

UNIVERSIDADE FEDERAL DE SANTA CATARINA DEPARTAMENTO
DE INFORMÁTICA E ESTATÍSTICA

**Aplicativo de comparação automática entre código fonte e
diagrama de classes**

Gabriel Martins Bergmann

Florianópolis - SC

2026 / 1

UNIVERSIDADE FEDERAL DE SANTA CATARINA DEPARTAMENTO DE
INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

**Aplicativo de comparação de código fonte e diagrama de classes
automatizado**

Trabalho de conclusão de curso apresentado
como parte dos requisitos para obtenção do
grau de Bacharel em Ciência da Computação.

Autor: Gabriel Martins Bergmann

Orientador: Prof. Dr. Jean Carlo Rossa Hauck

Florianópolis - SC

2026 / 1

Gabriel Martins Bergmann

Aplicativo de comparação de código fonte e diagrama de classes automatizado

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de
Bacharel em Ciência da Computação

Prof. Dr. Jean Carlo Rossa Hauck

Orientador

Universidade Federal de Santa Catarina

Prof. Dr. Ricardo Pereira e Silva

Avaliador

Universidade Federal de Santa Catarina

Prof.^a Dr.^a Thaís Bardini Idalino

Avaliadora

Universidade Federal de Santa Catarina

Prof. Dr. Cristian Koliver

Avaliador

Universidade Federal de Santa Catarina

AGRADECIMENTOS

Gostaria de agradecer aos voluntários que ajudaram a aplicar *labels* as imagens usadas na construção do dataset.

SUMÁRIO

AGRADECIMENTOS.....	4
SUMÁRIO.....	5
LISTA DE FIGURAS.....	7
LISTA DE TABELAS.....	8
LISTA DE ABREVIATURAS.....	9
RESUMO.....	10
1 INTRODUÇÃO.....	11
1.1 OBJETIVOS.....	12
1.1.1 Objetivo Geral.....	12
1.1.2 Objetivos Específicos.....	12
1.2 ABORDAGEM METODOLÓGICA.....	13
1.2.1 Especificação De Escopo.....	14
1.2.2 Uso de Inteligência Artificial Generativa.....	14
2 FUNDAMENTAÇÃO TEÓRICA.....	15
2.1 LINGUAGEM DE MODELAGEM UNIFICADA(UML).....	15
2.1.1 Histórico da UML.....	15
2.1.2 Diagramas da UML.....	16
2.1.3 Diagrama de Casos de Uso.....	17
2.1.4 Diagrama de Máquina de Estados.....	17
2.1.5 Diagrama de Atividades.....	17
2.1.6 Diagrama de Sequência.....	18
2.1.7 Diagrama de Comunicação.....	18
2.1.8 Diagrama de Visão Geral de Interação.....	19
2.1.9 Diagrama de Temporização.....	19
2.1.10 Diagrama de Utilização.....	19
2.1.11 Diagrama de Componentes.....	20
2.1.12 Diagrama de Estrutura Composta.....	20
2.1.13 Diagrama de Pacotes.....	20
2.1.14 Diagrama de Profile.....	21
2.1.15 Diagrama de Objetos.....	21
2.1.16 Diagrama de Classes.....	22
2.2 DIAGRAMA DE CLASSES.....	22
2.3 MACHINE LEARNING.....	26
2.3.1 Supervised Learning.....	27
2.3.2 Unsupervised Learning.....	27
2.3.3 Semi-supervised Learning.....	27
2.3.4 Reinforcement Learning.....	28
2.4 Mermaid.....	28
3 TRABALHOS CORRELATOS.....	31

3.1 DEFINIÇÃO DO PROTOCOLO DE REVISÃO.....	31
3.2 RESULTADOS DA PESQUISA.....	32
3.2.1 Qual o objetivo desses trabalhos?.....	32
3.2.2 Quais técnicas de IA foram usadas?.....	33
3.2.3 Quais tecnologias foram mencionadas?.....	34
3.2.4 Algum dataset existente foi utilizado?.....	34
3.2.5 Quais resultados foram encontrados?.....	35
3.3 DISCUSSÃO.....	36
4 FERRAMENTA DE AVALIAÇÃO DE CÓDIGOS-FONTE COM BASE EM DIAGRAMA DE CLASSES.....	37
4.1 ANÁLISE DE REQUISITOS.....	37
4.2 CRIAÇÃO DO MODELO DE VISÃO COMPUTACIONAL.....	38
4.2.1 Coleta de Imagens.....	39
4.2.2 Labelling.....	40
4.2.3 Pré processamento.....	43
4.2.4 Treinamento.....	44
4.2.5 Avaliação do Resultado.....	46
4.3 CONVERSÃO DE IMAGENS EM LINGUAGEM MERMAID.....	47
4.4 CONVERSÃO DE CÓDIGO PYTHON EM LINGUAGEM MERMAID.....	52
4.5 CORREÇÃO COM STRING MATCHING.....	55
4.6 EXEMPLO COMPLETO.....	57
4.7 DISCUSSÃO.....	63
5 CONCLUSÃO.....	67
REFERÊNCIAS.....	68
APÊNDICE A.....	71
APÊNDICE B.....	72
APÊNDICE C.....	73

LISTA DE FIGURAS

Figura 1: Taxonomia da estrutura de diagramas UML. Fonte: (SILVA, 2019).....	16
Figura 2: Exemplo de Diagrama de Sequência. Fonte: (SILVA, 2019).....	18
Figura 3: Exemplo de Diagrama de Comunicação. Fonte: (SILVA, 2019).....	19
Figura 4: Exemplo de Diagrama de Pacotes. Fonte: (SILVA, 2019).....	21
Figura 5: Exemplo de Diagrama de Classe. Fonte: (IVAR JACOBSON E GRADY BOOCH, 2000).....	23
Figura 6: Exemplo de diagrama de classes exibido pelo renderizador do mermaid. 30	
Figura 7: Um diagrama incluído no dataset.....	39
Figura 8: Exemplo de imagem do dataset após anotações.....	40
Figura 9: Opções de pré processamento utilizadas.....	44
Figura 10: Opções de augments utilizadas.....	44
Figura 11: Exemplo de um gráfico de curva PR. Fonte: (WANG, 2022).....	47
Figura 12: Gráficos de diversas métricas analisadas durante treinamento.....	47
Figura 13: Trecho de código coletando dados relevantes dos resultados da inferência.....	48
Figura 14: Código para ajustes de seção de classe.....	49
Figura 15: Separação da imagem contendo texto de nome, atributos ou método de classe em linhas.....	50
Figura 16: Chama o leitor do easyOCR somente para ler o texto, tratando a imagem inteira como detecção.....	51
Figura 17: Encontra os dois pontos da máscara mais distantes.....	51
Figura 18: Conexão de setas a relações, losangos tem uma função similar.....	52
Figura 19: Uso de uma expressão regular para encontrar declarações de classe... 53	
Figura 20: Verificação de associação e de composição.....	54
Figura 21: Verificação de herança.....	55
Figura 22: Ambos exemplos de mermaid descrevem o mesmo diagrama.....	56
Figura 23: Exemplo de uma seção do relatório apontando um erro encontrado.....	57

LISTA DE TABELAS

Tabela 1: Elementos do Diagrama de Classes. Fontes: (UNIFIED. . . , 2017), (THE. . . , s.d.).....	26
Tabela 2: Sintaxe mermaid para diagramas de classe.....	30
Tabela 3: Objetivos dos Artigos Relevantes.....	33
Tabela 4: Tecnologias Mencionadas em Artigos Relevantes.....	34
Tabela 5: Resumos de Resultados dos Artigos Relevantes.....	36
Tabela 6: Exemplos dos elementos de diagrama de classe no dataset.....	41
Tabela 7: Frequência de cada label no dataset.....	43

LISTA DE ABREVIATURAS

UML - Unified Modeling Language

IA - Inteligência Artificial

OCL - Object Constraint Language

XMI - XML Metadata Interchange

XML - Extensible Markup Language

CNN - Convolutional Neural Network

YOLO - You Only Look Once

CASE - Computer Aided Software Engineering

OCR - Optical Character Recognition

COCO - Common Objects in Context

mAP - Mean Average Precision

AP - Average Precision

PR - Precision-Recall

RESUMO

A correção de atividades avaliativas nos cursos da área de Computação toma uma quantia significativa de tempo de trabalho de professores fora de sala de aula. No campo de engenharia de software, atividades avaliativas frequentemente aplicadas são a criação de diagramas de classe UML baseadas em trechos de códigos e o desenvolvimento de código baseado em diagramas de classe, automatizar a correção dessas tarefas seria de extrema utilidade para reduzir o tempo de professores investido nesse processo, mas não é facilmente implementado devido a não haver uma única resposta possível, podendo ocorrer variações em respostas dadas que ainda são corretas, diferente de casos como questões de múltipla escolha ou com resultados de valores objetivos. Com esse problema em mente, o trabalho atual foi produzido visando usar modelos de visão computacional e identificação de padrões em código python, para permitir, ao menos de forma parcial, a comparação e correção de atividades envolvendo diagramas de classe UML.

Palavras-chave: *Universal Modelling Language*, Correção de atividades avaliativas, *Fine tuning*, Inteligência artificial, Diagrama de classes

1 INTRODUÇÃO

A correção de atividades avaliativas nos cursos da área de computação toma uma quantia significativa de tempo de trabalho de professores fora de sala de aula. A automatização desse processo de correção pode ser de grande utilidade nesse sentido, mas, de forma geral, só é facilmente aplicada a atividades executadas em computadores com questões de múltipla escolha ou resultados objetivos pré-determinados com menor ambiguidade na resposta. Com isso em mente, o uso de Inteligência Artificial (IA) pode apoiar a correção de atividades dissertativas ou que requerem a entrega de mídias que podem variar de forma significativa.

Um caso específico trata de atividades de disciplinas relacionadas à Engenharia de Software. Especialmente atividades que envolvem a modelagem de soluções computacionais, com a elaboração de diagramas UML, apresentam dificuldades adicionais para automatização, mesmo que parcial, da sua avaliação. Nesse sentido, os diagramas de classes são muito utilizados para representar uma visão estática do software, pois modelam conceitos do domínio da aplicação (JACOBSON & BOOCH, 2000). Avaliação de atividades relacionadas à elaboração de diagramas de classes tipicamente envolve primeiramente uma avaliação da sintaxe UML utilizada, bem como a comparação entre o modelo desenvolvido e a solução de software implementada.

Assim, este trabalho propõe o uso de inteligência artificial para automatizar a avaliação de trabalhos envolvendo a comparação de código fonte com diagramas de classes, importante para disciplinas da área de Engenharia de Software em cursos de graduação da área da Computação.

Já existem trabalhos detalhando tanto o uso de IA para ajudar professores na tarefa de correção, quanto no seu uso em reconhecer padrões de diagramas UML

(NAJAFI et al, 2025; CHEN et al., 2022), demonstrando que um projeto juntando os dois conceitos é possível e prático, e pode reduzir o tempo de trabalho de professores de engenharia de software dedicado a avaliação que poderá ser realocado para outras atividades importantes como planejamento de aulas.

1.1 OBJETIVOS

Nas seções abaixo estão descritos o objetivo geral e os objetivos específicos deste TCC.

1.1.1 Objetivo Geral

Esse trabalho busca desenvolver uma ferramenta capaz de avaliar um arquivo de código-fonte com base em um diagrama de classes. Nesse cenário, o diagrama de classes representa um gabarito feito pelo professor e o código-fonte representa uma resposta elaborada por um aluno. Ao fim da execução a saída deverá ser dada como uma comparação entre os dois, apontando se a resposta do aluno está de acordo com o gabarito.

Ao fim do projeto, o programa deve ser capaz de fazer a correção automática com sucesso na maioria dos casos, tendo sucesso se nenhuma ação adicional por parte do professor seja necessária para garantir uma avaliação correta.

1.1.2 Objetivos Específicos

Criar um modelo capaz de transcrever os elementos e relações semânticas de uma imagem representando um diagrama de classes para um formato que torne legível por computador em sintaxe *mermaid* (<https://mermaid.js.org/intro>). O programa também deve ser capaz de comparar a saída desse modelo a um código

fonte com representação escrita das classes, atributos, métodos e relações representadas no diagrama. Esse sistema deve ser integrado ao moodle para uso em correção de testes e exercícios automaticamente.

1.2 ABORDAGEM METODOLÓGICA

Este projeto se trata de uma pesquisa aplicada e multi métodos. Serão utilizados métodos quantitativos na implementação da avaliação dos códigos fonte e métodos qualitativos na interpretação das respostas dadas pelos alunos.

Em termos de etapas metodológicas, o desenvolvimento deste trabalho pode ser dividido nas seguintes etapas:

- analisar a literatura em relação à programação orientada a objetos e diagramas de UML, especialmente diagramas de classes;
- implementar a análise de uma imagem de um diagrama de classes, identificando os principais elementos de notação utilizados no diagrama;
- implementar a conversão dos elementos de notação presentes na imagem em um arquivo de formato legível pelo computador;
- implementar a conversão do código-fonte para o mesmo formato;
- implementar a comparação de similaridade dos dois arquivos gerados, e;
- Avaliar os resultados obtidos utilizando um métricas tipicamente utilizadas em desenvolvimento com IA.

1.2.1 Especificação De Escopo

O desenvolvimento será focado em desenvolver uma ferramenta útil para auxiliar na correção de atividades envolvendo a construção de um diagrama de classe a partir de código ou escrever código equivalente a um diagrama de classe, as quais são frequentes em cadeiras de programação orientada a objetos e tem uma única resposta correta e definida, apesar de esta resposta poder ser representada de forma variável quando expressada em uma imagem ou em código.

Devido a esse contexto, será focada a identificação de somente quatro tipos de relações encontradas em diagramas de classe, essas sendo: associação, agregação, composição e herança. Estas relações foram escolhidas porque são as mais frequentemente encontradas nas atividades avaliativas descritas.

1.2.2 Uso de Inteligência Artificial Generativa

Na elaboração deste artigo não foram utilizadas nenhuma ferramenta/modelo de Inteligência Artificial Generativa, porém Claude Sonnet 4.6 foi ocasionalmente utilizado para apoiar a produção de código.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são explorados conceitos relevantes para o desenvolvimento deste trabalho.

2.1 LINGUAGEM DE MODELAGEM UNIFICADA(UML)

A seguir, serão explorados um pouco da história de UML, os vários diagramas que a compõem e uma visão detalhada do diagrama de classes, um dos focos principais deste trabalho.

2.1.1 Histórico da UML

A primeira versão da *Unified Modelling Language*(UML), versão 0.8, foi publicada em 1995 pela empresa Rational, com Rumbaugh e Booch como autores, com Jacobson se juntando ao time na versão 0.9. Ela foi criada com o objetivo de padronizar e unificar as notações de metodologias voltadas ao desenvolvimento de especificações de análise de projeto orientadas a objetos, as quais existiam várias em uso na década de 80 e início da década de 90 gerando problemas para compartilhamento de modelagens entre profissionais que conheciam diferentes notações e limitando desenvolvedores de ferramentas de modelagem, que eram forçados a escolher apenas uma para focar em seus produtos, limitando a gama de escolha dos usuários.

UML não foi a primeira tentativa de unificar as notações de modelagem de projetos, porém foi a que foi aceita em larga escala e utilizada como padrão da indústria.

2.1.2 Diagramas da UML

A linguagem UML é composta por 14 diagramas divididos em duas categorias: diagramas de comportamento e diagramas estruturais.

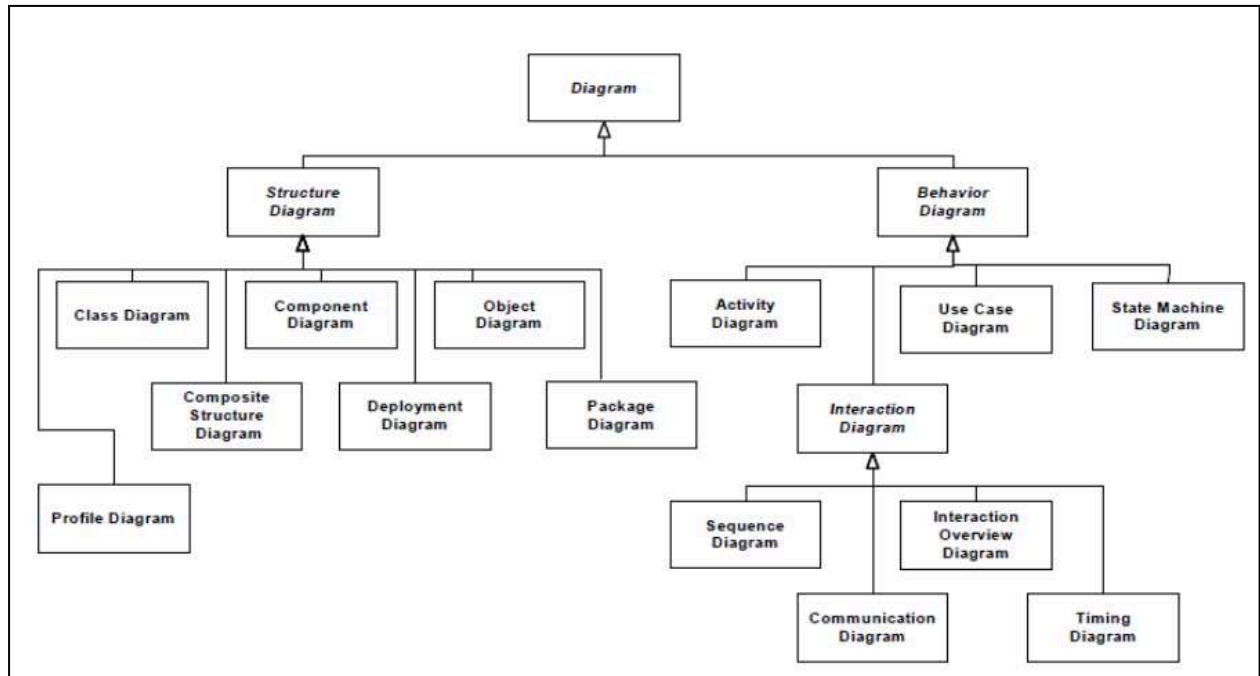


Figura 1: Taxonomia da estrutura de diagramas UML. Fonte: (SILVA, 2019).

Os diagramas de comportamento descrevem o sistema computacional em execução, representando a modelagem dinâmica do sistema, os seguintes diagramas pertencem a essa categoria: Diagrama de Casos de Uso, Diagrama de Máquina de Estados, Diagrama de Atividades e os diagramas pertencentes a subcategoria dos diagramas de interação: Diagrama de Sequência, Diagrama de Comunicação, Diagrama de Visão Geral de Interação e Diagrama de Temporização.

Diagramas estruturais tratam da estrutura e classes do sistema computacional, esses são: Diagrama de Utilização, Diagrama de Componentes, Diagrama de Estrutura Composta, Diagrama de Pacotes, Diagrama de Objetos e Diagrama de Classes (SILVA, 2019).

Na sequência, cada um dos diagramas da UML é brevemente apresentado (SILVA, 2019):

2.1.3 Diagrama de Casos de Uso

O Diagrama de Casos de Uso modela a dinâmica do sistema, representando a interação de atores externos ao sistema com elementos do sistema chamados casos de uso. Se encontra no nível mais alto de abstração oferecido pela UML (SILVA, 2019). Segundo (SHOEMAKER, 2004) o Diagrama de Casos de Uso mostra também o comportamento externamente visível do sistema.

2.1.4 Diagrama de Máquina de Estados

O Diagrama de Máquina de Estados faz uso de estados para descrever a situação atual de uma instância de classe, algoritmo de método de classe ou detalhar casos de uso. Seu segundo principal elemento sintático é a transição, que leva o objeto modelado de um estado a outro.

2.1.5 Diagrama de Atividades

O Diagrama de Atividades descreve uma fração do comportamento do sistema sendo modelado, essa modelagem descrita por um conjunto de atividades e ações. A diferença entre esses dois elementos está em sua atomicidade, ações são atômicas e não podem ser divididas, enquanto atividades podem ser separadas em ações ou atividades menores.

Além de seus componentes principais, esse diagrama faz uso de elementos de fluxo de controle que levam da execução de um elemento descritor para outro, e nodos para estabelecer o percurso de execução incluindo repetições, condicionais e concorrências.

2.1.6 Diagrama de Sequência

O Diagrama de Sequência representa as interações entre objetos ao longo do tempo, através de mensagens enviadas de um para o outro. Tem como objetivo refinar casos de uso. A passagem do tempo é representada pela altura dos eventos, quanto mais acima a mensagem foi enviada ou recebida, mais cedo ela ocorre.

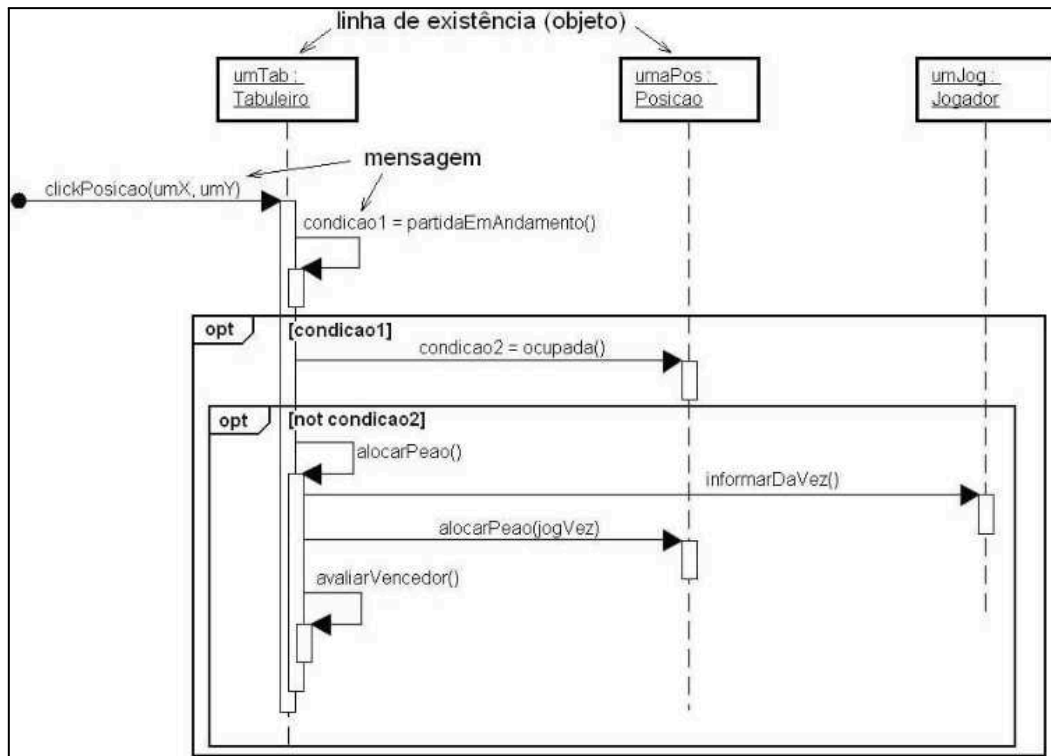


Figura 2: Exemplo de Diagrama de Sequência. Fonte: (SILVA, 2019).

2.1.7 Diagrama de Comunicação

O Diagrama de Comunicação usa os mesmos elementos sintáticos principais que o diagrama de sequência, mensagens e objetos, e serve uma função similar, agindo como uma alternativa para representação da interação de objetos. Diferente do diagrama de sequência, o diagrama de comunicação não modela a passagem do tempo.

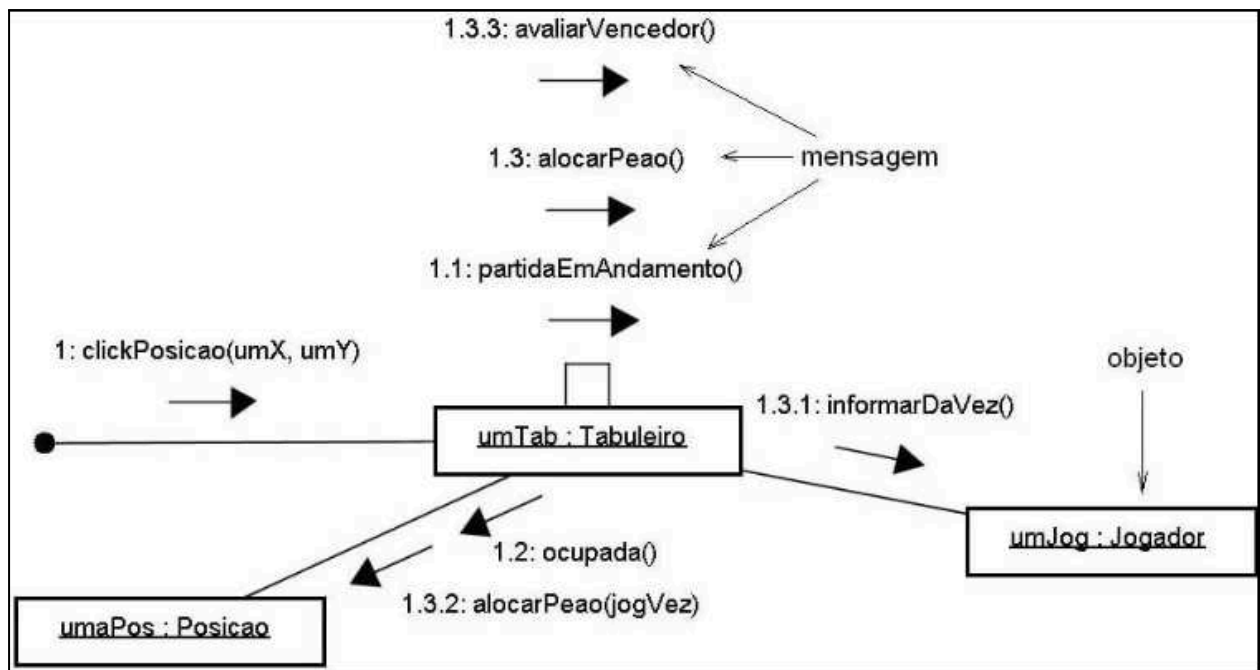


Figura 3: Exemplo de Diagrama de Comunicação. Fonte: (SILVA, 2019).

2.1.8 Diagrama de Visão Geral de Interação

O Diagrama de Visão Geral de Interação é uma modificação do diagrama de atividades, que faz uso de interações no lugar de ações e atividades. É focado em refinar casos de uso com o uso de interações entre elementos e atores.

2.1.9 Diagrama de Temporização

O Diagrama de Temporização tem como objetivo modelar restrições temporais do sistema, usando objetos como seu elemento sintático principal e descrevendo suas alterações de estado no tempo, incluindo limitações de tempo impostas no sistema.

2.1.10 Diagrama de Utilização

O Diagrama de Utilização busca mostrar a organização dos elementos necessários para a execução de um sistema computacional, usando como principal

elemento sintático o nodo, que representa um recurso computacional. Tanto nodos em si como instâncias de nodo podem ser usados na construção de um diagrama de utilização.

2.1.11 Diagrama de Componentes

O Diagrama de Componentes modela componentes e suas interações. Componentes representam conjuntos de classes, e assim podem apresentar a modelagem um nível de abstração acima do diagrama de classes.

2.1.12 Diagrama de Estrutura Composta

Os Diagramas de Estrutura Composta detalham elementos de modelagem estrutural, como classes, pacotes e componentes. Faz uso de um elemento sintático chamado porto, onde uma interface pode ser conectada ou pode ser usada para interligar elementos em colaborações, tornando esse diagrama útil para modelagem de padrões de projeto.

2.1.13 Diagrama de Pacotes

O pacote é um elemento sintático definido na primeira versão de UML, com o objetivo de conter outros elementos sintáticos, mas na segunda versão foi introduzido um diagrama dedicado a pacotes, chamado Diagrama de Pacotes, e suas conexões exclusivamente, que trata da modelagem estrutural do sistema. Um exemplo pode ser visto abaixo.

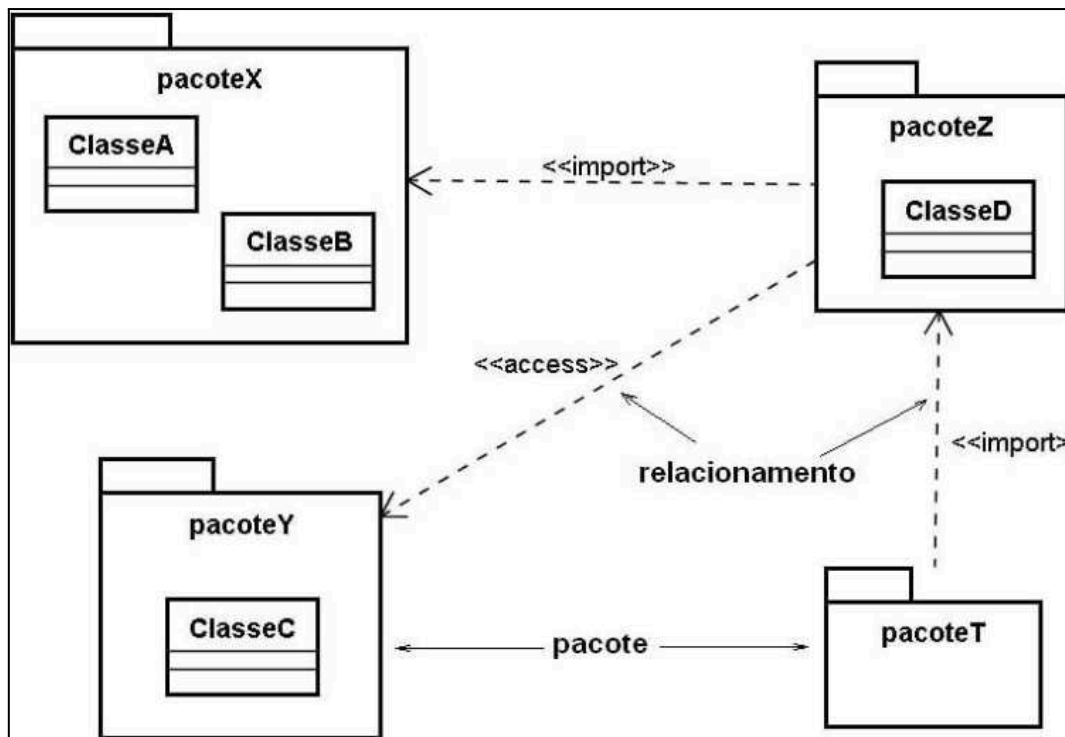


Figura 4: Exemplo de Diagrama de Pacotes. Fonte: (SILVA, 2019).

2.1.14 Diagrama de Profile

O Diagrama de Profile é o mais recente adicionado a UML, introduzido oficialmente a taxonomia de diagramas UML na versão 2.2.

Ele é usado como um mecanismo de extensão para metamodelos UML, permitindo sua adaptação para diferentes plataformas e domínios. Tem como elementos principais a Profile e Estereótipo (THE. . . , s.d.).

2.1.15 Diagrama de Objetos

O Diagrama de Objetos representa instâncias de objetos de classe existentes em um dado momento de execução do código modelado, funciona como uma fotografia do tempo de execução. É composto por objetos e suas interligações, podendo também mostrar atributos desses objetos com seus valores no momento representado.

2.1.16 Diagrama de Classes

O Diagrama de Classes é o modelo fundamental de uma especificação orientada a objetos. É o diagrama mais próximo do código, representando classes, seus atributos, métodos e relacionamentos entre classes interligadas. Como o foco deste trabalho é o Diagrama de Classes, este diagrama será descrito em mais detalhes na próxima sessão.

2.2 DIAGRAMA DE CLASSES

O Diagrama de Classes representa a visão estática do modelo, assim chamada pois não descreve comportamento dependente do tempo, ela modela conceitos do domínio da aplicação, assim como conceitos internos inventados como parte da implementação da aplicação (IVAR JACOBSON & BOOCH, 2000). Segue um exemplo de um Diagrama de Classe:

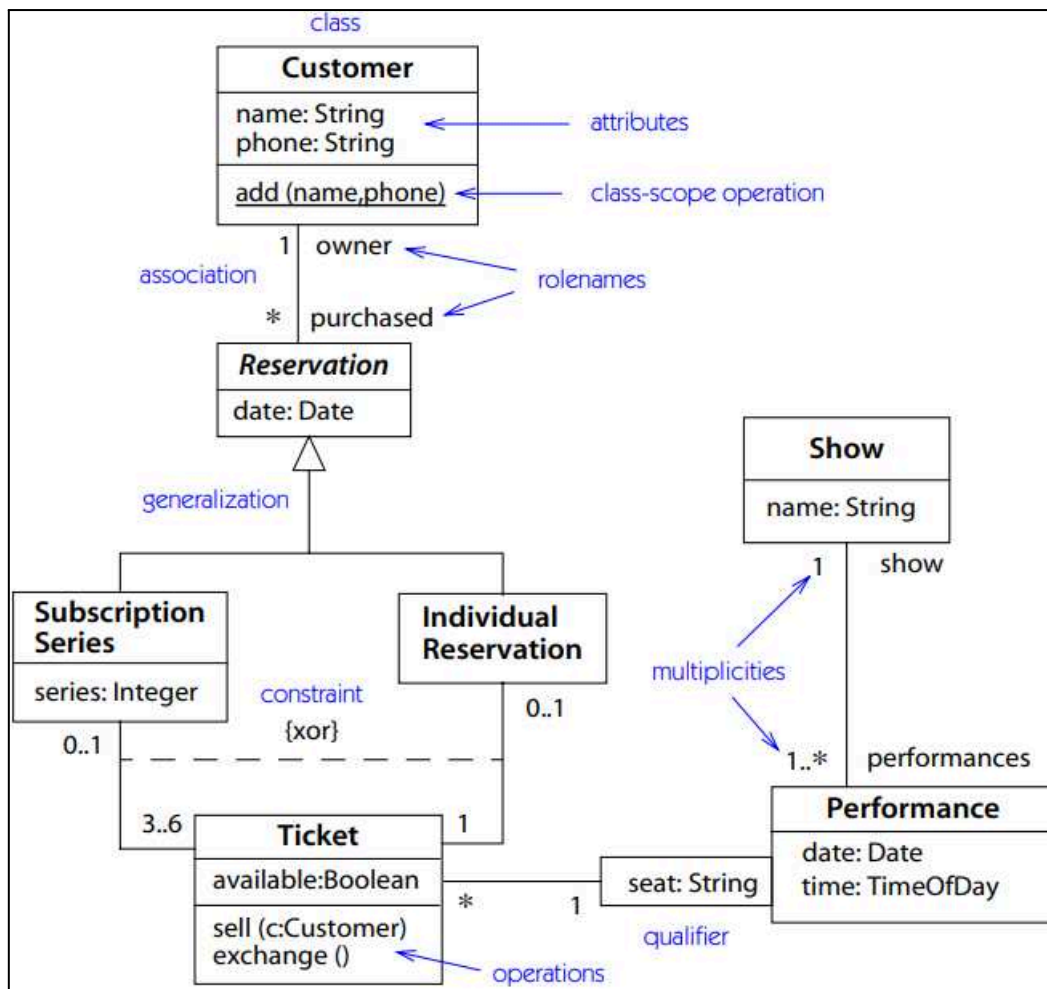
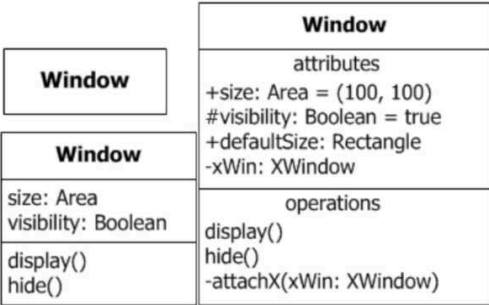
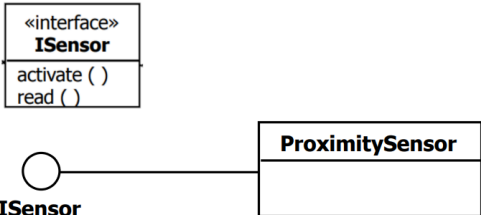
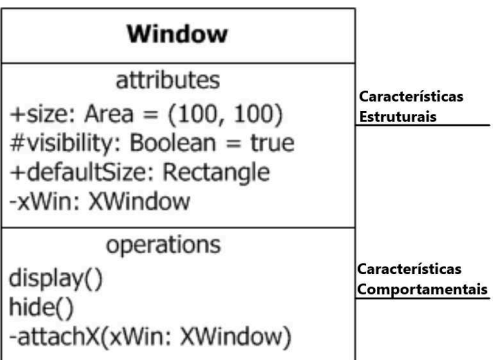
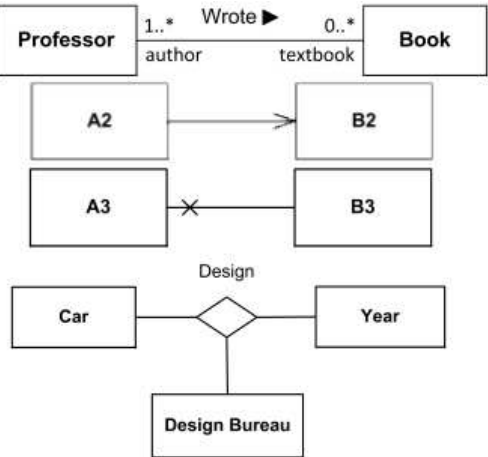


Figura 5: Exemplo de Diagrama de Classe. Fonte: (IVAR JACOBSON E GRADY BOOCH, 2000).

O Diagrama de Classes mostra a estrutura do sistema, subsistema ou componente com relação a classes e interfaces, com suas características, restrições e relações (THE. . . , s.d.). Na tabela a seguir, serão definidos os principais elementos do diagrama de classes junto de suas representações(Tabela 1):

Elemento	Representação	Descrição
----------	---------------	-----------

Classe	 <pre> classDiagram class Window { +size: Area = (100, 100) +visibility: Boolean = true +defaultSize: Rectangle -xWin: XWindow +display() +hide() -attachX(xWin: XWindow) } </pre>	A classe é um classificador que descreve um conjunto de objetos que compartilham as mesmas características, restrições e semântica. A classe é representada como um retângulo de linhas sólidas com seu nome escrito em negrito no centro, com a primeira letra em maiúsculo, opcionalmente pode conter compartimentos separados por linhas horizontais que contenham características ou outros membros da classe, como o classificador mais utilizado, não é necessário adicionar uma palavra-chave <<classe>> para identificá-la.
Interface	 <pre> classDiagram class ISensor { <<interface>> +activate() +read() } class ProximitySensor ISensor -- > ProximitySensor </pre>	Interface é um classificador que declara um conjunto de obrigações e características que juntos formam um serviço coerente. Ela especifica um contrato e qualquer instância de um classificador que realiza a interface, deve cumprir com esse contrato. Uma interface não pode ser instanciada, ao invés disso ela é implementada por outro classificador que se conforme a interface. Interfaces podem ser representadas de forma similar a classes, com a adição da palavra chave <<interface>>, alternativamente pode ser representada por uma bola com o nome da interface, com uma linha conectando-a ao classificador que a implementa.
Característica	 <pre> classDiagram class Window { +size: Area = (100, 100) +visibility: Boolean = true +defaultSize: Rectangle -xWin: XWindow +display() +hide() -attachX(xWin: XWindow) } </pre> Características Estruturais Características Comportamentais	Características representam aspectos estruturais (atributos) e comportamentais (operações) de classificadores. Uma característica estrutural é uma característica tipada que descreve a estrutura do classificador, define que instâncias do classificador do qual faz parte possuem um slot para valores de um dado tipo. Uma característica comportamental especifica um aspecto do comportamento das instâncias de um classificador e especifica que uma instância do classificador responderá a requisições específicas invocando um comportamento, sua lista de parâmetros descreve o tipo, a ordem e a direção de argumentos que podem ser dados quando um comportamento é chamado ou que serão retornados quando o comportamento finalizar. Não há um padrão para notação de características.

Restrição	<pre>{self.boss->isEmpty() or self.employer = self.boss.employer}</pre>	Uma restrição é uma asserção que indica uma limitação que deve ser seguida por qualquer implementação válida do modelo, ela é conectada a um conjunto de elementos restringidos. É normalmente representada por uma expressão booleana que deve resultar em um valor verdadeiro em uma implementação correta do modelo, opcionalmente pode ter nome mas costumam ser anônimas. A UML não aplica restrições à linguagem usada para escrever a restrição, podendo ser usada linguagens formais como OCL ou de programação como Java, mas também sendo válido o uso de linguagem coloquial. Ela pode ser inserida em um modelo através do símbolo de nota, também usado para comentários, diretamente ao lado de uma característica que ela restringe ou entre dois elementos restringidos por ela.
Associação	 <pre> classDiagram class Professor class Book Professor "1..*" -- "0..*" Book : Wrote Professor --> Book : author, textbook class A2 class B2 A2 --> B2 class A3 class B3 A3 -- B3 class Car class Year class DesignBureau[Design Bureau] Car -- > Year : Design DesignBureau -- Car DesignBureau -- Year </pre>	Uma associação é a representação de uma relação entre classificadores, que indica uma conexão entre eles ou que eles podem ser combinados física ou logicamente em uma agregação. São categorizadas pela especificação UML como relações semânticas. É frequentemente representada como uma linha sólida conectando dois classificadores ou um classificador a si mesmo, o nome da associação pode ser escrito no meio da linha, sem ser muito próximo de nenhum dos classificadores, cada ponta da linha pode ser decorada com os nomes das pontas da associação. A ponta da associação define a função do classificador na associação. Para indicar navegabilidade, isso é, acessibilidade eficiente da instância de um lado da associação pela instância do outro durante execução, usa-se um símbolo de seta para indicar que é possível navegar em uma direção, e um X para identificar não navegabilidade, o não uso de nenhum dos dois símbolos indica navegabilidade incerta. Quando uma associação envolve mais de 2 classificadores, ela pode ser representada por um diamante com o nome da associação, com linhas sólidas conectando-o aos classificadores.

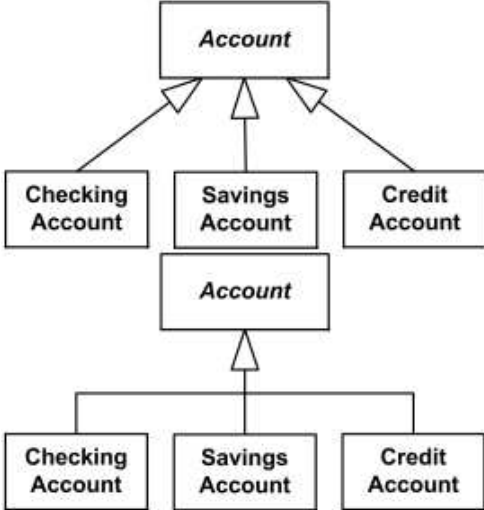
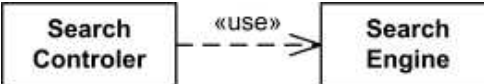
Generalização		A generalização é uma relação direcionada entre um classificador genérico(superclasse) e um ou mais classificadores específicos(subclasse). Cada instância do classificador específico também é indiretamente uma instância do classificador genérico(um estudante é uma pessoa por exemplo).A generalização é representada por uma linha com um triângulo vazio na ponta, formando uma seta saindo do classificador específico e apontando para o classificador genérico.
Dependência		Uma dependência é uma relação que significa que um ou mais elementos do modelo dependem de outros elementos para sua implementação ou especificação. Isso gera uma relação provedor - cliente, em que o provedor fornece algo para o cliente, levando o cliente a ficar incompleto enquanto dependente no provedor. Modificação do provedor pode afetar o cliente. A dependência é representada por uma linha tracejada com uma seta na ponta, saindo do cliente e apontando para o provedor, opcionalmente podendo ser identificada com um estereótipo.

Tabela 1: Elementos do Diagrama de Classes. Fontes: (UNIFIED. . . , 2017), (THE. . . , s.d.).

2.3 MACHINE LEARNING

Machine Learning, ou Aprendizado de Máquina, é o estudo científico de algoritmos e modelos estatísticos que sistemas computacionais usam para executar uma tarefa específica sem ser explicitamente programado. Esses algoritmos são usados para vários propósitos, como data mining, processamento de imagens e análise preditiva, entre outros (MAHESH, 2020). Algoritmos de Machine Learning constroem um modelo a partir de dados de treinamento, para poderem fazer previsões e tomar decisões. Machine Learning pode ser dividido nos seguintes tipos

de algoritmos: Supervised Learning, Unsupervised Learning, Semi-supervised Learning e Reinforcement Learning (SAMOILI et al., 2021).

2.3.1 Supervised Learning

Supervised Learning mapeia valores de entrada a valores de saída baseados em exemplos etiquetados de pares de entrada-saída. Supervised Learning requer grandes bases de dados etiquetadas por humanos para funcionar corretamente. Alguns algoritmos de Supervised Learning são: Regression, Graphical Models, Neural Networks, Deep Learning, Decisions Trees, Classifiers and Ensemble Learning (SAMOILI et al., 2021).

2.3.2 Unsupervised Learning

Unsupervised Learning busca achar padrões previamente desconhecidos em datasets sem etiquetas, com o objetivo de descobrir a estrutura dos dados, por exemplo agrupando itens similares formando um cluster. Unsupervised Learning não faz uso de dados etiquetados e não precisando de influência humana, é capaz de aprender por conta própria. Alguns algoritmos de Unsupervised Learning são: Clustering, Data Compression, Generative Adversarial Networks e Autoencoders (SAMOILI et al., 2021).

2.3.3 Semi-supervised Learning

Semi-supervised Learning é uma categoria de Machine Learning entre Supervised e Unsupervised Learning, fazendo uso de técnicas de ambos e datasets com e sem etiquetas (SAMOILI et al., 2021).

2.3.4 Reinforcement Learning

Reinforcement Learning explora como agentes tomam ações em um ambiente com o objetivo de maximizar uma recompensa. Alguns exemplos de algoritmos de Reinforcement Learning são: Deep Reinforcement Learning, Inverse Reinforcement Learning, Q-Learning.

2.4 Mermaid

Mermaid é uma sintaxe e também uma ferramenta de diagramação e criação de gráficos baseada em JavaScript, que renderiza definições textuais para criar e modificar diagramas dinamicamente (MERMAID, 2026).

A sintaxe mermaid, juntamente com a ferramenta, constituem a totalidade do Mermaid. Todas as definições de diagramas começam com uma declaração do tipo de diagrama, seguida pelas definições do diagrama e seu conteúdo. Essa declaração informa ao analisador qual tipo de diagrama o código deve gerar. Os comentários de linha podem ignorar tudo o que estiver na linha após '%%'. Palavras desconhecidas e erros de ortografia quebrarão um diagrama, enquanto os parâmetros falharão silenciosamente (MERMAID, 2026).

O mermaid pode definir e renderizar diversos tipos de diagramas, incluindo UML. Diagramas de classe são definidos em sintaxe mermaid por meio da definição do termo: classDiagram. O código a seguir, retirado da documentação oficial do mermaid¹ define um diagrama de classes para as classes: Animal, Pato, Peixe e Zebra.

```
---
title: Animal example
---
classDiagram
    note "From Duck till Zebra"
```

¹ <https://mermaid.js.org/syntax/classDiagram.html>

```
Animal <|-- Duck
    note for Duck "can fly<br>can swim<br>can dive<br>can
help in debugging"
Animal <|-- Fish
Animal <|-- Zebra
Animal : +int age
Animal : +String gender
Animal: +isMammal()
Animal: +mate()
class Duck{
    +String beakColor
    +swim()
    +quack()
}
class Fish{
    -int sizeInFeet
    -canEat()
}
class Zebra{
    +bool is_wild
    +run()
}
```

A renderização dessa especificação de diagrama de classes em texto por meio do motor de renderização do mermaid, resulta da seguinte imagem:

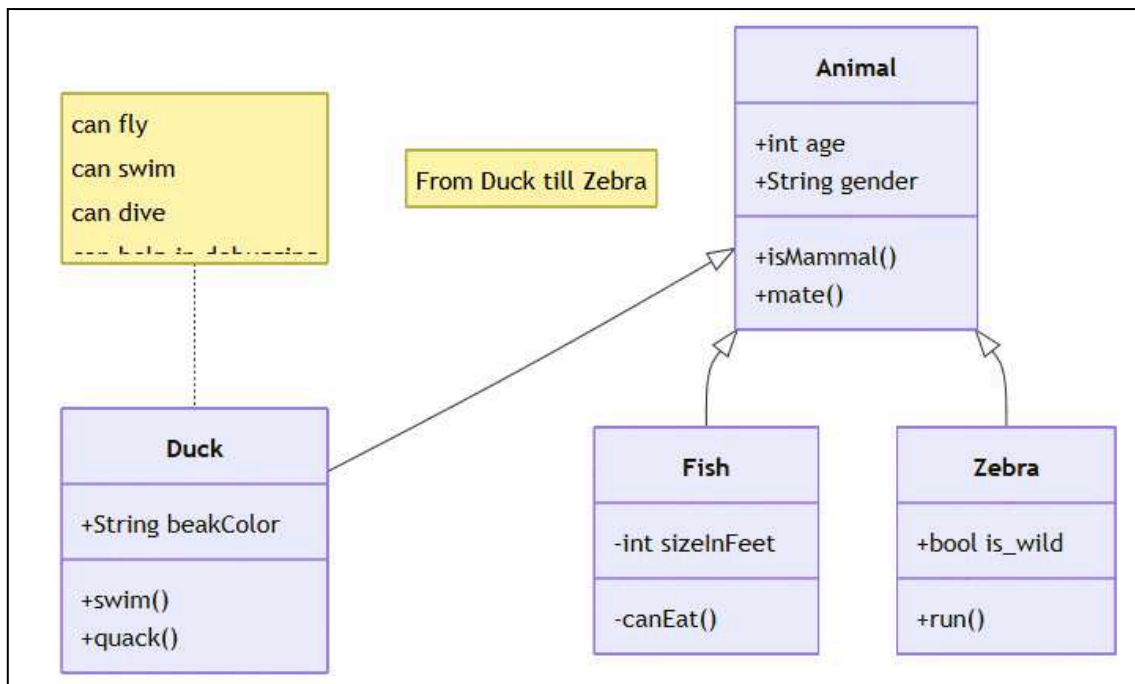


Figura 6: Exemplo de diagrama de classes exibido pelo renderizador do mermaid

A tabela 2 apresenta os principais elementos de notação de um diagrama de classes e sua respectiva notação em sintaxe mermaid².

Elemento	Sintaxe mermaid
Classe	<code>class BankAccount{</code> <code>}</code>
Interface	<code>bar ()-- foo</code>
Característica	<code>class BankAccount{</code> <code> +String owner</code> <code> +BigDecimal balance</code> <code> +deposit(amount) bool</code> <code> +withdrawal(amount) int</code> <code>}</code>
Associação	<code>classA --> classB</code>
Generalização	<code>classA < -- classB</code>
Dependência	<code>classK <.. classL</code>

Tabela 2: Sintaxe mermaid para diagramas de classe.

² <https://mermaid.js.org/syntax/classDiagram.html>

3 TRABALHOS CORRELATOS

Neste capítulo será discutida uma pesquisa do estado da arte relacionada a geração e análise de diagramas de classe a partir de imagens, junto a metodologia para essa pesquisa. Devido às limitações de um trabalho de conclusão de curso, não é realizado uma revisão sistemática de literatura, mas uma um mapeamento ad hoc de literatura, guiado por um protocolo simples mercado de busca.

3.1 DEFINIÇÃO DO PROTOCOLO DE REVISÃO

O objetivo da pesquisa feita neste capítulo é responder às seguintes perguntas com relação a literatura já existente com premissas similares ao do trabalho atual:

- Qual o objetivo dos trabalhos encontrados na literatura?
- Quais técnicas de IA foram usadas?
- Quais tecnologias foram mencionadas?
- Algum dataset existente foi utilizado?
- Quais resultados foram encontrados?

A pesquisa foi feita através da base de pesquisa Google Scholar, com um filtro para apenas mostrar resultados a partir de 2010, usando as seguintes strings de busca:

- "image recognition on uml class diagram"
- "uml class diagram generation from images"

Essas pesquisas foram feitas nos meses de Novembro e Dezembro de 2024, para cada *string* de busca foram exploradas as 10 primeiras páginas de resultados buscando artigos a serem incluídos nesse estudo. Segue uma lista dos critérios de inclusão aplicados na pesquisa:

- Foram considerados artigos que contenham modelos de geração de diagramas de classe ou reconhecimento dos mesmos em uma imagem.
- Somente foram considerados artigos de geração de modelos a partir de representações de diagrama de classes.
- As imagens usadas nos artigos podem ser tanto digitais, geradas por software ou capturas de tela, quanto imagens de desenhos físicos.
- Para tornar a pesquisa mais ampla, também foram incluídos artigos que geram outros diagramas UML além do diagrama de classes.

3.2 RESULTADOS DA PESQUISA

Para cada *string* de busca, foram avaliados 100 resultados, porém havendo artigos presentes em mais de uma pesquisa, 6 foram selecionados como relevantes baseado nos critérios de inclusão, através de um processo de avaliação primeiro dos títulos de cada artigo, seguidos de uma avaliação de seus *abstracts*. Em sequência é feita a análise dos resultados, na forma de respostas às perguntas propostas no início do capítulo.

3.2.1 Qual o objetivo desses trabalhos?

Todos artigos relevantes lidam com a identificação de elementos e semântica de diagramas UML, mas possuem diferentes motivações e métodos para fazê-lo. Os objetivos de cada artigo são apresentados na Tabela 3.

Citação	Objetivos
(KARASNEH, Bilal; CHAUDRON, M. R., 2013a)	Devido a diagramas UML serem frequentemente salvos como imagens, eles não são atualizados ao longo de um projeto porque recriar diagramas em ferramentas CASE custa tempo e não é prático, logo ter uma ferramenta para converter imagens geradas por ferramentas CASE de volta a arquivos legíveis por elas seria útil, o artigo apresenta a criação da

	ferramenta Img2UML como solução.
(KARASNEH, Bilal; CHAUDRON, M. R., 2013b)	Criado pelos mesmos autores de "Extracting UML Models from Images", apresenta o sistema Img2UML em detalhe.
(KARASNEH, B.; CHAUDRON, M., 2013)	Também relacionado ao sistema Img2UML, apresenta um repositório de diagramas UML em formato XMI para estudo.
(GOSALA et al., 2021)	Busca fazer uso de algoritmos de Deep Learning para atingir maior acurácia na classificação de diagramas de classe do que modelos existentes.
(CHEN et al., 2022)	Após uma pesquisa identificando diagramas de classe UML como os mais comuns utilizados, e que em sua maioria diagramas são colocados em documentos como imagens, apresentam uma solução para automaticamente identificar a semântica dos elementos de diagramas de classe fazendo uso de um método chamado ReSECDI(<i>Recognition of Semantic Elements in Class Diagram Images</i>)
(RAŽINSKAS et al., 2024)	Transformar imagens de rascunhos de diagramas de caso de uso em arquivos XMI.

Tabela 3: Objetivos dos Artigos Relevantes.

3.2.2 Quais técnicas de IA foram usadas?

Dois dos artigos descrevem métodos de IA e treinamento em detalhe: "Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network" (GOSALA et al., 2021) e "Transforming Sketches of UML Use Case Diagrams to Models" (RAŽINSKAS et al., 2024). Ambos fazem uso de técnicas de Redes Neurais e Deep Learning para desenvolver um modelo de classificação de diagramas de classe, mais especificamente, *Convolutional Neural Networks* ou CNN. Os outros trabalhos descrevem o uso de ferramentas e tecnologias já existentes que serão mencionados na próxima subseção.

3.2.3 Quais tecnologias foram mencionadas?

A seguir são citadas as tecnologias mencionadas por cada artigo estudado(Tabela 4). A maioria das tecnologias mencionadas são bibliotecas ou frameworks para detectar formas, linhas e texto em imagens(AForge.NET, MODI, OpenCV, Point-Polygon, TesseractOCR, Google Cloud Vision API) mas também há ferramentas completas de converter imagens em diagramas de vários tipos(Img2UML, Flowmind2Digital, Sketch2Process, DrawNet), tecnologias relacionadas a detecção de objetos(YOLO, SSD, Arrow R-CNN, Faster R-CNN) e a redes neurais(TensorFlow).

Citação	Ferramentas/Tecnologias
(KARASNEH, Bilal; CHAUDRON, M. R., 2013a)	Img2UML, StarUML, Aforge.NET, Microsoft Office Document Imaging(MODI)
(KARASNEH, Bilal; CHAUDRON, M. R., 2013b)	Img2UML, StarUML, Microsoft Office Document Imaging(MODI), VB.NET, AForge.NET
(KARASNEH, B.; CHAUDRON, M., 2013)	Img2UML
(GOSALA et al., 2021)	Google-Colab, TensorFlow
(CHEN et al., 2022)	Img2UML, OpenCV, AForge.NET, Point-Polygon, TesseractOCR
(RAŽINSKAS et al., 2024)	Arrow R-CNN, DrawNet, YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), Sketch2Process, Flowmind2Digital, Img2UML, Faster R-CNN, Google Cloud Vision API, S2D, TensorFlow

Tabela 4: Tecnologias Mencionadas em Artigos Relevantes.

3.2.4 Algum dataset existente foi utilizado?

Em "Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network" (GOSALA et al., 2021) faz uso de um dataset de um artigo de conferência chamado "Automatically Classifying UML Class Diagrams from Images using Deep Learning" (DATASET. . . , 2021) que contém 3298 imagens das quais metade são de diagramas de classe UML e a outra metade

são outras imagens que não são diagramas de classe, contendo gráficos, fluxogramas, imagens de animais entre outras. Infelizmente esse dataset é restrito. No artigo "Transforming Sketches of UML Use Case Diagrams to Models" (RAŽINSKAS et al., 2024), foi criado um dataset de imagens de rascunhos de diagramas de casos de uso, esse dataset não foi disponibilizado e não seria muito útil para esse trabalho, devido a focos em diagramas UML diferentes.

3.2.5 Quais resultados foram encontrados?

As soluções encontradas nos trabalhos correlatos(Tabela 5) tem alta precisão para identificação de elementos do diagrama de classes. Os modelos baseados em redes neurais tem como principal limitação o pequeno tamanho de seus datasets usados no treinamento.

Citação	Resultados
(KARASNEH, Bilal; CHAUDRON, M. R., 2013a)	100% de precisão em detectar classes, 97% em detecção de relacionamentos e 85% em detecção de símbolos.
(KARASNEH, Bilal; CHAUDRON, M. R., 2013b)	Precisão de Img2UML para reconhecimento em imagens de modelos de classe é aproximadamente 89%.
(KARASNEH, B.; CHAUDRON, M., 2013)	Apenas apresenta um repositório de Img2UML, não há resultados a relatar.
(GOSALA et al., 2021)	Com regularização, precisão de 96% nos dados de treinamento e 86,6% nos de teste, sem regularização, precisão de 94,5% nos dados de treinamento e 85,4% nos de teste. Precisão comparável a estudos similares, com potencial para melhorar, mas a qualidade é limitada pelo tamanho do dataset.
(CHEN et al., 2022)	Para todas as categorias, tem performance com precisão acima de 95% quando testado com exemplos de UML da comunidade open source, mantendo uma performance estável mesmo em diagramas com estilos diferentes, precisão cai com modelos mais complexos, ou seja, que contém muitas classes e relações entre elas.
(RAŽINSKAS et al., 2024)	Precisão geral de 93%, com precisão de detecção

	de elementos individuais em 97% colocando a solução a par de outras como DrawNet e Arrow R-CNN.
--	---

Tabela 5: Resumos de Resultados dos Artigos Relevantes.

3.3 DISCUSSÃO

Os resultados dessa pesquisa demonstram alguns padrões úteis para o desenvolvimento deste trabalho, primeiramente apesar das ferramentas e metodologias variam, há uma sequência de passos seguida por todos os artigos estudados para o desenvolvimento de suas soluções: identificar as formas da classe, subdividir essa forma em até 3 partes para considerar as áreas usadas para atributos e métodos, identificar texto, identificar linhas e identificar relações. A forma como esses passos são executados foi desenvolvida de formas diferentes em cada artigo, demonstrando uma grande variedade de meios para se chegar a solução, mas todos incluem esses passos.

O segundo é a existência de múltiplas ferramentas dedicadas a transformar imagens de diagramas de classe em formatos legíveis por ferramentas CASE já prontas, possibilitando usá-las de forma integrada com o moodle para atingir o objetivo deste trabalho.

Apenas dois trabalhos oferecem soluções com modelos de machine learning, e só um deles é dedicado a identificação de diagramas de classe, limitando as fontes que podem ser usadas para o desenvolvimento e treinamento de uma solução com IA, porém essas soluções tinham alguns dos melhores resultados entre os trabalhos estudados, levando a conclusão de que usar uma rede neural seria o método mais efetivo.

4 FERRAMENTA DE AVALIAÇÃO DE CÓDIGOS-FONTE COM BASE EM DIAGRAMA DE CLASSES

Este capítulo apresenta a ferramenta desenvolvida para a avaliação de códigos-fonte desenvolvido por alunos por meio da comparação entre a solução implementada em linguagem de programação Python e um diagrama de classes definido. Essa avaliação se dá por meio da conversão tanto das imagens de diagramas de classe UML quanto dos códigos Python contendo as implementações das classes, em formato *mermaid* da sua comparação.

4.1 ANÁLISE DE REQUISITOS

O principal requisito desta ferramenta é a geração de arquivos de texto em sintaxe compatível com o editor de diagramas e gráficos *mermaid* (<https://mermaid.js.org/intro/>) a partir de dois tipos de entradas diferentes: imagens de diagramas UML e arquivos de código em python. Após construídos os arquivos *mermaid*, o programa deve comparar ambos, tratando um como gabarito de avaliação e outro como a resposta a ser testada, expressando se ambos descrevem o mesmo diagrama de classe, e se não, apontar os erros encontrados no arquivo resposta. O programa pode ser dividido em 3 fluxos de trabalho:

- Conversão de imagem em arquivos *mermaid*
- Conversão de código em arquivos *mermaid*.
- Comparação de similaridade entre dois arquivos *mermaid*.

As conversões de imagem e código para *mermaid* podem ser efetuadas em qualquer ordem, ou podem ser executadas múltiplas vezes ou puladas em casos que ambos lados da comparação são o mesmo tipo de entrada ou documentos

mermaid são providenciados como entradas diretamente, mas devem ocorrer antes da comparação.

Durante a conversão de imagens de diagramas de classe em sintaxe *mermaid*, serão identificados os elementos do diagrama e suas posições e, através dessa informação, as relações entre classes. Para propriamente converter as imagens em texto, um processo de OCR(*optical character recognition*) deve ser aplicado a elementos de texto identificados, permitindo anotação de atributos, métodos e nomes de classes.

A conversão de código para *mermaid* ocorre por identificação de padrões através do uso de expressões regulares, fazendo uso da gramática e sintaxe de python como uma linguagem de programação para reconhecer a estrutura e relações entre classes.

Por fim com dois arquivos contendo descrições de diagramas de classe UML em *mermaid*, a etapa de comparação é tratada com uso de *string matching*, encontrando e comparando strings similares nas duas representações de diagrama, e pontuando sua similaridade, casos em que a linha mais similar encontrada tem pontuação baixa, levam a uma indicação de onde a discrepância ocorre no gabarito e na resposta.

4.2 CRIAÇÃO DO MODELO DE VISÃO COMPUTACIONAL

Esta subseção detalha o processo de criação do modelo de IA e código envolvidos na conversão de uma imagem de diagrama de classes em um documento *mermaid*. Grande parte desse processo foi feito por dentro da plataforma Roboflow (<https://roboflow.com>), uma plataforma online para criação de modelos de

inteligência artificial para visão computacional, antes de exportar o modelo resultante para uso na ferramenta.

4.2.1 Coleta de Imagens

A construção do modelo de detecção visual de objetos foi iniciada com a preparação de um *dataset* para ser usado no processo de treinamento. Imagens de diagramas de classe produzidos digitalmente foram obtidas manualmente a partir de diversas pesquisas no Google Imagens e suplantadas por imagens disponíveis publicamente em projetos existentes no Roboflow.

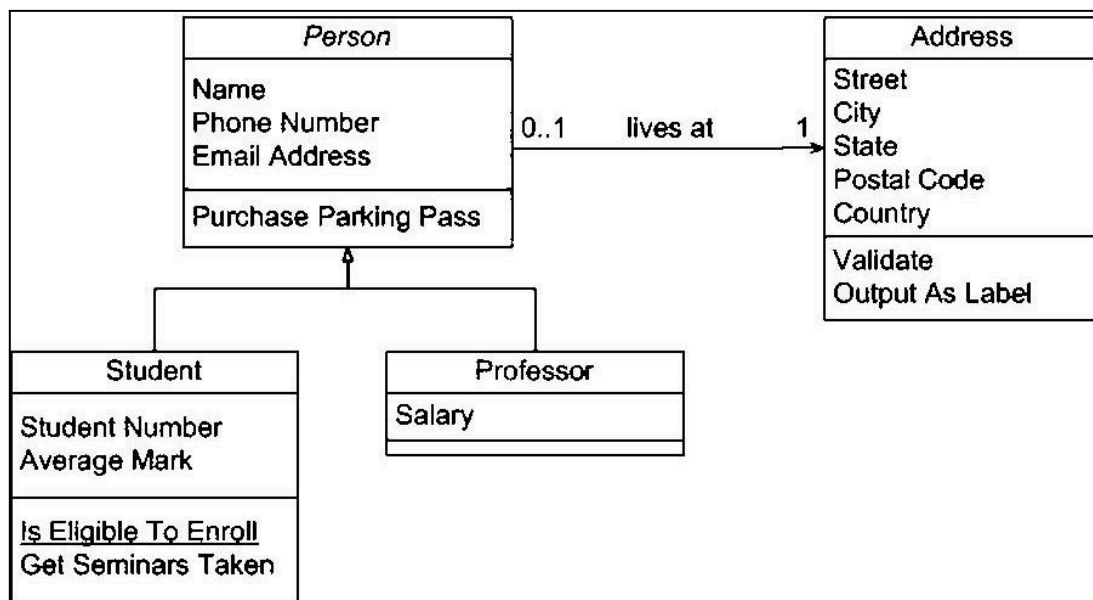


Figura 7: Um diagrama incluído no *dataset*³.

Das imagens obtidas, foi realizada uma filtragem manual para retirar qualquer uma com qualidade baixa demais para seu texto ser legível, resultando no fim em um banco com 102 imagens no total consideradas apropriadas para uso.

³ <https://universe.roboflow.com/image2mermaid/img2mermaid>

4.2.2 Labelling

Para uso no processo de *machine learning*, além de imagens é preciso de um arquivo YAML⁴ contendo as *labels* ou etiquetas que identificam objetos em todas as imagens e suas posições em cada uma delas, o processo de criar essas etiquetas precisa ser feito manualmente.

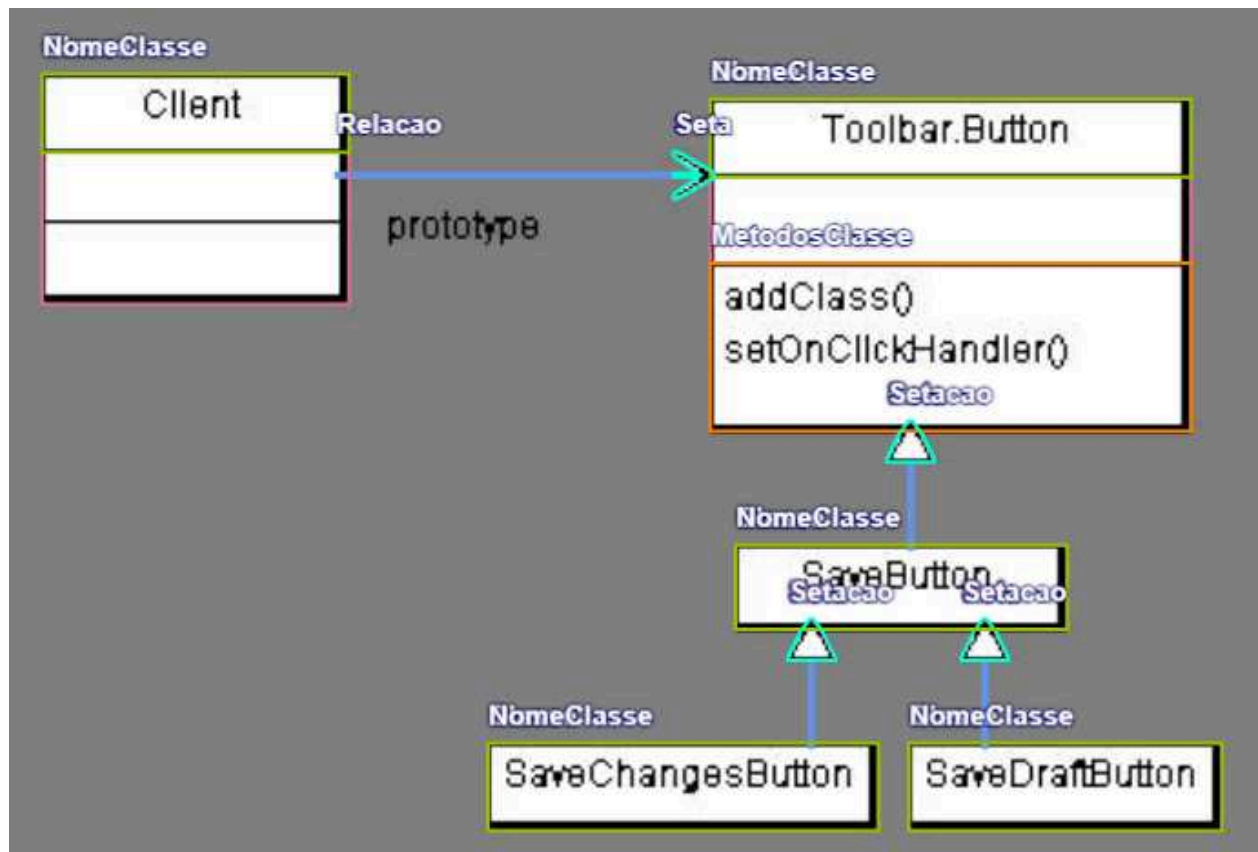


Figura 8: Exemplo de imagem do *dataset* após anotações.

Foram selecionados como objetos a identificar com *labels* o mínimo necessário para identificar a presença de classes, suas seções de texto internas (nome, atributos, métodos), relações entre classes e a direção dessas relações. Esses objetos são: Classe, Nome de Classe, Atributos de Classe, Métodos de Classe, Relação entre Classes, Seta e Losango.

⁴ <https://yaml.org/>

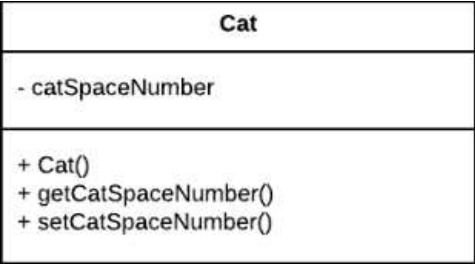
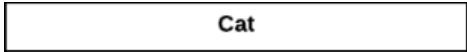
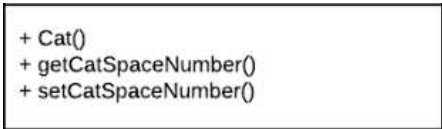
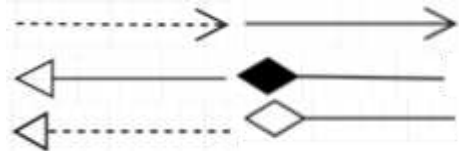
Elemento	Descrição	Exemplos
Classe	Formas retangulares que contêm texto dentro de sua área. Classes podem ser subdivididas em até 3 partes, essas divisões sendo feitas por linhas horizontais dentro da forma principal.	 <pre> classDiagram class Cat { - catSpaceNumber + Cat() + getCatSpaceNumber() + setCatSpaceNumber() } </pre>
Nome de Classe	Somente existe dentro de uma Classe e em casos que a classe não contém subdivisões, é equivalente a ela, em caso de existirem subdivisões, o texto em Nome de Classe é iniciado com letra maiúscula e se encontra no topo da Classe.	
Atributos de Classe	Somente existe dentro de uma Classe com uma ou duas subdivisões. Se encontra diretamente abaixo do Nome de Classe e não inclui parêntesis.	
Métodos de Classe	Somente existe dentro de uma Classe com uma ou duas subdivisões. Cada linha de texto termina com parêntesis, podendo ou não ter texto entre "(" e ")". Em casos com somente duas subdivisões, fica abaixo do Nome de Classe, porém se houver três, fica abaixo de Atributos de Classe.	
Relação	Uma linha, contínua ou pontilhada, com ou sem elementos adicionais (como setas ou losangos em suas pontas) que conecta duas classes é chamada de relação.	
Seta	Existe como parte de uma relação, aparecendo em uma de suas pontas, e pode ser composta por um triângulo (seta "fechada") ou um formato de "V" (seta "aberta").	
Losango	Existe como parte de uma relação, aparecendo em uma de suas pontas, como o nome indica tem o formato de um losango, podendo ser ou não preenchido por preto.	

Tabela 6: Exemplos dos elementos de diagrama de classe no *dataset*.

Classes são o elemento central de um diagrama de classe, e estão sempre presentes nos diagramas de classes, com seus componentes recebendo etiquetas para auxílio no processo de identificar o texto escrito. A posição de linhas contínuas ou pontilhadas, chamadas conjuntamente no *dataset* de relações, e quais classes

intersectam com elas são necessárias para encontrar classes conectadas. Setas e losangos dão direção para relações e auxiliam em dar nome a relação de forma mais específica, com setas identificando a classe mais próxima como destino da relação, e losangos identificando a classe mais próxima como origem.

Durante a produção desse trabalho houveram tentativas de construir modelos a partir de etiquetas e objetos um pouco mais específicos, como separar a etiqueta de losangos em duas etiquetas diferentes, uma para losangos brancos e outra para losangos pretos, o que permitiria a correta identificação do tipo de relação em mais casos, porém isso levou a uma redução na habilidade do modelo de identificar os demais elementos de notação, o que levou à decisão de manter essas classificações mais gerais.

O processo de anotação foi feito através do anotador do Roboflow⁵, e com ajuda de 4 voluntários. As etiquetas foram marcadas com caixas simples onde possível, mas fazendo uso de polígonos mais complexos para seguir a forma de elementos mais complexos. Essa atenção extra se deve a necessidade de o modelo final ser capaz de segmentação além de detecção simples, produzindo máscaras das formas encontradas além da posição geral do objeto para propriamente identificar relações de classe. A partir do resultante conjunto de etiquetas, é possível identificar a frequência de cada elemento de diagrama(Tabela 7).

Label	Ocorrências	Frequência
Classe	650	20,6%
NomeClasse	651	20,6%
AtributosClasse	338	10,7%
MetodosClasse	364	11,5%
Relacao	626	20%

⁵ <https://roboflow.com/annotate>

Seta	366	11,6%
Losango	155	5%
Total	3150	100%

Tabela 7: Frequência de cada *label* no *dataset*.

4.2.3 Pré processamento

O dataset foi dividido aleatoriamente em 3 subconjuntos, com 72 imagens sendo reservadas para treino, 20 para validação e 10 para teste. Os tamanhos de cada conjunto foram definidos com a intenção de maximizar a quantidade de imagens envolvidas em treinamento, mas mantendo um número suficiente separadas para validação e teste, dos quais validação recebeu prioridade porque é utilizada como parte ativa do processo de treinamento para identificar a qualidade atual do modelo em cada *epoch*, enquanto o conjunto de teste somente é utilizado após o modelo estar pronto para medir performance com dados que ele não é familiar. Fora seguir esses princípios de quais conjuntos devem ser priorizados, não houve um método claro para definição de quantas imagens cada conjunto receberia, e a decisão foi feita de forma arbitrária, com a intenção de modificar esses valores se resultados dos treinamentos fossem insatisfatórios.

Para aprimorar o treinamento do modelo foram aplicados alguns passos de pré processamento oferecidos pelo Roboflow, padronizando entradas a condições mais similares e previsíveis:

- Aplicação de *greyscale*, já que é esperado que a grande maioria dos diagramas de classe vão ser produzidos em tons de cinza;
- Ajuste de contraste automático, tornando limitações de linhas mais claras e especialmente deixando a forma de caracteres mais definidos;
- Ajuste de tamanho para uma resolução de 640 pixels por 640 pixels sem distorção, preenchendo a imagem com a cor branca onde

necessário para encaixar a imagem, esse é o tamanho de imagem que o modelo aceitará como entrada após pronto, logo treinar em imagens nesta resolução é ideal.

Preprocessing	Auto-Orient: Applied
	Resize: Fit (white edges) in 640x640
	Grayscale: Applied
	Auto-Adjust Contrast: Using Contrast Stretching
	Random Sample: Include 100% train, 100% valid, 100% test

Figura 9: Opções de pré processamento utilizadas.

Além dos passos já mencionados, também foi importante expandir artificialmente o *dataset* através de modificações devido a ele ser relativamente pequeno, através de aplicação de espelhagem vertical ou horizontal, rotações e adição de pixels de interferência ou *noise* a imagens do conjunto de treino, expandindo o mesmo para 216 imagens, e aumentando o tamanho total do banco a 246 imagens.

Augmentations	Outputs per training example: 3
	Flip: Horizontal, Vertical
	90° Rotate: Clockwise, Counter-Clockwise, Upside Down
	Noise: Up to 0.5% of pixels

Figura 10: Opções de *augments* utilizadas.

4.2.4 Treinamento

Treinar um novo modelo somente a partir do *dataset* preparado não seria viável. Então, ao invés disso, foi feito um processo de *fine-tuning* em um modelo já existente e com apoio de um dataset maior.

O *dataset* oferecido pelos servidores do Roboflow correspondente à mesma tarefa que o modelo a ser treinado é o COCO-seg, uma extensão de COCO(*Common Objects in Context*) adaptada para treinamento de modelos de

segmentação com anotações mais detalhadas, mas mantendo as mesmas 330.000 imagens do original (LIN et al., 2015).

O modelo usado para treinamento foi YOLOv11, sendo a versão de YOLO mais recente disponível no Roboflow⁶.

⁶ <https://playground.roboflow.com/models/tencent-ai-lab/yolo-world>

4.2.5 Avaliação do Resultado

Modelos de visão computacional podem ser avaliados por meio de diversas métricas com diferentes propósitos, as usadas para medir o desempenho desse modelo foram mAP (*mean Average Precision*) e o *F1-score*.

Ambas métricas são baseadas em dois conceitos mais simples: precisão e *recall*. Precisão pode ser definida como a fração de todas inferências feitas que foram corretas, enquanto *recall* é a fração de todos acertos que foram previstos (LIPTON et al., 2014). No contexto de um modelo de visão computacional, inferências são as detecções que atingem um valor de IoU (*Intersection over Union*) mínimo com a etiqueta que aponta para o objeto verdadeiro, sendo a detecção com pontuação mais alta considerada correta enquanto todas outras são consideradas falsas positivas (HENDERSON et al., 2016).

Para calcular o mAP de um modelo, é primeiro necessário calcular a precisão média, ou AP, de cada classe de objeto detectado, o que é feito a partir da formação de pares de valores de precisão e *recall* para cada detecção feita e as ordenando de forma decrescente, formando uma curva de *precision-recall*(PR), cuja área abaixo dessa curva define AP da classe, a média de todos valores AP de cada classe de objeto resulta em mAP (WANG, 2022). O valor de AP, e por consequência de mAP, pode ser alterado pelo limiar de IoU utilizado, para esse projeto, a medida foi feita com um limiar de 50%, o que pode ser representado no nome da métrica como mAP@50.

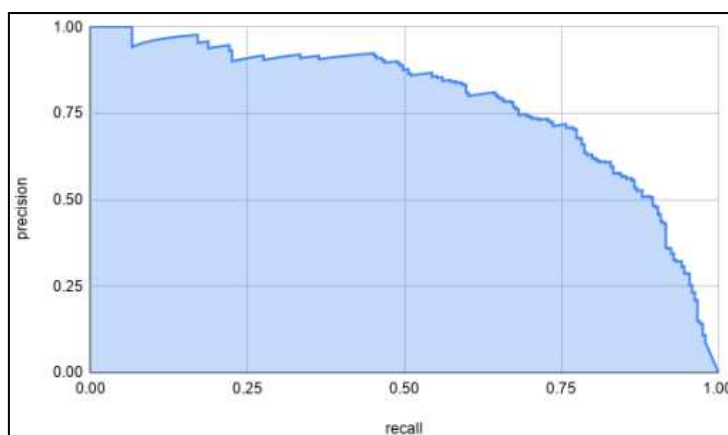


Figura 11: Exemplo de um gráfico de curva PR. Fonte: (WANG, 2022).

O *F1-score* é a média harmônica entre precisão e *recall* em que ambos possuem peso igual (LAM et al., 2023).

O modelo final produzido pontuou **96,4% em precisão**, **92,5% em *recall***, **97% em mPA@50** e **94,4% em *F1-score*** quando testados sobre o conjunto de validação do dataset, indicando boa performance.

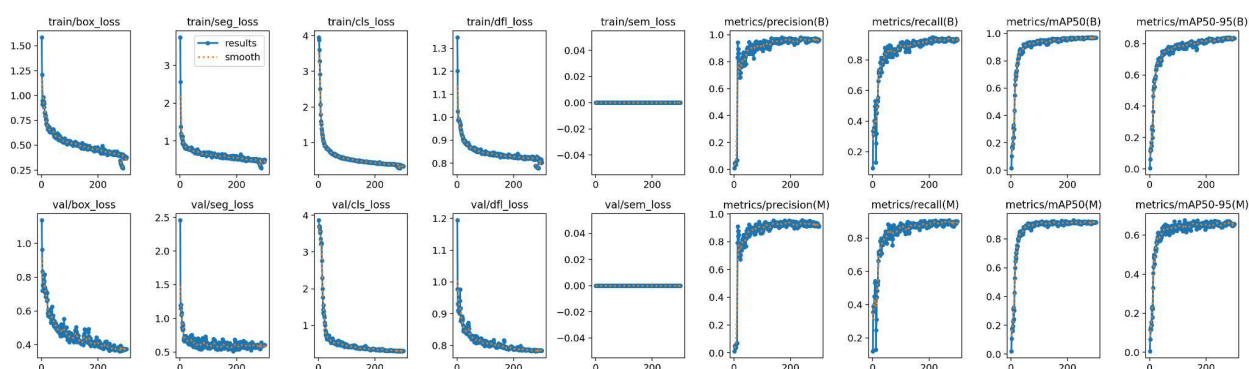


Figura 12: Gráficos de diversas métricas analisadas durante treinamento.

4.3 CONVERSÃO DE IMAGENS EM LINGUAGEM *MERMAID*

Com o modelo treinado e qualidade suficiente, o desenvolvimento foi avançado para a construção do módulo do programa que faz uso das previsões do modelo para construir uma representação do diagrama em *mermaid*.

Fazendo uso da biblioteca *ultralytics*⁷ para carregar e executar inferências com os pesos do modelo treinado, produz-se uma lista de caixas de detecção e máscaras baseadas na imagem de entrada, que foram consumidas para a criação de um objeto que representa o diagrama de classes a ser descrito, conforme mostra o trecho de código da Figura 13.

```
112 def get_results_diagram(result, reader):
113     detections = []
114     boxes = result.boxes
115     masks = result.masks
116     original_image = result.orig_img
117     #Coloca a imagem em tons de cinza para operações OCR se não tiver só um\
118     # canal já
119     if len(original_image.shape) >= 3:
120         grey_img = cv2.cvtColor(original_image.copy(), cv2.COLOR_BGR2GRAY)
121     else:
122         grey_img = original_image
123
124     for i, box in enumerate(boxes):
125         c_name = result.names[int(box.cls.item())]
126
127         new_detection = DetectionObjClass(grey_img, box, masks[i], c_name, i)
128         detections.append(new_detection)
129
130     diagram = DiagramaUML(grey_img, detections, reader)
131     return diagram.mermaid
```

Figura 13: Trecho de código coletando dados relevantes dos resultados da inferência.

O primeiro passo após adquirir as coordenadas das detecções e construir os elementos do diagrama de classes dentro do objeto, é extrair o texto de cada porção das classes e associá-lo à classe correta. Isso foi feito por um processo de combinar caixas de detecção sobrepostas e identificadas como mesma parte da classe, antes de então ajustá-las para garantir que cobrem somente a parte que correspondem a sua classificação, cortando também linhas externas e divisórias, idealmente deixando cada seção apontando para um pedaço da imagem contendo nada além de texto em fundo branco (Figura 14).

⁷ <https://www.ultralytics.com>

```

def adjust_sections(self):
    #Se houver mais de uma detecção para uma seção da classe, combinar as\
    # que tem intersecção com a de mais alta confiança, e retirar o resto
    if not self.__has_name():
        print(f'Nome Inexistente Para Classe: {self.__detect_id}')
    elif len(self.__umlclass_name_list) > 1:
        self.__combine_detections('name')
    if len(self.__umlclass_attribute_list) > 1:
        self.__combine_detections('att')
    if len(self.__umlclass_method_list) > 1:
        self.__combine_detections('method')
    if self.__has_name():
        self.__name = self.__umlclass_name_list[0]
    if self.__has_attributes():
        self.__attributes = self.__umlclass_attribute_list[0]
    if self.__has_methods():
        self.__methods = self.__umlclass_method_list[0]
    #Checa se há um objeto nome existente, e se não cria um com as\
    # coordenadas da classe se não houver métodos ou atributos também e\
    # retorna False, se houverem, levanta uma exceção
    if not self.__class_is_name():
        #Modifica coordenadas dos blocos de classe para serem o maior\
        # possíveis e conectadas a suas seções vizinhas
        self.__subsection_bbox_maximize()
    # Enquadra a bounding box para só manter o texto em fundo branco
    self.__fit_to_sections()

```

Figura 14: Código para ajustes de seção de classe.

Com as peças de texto isoladas, o processo de OCR pode ser executado. Foi usada a biblioteca *easyOCR*⁸ para esse processo. Para melhores resultados, os recortes da imagem contendo o texto são divididos a cada linha por uma função identificando espaços em branco entre cada conjunto de caracteres.

⁸ <https://github.com/jaidedai/easyocr>

```

def cut_lines(self):
    #Soma valor de cada linha e encontra a soma máxima indicando linha\
    # completamente branca
    bin_crop = cv2.threshold(self._crop, 0, 255, cv2.THRESH_BINARY
                             + cv2.THRESH_OTSU)
    row_sums = np.sum(bin_crop, axis=1)
    max_sum = 255 * bin_crop.shape[1]
    #Encontra linhas vazias
    empty_rows = np.where(row_sums == max_sum)[0]
    #Encontra linhas de conteúdo entre os espaços vazios
    split_points = []
    gap_tolerance = 2
    if len(empty_rows) > 0:
        u_empty_rows = np.unique(empty_rows)
        run_start = u_empty_rows[0]
        run_end = u_empty_rows[0]
        for r in u_empty_rows[1:]:
            if r == run_end + 1:
                run_end = r
            else:
                #Espaços vazios pequenos são ignorados
                if (run_end - run_start + 1) >= gap_tolerance:
                    split_points.append((run_start, run_end))
                run_start = r
                run_end = r
            if (run_end - run_start + 1) >= gap_tolerance:
                split_points.append((run_start, run_end))
    #Cria e armazena os recortes de cada linha
    prev = 0
    for run_start, run_end in split_points:
        line = self._crop[prev:run_start, :]
        #Só armazena se houver texto no intervalo
        self._append_lines(line)
        prev = run_end + 1
    #Pega última linha além do intervalo final se existir
    if prev < self._crop.shape[0]:
        line = self._crop[prev:, :]
        self._append_lines(line)

```

Figura 15: Separação da imagem contendo texto de nome, atributos ou método de classe em linhas.

Cada um desses recortes da imagem original, são então *upscaled* em 3 vezes, para dar mais detalhes ao modelo de OCR. Eles passam por um processo de binarização, em que todos pixels são forçados a um valor absoluto de branco ou preto, baseado em um cálculo providenciado pelo *opencv*⁹, que encontra um limiar para identificar quais pixels devem ser maximizados ao valor de 255 (brancos) ou minimizados a 0 (pretos). Esse processo cria imagens e texto com bordas mais definidas e assim facilitando acertos para identificação.

⁹ <https://github.com/opencv/opencv>

```
def detect_text_ocr(self, reader:ez.Reader):
    line_text = []
    for l in self._text_lines:
        img_to_read = self._preprocess_line(l)
        new_line = reader.recognize(img_to_read, detail=0, paragraph=True)
        if new_line:
            for text in new_line:
                line_text.append(text)
    self._text = line_text
```

Figura 16: Chama o leitor do *easyOCR* somente para ler o texto, tratando a imagem inteira como detecção.

As *strings* resultantes são reestruturadas para encaixarem com a descrição de nomes, atributos e métodos de classe em *mermaid*, produzindo próprias declarações de classe e seus componentes.

O próximo passo envolve a conexão das classes encontradas entre si, através das máscaras de relações encontradas, mais especificamente, foram identificados os dois pontos mais extremos da máscara, como suas pontas de início e fim.

```
def __find_endpoints(self) -> List[np.ndarray]:
    pts = self.__mask_points
    if len(pts) < 2:
        return (pts[0], pts[0])
    centroid = pts.mean(axis=0)
    d = np.linalg.norm(pts - centroid, axis=1)
    p1 = pts[d.argmax()]
    d2 = np.linalg.norm(pts - p1, axis=1)
    p2 = pts[d2.argmax()]
    return [p1, p2]
```

Figura 17: Encontra os dois pontos da máscara mais distantes.

A classe mais próxima de cada ponto, encontrada por comparar a distância do ponto a sua caixa de detecção, ignorando uma classe se já foi associada ao outro, foi conectada aquela relação, mas ainda é necessário informação de losangos e setas para identificar a direção e aproximar o tipo da relação. Para associar esses elementos com a relação, as suas máscaras são comparadas para

identificar sobreposição entre elas, e então é medida a distância da seta ou losango até uma das duas classes da relação para identificar a sua direção.

```
def __attach_arrows(self, all_arrows:List[SetasUML]):
    for arr in all_arrows:
        overlap = cv2.bitwise_and(self.__b_mask, arr.b_mask)
        overlap_n = cv2.countNonZero(overlap)
        if overlap_n > 0:
            self.__connected_arrows.append(arr)
            x, y, _ = arr.coordinates
            pts = np.array([x, y])
            dis_a = self.__point_to_bbox_distance(pts, self.__class_a)
            dis_b = self.__point_to_bbox_distance(pts, self.__class_b)
            if dis_a <= dis_b:
                if self.__edgepiece_a is None:
                    self.__edgepiece_a = arr
                elif self.__edgepiece_b is None:
                    self.__edgepiece_b = arr
            else:
                if self.__edgepiece_b is None:
                    self.__edgepiece_b = arr
                elif self.__edgepiece_a is None:
                    self.__edgepiece_a = arr
            if len(self.__connected_arrows) >= 2:
                break
```

Figura 18: Conexão de setas a relações, losangos tem uma função similar.

Com as classes e direção da relação encontradas, é escrita a string representando a relação em *mermaid* para ser adicionada a saída, ao fim gerando um documento contendo uma descrição do diagrama de classes representado na imagem dada como entrada.

4.4 CONVERSÃO DE CÓDIGO PYTHON EM LINGUAGEM MERMAID

Apesar de mais simples, a capacidade de converter código para o formato *mermaid* também foi fundamental. Este módulo recebe como entrada uma pasta contendo um ou mais arquivos de código python, os quais têm seu conteúdo extraído como *strings* para serem processados, com o primeiro passo sendo identificar e separar todo código que é parte de uma declaração de classe.

```
def find_classes(code):
    classes = []
    #Separa as strings originais do código em uma nova lista de strings, cada\
    # str sendo uma classe, seus atributos e métodos, ignorando código que não\
    # declara classes
    pattern = r'(?m)^(class\s+[A-Z]\w*(?:\([^\)]*\))?:\n(?:[:\s\t].*\n|^[ \t\r\f\v]*\n)*'
    for i in range(len(code)):
        x = re.findall(pattern, code[i])
        for i in range(len(x)):
            classes.append(x[i])
    return classes
```

Figura 19: Uso de uma expressão regular para encontrar declarações de classe.

Na Figura 19 acima é representado o primeiro uso também de expressões regulares, ou *Regex*, em que um padrão de texto é criado, e aplicada a função com o padrão criado em uma *string*, que retorna qualquer *substring* que está de acordo com o padrão. Neste caso, o padrão usado descreve o texto iniciado pela palavra “class” que, em python, inicia a definição de uma nova classe, seguida por qualquer palavra que é iniciada com uma letra maiúscula. Opcionalmente pode ser aceita a possibilidade de abertura de parênteses após o nome da classe se também houver um fecha parênteses em algum ponto a frente antes de haver quebra de linha. Após uma quebra de linha é simplesmente aceito todo texto que está na indentação dentro da classe, parando assim que texto fora dessa indentação é encontrado, indicando o fim da declaração da classe.

Cada *string* de classe é explorada de forma similar para encontrar seu nome, seus atributos, métodos e verificando se é uma classe abstrata, o que é rapidamente transformado em uma descrição *mermaid* da classe e seus componentes. São observadas tanto as declarações de métodos quanto o corpo dos métodos de classe.

Isto é seguido da busca por identificação de relações entre classes, mas devido a python não demonstrar de forma explícita somente por sua estrutura todo

tipo de relação entre classes, somente 3 foram cobertas como garantidas a ser identificadas pelo método utilizado: associações, composições e herança

```
def check_association(method_param, method_output, name_origin):
    #Procura por variáveis locais entre os parâmetros do método que pertencem a\
    # outra classe, ou por saídas de métodos que pertencem a outra classe
    param_as = r'\w+\s*:\s*([A-Z]\w*)|.*(?:\s*->\s*([A-Z]\w*))'
    associations_found = []
    #Se o tipo da saída não foi declarado, é necessário trocar a variável local\
    # por uma str vazia para evitar erro na busca RegEx
    if not method_output:
        method_output = ""
    for cls_parameter in re.finditer(param_as, method_param + method_output):
        name_target_as = cls_parameter.group(1) or cls_parameter.group(2)
        #Formatação pronta para o texto do arquivo mermaid
        as_code = f"    {name_origin} --> {name_target_as}"
        associations_found.append(as_code)
    #Lista retornada pode conter associações repetidas que serão eliminadas\
    # depois
    return associations_found

def check_composition(method_body, name_origin):
    #Procura por classes construídas dentro do corpo do método dado e\
    # atribuídas a ou adicionadas a lista ou outra estrutura similar atribuída\
    # a um atributo da classe
    parameter_comp = r'(?m)self\._*\w+(?:\s*-\s*([A-Z]\w*)\s*\.\s*)|(?:\s*\.\s*)(?:\s*-\s*)([A-Z]\w*)\s*\.\s*(?:\s*\.\s*)?'
    compositions_found = []
    for att_assignment in re.finditer(parameter_comp, method_body):
        name_target_comp = att_assignment.group(1) or att_assignment.group(2)
        #Formatação pronta para o texto do arquivo mermaid
        comp_code = f"    {name_origin} *-- {name_target_comp}"
        compositions_found.append(comp_code)
    #Lista retornada pode conter composições repetidas que serão eliminadas\
    # depois
    return compositions_found
```

Figura 20: Verificação de associação e de composição.

```

def extract_class_values(class_code):
    #RegEx usado para identificar o nome da classe atualmente em processamento,\
    # e a sua superclasse se existir
    parameter = r'(?m)^class\s+([A-Z]\w*)(?:\s+(\w+))?'
    class_string = re.search(parameter, class_code)
    class_name = class_string.group(1)
    class_inh = class_string.group(2)
    #Adição de relações entre classes, especificamente, associações e\
    # composições na primeira função, e herança na segunda
    class_block += ( add_associations(class_code)
                    + add_inheritance(class_name, class_inh) + '\n' )
def add_inheritance(name, inheritance):
    if inheritance:
        inheritance_list = inheritance.split(',')
        output_list = []
        for i in range(len(inheritance_list)):
            if inheritance_list[i].strip() != 'ABC':
                inher_code = f"    {name} --|> {inheritance_list[i].strip()}"
                output_list.append(inher_code)
        if output_list:
            output = '\n'.join(output_list) + '\n'
        else:
            output = ''
        return output
    else:
        return ''

```

Figura 21: Verificação de herança.

Com cada padrão identificado para cada classe, os resultados são combinados formando uma descrição de um diagrama de classe equivalente ao que seria representado pelo código.

4.5 CORREÇÃO COM *STRING MATCHING*

Após adquirir dois conjuntos de texto em mermaid, e estabelecer um como o gabarito e o outro como resposta, o programa faz uso de *rapidfuzz* para encontrar pedaços de texto entre os dois que são iguais ou similares, e em casos de *strings* similares, baseado na pontuação da diferença e no elemento representado por aquele trecho de texto, identifica se é um erro genuíno ou somente uma formatação diferente do mesmo conceito ou pequenas falhas resultantes do processo de OCR.

O texto das duas entradas é transformado em duas listas, com cada item contendo uma linha do arquivo original, para cada linha do arquivo de resposta, é

procurada a linha mais similar no arquivo de gabarito, se a pontuação for 100, ou seja uma equivalência exata, a linha é considerada um acerto e o *loop* continua para a próxima.

Se as duas linhas não forem idênticas, ambas são quebradas em listas de palavras individuais, a linha da resposta é usada como base para identificar que elemento de classe está sendo representado por essa *string*. Se for uma relação, a linha da resposta é invertida antes de testar a similaridade de novo, para cobrir casos em que ambas descrevem a mesma relação mas trocando a ordem em que as classes são nomeadas, se após a inversão a pontuação aumentar, a nova versão é inserida no lugar da original.

<pre>classDiagram class Classe1 class Classe2 Classe1 --> Classe2</pre>	<pre>classDiagram class Classe1 class Classe2 Classe2 <-- Classe1</pre>
--	--

Figura 22: Ambos exemplos de mermaid descrevem o mesmo diagrama

Em casos que o elemento não é uma relação, ou simplesmente inversão não solucionar a pontuação, uma checagem ocorre em que uma pontuação de 95 é considerada similar o suficiente, com a diferença provavelmente sendo causa de letras trocadas durante OCR ou um erro gramático pequeno por parte do criador de um dos documentos *mermaid*, senão, uma última checagem é feita dessa vez olhando se a pontuação é 90 ou acima, ainda pode ser considerada correta se ambas linhas tiverem a mesma quantia de caracteres, novamente erros nesse caso podendo ser considerados pequenas discrepâncias em OCR, porém caracteres faltantes ou em excesso nessa pontuação podem indicar erros mais notáveis, assim como no caso anterior, se algum dos casos anteriores retornar como verdadeiro, a linha original é substituída, mas dessa vez pela linha do gabarito, para marcá-la como correta. Os valores foram escolhidos de forma parcialmente arbitrária, com

base em testes feitos durante o desenvolvimento demonstrarem que são valores de corte aparentemente confiáveis.

Após essas comparações e substituições, a pontuação de cada linha é testada novamente, qualquer caso remanescente com pontuação abaixo de 100 sendo anotado para um relatório, demonstrando a linha correta no gabarito e a linha original na resposta, junto da pontuação de similaridade entre as duas, por fim adicionando uma pontuação média do documento como um todo calculada durante esse processo.

```
- Person --> Address // line: 26  
+ ~ 6<Person --> |Address7 // line: 29  
Similarity: 85.71428571428572
```

Figura 23: Exemplo de uma seção do relatório apontando um erro encontrado.

4.6 EXEMPLO COMPLETO

Nesta seção será descrito um exemplo do programa em funcionamento, recebendo como gabarito o código da Figura 24 e como “resposta do aluno” o diagrama da Figura 25. O exemplo foi executado na máquina virtual do windows 10(WSL2) executando o sistema operacional ubuntu 22.04.5, fazendo uso de uma CPU intel core i7 de 12ª geração e sem acesso a uma GPU.

```

1  class ClassB:
2      def __init__(self, name: str, model: int):
3          self.__name = name
4          self.__model = model
5
6      def get_name(self) -> str:
7          return self.__name
8
9      def get_model(self) -> int:
10         return self.__model
11
12  class ClassA:
13      def __init__(self, name: str, id: int, obj_b: ClassB):
14          self.__name = name
15          self.__id = id
16          self.__obj_b = obj_b
17
18      def get_name(self) -> str:
19          return self.__name
20
21      def get_id(self) -> int:
22          return self.__id
23
24      def get_obj_b(self) -> ClassB:
25          return self.__obj_b
26
27      def act(self):
28          print(self.__name)
29          print(self.__id)
30
31  class SubclassB(ClassB):
32      def __init__(self, name: str, model: int, serial_num: int):
33          super().__init__(name, model)
34          self.__serial_num = serial_num
35
36      def get_snum(self) -> int:
37          return self.__serial_num
38
39  class ClassC:
40      def __init__(self, value: int, name: str, model: int, serial_num: int):
41          self.__value = value
42          self.__obj_sb = SubclassB(name, model, serial_num)
43
44      def get_value(self) -> int:
45          return self.__value
46
47      def set_value(self, n: int):
48          self.__value = n

```

Figura 24: Código para exemplo

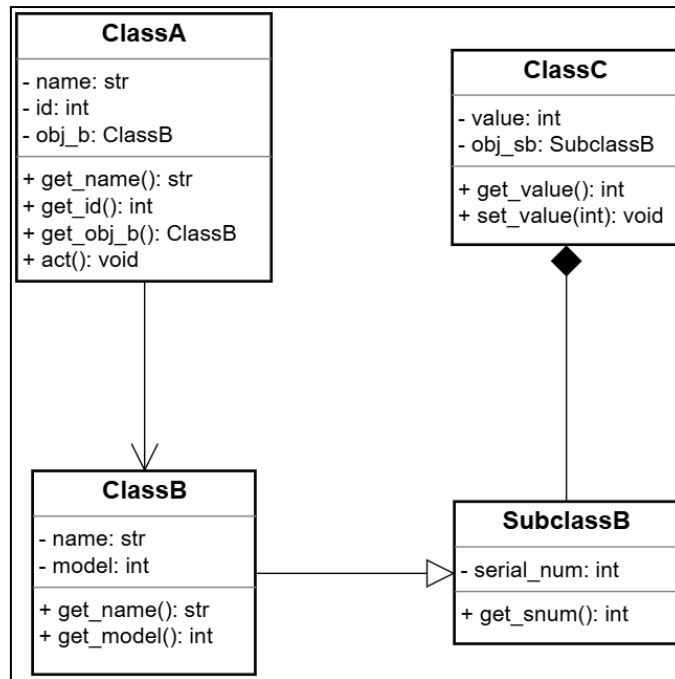


Figura 25: Diagrama para exemplo

Após iniciada a execução, o gabarito de código é convertido primeiro, sendo dividido em blocos de texto, cada um contendo uma classe, os quais são analisados individualmente. Ao fim da conversão, é dada a saída a seguir em formato *mermaid*:

```

1  classDiagram
2      class ClassB
3      ClassB : -str name
4      ClassB : -int model
5      ClassB : +get_name() str
6      ClassB : +get_model() int
7
8      class ClassA
9      ClassA : -str name
10     ClassA : -int id
11     ClassA : -ClassB obj_b
12     ClassA : +get_name() str
13     ClassA : +get_id() int
14     ClassA : +get_obj_b() ClassB
15     ClassA : +act()
16     ClassA --> ClassB
17
18     class SubclassB
19     SubclassB : -int serial_num
20     SubclassB : +get_snum() int
21     SubclassB --|> ClassB
22
23     class ClassC
24     ClassC : -int value
25     ClassC : -SubclassB obj_sb
26     ClassC : +get_value() int
27     ClassC : +set_value(int n)
28     ClassC *-- SubclassB

```

Figura 26: Saída da conversão de código para *mermaid* no exemplo

Após o término dessa conversão, a imagem dada como resposta segue para leitura pelo modelo de IA e subsequente processo de reconhecimento de caracteres, resultando nas seguintes detecções de imagem e texto *mermaid*:

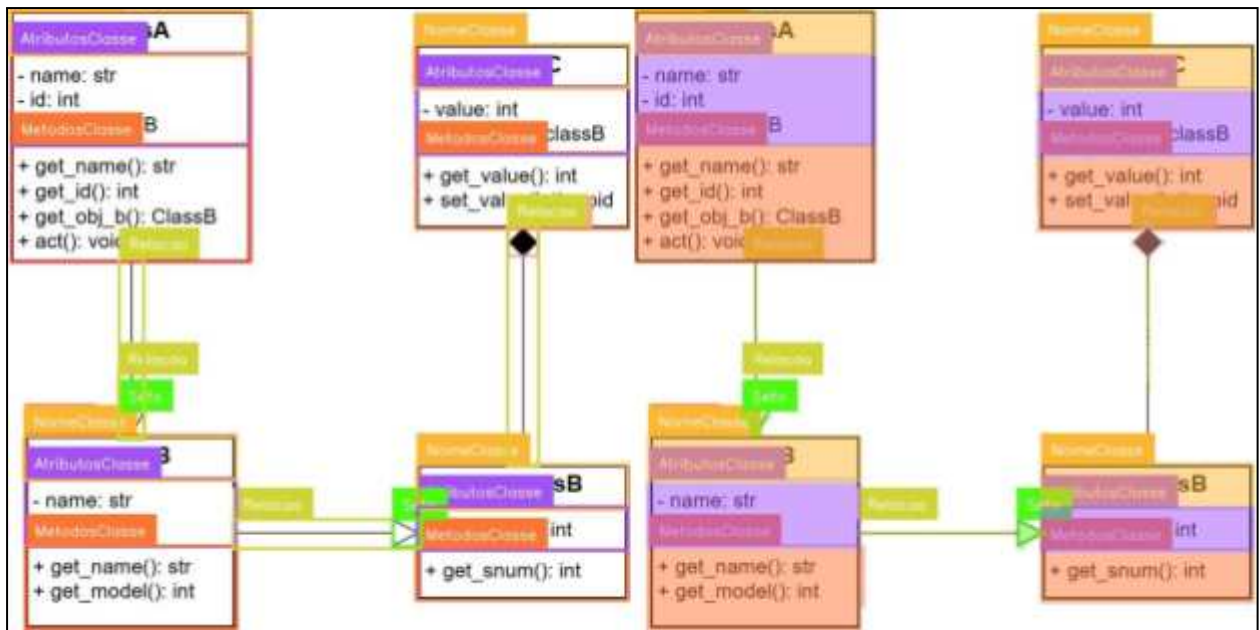


Figura 27: A esquerda, as caixas de detecção sobre o diagrama de classe observado, a direita as máscaras de detecção do mesmo.

```

1 classDiagram
2     class Classc
3     Classc : -int value
4     Classc : -SubclassB obj_sb
5     Classc : +get_value() int
6     Classc : +set_value(int) void
7
8     class ClassB
9     ClassB : -str name
10    ClassB : -int model
11    ClassB : +get_name() str
12    ClassB : +get_model() int
13
14    class ClassA
15    ClassA : -str name
16    ClassA : -int id
17    ClassA : -ClassB obj_b
18    ClassA : +get_name() str
19    ClassA : +get_id() int
20    ClassA : +get_obj_b() ClassB
21    ClassA : +act() void
22
23    class SubclassB
24    SubclassB : -int serial num
25    SubclassB : +=
26    SubclassB : +get_snum() int
27
28    ClassB --> SubclassB
29    SubclassB --* Classc
30    ClassA --> ClassB

```

Figura 28: Saída da conversão de imagem para *mermaid* no exemplo

Apesar das duas entradas serem equivalentes em termos da estrutura do diagrama de classe que as representa, as saídas de cada foram diferentes, isso se

dá a pequenos erros na detecção de texto em imagem, mas também algumas limitações que serão exploradas mais a fundo na próxima seção.

Após ambas entradas serem ajustadas a um formato comum, elas são propriamente comparadas, gerando um relatório ressaltando as diferenças entre si e dando uma pontuação média de precisão. A ordem em que as classes foram declaradas em cada saída *mermaid* não tem impacto nesse processo já que o pareamento de *strings* busca comparar cada linha do gabarito com a linha mais similar na resposta.

O relatório final se encontra a seguir:

```
1  - ClassB : -str name // line: 3
2  + Classc : +set_value(int) void // line: 6
3  Similarity: 85.5
4  - SubclassB : -int serial_num // line: 17
5  + SubclassB : -int serial num // line: 21
6  Similarity: 94.54545454545455
7  - SubclassB : +get_snum() int // line: 18
8  + SubclassB : += // line: 22
9  Similarity: 86.66666666666667
10 - SubclassB --|> ClassB // line: 19
11 + SubclassB <-- ClassB // line: 24
12 Similarity: 92.6829268292683
13 - SubclassB --|> ClassB // line: 19
14 + SubclassB --* Classc // line: 25
15 Similarity: 87.8048780487805
16 Average Score: 97.96922792654499
```

Figura 29: Relatório do exemplo

Linhas iniciadas com “-” representam a linha do gabarito estudada, são seguidas por uma linha iniciada com “+” representando a linha da resposta com maior similaridade encontrada, e abaixo dela é dada a pontuação de similaridade entre as duas. No fim do relatório é dada uma pontuação média da precisão da resposta como um todo.

4.7 DISCUSSÃO

Em geral, os resultados do programa desenvolvido foram bem sucedidos, porém insuficientes para fazer uso do mesmo em produção, devido a isso, não ocorreu a integração do sistema ao moodle como originalmente planejado.

O modelo de IA produzido apresentou alta precisão na parte de visão computacional, com AP de classes de objeto mais baixo encontrado quando calculados no split de validação, foi 91% com a detecção de relações, e testado no split de teste, o AP mais baixo foi 82% com a classe de setas.

A performance relativamente mais baixa em detectar relações provavelmente se dá por esses objetos serem os mais variáveis em forma e, apesar de poderem ocupar uma grande área da imagem, são compostos em maior parte por uma linha fina de pixels que pode ser difícil de identificar consistentemente, especialmente se elementos como setas ou losangos que podem servir como marcadores de início e fim de uma relação, não estiverem presentes.

Sobre a menor performance com setas quando aplicado em imagens do conjunto de teste, pode ser explicada pelo tamanho do dataset ainda ser pequeno comparado a outros projetos similares a este, em conjunto com a variação maior em formas de serem desenhadas, com diferentes diagramas representando setas de formas diferentes, além da variação já prevista pelas regras de UML. Isso torna a classe de setas particularmente mais propensa a sofrer por ter menos exemplos no dataset que outras classes com menos variação.

Apesar dos resultados satisfatórios, ainda há limitações e aspectos do modelo e do módulo construído ao seu redor que podem ser melhorados, o potencial maior deles sendo que atualmente não é possível através da detecção de objetos somente, identificar o tipo da relação entre classes com exatidão, diagramas

de classe possuem diversas variações de elementos que foram truncados em etiquetas de objeto comuns durante o treinamento, impedindo que, por exemplo uma associação possa ser diferenciada de uma dependência, já que a diferença entre os dois é que associação faz uso de uma linha contínua e dependência uma linha pontilhada, ou a diferença entre agregação e composição, ambos têm um losango marcando seu ponto de origem, mas em uma agregação é um losango branco e em composição é um losango preto. A solução mais direta para essa questão é a expansão do dataset para que haja exemplos suficientes de cada variação de cada elemento de diagrama que atualmente está combinado, possa ser separado em uma classe diferente. Alternativamente poderia ser possível implementar alguma verificação analisando o recorte do elemento detectado buscando por aspectos que indicariam qual variação de cada objeto foi encontrada em nível de código, porém isso está fora do escopo do trabalho atual.

Imagens com certas propriedades também podem esbarrar em limitações do modelo, como em caso de resolução, entradas para o modelo precisam estar em formato 640 x 640, então qualquer imagem que não se enquadra precisa ter seu tamanho e resolução alterados antes de ser usada, em particular causando problemas para imagens muito maiores que a resolução esperada e conteúdo esparso, testes informais demonstraram que entradas que precisam ser reduzidas de uma alta resolução e contém relações entre classes distantes uma da outra, ou um alto número de elementos a detectar, têm uma tendência a detecções significativamente piores que o esperado com base nas métricas calculadas do modelo, em especial frequentemente falhando em encontrar relações, muito provavelmente devido a perda de informação durante downscaling com esses objetos já sendo pequenos em comparação ao restante. Há uma solução teórica

para o problema que não foi implementada em prática devido ao tempo disponível ser insuficiente, que seria o uso de tiling para alimentar pedaços da imagem ao modelo que se enquadram no tamanho esperado sem nenhuma forma de modificação em resolução, para depois combinar os resultados, se implementado poderia aumentar a performance do modelo sem necessidade de modificações no dataset ou repetição do treinamento, porém a tarefa foi considerada complexa demais para executar em tempo hábil para a entrega deste trabalho.

Outras possíveis entradas problemáticas são imagens com fundo não branco para seções de texto, tornando o processo de binarização para facilitar OCR ocasionalmente levar ao fundo ser forçado a cor preta junto com o texto, deixando-o invisível e diagramas contendo relações que se cruzam uma por cima da outra ou se combinam em um elemento só, podendo dificultar a detecção desses objetos devido a sua sobreposição.

Para o aspecto de conversão do código para um diagrama, a maior limitação é o método utilizado, apesar de expressões Regex serem um método forte de identificação de padrões em texto, só se pode identificar alguns aspectos de um diagrama de classe através da estrutura de código, sem entendimento próprio do que as classes representam e como foram projetadas para interagir, necessitando intervenção humana para discernir mais detalhes.

Por fim, a comparação das duas representações de diagrama em mermaid foi testada de forma informal, com casos de erros fabricados ou exemplos específicos de erros encontrados durante a conversão de imagens para diagrama que foram considerados pequenos o suficiente para serem aceitos como corretos, uma pesquisa mais ampla sobre quais casos de erro professores e alunos consideram

aceitáveis e quais deveriam ser marcados, poderia dar uma pontuação mais clara como parâmetro para aceitação ou considerar um erro.

5 CONCLUSÃO

Este trabalho apresenta uma ferramenta para avaliação de códigos-fonte com base em imagens de diagramas de classes. A ferramenta desenvolvida é capaz de parcialmente converter imagens de diagramas de classe UML e arquivos de código python em representações textuais de diagramas equivalentes em formato mermaid, além de compará-los, servindo como uma prova de conceito e base para a correção automatizada de exercícios envolvendo como resposta um desses formatos de entrada, a qual pode ser expandida no futuro para ser útil em ambientes de produção para professores. As métricas identificadas demonstram uma confiável capacidade em detectar e classificar elementos em uma imagem de diagrama de classe, junto com uma correta transformação em texto das classes encontradas, seus atributos e métodos, além de suas relações e a direção dessas relações.

Trabalhos futuros possivelmente podem expandir as capacidades do modelo para identificar também mais especificamente o tipo de relação encontrada, diferenciar elementos que atualmente são agrupados em uma categoria só, e expandir a funcionalidade da conversão de código para mermaid com um método diferente, capaz de identificar além de padrões de texto, a estrutura provável baseada em um melhor entendimento de como classes se conectam, provavelmente fazendo uso de um segundo modelo de IA.

REFERÊNCIAS

- CHEN, Fangwei et al. Automatically recognizing the semantic elements from UML class diagram images. *Journal of Systems and Software*, v. 193, p. 111431, 2022. ISSN 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2022.111431>. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121222001340>.
- [S.I.].DATASET of the Paper "Automatically Classifying UML Class Diagrams from Images using Deep Learning". [S.I.]: Zenodo, fev. 2021. DOI: 10.5281/zenodo.4549406. Disponível em: <https://doi.org/10.5281/zenodo.4549406>.
- GOSALA, Bethany et al. Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network. *Applied Sciences*, v. 11, n. 9, 2021. ISSN 2076-3417. DOI: 10.3390/app11094267. Disponível em: <https://www.mdpi.com/2076-3417/11/9/4267>.
- IVAR JACOBSON E GRADY BOOCH, James Rumbaugh e. The unified modeling language reference manual. 1. ed. [S.I.]: Addison-Wesley Longman, 2000.
- KARASNEH, B.; CHAUDRON, Michel. Online Img2UML Repository: An Online Repository for UML Models. In:
- KARASNEH, Bilal; CHAUDRON, Michel R.V. Extracting UML models from images. In: 2013 5th International Conference on Computer Science and Information Technology. [S.I.: s.n.], 2013. P. 169–178. DOI: 10.1109/CSIT.2013.6588776.
- Img2UML: A System for Extracting UML Models from Images. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. [S.I.: s.n.], 2013. P. 134–137. DOI: 10.1109/SEAA.2013.45.

MAHESH, Batta. Machine learning algorithms-a review. International Journal of Science and Research (IJSR).[Internet], v. 9, n. 1, p. 381–386, 2020.

MERMAID. Disponível em: <https://mermaid.js.org>. Acessado em Junho de 2026.

OMG. Unified Modeling Language. [S.l.], 2017.

RAŽINSKAS, Mantas et al. Transforming Sketches of UML Use Case Diagrams to Models. IEEE Access, v. 12, p. 185826–185837, 2024. DOI: 10.1109/ACCESS.2024.3514455.

SAMOILI, Sofia et al. AI Watch. Defining Artificial Intelligence 2.0. Towards an operational definition and taxonomy of AI for the AI landscape. JRC Research Reports, Joint Research Centre (Seville site), JRC126426, 2021.

SHOEMAKER, Martin L. UML Applied: A .NET Perspective. 1. ed. [S.l.]: Springer Link, 2004.

SILVA, Ricardo Pereira e. UML 2 em Modelagem Orientada a Objetos. 2. Ed. Florianópolis: Ricardo Pereira e Silva, 2019.

THE Unified Modeling Language. Disponível em: <https://www.uml-diagrams.org>. (acessado: 02/11/2024).

LIN, Tsung-Yi et al. Microsoft COCO: Common Objects in Context. [S.l.: s.n.], 2015. arXiv: 1405.0312 [cs.CV].

LIPTON, Zachary Chase; ELKAN, Charles; NARAYANASWAMY, Balakrishnan. Thresholding classifiers to maximize F1 score. arXiv preprint arXiv:1402.1892, 2014.

HENDERSON, Paul; FERRARI, Vittorio. End-to-end training of object class detectors for mean average precision. In: Conference paper apresentado em Asian conference on computer visionAsian conference on computer vision. 2016.

WANG, Beinan. A parallel implementation of computing mean average precision. arXiv preprint arXiv:2206.09504, 2022.

LAM, Kevin Fu Yuan; GOPAL, Vikneswaran; QIAN, Jiang. Confidence intervals for the F1 score: a comparison of four methods. arXiv preprint arXiv:2309.14621, 2023.

NAJAFI, Fazil T. et al. AI-Assisted Grading—A Study on Efficiency and Fairness. In: Conference paper apresentado em 2025 ASEE Southeast Conference2025 ASEE Southeast Conference. 2025.

APÊNDICE A

Repositório do código fonte para o programa desenvolvido:

<https://codigos.ufsc.br/gqs/image2mermaid>

APÊNDICE B

Página pública do Roboflow contendo o *dataset* criado:

<https://universe.roboflow.com/image2mermaid/img2mermaid>

APÊNDICE C

Aplicativo de comparação automática entre código fonte e diagrama de classes

Gabriel Martins Bergmann¹

¹Departamento de Informática e Estatística (INE) Universidade Federal de Santa Catarina (UFSC) Florianópolis SC Brasil

gabriel.bergmann@grad.ufsc.br

Abstract. *In the field of software engineering, the creation of UML class diagrams based on snippets of code and the development of code based on class diagrams are frequently applied as graded activities, automating the grading process of these tasks would be extremely useful for saving the time of teachers that would be involved in doing so manually, but it can't be easily implemented. With that in mind, this article was produced with the goal of using computer vision models and identification of patterns in python code, to allow for partial comparison and grading of activities involving UML class diagrams.*

Resumo. *No campo de engenharia de software, atividades avaliativas frequentemente aplicadas são a criação de diagramas de classe UML baseadas em trechos de códigos e o desenvolvimento de código baseado em diagramas de classe, automatizar a correção dessas tarefas seria de extrema utilidade para reduzir o tempo de professores investido nesse processo, mas não é facilmente implementado. Com esse problema em mente, o trabalho atual foi produzido visando usar modelos de visão computacional e identificação de padrões em código python, para permitir, ao menos de forma parcial, a comparação e correção de atividades envolvendo diagramas de classe UML.*

1. Introdução

A correção de atividades avaliativas nos cursos da área de computação toma uma quantia significativa de tempo de trabalho de professores fora de sala de aula. A automatização desse processo de correção pode ser de grande utilidade nesse sentido, mas, de forma geral, só é facilmente aplicada a atividades executadas em computadores com questões de múltipla escolha ou resultados objetivos pré-determinados com menor ambiguidade na resposta. Com isso em mente, o uso de Inteligência Artificial (IA) pode apoiar a correção de atividades dissertativas ou que requerem a entrega de mídias que podem variar de forma significativa.

Um caso específico trata de atividades de disciplinas relacionadas à Engenharia de Software. Especialmente atividades que envolvem a modelagem de soluções computacionais, com a elaboração de diagramas UML, apresentam dificuldades adicionais para automatização, mesmo que parcial, da sua avaliação. Nesse sentido, os diagramas de classes são muito utilizados para representar uma visão estática do software, pois modelam conceitos do domínio da aplicação (JACOBSON & BOOCH, 2000). Avaliação de atividades

relacionadas à elaboração de diagramas de classes tipicamente envolve primeiramente uma avaliação da sintaxe UML utilizada, bem como a comparação entre o modelo desenvolvido e a solução de software implementada.

Assim, este trabalho propõe o uso de inteligência artificial para automatizar a avaliação de trabalhos envolvendo a comparação de código fonte com diagramas de classes, importante para disciplinas da área de Engenharia de Software em cursos de graduação da área da Computação.

2. Estado Da Arte

Neste capítulo será discutida uma pesquisa do estado da arte relacionada a geração e análise de diagramas de classe a partir de imagens, junto a metodologia para essa pesquisa. Devido às limitações de um trabalho de conclusão de curso, não é realizado uma revisão sistemática de literatura, mas uma um mapeamento ad hoc de literatura, guiado por um protocolo simples mercado de busca.

A pesquisa foi feita através da base de pesquisa Google Scholar, com um filtro para apenas mostrar resultados a partir de 2010, usando as seguintes strings de busca: “image recognition on uml class diagram” e “uml class diagram generation from images”. Essas pesquisas foram feitas nos meses de Novembro e Dezembro de 2024, para cada string de busca foram exploradas as 10 primeiras páginas de resultados buscando artigos a serem incluídos nesse estudo.

Para cada string de busca, foram avaliados 100 resultados, porém havendo artigos presentes em mais de uma pesquisa, 6 foram selecionados como relevantes, todos os quais buscavam identificar elementos UML em imagens, porém focando em métodos e diagramas diferentes.

Três dos artigos estudados foram desenvolvidos pelos mesmos autores e se referem ao desenvolvimento de um programa chamado Img2UML(KARASNEH, Bilal; CHAUDRON, M. R., 2013a)(KARASNEH, Bilal; CHAUDRON, M. R., 2013b)(KARASNEH, B.; CHAUDRON, M., 2013), o exemplo mais antigo explorado de converter imagens de diagramas UML em um formato legível por um computador, nesse caso XMI, mas não faz uso de técnicas de inteligência artificial. O trabalho “Automatically recognizing the semantic elements from UML class diagram images”(CHEN et al., 2022) produz um método similar, porém mais avançado, que também não utiliza IA de visão computacional.

O artigo "Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network" (GOSALA et al., 2021) busca identificar se uma imagem contém um diagrama de classe ou não com alta acurácia fazendo uso de técnicas de *Deep Learning*, mas não busca identificar elementos individuais do diagrama ou como eles se relacionam.

A pesquisa encontrada que mais se assemelha à proposta deste artigo é “Transforming Sketches of UML Use Case Diagrams to Models”(RAŽINSKAS et al., 2024) em que foi desenvolvido um programa capaz de identificar elementos de um diagrama de casos de uso em rascunhos através de técnicas de IA, fazendo uso de *Faster R-CNN*.

Os resultados dessa pesquisa demonstram alguns padrões úteis para o desenvolvimento deste trabalho, apesar da variação em metodologia, há uma sequência de passos seguida por todos os artigos estudados para o desenvolvimento de suas soluções: identificar as formas da classe, subdividir essa forma em até 3 partes para considerar as áreas usadas para atributos e métodos, identificar texto, identificar linhas e identificar relações.

Apenas dois trabalhos oferecem soluções com modelos de machine learning, e só um deles é dedicado a identificação de diagramas de classe, limitando as fontes que podem ser usadas para o desenvolvimento e treinamento de uma solução com IA, porém essas soluções tinham alguns dos melhores resultados entre os trabalhos estudados, levando a conclusão de que usar uma rede neural seria o método mais efetivo.

3. Ferramenta De Avaliação De Códigos-Fonte Com Base Em Diagrama De Classes

O objetivo deste projeto é a construção de um programa capaz de converter código *python* e imagens de diagramas UML em um formato de texto que permita comparação entre ambos para correção automatizada. Para isso foi escolhido o formato de markdown usado por *mermaid*(MERMAID, 2026), nome o qual se refere a uma ferramenta web de construir diagramas e a linguagem utilizada por ela. Segue uma tabela demonstrando a sintaxe de mermaid relevante para descrição de diagramas de classe.

Tabela 1. Sintaxe mermaid para diagramas de classe.

Elemento	Sintaxe mermaid
Classe	<code>class BankAccount{ }</code>
Interface	<code>bar ()-- foo</code>
Característica	<code>class BankAccount{ +String owner +BigDecimal balance +deposit(amount) bool +withdrawal(amount) int }</code>
Associação	<code>classA --> classB</code>
Generalização	<code>classA < -- classB</code>
Dependência	<code>classK <.. classL</code>

Durante a conversão de imagens de diagramas de classe em sintaxe mermaid, serão identificados os elementos do diagrama e suas posições e, através dessa informação, as relações entre classes. Para propriamente converter as imagens em texto, um processo de

OCR(optical character recognition) deve ser aplicado a elementos de texto identificados, permitindo anotação de atributos, métodos e nomes de classes.

A conversão de código para mermaid ocorre por identificação de padrões através do uso de expressões regulares, fazendo uso da gramática e sintaxe de python como uma linguagem de programação para reconhecer a estrutura e relações entre classes.

Por fim com dois arquivos contendo descrições de diagramas de classe UML em mermaid, a etapa de comparação é tratada com uso de string matching, encontrando e comparando strings similares nas duas representações de diagrama, e pontuando sua similaridade, casos em que a linha mais similar encontrada tem pontuação baixa, levam a uma indicação de onde a discrepância ocorre no gabarito e na resposta.

3.1. Criação Do Modelo De Visão Computacional

A construção do modelo de detecção visual de objetos foi iniciada com a preparação de um dataset para ser usado no processo de treinamento. Imagens de diagramas de classe produzidos digitalmente foram obtidas manualmente a partir de diversas pesquisas no Google Imagens e suplantadas por imagens disponíveis publicamente em projetos existentes no Roboflow(<https://roboflow.com>).

Das imagens obtidas, foi realizada uma filtragem manual para retirar qualquer uma com qualidade baixa demais para seu texto ser legível, resultando no fim em um banco com 102 imagens no total consideradas apropriadas para uso.

Foram selecionados como objetos a identificar com labels o mínimo necessário para identificar a presença de classes, suas seções de texto internas(nome, atributos, métodos), relações entre classes e a direção dessas relações. Esses objetos são: Classe, Nome de Classe, Atributos de Classe, Métodos de Classe, Relação entre Classes, Seta e Losango.

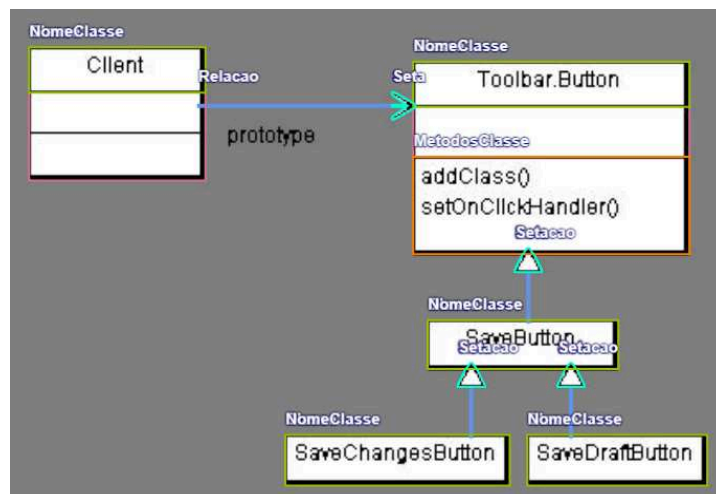


Figura 1. Exemplo de imagem do dataset após anotações.

O processo de anotação foi feito através do anotador do Roboflow, e com ajuda de 4 voluntários. As etiquetas foram marcadas com caixas simples onde possível, mas fazendo uso de polígonos mais complexos para seguir a forma de elementos mais complexos. Essa atenção extra se deve a necessidade de o modelo final ser capaz de segmentação além de

detecção simples, produzindo máscaras das formas encontradas além da posição geral do objeto para propriamente identificar relações de classe.

O dataset foi dividido aleatoriamente em 3 subconjuntos, com 72 imagens sendo reservadas para treino, 20 para validação e 10 para teste. Para aprimorar o treinamento do modelo foram aplicados alguns passos de pré processamento, transformando as imagens em tons de cinza, ajustando seu contraste e forçando uma resolução de 640x640.

O conjunto de treino também foi expandido através de transformações como espelhagem vertical ou horizontal, rotações e adição de pixels de interferência ou noise. Ao fim desse processo o conjunto de treino foi expandido para 216 imagens, totalizando em um *dataset* de 246 imagens.

Foi feito um processo de fine-tuning em um modelo já existente e com apoio de um dataset maior. O dataset oferecido pelos servidores do *Roboflow* correspondente à mesma tarefa que o modelo a ser treinado é o COCO-seg, uma extensão de COCO(Common Objects in Context) adaptada para treinamento de modelos de segmentação com anotações mais detalhadas, mas mantendo as mesmas 330.000 imagens do original (LIN et al., 2015). O modelo usado para treinamento foi *YOLOv11*, sendo a versão de YOLO mais recente disponível no *Roboflow*.

O modelo final produzido pontuou 96,4% em precisão, 92,5% em recall, 97% em mPA@50 e 94,4% em F1-score quando testados sobre o conjunto de validação do dataset, indicando boa performance.

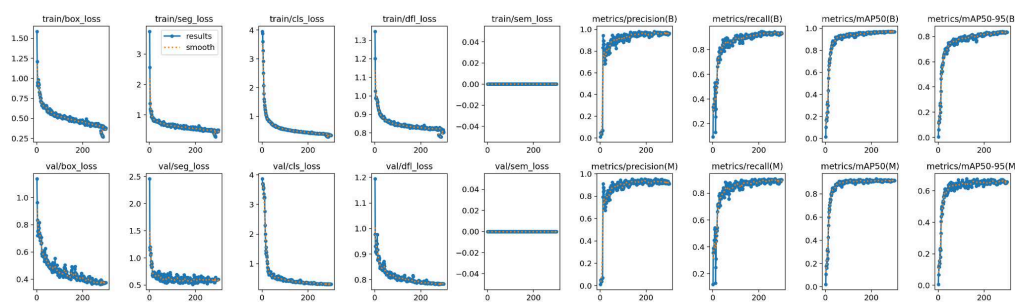


Figura 2: Gráficos de diversas métricas analisadas durante treinamento.

3.2. Desenvolvimento

O processo de desenvolvimento da ferramenta foi dividido em três partes, a conversão de imagem para *mermaid*, a conversão de código para *mermaid* e a comparação das saídas dos processos anteriores.

Para a conversão de imagens de diagramas de classe, com o uso da biblioteca *ultralytics* para carregar e executar inferências com os pesos do modelo treinado, produz-se uma lista de caixas de detecção e máscaras baseadas na imagem de entrada, que foram consumidas para a criação de um objeto que representa o diagrama de classes a ser descrito.

O texto de cada classe é isolado e recortado individualmente do resto da imagem para servir de entrada para a biblioteca *easyOCR* que identifica os caracteres e produz *strings* equivalentes ao que estava escrito na imagem. Esse texto resultante é estruturado para encaixar na sintaxe de *mermaid* declarando a classe, seus atributos e métodos.

Usando as máscaras das identificando a posição das relações, é identificado os dois pontos mais extremos da máscara, os quais se conectam cada um a uma classe diferente, identificando quais estão envolvidas em cada relação, checando sobreposição com elementos como setas ou losangos, é possível também identificar a direção da relação. Esse processo resulta na descrição das relações em formato *mermaid* que combinadas com as declarações de classe, forma a saída parcial deste módulo.

Em questão da conversão de *python* para *mermaid*, recebe como entrada uma pasta contendo um ou mais arquivos de código python, os quais têm seu conteúdo extraído como strings para serem processados, com o primeiro passo sendo identificar e separar todo código que é parte de uma declaração de classe.

Cada string de classe é explorada de forma similar, usando expressões regulares, para encontrar seu nome, seus atributos, métodos e verificando se é uma classe abstrata, o que é rapidamente transformado em uma descrição *mermaid* da classe e seus componentes. São observadas tanto as declarações de métodos quanto o corpo dos métodos de classe. Isto é seguido da busca por identificação de relações entre classes, mas devido a python não demonstrar de forma explícita somente por sua estrutura todo tipo de relação entre classes, somente 3 foram cobertas como garantidas a ser identificadas pelo método utilizado: associações, composições e herança

Com cada padrão identificado para cada classe, os resultados são combinados formando uma descrição de um diagrama de classe equivalente ao que seria representado pelo código.

Após adquirir dois conjuntos de texto em *mermaid*, e estabelecer um como o gabarito e o outro como resposta, o programa faz uso de *rapidfuzz* para encontrar pedaços de texto entre os dois que são iguais ou similares, e em casos de strings similares, baseado na pontuação da diferença e no elemento representado por aquele trecho de texto, identifica se é um erro genuíno ou somente uma formatação diferente do mesmo conceito ou pequenas falhas resultantes do processo de OCR.

4. Discussão

Em geral, os resultados do programa desenvolvido foram bem sucedidos, porém insuficientes para fazer uso do mesmo em produção, devido a isso, não ocorreu a integração do sistema ao moodle como originalmente planejado.

O modelo de IA produzido apresentou alta precisão na parte de visão computacional, com AP de classes de objeto mais baixo encontrado quando calculados no split de validação, foi 91% com a detecção de relações, e testado no split de teste, o AP mais baixo foi 82% com a classe de setas.

Apesar dos resultados satisfatórios, ainda há limitações e aspectos do modelo e do módulo construído ao seu redor que podem ser melhorados, o potencial maior deles sendo que atualmente não é possível através da detecção de objetos somente, identificar o tipo da relação entre classes com exatidão, diagramas de classe possuem diversas variações de elementos que foram truncados em etiquetas de objeto comuns durante o treinamento,

impedindo que, por exemplo uma associação possa ser diferenciada de uma dependência, já que a diferença entre os dois é que associação faz uso de uma linha contínua e dependência uma linha pontilhada, ou a diferença entre agregação e composição, ambos têm um losango marcando seu ponto de origem, mas em uma agregação é um losango branco e em composição é um losango preto. A solução mais direta para essa questão é a expansão do dataset para que haja exemplos suficientes de cada variação de cada elemento de diagrama que atualmente está combinado, possa ser separado em uma classe diferente.

Alternativamente poderia ser possível implementar alguma verificação analisando o recorte do elemento detectado buscando por aspectos que indicariam qual variação de cada objeto foi encontrada em nível de código, porém isso está fora do escopo do trabalho atual.

Para o aspecto de conversão do código para um diagrama, a maior limitação é o método utilizado, apesar de expressões Regex serem um método forte de identificação de padrões em texto, só se pode identificar alguns aspectos de um diagrama de classe através da estrutura de código, sem entendimento próprio do que as classes representam e como foram projetadas para interagir, necessitando intervenção humana para discernir mais detalhes.

Por fim, a comparação das duas representações de diagrama em mermaid foi testada de forma informal, com casos de erros fabricados ou exemplos específicos de erros encontrados durante a conversão de imagens para diagrama que foram considerados pequenos o suficiente para serem aceitos como corretos, uma pesquisa mais ampla sobre quais casos de erro professores e alunos consideram aceitáveis e quais deveriam ser marcados, poderia dar uma pontuação mais clara como parâmetro para aceitação ou considerar um erro.

5. Conclusão

Este trabalho apresenta uma ferramenta para avaliação de códigos-fonte com base em imagens de diagramas de classes. A ferramenta desenvolvida é capaz de parcialmente converter imagens de diagramas de classe UML e arquivos de código python em representações textuais de diagramas equivalentes em formato mermaid, além de compará-los, servindo como uma prova de conceito e base para a correção automatizada de exercícios envolvendo como resposta um desses formatos de entrada, a qual pode ser expandida no futuro para ser útil em ambientes de produção para professores. As métricas identificadas demonstram uma confiável capacidade em detectar e classificar elementos em uma imagem de diagrama de classe, junto com uma correta transformação em texto das classes encontradas, seus atributos e métodos, além de suas relações e a direção dessas relações.

Trabalhos futuros possivelmente podem expandir as capacidades do modelo para identificar também mais especificamente o tipo de relação encontrada, diferenciar elementos que atualmente são agrupados em uma categoria só, e expandir a funcionalidade da conversão de código para mermaid com um método diferente, capaz de identificar além de padrões de texto, a estrutura provável baseada em um melhor entendimento de como classes se conectam, provavelmente fazendo uso de um segundo modelo de IA.

Referências

- CHEN, Fangwei et al. Automatically recognizing the semantic elements from UML class diagram images. *Journal of Systems and Software*, v. 193, p. 111431, 2022. ISSN 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2022.111431>. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121222001340>.
- [S.l.]. DATASET of the Paper "Automatically Classifying UML Class Diagrams from Images using Deep Learning". [S.l.]: Zenodo, fev. 2021. DOI: 10.5281/zenodo.4549406. Disponível em: <https://doi.org/10.5281/zenodo.4549406>.
- GOSALA, Bethany et al. Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network. *Applied Sciences*, v. 11, n. 9, 2021. ISSN 2076-3417. DOI: 10.3390/app11094267. Disponível em: <https://www.mdpi.com/2076-3417/11/9/4267>.
- IVAR JACOBSON E GRADY BOOCH, James Rumbaugh e. The unified modeling language reference manual. 1. ed. [S.l.]: Addison-Wesley Longman, 2000.
- KARASNEH, B.; CHAUDRON, Michel. Online Img2UML Repository: An Online Repository for UML Models. In:
- KARASNEH, Bilal; CHAUDRON, Michel R.V. Extracting UML models from images. In: 2013 5th International Conference on Computer Science and Information Technology. [S.l.: s.n.], 2013. P. 169–178. DOI: 10.1109/CSIT.2013.6588776.
- Img2UML: A System for Extracting UML Models from Images. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. [S.l.: s.n.], 2013. P. 134–137. DOI: 10.1109/SEAA.2013.45.
- MAHESH, Batta. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR)*. [Internet], v. 9, n. 1, p. 381–386, 2020.
- MERMAID. Disponível em: <https://mermaid.js.org>. Acessado em Junho de 2026.
- OMG. Unified Modeling Language. [S.l.], 2017.
- RAŽINSKAS, Mantas et al. Transforming Sketches of UML Use Case Diagrams to Models. *IEEE Access*, v. 12, p. 185826–185837, 2024. DOI: 10.1109/ACCESS.2024.3514455.
- SAMOILI, Sofia et al. AI Watch. Defining Artificial Intelligence 2.0. Towards an operational definition and taxonomy of AI for the AI landscape. JRC Research Reports, Joint Research Centre (Seville site), JRC126426, 2021.
- SHOEMAKER, Martin L. UML Applied: A .NET Perspective. 1. ed. [S.l.]: Springer Link, 2004.
- SILVA, Ricardo Pereira e. UML 2 em Modelagem Orientada a Objetos. 2. Ed. Florianópolis: Ricardo Pereira e Silva, 2019.

THE Unified Modeling Language. Disponível em: <https://www.uml-diagrams.org>. (acessado: 02/11/2024).

LIN, Tsung-Yi et al. Microsoft COCO: Common Objects in Context. [S.l.: s.n.], 2015. arXiv: 1405.0312 [cs.CV].

LIPTON, Zachary Chase; ELKAN, Charles; NARAYANASWAMY, Balakrishnan. Thresholding classifiers to maximize F1 score. arXiv preprint arXiv:1402.1892, 2014.

HENDERSON, Paul; FERRARI, Vittorio. End-to-end training of object class detectors for mean average precision. In: Conference paper apresentado em Asian conference on computer visionAsian conference on computer vision. 2016.

WANG, Beinan. A parallel implementation of computing mean average precision. arXiv preprint arXiv:2206.09504, 2022.

LAM, Kevin Fu Yuan; GOPAL, Vikneswaran; QIAN, Jiang. Confidence intervals for the F1 score: a comparison of four methods. arXiv preprint arXiv:2309.14621, 2023.

NAJAFI, Fazil T. et al. AI-Assisted Grading—A Study on Efficiency and Fairness. In: Conference paper apresentado em 2025 ASEE Southeast Conference2025 ASEE Southeast Conference. 2025.