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

Larman (2002) descreve três modos de desenvolvedores projetarem objetos: (i) projetar enquanto codifica, (ii) desenhar e projetar, (iii) apenas desenhar. Este trabalho propõe uma ferramenta de suporte para desenhar e projetar.

Quero que a introdução tenha contextualização, poblematização, questões/hipóteses, objetivo(s) e relevância

Larman (2002) descreve três modos de desenvolvedores projetarem objetos: (i) projetar enquanto codifica, (ii) desenhar e projetar, (iii) apenas desenhar. Este trabalho propõe uma ferramenta de suporte para desenhar e projetar.

Quero que a introdução tenha contextualização, poblematização, questões/hipóteses, objetivo(s) e relevância

2 Introdução

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

Esse estudo fundamenta-se em autores que defendem o desenvolvimento estruturado e orientado ao usuário, projetado a partir da visão e das necessidades de quem o utiliza, 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, Figura 1.

Cada etapa dessa cadeia implica na 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.

2.1 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.

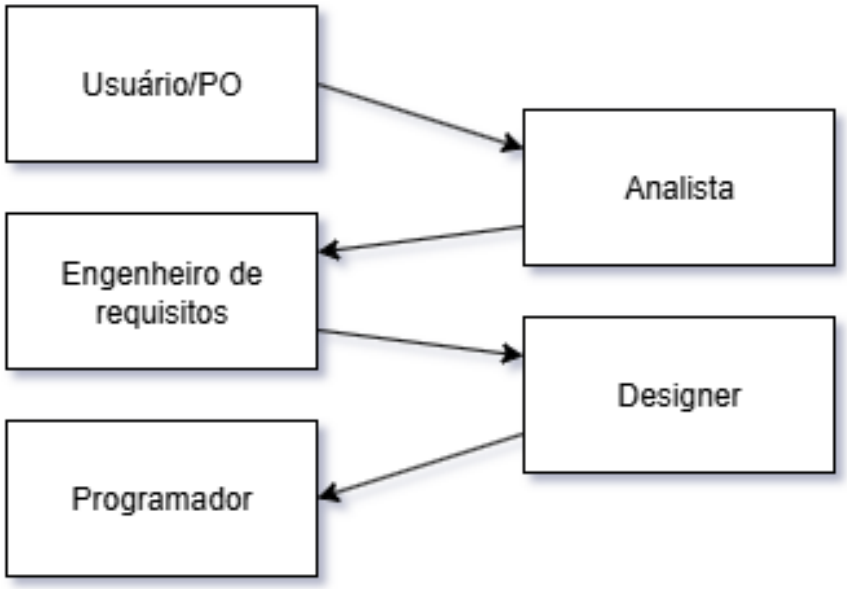

Figura 1: cadeia de tradução de requisitos segundo Constantine 1979.

Autores posteriores ampliaram essa discussão ao 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 o 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, um desafio na manutenção e evolução de sistemas legados.

2.2 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, questiona-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 permitir a reconstrução de artefatos de alto nível, como diagramas de casos de uso, mesmo na ausência de documentação formal.

2.3 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 representações para uma representação uniforme 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 sintática e os diagramas de casos de uso, permitindo representar classes, métodos, atributos e interações de forma compreensível e consistente.

Esse tipo de investigação é definido por Chikofsky e Cross (1990) como *Redocumentation* 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. Aqui propõe-se criar diagramas de casos de uso, preservando a semântica do sistema original, afim de compreender o comportamento observável do ponto de vista do usuário.

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.

2.4 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 programadore. 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.

3 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 propõe 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 aperfeiçoa)
- 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).

3.1 Análise sistemática da literatura

A partir da síntese da Tabela 1, organizamos os achados por eixo (RQ1–RQ6), destacando tendências, limitações e implicações para a presente pesquisa.

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 relevante e contemporâneo, porém ainda subexplorado.

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.→OFG; CFG+OFG→Sequência	J2UML; JavaCC; Dom4j; J2X (DTD/XML)	5 casos pequenos; acurácia: classes 96,4–100%; relações 65,0–90,4%
Yang *et al.* (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	MagicDraw; 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 Sunindyo (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.x; instrumentação/VM/debugger; CPN (IR)	Sem validação; futuro
Leatongkam, Nanthaamornphong 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;	*Work in progress*
Sabir *et al.* (2019)	Java (sistemas legados orientados a objetos)	UML *Class Diagram* + *Activity Diagram* (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 Amadeus 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.

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* → *J2X* → *OFG/CFG* → *UML* (Classe+Sequência) . Leatongkam, Nanthaamornphong e Rouson (2017) propõem 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 (e.g., "SDs mostram interações num *use case* específico"), 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), o padrão T2M/M2M consolida o fluxo código→AST→IM→UML, com *Activity* agregada ao escopo estrutural. 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 comportamental em *Colored Petri Nets* e, então, mapeia-se para UML 2.x, obtendo maior fidelidade comportamental, embora sem validação empírica robusta reportada. 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 (*Intermeadiate Representation*) que padroniza elementos de linguagem; o diagrama de classes surge de metadados extraídos dessa IR (Interme), 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 (*Intermeadiate Model*) para **UML2** (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 com maior fidelidade às execuções reais. 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).

Em termos de *trade-offs*, IRs sintáticas (J2X, XMI, EMF) maximizam portabilidade e auditabilidade do pipeline; CPNs elevam a *fidelidade comportamental* à custa de instrumentação; e LLMs ampliam a *capacidade explicativa* ao incorporar indícios semânticos (nomes, comentários, docstrings).

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.x**. Em Lea-

tongkam, Nanthaamornphong 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.

Leitura crítica (versão mais explícita). Os fluxos T2M/M2M sustentam traçabilidade e reprodutibilidade quando ancorados em IRs explícitas (J2X, EMF, XMI). Abordagens híbridas elevam a fidelidade comportamental (Petri Nets), mas dependem de instrumentação. As abordagens com LLMs (e.g., Siala 2024) inserem uma etapa semântica — código → representação textual próxima de UML/OCL — sem quebrar a cadeia MDA: os modelos continuam conformes a metamodelos OMG (via AgileUML) e podem ser materializados/validados em ferramentas padrão (Modelio, PlantUML, Graphviz), preservando a interoperabilidade do pipeline.

Implicações para este trabalho. As lacunas identificadas fundamentam a proposta de **redocumentação semântica** a partir de **Python**, combinando (i) **AST** como *IM/IR* e (ii) **LLMs/embeddings** para inferir *intenção e papel* (a partir de docstrings, comentários e nomenclaturas), visando a reconstrução de **diagramas de casos de uso**.

4 Escolha dos repositórios

Para 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 possui uma documentação completa no próprio repositório, facilitando a compreensão do software construido. Já o repositório do Brandon Rhodes não contém documentação, contudo o conteúdo é a portabilidade do jogo Colossal Cave Adventure de Fortran para Python.

4.1 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.

4.2 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 labiríntos, 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.

4.3

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 aombiente 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 entra 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 veros 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 propriedado 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 exibe 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). Timişoara: IEEE, 2010. p. 173–182.

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

CHIKOFSKY, 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.

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.