JAKIMI, A. Towards a New Hybrid Approach of the Reverse Engineering of UML Sequence Diagram. *In: 2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. Beijing, China: IEEE, 2016. p. 164–168.

LEATONGKAM, A.; NANTHAAMORNPHONG, A.; ROUSON, D. W. WIP: Generating Sequence Diagrams for Modern Fortran. *In: 2017 IEEE/ACM 12th International Workshop on Software Engineering for Science (SE4Science)*. Buenos Aires, Argentina: IEEE, 2017. p. 22–25.

SABIR, U. *et al.* A Model Driven Reverse Engineering Framework for Generating High Level UML Models from Java Source Code. *IEEE Access*, v. 7, p. 158931–158950, 2019.

SIALA, H. A. Enhancing Model-Driven Reverse Engineering Using Machine Learning. Versão inglesa. *In: 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Lisbon, Portugal: IEEE/ACM, 2024. p. 1–13. King’s College London, London, UK.

PEREIRA, C.; MARTÍNEZ, L.; FAVRE, L. Recovering Use Case Diagrams from Object-Oriented Code: An MDA-Based Approach. *In: 2011 Eighth International Conference on Information Technology: New Generations (ITNG)*. Las Vegas, NV: IEEE, 2011.

LIANG, H.-M.; HUANG, C.-Y. Integrating Non-Fourier and AST-Structural Relative Position Representations Into Transformer-Based Model for Source Code Summarization. *IEEE Access*, v. 12, p. 9871–9889, jan. 2024.

RHODES, B. *Adventure (Python 3 Port): A Faithful Port of Crowther and Woods’s 1977 FORTRAN Adventure*. [S. l.: s. n.], 1 jan. 2010–31 dez. 2015.

CROWTHER, W.; WOODS, D. *Original Adventure Sources (FORTRAN) and Data*. [S. l.: s. n.], 1977. Archive of original sources. Linked from the historical page curated by Rick Adams.

DIBBELL, J. *My Tiny Life: Crime and Passion in a Virtual World*. New York: Holt, 1998.

LEVY, S. *Hackers: Heroes of the Computer Revolution*. 25th Anniversary Edition. Sebastopol: O’Reilly Media, 2010.

PETERSON, D. *Genesis II: Creation and Recreation with Computers*. Reston, VA: Reston Publishing Company, 1983.

JERZ, D. G. Somewhere Nearby is Colossal Cave: Examining Will Crowther’s Original “Adventure” in Code and in Kentucky. Versão inglesa. *Digital Humanities Quarterly*, 2007.