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 AAAH

Apesar dessas práticas, a perda de significado ainda é comum em ambientes colaborativos. Avanços recentes em modelos de linguagem e em representações semânticas (embeddings) abrem uma oportunidade: tratar o repositório de código como um corpus rico em sinais de intenção (identificadores, comentários, *docstrings*, *issues*, *commits*). Combinando *embeddings* e *Retrieval-Augmented Generation* (RAG), podemos recuperar e recompor essas intenções em artefatos de engenharia de requisitos.

Este trabalho propõe uma abordagem de **engenharia reversa orientada por aprendizado de máquina** para *derivar casos de uso* a partir do código-fonte, conectando-o a uma base curada de boas práticas e obras clássicas. Como estudo de caso, analisamos o *Colossal Cave Adventure* (Crowther/Woods), cujo desenho em tabelas facilita a correspondência entre elementos do código, comportamentos observáveis e narrativas de uso. Nossas contribuições são: (i) um método de extração e alinhamento semântico (código $\rightarrow$ casos de uso), (ii) um *pipeline* RAG reprodutível, e (iii) evidências de que a técnica aumenta a precisão e a compreensibilidade da documentação gerada.

2 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.

Esse estudo fundamenta-se nas constribuições de Larry Constantine, Edward Yourdon, Ivar Jacobson e Alistar Cockburn, autores que defendem o desenvolvimento estruturado e orientado ao usuário. Isto é, 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 ocorre entre o proprietário do produto, o usuário, e o analista responsável por capturar suas ideias, conhecimentos de usabilidade e o comportamento esperado do software. Em seguida, o engenheiro de requisitos traduz essa concepção em um conjunto de requisitos funcionais e não funcionais, que por sua vez são reinterpretados pelo designer de sistemas em estruturas lógicas e técnicas. Por fim, o programador concretiza essas decisões, implementando no código as interpretações que compreendeu a partir do trabalho do designer. Cada etapa dessa cadeia de traduções implica na perda ou distorção de parte do significado original do usuário, o que pode resultar em comportamentoa apenas próximo ao desejado.

Como solução desse problema, os autores apresentam o conceito de *projeto estruturado*,

entendido como "a arte de definir os componentes de um sistema e as interrelações entre esses componentes da melhor forma possível"Yourdon e Constantine

(1979). Esse processo tem início na clareza e na visibilidade das decisões e atividades envolvidas, promovendo uma compreensão compartilhada entre os membros da equipe e garantindo que o design reflita as intenções originais do sistema.

"Introduzindo uma atividade específica de design formal para descrever completamente, e com antecedência, todas as partes de um sistema e suas inter-relações, não criamos uma nova atividade no ciclo de desenvolvimento de programas. O design estruturado apenas consolida, formaliza e torna visíveis as atividades e decisões de design que inevitavelmente acontecem - e de forma invisível - no curso de todo projeto de desenvolvimento de sistemas. Em vez de ocorrerem por tentativa e erro, sorte e padrão, essas decisões podem ser abordadas deliberadamente como compensações técnicas."

"Reunindo todas as decisões que afetam a escolha de módulos e inter-relações em um sistema, inevitavelmente influenciamos a maneira como outras decisões são organizadas e resolvidas. Assim, algumas questões que tradicionalmente eram abordadas de uma certa forma durante a fase inicial de um projeto podem ter que ser tratadas de uma maneira completamente diferente em uma fase muito posterior, uma vez que o designer adota uma abordagem de design estruturado."

"O design estruturado é o processo de decidir quais componentes interconectados de qual maneira resolverão algum problema bem especificado."

Para Constantine, propriedades de qualidade como coesão e acoplamento não são apenas métricas técnicas, mas indicadores de quão bem o software reflete o domínio do problema e promove a clareza de propósito de cada módulo. Um sistema altamente coeso e fracamente acoplado tende a reproduzir com maior fidelidade a lógica do usuário, tornando-se mais previsível, transparente e alinhado às suas necessidades operacionais.

Afim de alcançar o resultado de um software onde observamos o comportamento esperado, modelagem de *casos de uso* surge como uma estratégia orientada ao usuário. Introduzido por Ivar Jacobson em 1989, o conceito propõe que para compreendermos o que um sistema deve fazer, é preciso identificar quem o utilizará e quais objetivos esse usuário pretende alcançar.

Segundo Jacobson e Cockburn, um caso de uso "conta a história completa", uma sequência de eventos que se inicia com uma necessidade e termina com a geração de valor, incluindo também alternativas, desvios e exceções. Essa narrativa funcional estabelece o elo entre o domínio do problema e a estrutura do software, permitindo que analistas e desenvolvedores compartilhem uma mesma linguagem de entendimento sobre o sistema.

Os princípios subjacentes aos casos de uso refletem diretamente o ideal do projeto estruturado defendido por Constantine e Yourdon. Ambos valorizam a decomposição do sistema em unidades compreensíveis e interdependentes, com ênfase na clareza das intenções e na previsibilidade das interações. Enquanto o projeto estruturado busca representar a estrutura do sistema de forma coerente e modular, os casos de uso descrevem o comportamento dessa estrutura na perspectiva do usuário.

Autores posteriores ampliaram essa discussão ao destacar o papel humano da comunicação no próprio ato de programar. Autores como Frederick P. Brooks Jr., Martin Fowler, Kent Beck e Robert C. Martin ressaltam que o desenvolvimento de software não é apenas um esforço técnico, mas também um ato de comunicação

entre mentes humanas mediado pelo código. A produtividade e a clareza de um sistema dependem, portanto, não apenas da habilidade individual do programador, mas da forma como o conhecimento é transmitido, compreendido e preservado entre diferentes desenvolvedores ao longo do tempo.

Nesse sentido, a documentação, a estrutura do código, a nomenclatura e o comportamento esperado funcionam como pontes entre a intenção original do autor e a compreensão dos futuros mantenedores. Sommerville e Wiegers e Beatty destacam que a documentação é um artefato essencial da engenharia de requisitos, atuando como quem nos diz quem é o software, ou quem deveria ser. Ela traduz como o usuário pretende utilizar o sistema e quais são as características de negócio que o software deve refletir. A ausência de clareza nesse elo resulta em sistemas de difícil manutenção, inconsistências entre requisitos e implementação e perda de rastreabilidade, um problema recorrente em ambientes de desenvolvimento complexos e colaborativos.

No cenário contemporâneo, o avanço dos modelos de linguagem e das técnicas de representação semântica, como os *embeddings*, oferece novas possibilidades. Trabalhos recentes exploram como representações vetoriais de elementos de software, tais como identificadores, comentários, *issues*, *commits* e *docstrings*, podem capturar aspectos semânticos úteis à engenharia de requisitos e à geração automatizada de artefatos. Essa evolução permite observar o código não apenas como uma estrutura sintática, mas como um repositório de intenções comunicativas, em que cada elemento textual contribui para reconstruir a lógica, o propósito e o comportamento esperados do sistema.

É nesse contexto que se insere o presente trabalho, propondo uma abordagem de engenharia reversa orientada por aprendizado de máquina, fundamentada em *embeddings* e *Retrieval-Augmented Generation* (RAG). O objetivo é investigar como informações semânticas presentes em comentários, identificadores e estruturas de código podem ser utilizadas para recuperar a intenção do desenvolvedor e derivar artefatos de engenharia de requisitos, em especial casos de uso. Ao conectar o código-fonte a uma base de conhecimento composta por obras clássicas e boas práticas de engenharia de software, busca-se contribuir para a construção de um processo de documentação e elicitação de casos de uso mais preciso, compreensível e automatizável.

2.1 Fundamentos da Engenharia de Software e Projeto Estruturado

Desde as décadas de 1970 e 1980, a engenharia de software vem se consolidando como um campo preocupado em reduzir a distância entre o comportamento esperado do sistema e sua implementação técnica. Yourdon e Constantine introduziram os princípios de projeto estruturado, destacando a importância da coesão e do baixo acoplamento entre módulos como indicadores de qualidade.

Posteriormente, Jacobson, Booch, Rumbaugh propuseram a UML e os casos de uso como meio de representar o comportamento do sistema a partir da perspectiva do usuário. Essa abordagem complementa o projeto estruturado ao oferecer uma visão funcional do software, alinhada às intenções e objetivos do usuário.

Autores como Brooks Jr., Fowler e Martin reforçaram a dimensão humana do desenvolvimento de software, tratando o código como um meio de comunicação entre desenvolvedores e um reflexo das intenções de design. Essas obras clássicas sustentam a premissa deste trabalho: o código-fonte contém, em sua estrutura e documentação, pistas da intenção do autor que podem ser extraídas e formalizadas por técnicas de engenharia reversa.

3 Metodologia de Mapeamento Sistemático

3.1 Capítulo 17 CASOS DE USO - UML

Nenhum sistema existe isoladamente. Todo sistema interessante interage com atores humanos ou autômatos que utilizam esse sistema para algum propósito e esses atores esperam que o sistema se comporte de acordo com as maneiras previstas. Um caso de uso especifica o comportamento de um sistema ou de parte de um sistema e é uma descrição de um conjunto de seqüências de ações, incluindo variantes realizadas pelo sistema para produzir um resultado observável do valor de um ator. Os casos de usos podem ser aplicados para captar o comportamento pretendido do sistema que está sendo desenvolvido, sem ser necessário especificar como esse comportamento é implementado. Os casos de uso fornecem uma maneira para os desenvolvedores chegarem a uma compreensão comum com os usuários finais do sistema e com os especialistas do domínio. Além disso, os casos de uso servem para ajudar a validar a arquitetura e para verificar o sistema à medida que ele evolui durante seu desenvolvimento. À proporção que você implementa o seu sistema, esses casos de uso são realizados por colaborações cujos elementos trabalham em conjunto para a execução de cada caso de uso. Casos de uso bem-estruturados denotam somente o comportamento essencial do sistema ou subsistema e não são amplamente gerais, nem muito específicos.

Um caso de uso executa alguma quantidade tangível de trabalho. Sob a perspectiva de um determinado ator, um caso de uso realiza algo que é de valor para um ator, como o cálculo de um resultado, a geração de um novo objeto ou a modificação do estado de outro objeto. Por exemplo, na modelagem de um banco, o processamento de um empréstimo resulta na entrega de um empréstimo aprovado, manifestada como uma pilha de dinheiro entregue nas mãos do cliente.

Você poderá aplicar os casos de uso a todo o seu sistema. Também pode aplicá-los a uma parte do sistema, incluindo subsistemas e até interfaces e classes individuais. Em cada situação, os casos de uso não apenas representam o comportamento desejado desses elementos, mas também podem ser utilizados como a base de casos de teste para esses elementos, à medida que evoluem durante o desenvolvimento. Casos de uso aplicados aos subsistemas são excelentes fontes de testes de regressão; casos de uso aplicados a todo o sistema são excelentes fontes de testes de sistema e de integração. A UML fornece a representação gráfica de um caso de uso e de um ator, conforme mostra a Figura 17.1. Essa notação permite visualizar um caso de uso em separado de sua realização e no contexto com outros casos de uso.

4 Escolha da linguagem e porque OO

5 Escolha de repositório

5.1 Colossal Cave Adventure

"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 de Flint Ridge, na Mammoth Cave, localizada em Kentucky. No verão de 1974, enquanto jogava campanhas de Dungeons and Dragons, Will desenvolveu em Fortran o que viria a ser o primeiro jogo de aventura interativa, originalmente chamado de ADVENT, Colossal Cave Adventure. A jogabilidade baseava-se em comandos de texto digitados pelo jogador, que descreviam ações em um ambiente simulado. O sistema interpretava instruções como "go north", "take lamp" ou "open door" e retornava descrições das consequências, criando uma interação textual. O mapa utilizado no jogo foi inspirado diretamente nos levantamentos realizados pelo casal durante as expedições à Mammoth Cave, refletindo 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) mostra como inicia a colaboração de Donald Woods em 1976, um pesquisador da *Stanford Artificial Intelligence Laboratory* (SAIL). 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 mensurável ao jogador. A versão combinada de Crowther e Woods é um marco na história da interação humano-computador.

Além do arquivos Crowther e Woods (1977), como o jogo não possue documentação técnica original, usaremos o artigo de Jerz (2007) para orientar a análise da portabilidade em Python. O estudo recupera e examina o código Fortran original de Crowther, a partir de um backup do SAIL. Jerz detalha as seis tabelas centrais do código fonte em Fortran: descrições longas, rótulos curtos das salas, dados do mapa, palavras-chave agrupadas do vocabulário, estados estáticos e dicas/eventos. Exatamente o tipo de estrutura que é preservada no versão em Python através do arquivo `advent.dat`. Permitindo mapear conceitos do Fortran, tabelas, palavras-chave e transições, para as construções equivalentes no código Python analisado neste trabalho.

5.2 Código-fonte em Fortran de Crowther

Conforme analisado por Jerz (2007), o programa é dividido em dois arquivos principais: um dedicado à lógica e outro aos dados, estes organizados em seis tabelas

distintas.

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.

Tabela 1 – Long Descriptions. A tabela contém descrições extensas dos ambientes do jogo. Com 302 linhas e 78(confirmar o número no dat) entradas numeradas 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
- 1 DOWN A GULLY.
- 2 YOU HAVE WALKED UP A HILL, STILL IN THE FOREST. THE ROAD SLOPES BACK
- 2 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 segunda tabela contém rótulos curtos correspondentes às localizações/ambientes do jogo. Com 67 entradas numeradas de 1 a 130, indicando que 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.
- 33 YOU'RE AT Y2.

Tabela 3 – Map Data. A terceira tabela 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.

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 jogo original em Fortran, todo input era trucado nos seus 5 primeiros caracteres, a versão de Rhodes (2010) aceita inputs de 5 letras ou palavras completas. Por exemplo o comando "inventory"pode ser substituido por "inven".

A quarta tabela representa o vocabulário reconhecido pelo jogo, funcionando como o léxico central do sistema de interpretação de comandos. Ela contém 193 itens, todos truncados aos cinco primeiros caracteres, e agrupa palavras sinônimas sob o mesmo identificador numérico, permitindo que diferentes entradas do jogador acionem a mesma ação.

O primeiro grupo, numerado entre 2 e 70, reúne palavras associadas ao movimento. Nessa seção, comandos como `ENTER`, `DOOR` e `GATE` compartilham o identificador 3, sendo tratados como equivalentes. Termos que indicam direções cardeais (`NORTH`, `EAST`, `SOUTH`, `WEST`) ocupam os números 43–46, enquanto os movimentos verticais (`UP`, `DOWN`) aparecem em 29 e 30. Essa codificação evidencia uma transição de um estilo linear de navegação para um modelo espacial baseado em coordenadas, característica que se tornaria definidora do gênero das aventuras de texto.

O segundo grupo, numerado entre 1001 e 1023, abrange objetos e entidades do jogo — tanto itens manipuláveis (como `KEYS`, `LAMP`, `ROD`) quanto elementos de cenário (`STEPS`, `GRATE`) e adversários (`SNAKE`, `DWARVES`). Alguns objetos compartilham

o mesmo código, indicando equivalência funcional: por exemplo, `BOTTL` e `WATER` possuem o identificador 1020, sendo tratados como o mesmo objeto.

O terceiro grupo (prefixo 2000) reúne verbos de ação, incluindo doze sinônimos para "pegar" (`TAKE`, `GET`), cinco para "soltar" (`RELEASE`) e nove para "andar" (`WALK`). Um agrupamento curioso inclui `CALM`, `WAVE`, `SHAKE`, `SING` e `CLEAVE`, todos com o código 2010 — indicando um mapeamento genérico de gestos ou interações sociais.

A análise da Tabela 4 revela ainda que o código original de Crowther não possuía comandos para salvar o jogo, verificar o placar ou listar o inventário, funções que só seriam adicionadas por Don Woods em sua expansão posterior. O último termo adicionado ao vocabulário é um palavrão de quatro letras, indício de que Crowther testava também respostas a comandos improváveis de jogadores frustrados — uma evidência da natureza experimental e humorada do projeto.

Essa estrutura lexical demonstra uma clara separação entre a camada de linguagem (Tabela 4) e a lógica de jogo (Tabela 3), antecipando princípios modernos de design modular e interpretadores baseados em dicionários.

Referências

YOURDON, E.; CONSTANTINE, L. L. *Structured design: fundamentals of a discipline of computer program and systems design*. [*S. l.*]: Prentice-Hall, Inc., 1979.

LEVY, S. *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*. [*S. l.*]: O'Reilly Media, 2010.

PETERSON, D. *Genesis II, Creation and recreation with computers*. [*S. l.*]: Reston Publishing Company, 1983.

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.

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.

RHODES, B. *Adventure (Python 3 port) — a faithful port of Crowther and Woods's 1977 FORTRAN Adventure*. [*S. l.: s. n.*], 2010.