UNIVERSIDADE FEDERAL DO PARANÁ

DIOGO PARIS KRAUT, JEAN CARLO SANCHUKI FILHO

AVOC (AUTOMATIC VALIDATION OF CODE): UMA FERRAMENTA PARA A VALIDAÇÃO AUTOMATIZADA DE CONSISTÊNCIA ENTRE DIAGRAMAS UML E CÓDIGO JAVA.

CURITIBA PR

2023

DIOGO PARIS KRAUT, JEAN CARLO SANCHUKI FILHO

AVOC (AUTOMATIC VALIDATION OF CODE): UMA FERRAMENTA PARA A VALIDAÇÃO AUTOMATIZADA DE CONSISTÊNCIA ENTRE DIAGRAMAS UML E CÓDIGO JAVA.

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Andrey Pimentel.

CURITIBA PR

2023

Gostaria de dedicar este trabalho a nossas famílias, pelo apoio durante todo o período de universidade.
A todos com qual compartilhamos risadas, descontração e conhecimento dentro e fora da universidade. Sem os mesmos, não seria possível completar todo o período de estudos.
Muito obrigado.

AGRADECIMENTOS

É com profunda gratidão que expressamos nossos sinceros agradecimentos a todas as pessoas que contribuíram para a realização deste trabalho. Em especial, gostaríamos de destacar a importância dos professores que nos acompanharam ao longo desta jornada acadêmica e que compartilharam seus conhecimentos em sala de aula. Cada aula, cada discussão e cada desafio proposto contribuíram para a nossa formação acadêmica e pessoal. Principalmente a nosso professor orientador Andrey Ricardo Pimentel. Suas sugestões e feedback enriqueceram significativamente este trabalho, tornando-o mais consistente e relevante.

Não podemos deixar de mencionar todos aqueles que, de uma forma ou de outra, contribuíram para o desenvolvimento deste estudo.

Este trabalho não teria sido possível sem o apoio dessas pessoas incríveis. Mais uma vez, expressamos nossa profunda gratidão a todos os professores e pessoas que contribuíram para o nosso crescimento acadêmico e pessoal.

RESUMO

Este trabalho aborda a validação automatizada da consistência entre código Java e diagramas UML por meio de comparação XML. O objetivo principal é desenvolver uma ferramenta que possa verificar se um código Java está em conformidade com o diagrama UML correspondente em formato XML, sendo importante para a área de UML, diagramas de classes e engenharia de software, ao oferecer uma solução para verificar a coerência entre a implementação do código Java e a representação visual do diagrama UML. O processo envolve a análise do código Java, que é parseado em uma estrutura de dados personalizada, assim como o XML do diagrama UML. Em seguida, ocorre a comparação entre essas estruturas, identificando discrepâncias e destacando as partes que estão alinhadas e as que não estão. A abordagem proposta se assemelha a um teste unitário, fornecendo uma forma automatizada de garantir a consistência entre o código e o diagrama UML. Testes foram realizados para validar a ferramenta. Os mesmos abordaram diferentes categorias de erro, como erros de associação, falta/excesso de elementos, nomenclatura, visibilidade e tipos. Seus resultados demonstram eficácia na detecção de problemas, porém algumas limitações foram identificadas em cenários específicos.

Palavras-chave: UML. Diagrama de Classes. Interpretador. Validação de Código.

ABSTRACT

This work addresses the automated validation of consistency between Java code and UML diagrams through XML comparison. The main objective is to develop a tool that can verify if Java code complies with the corresponding UML diagram in XML format. It is important for the UML field, class diagrams, and software engineering, as it offers a solution to check the coherence between the implementation of Java code and the visual representation of the UML diagram. The process involves the analysis of the Java code, which is parsed into a custom data structure, as well as the XML of the UML diagram. Then, a comparison is made between these structures, identifying discrepancies and highlighting the parts that are aligned and those that are not. The proposed approach resembles a unit test, providing an automated way to ensure consistency between the code and the UML diagram. Tests were conducted to validate the tool, covering different categories of errors such as association errors, missing/excessive elements, naming issues, visibility, and types. The results demonstrate effectiveness in detecting problems, although some limitations were identified in specific scenarios.

Keywords: UML. Class Diagram. Parser. Code Validation.

LISTA DE FIGURAS

4.1 Exemplo de diagrama de classe. 17
4.2 Exemplo de diagrama de classe. 18
4.3 Exemplo de navegabilidade. 22
4.4 Exemplo de comparação: associação. 22
4.5 Exemplo de comparação: nomenclatura. 24
4.6 Exemplo de comparação: tipo. 26

LISTA DE ACRÔNIMOS

DINF	Departamento de Informática
UFPR	Universidade Federal do Paraná
JSON	JavaScript Object Notation
ID	Identifier (identificador)
XML	eXtensible Markup Language
UML	Unified Modeling Language (Linguagem de Modelagem Unificada)
XML	Extensible Markup Language (Linguagem de Marcação Extensível)
MDSE	Model-Driven Software Engineering (Engenharia de Software Orientada a Modelos)
OCL	Object Constraint Language (Linguagem de Restrição de Objetos)
M2M	Model-to-Model (Modelo para Modelo)
M2T	Model-to-Text (Modelo para Texto)
EMF	Eclipse Modeling Framework
ATL	Atlas Transformation Language
XMI	XML Metadata Interchange

SUMÁRIO

1 INTRODUÇÃO 10
1.1 OBJETIVOS 10
2 TRABALHOS CORRELATOS. 12
2.1 UJECTOR. 12
2.2 VALIDATION AUTOMATION OF UML DIAGRAMS CREATED BY STUDENTS 12
2.3 A PRECISE APPROACH TO VALIDATING UML MODELS AND OCL CONSTRAINTS 13
3 FUNDAMENTAÇÃO TEÓRICA 14
3.1 UML 14
3.2 XML 14
3.3 ENGENHARIA DE SOFTWARE ORIENTADA A MODELOS 14
3.3.1 Metamodelos 15
3.3.2 Conversão de Modelos 15
3.4 ASTAH 15
3.5 JSON 16
3.6 TYPESCRIPT. 16
3.7 VERIFICAÇÃO E VALIDAÇÃO. 16
3.8 CONCLUSÃO 16
4 IMPLEMENTAÇÃO 17
4.1 ASTAH E CÓDIGO JAVA 18
4.2 XML 18
4.3 CONVERSÃO EM JSON. 19
4.4 METAMODELO 19
4.4.1 Classes 19
4.4.2 Atributos 20
4.4.3 Associações 20
4.4.4 Operações 21
4.4.5 Parâmetros 21
4.5 COMPARAÇÃO 21
4.5.1 Erros de Associação. 21
4.5.2 Erros de Falta/Excesso e Nomenclatura 24
4.5.3 Erros de Visibilidade 25
4.5.4 Erros de Tipo 26

5 TESTES 28
5.1 ERROS DE ASSOCIAÇÃO 28
5.2 ERROS DE FALTA/EXCESSO E NOMENCLATURA 28
5.3 ERROS DE VISIBILIDADE 28
5.4 ERROS DE TIPO 29
5.5 CONCLUSÃO 29
6 CONCLUSÃO 30
REFERÊNCIAS 31

1 INTRODUÇÃO

A validação da consistência entre diagramas UML e código é um processo fundamental no desenvolvimento de software. Garantir que o código implementado esteja em conformidade com o design previamente especificado nos diagramas UML é essencial para assegurar a qualidade e integridade do sistema. Nesse contexto, a automação desse processo de validação pode proporcionar benefícios significativos, como a redução de erros e o aumento da eficiência no desenvolvimento de software.

Além disso, a aplicação desse código pode ser estendida além do desenvolvimento de software comercialmente, para a área educacional. Professores que precisam corrigir e avaliar o código escrito por vários alunos podem se beneficiar da validação automatizada entre diagramas UML e o código produzido pelos estudantes. Isso pode facilitar e agilizar o processo de correção, proporcionando feedback consistente e objetivo aos alunos.

A Linguagem de Modelagem Unificada (UML) é uma linguagem padronizada amplamente utilizada na Engenharia de Software para representar visualmente o design de sistemas. Ela oferece uma série de diagramas, como diagramas de classes, diagramas de sequência e diagramas de atividades, que auxiliam na compreensão e comunicação entre os membros da equipe de desenvolvimento.

Este trabalho propõe uma abordagem de validação automatizada de consistência entre diagramas UML e código Java, utilizando a comparação entre arquivos XML gerados a partir dos diagramas UML e o código fonte escrito em Java. Através da conversão dos diagramas UML para XML e do parse do código Java em uma estrutura de dados, torna-se possível realizar comparações precisas entre o design especificado e a implementação realizada.

A validação automatizada traz vantagens significativas, como a detecção precoce de inconsistências entre diagramas e código, a redução do tempo necessário para revisões manuais e a facilitação da manutenção e evolução do sistema. Além disso, a utilização de ferramentas como o Astah, que oferece suporte para a geração de diagramas e armazenamento de arquivos em formato XML, juntamente com a facilidade de manipulação de dados estruturados proporcionada pelo formato JSON, contribui para a efetividade da abordagem proposta.

Ao combinar a Engenharia de Software Orientada a Modelos (MDSE), a validação automatizada e a comparação entre XML e código Java, este trabalho visa contribuir para a melhoria dos processos de desenvolvimento de software, especialmente na área de Engenharia de Software baseada em UML. Além disso, busca-se explorar a aplicação dessa abordagem na área educacional, oferecendo uma ferramenta útil para professores corrigirem e avaliarem o código de vários alunos de forma consistente e eficiente.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver um sistema de verificação de corretude de códigos em Java em relação a um diagrama de classes nos padrões UML. Para alcançar esse objetivo principal, os seguintes objetivos específicos foram estabelecidos:

1. Projetar e implementar um metamodelo para armazenar as diferentes partes do diagrama e do código Java, permitindo a comparação direta entre eles.

2. Desenvolver algoritmos para a comparação automática do código Java com o diagrama de classes em UML, identificando discrepâncias, como atributos, operações ou classes faltantes ou com tipos de dados diferentes.

3. Criar um sistema de geração de relatórios que apresente as inconsistências encontradas entre o código Java e o diagrama de classes, fornecendo informações detalhadas sobre os problemas identificados.

4. Validar o sistema por meio de testes com exemplos de códigos Java e diagramas de classes em UML, verificando sua capacidade de detectar corretamente as discrepâncias e gerar relatórios precisos.

Através desses objetivos, espera-se que o sistema desenvolvido possa auxiliar no processo de verificação de corretude de códigos em Java, proporcionando uma análise automatizada e precisa em relação ao diagrama de classes em UML, reduzindo erros e agilizando o desenvolvimento de software.

2 TRABALHOS CORRELATOS

Nesta seção, são apresentados trabalhos relacionados à geração automática de código a partir de diagramas UML, com foco na verificação de consistência e na qualidade do código gerado.

2.1 UJECTOR

O trabalho realizado por (Usman e Nadeem, 2009) propõe uma abordagem para a geração automática de código Java a partir de diagramas UML, incluindo diagramas de classe, diagramas de sequência e diagramas de atividade. O objetivo principal é garantir a consistência entre os diagramas UML e o código gerado, assegurando a precisão, manutenibilidade, eficiência e completude do código resultante. O estudo apresenta a ferramenta UJECTOR (UML to Java Executable Code generaTOR), que implementa a abordagem proposta. A ferramenta utiliza o diagrama de classe para estruturar o código, o diagrama de sequência para controlar o fluxo de execução dos métodos e o diagrama de atividade para incluir manipulações de objetos e interações com o usuário.

Para validar a abordagem proposta, foram realizados dois estudos de caso, envolvendo sistemas reais, como um sistema de ponto de venda e um sistema universitário. A avaliação experimental demonstrou que o código gerado pelo UJECTOR é consistente com os diagramas UML de entrada, além de ser totalmente funcional e compreensível. Além disso, uma comparação foi realizada com outras ferramentas existentes de geração de código UML, tanto baseadas em pesquisa quanto comerciais e de código aberto. Os resultados mostraram que o código gerado pelo UJECTOR apresentou maior abrangência e compreensibilidade em relação às demais ferramentas avaliadas.

Em suma, o trabalho de (Usman e Nadeem, 2009) contribui para a área de geração automática de código a partir de diagramas UML, fornecendo uma abordagem integrada que incorpora tanto os aspectos estruturais quanto os comportamentais de um aplicativo orientado a objetos. Através da análise comparativa e da validação experimental, o estudo demonstrou a eficácia e a qualidade do código gerado pela ferramenta UJECTOR.

2.2 VALIDATION AUTOMATION OF UML DIAGRAMS CREATED BY STUDENTS

Este trabalho (Gasheva et al., 2021) aborda a classificação de erros cometidos por estudantes ao criar diagramas de UML, com foco em verificar a consistência, o layout e a gravidade desses erros. A classificação proposta é composta por várias categorias, incluindo Modelos, Consistência e Layout.

A abordagem para validação de diagramas de classes descrita no trabalho utiliza três etapas: análise léxica, análise sintática e análise semântica. Essas etapas se alinham com o processo de validação de código em comparação com diagramas de classes, pois você precisa garantir o uso correto dos elementos, relacionamentos e semântica da UML tanto no código quanto nos diagramas.

Para validar os diagramas de Casos de Uso (UCD) e Diagramas de Atividades (AD), foram utilizadas abordagens específicas. Para o UCD, foram identificados os métodos de Leitura Orientada a Objetos (OORT) e CBR. Já para o AD, foi realizado um estudo das metodologias

existentes, como a construção de uma rede de Petri (Baresi e Pezzè, 2001) e o uso de lógica temporal (Araujo e Moreira, 2005).

Em resumo, este trabalho busca identificar erros cometidos pelos estudantes na criação de diagramas propondo uma abordagem automatizada para verificar a qualidade dos diagramas UML, identificando erros léxicos, sintáticos e semânticos nos diagramas de caso de uso, diagramas de atividades e diagramas de classes.

2.3 A PRECISE APPROACH TO VALIDATING UML MODELS AND OCL CONSTRAINTS

A Linguagem de Modelagem Unificada (UML) se tornou um padrão amplamente adotado para a modelagem de sistemas de software. No entanto, o significado preciso dos modelos UML e das especificações da Linguagem de Restrição de Objetos (OCL) ainda carece de uma fundamentação formal. Isso tem levado a desafios, como construções com especificação insuficiente, ambiguidades e contradições. Esta pesquisa tem como objetivo abordar essas questões fornecendo um framework preciso para análise e validação de restrições OCL em modelos UML (Richters, 2002).

O principal objetivo deste trabalho é definir uma sintaxe e semântica formais para a OCL e partes essenciais do núcleo da UML relacionadas às restrições OCL. Essa formalização permite uma interpretação clara e consistente de conceitos-chave na modelagem de software. As contribuições desta pesquisa são as seguintes:

- **Formalização do Núcleo da UML:** Partes do núcleo da UML, focando em propriedades estruturais e dinâmicas de objetos, são formalmente definidas. Isso permite uma interpretação precisa e consistente de conceitos essenciais na modelagem.
- **Definição Precisa da OCL:** Pela primeira vez, é fornecida uma definição precisa da OCL, abordando ambiguidades, especificações insuficientes e contradições. São propostas várias extensões leves para aprimorar a ortogonalidade da linguagem.
- **Melhoria na Integração da UML e OCL:** As relações e dependências entre a UML e a OCL são tornadas explícitas, aprimorando a integração dessas linguagens.
- **Abordagem Baseada em Ferramentas para Validação:** É desenvolvida uma base sólida para ferramentas que suportam análise, simulação, transformação e validação de modelos UML com restrições OCL. Uma ferramenta específica para validar restrições OCL é implementada, demonstrando os benefícios práticos da formalização.
- **Validação das Regras de Boa-Formação do Padrão UML:** A abordagem de pesquisa é aplicada para validar as regras de bem-formação no padrão UML. Os resultados fornecem insights valiosos para a melhoria de futuras versões da UML.

Ao abordar os desafios de formalização e validação de modelos UML e restrições OCL, esta pesquisa contribui para aprimorar a qualidade geral de sistemas de software. Os resultados integrados abrem caminho para linguagens de modelagem bem definidas e fornecem uma base para avanços adicionais no desenvolvimento de software.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 UML

A UML (Unified Modeling Language), em tradução livre "Linguagem de Modelagem Unificada", é uma linguagem de propósito geral usada na área de Engenharia de Software para visualizar de forma padronizada o design de um sistema. A UML tem como objetivo fornecer aos arquitetos de sistema, engenheiros de software e desenvolvedores de software ferramentas para análise, design e implementação de sistemas baseados em software, assim como para modelagem de processos de negócios e processos similares (Object Management Group, 2017).

3.2 XML

O XML (Extensible Markup Language), em tradução livre "Linguagem de Marcação Extensível", é uma linguagem de marcação que define regras para a criação de documentos estruturados e legíveis por máquina e por humanos. O XML utiliza tags para delimitar elementos e permite que os desenvolvedores definam suas próprias tags personalizadas, o que o torna uma linguagem flexível e extensível. Ele é amplamente utilizado para representar e estruturar dados de forma hierárquica, tornando-o adequado para armazenar e transmitir informações complexas. O XML é independente de plataforma e de linguagem de programação, o que o torna uma escolha popular para troca de dados entre sistemas heterogêneos. Além disso, o XML é amplamente suportado por ferramentas e bibliotecas, o que facilita a análise, a manipulação e a validação dos dados em formato XML(Bray et al., 2013). No exemplo a seguir, sabemos que o título do livro pode ser acessado seguindo a sequência "livro.titulo"

```
1 <livro>
2   <titulo>Harry Potter and the Philosopher's Stone</titulo>
3   <autor>J.K. Rowling</autor>
4   <ano>1997</ano>
5 </livro>
```

Código 3.1: Exemplo de XML

3.3 ENGENHARIA DE SOFTWARE ORIENTADA A MODELOS

A Engenharia de Software Orientada a Modelos (Model-Driven Software Engineering - MDSE) é uma abordagem que busca melhorar o processo de desenvolvimento de software por meio do uso intensivo de modelos(Brambilla et al., 2017). Nessa abordagem, os modelos são criados e manipulados de forma sistemática, permitindo uma representação mais precisa e abstrata dos sistemas em estudo.

A MDSE baseia-se na ideia de que os modelos são uma representação mais compreensível e manipulável do sistema do que o próprio código-fonte. Ao utilizar modelos como artefatos principais, a MDSE visa aprimorar a qualidade do software, promover a reutilização de componentes e facilitar a comunicação entre os membros da equipe de desenvolvimento.

A Engenharia de Software Orientada a Modelos é fundamentada em alguns conceitos-chave. Um desses conceitos é o metamodelo, que define os elementos, as relações e as restrições de uma linguagem de modelagem específica. O metamodelo estabelece as regras e convenções para a criação e manipulação dos modelos dentro de um determinado domínio.

Outro conceito importante é o de transformações de modelos. As transformações de modelos são processos que permitem a conversão de modelos de um formato para outro, a partir de regras definidas. Essas transformações podem ser utilizadas para gerar código-fonte a partir de modelos, realizar análises e verificações nos modelos, ou até mesmo para a geração automática de documentação.

Uma das principais vantagens da Engenharia de Software Orientada a Modelos é a possibilidade de promover a reutilização de modelos e componentes. Com modelos bem definidos e padronizados, é possível criar bibliotecas de modelos que podem ser utilizadas em diferentes projetos, acelerando o processo de desenvolvimento e aumentando a produtividade da equipe.

3.3.1 Metamodelos

Um metamodelo é um modelo que representa de forma abstrata a estrutura e o comportamento de outros modelos. Ele define os elementos, suas propriedades e relacionamentos, bem como as regras e restrições que regem sua utilização. Os metamodelos são amplamente utilizados na Engenharia de Software Orientada a Modelos (MDSE) para analisar, manipular e representar modelos de forma consistente e padronizada (Brambilla et al., 2017). Fornecendo uma base sólida para a geração automática de código, validação e análise.

Um exemplo comum de metamodelo é a Linguagem de Modelagem Unificada (UML), que define um conjunto de conceitos e regras para representar visualmente diferentes aspectos de um sistema de software (Object Management Group, 2011). O metamodelo da UML especifica elementos como classes, associações, herança e comportamentos, permitindo que os modeladores criem diagramas que representem essas estruturas e interações.

3.3.2 Conversão de Modelos

A conversão de modelos envolve a transformação de um modelo em outro, geralmente com o objetivo de obter uma representação em um nível de abstração diferente ou em uma linguagem específica.

A conversão de modelos desempenha um papel fundamental na MDSE, permitindo a geração automática de código a partir de modelos, a integração de diferentes ferramentas de modelagem e a sincronização de modelos em diferentes fases do desenvolvimento de software.

Existem várias abordagens e técnicas para a conversão de modelos, incluindo transformações de modelo a modelo (Model-to-Model, M2M) e transformações de modelo a texto (Model-to-Text, M2T). As transformações M2M envolvem a definição de regras que mapeiam elementos de um modelo-fonte para elementos de um modelo-alvo (Jouault e Kurtev, 2006). Já as transformações M2T são usadas para gerar código ou texto a partir de um modelo.

A conversão de modelos pode ser realizada manualmente, com desenvolvedores definindo as regras de transformação, ou pode ser automatizada usando ferramentas de modelagem e transformação de modelos, como o Eclipse Modeling Framework (EMF), o ATL (Atlas Transformation Language) e o Acceleo (Brambilla et al., 2017).

Ao converter modelos, é importante garantir a preservação das propriedades e restrições do modelo original, bem como a corretude e a integridade do modelo resultante (France e Rumpe, 2007).

3.4 ASTAH

O Astah é uma ferramenta utilizada para criar diagramas que auxiliam no desenvolvimento de software e design de sistemas (Astah, sd). É fácil de usar e oferece recursos como

geração de código e simulação de modelo. É amplamente utilizado na área de desenvolvimento de software e engenharia de software, devido ao seu uso frequente para a criação de diagramas de software e suporte ao armazenamento de arquivos em um formato semelhante ao XML, o XMI.

3.5 JSON

JSON (JavaScript Object Notation), em tradução livre "Notação de Objetos JavaScript", é um formato baseado em texto que é independente de linguagem e é usado para transmitir e armazenar dados de forma estruturada (JSON, sd). Ele utiliza uma estrutura de pares chave-valor para organizar dados e é frequentemente utilizado como um formato de troca de dados entre aplicativos da web e servidores. O JSON é fácil de ler e escrever, e pode ser facilmente convertido em outros formatos de dados. É amplamente utilizado no desenvolvimento web para construir aplicativos e APIs dinâmicos.

O JSON foi escolhido devido à sua fácil visualização e manipulação de dados.

3.6 TYPESCRIPT

TypeScript é uma linguagem de programação desenvolvida pela Microsoft que é baseada em JavaScript (TypeScript, sd). Ela adiciona recursos adicionais, como tipagem estática opcional, interfaces, classes e módulos para ajudar na construção de aplicativos em larga escala. O TypeScript é compilado em JavaScript e pode ser usado em aplicativos cliente e servidor, e é frequentemente usado com estruturas front-end populares. Ele ajuda a detectar erros de código durante a compilação e oferece suporte a padrões de programação orientada a objetos e funcionais.

3.7 VERIFICAÇÃO E VALIDAÇÃO

Os processos de verificação e validação, são fundamentais para garantir a precisão e correção do código de computador e diagramas. Verificação é a avaliação do produto intermediário ou final para determinar se atende aos requisitos e padrões especificados (IEEE, 1990), enquanto validação é a avaliação do produto final para garantir que atenda às necessidades e expectativas do cliente. Ambos os processos são críticos para garantir que o produto final seja preciso, funcione corretamente e atenda às necessidades do cliente (Sommerville, 2011).

3.8 CONCLUSÃO

Em suma, esses são os conceitos fundamentais relacionados à validação de consistência entre diagramas UML e código, bem como ferramentas e tecnologias relevantes para a abordagem proposta neste trabalho. A UML foi introduzida como uma linguagem padronizada utilizada na Engenharia de Software para representar o design de sistemas. O XML foi abordado como uma linguagem de marcação para estruturar dados para exportar os dados do diagrama. A conversão de modelos, devido à necessidade de realizar a comparação de um diagrama com linhas de código. O Astah como ferramenta para gerar o diagrama e o JSON como formato de comparação de dados também foram discutidos. Por fim, a verificação e validação foram destacadas como processos cruciais para garantir a precisão do código e dos diagramas.

4 IMPLEMENTAÇÃO

O objetivo principal do sistema é verificar a corretude de códigos em Java quando comparados com o diagrama de classes em UML no qual se baseiam.

Para realizar a análise foi projetado um metamodelo para armazenar as diferentes partes do diagrama e do código de forma padrão, permitindo a comparação direta das duas partes. Dessa comparação são extraídos pontos problemáticos da implementação, como por exemplo a falta (ou sobra) de atributos, operações ou classes, além dos tipos dos dados definidos no diagrama.

O artigo descreve uma abordagem para gerar padrões de classes a partir de um diagrama UML gerado pelo Astah, que então é exportado em XMI (Derivação do XML). O diagrama, juntamente com o código Java, são transformados em um modelo representado em JSON, criado a partir do metamodelo. Finalmente, os dois modelos resultantes são comparados e um relatório das discrepâncias encontradas é gerado.

Exemplo concreto: Suponha que temos a seguinte classe em Java:

```
1 public class Person {
2    public void setName(String name) {
3       this.name = name;
4    }
5    public void setAge(int age) {
6       this.age = age;
7    }
8 }
```

Código 4.1: Exemplo de classe em Java

E temos o seguinte diagrama de classes em UML:

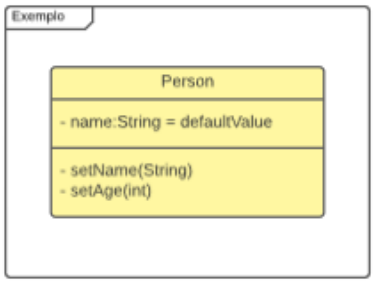


Figura 4.1: Exemplo de diagrama de classe.

O sistema de verificação identificaria que o atributo *name* está faltando no código e geraria um relatório indicando essa inconsistência.

```
Analysing class Person...
   Matching class FOUND
   Comparing attributes...
      Attribute name NOT FOUND
   Comparing operations...
      Operation setName FOUND
      Operation setAge FOUND
```

Código 4.2: Exemplo de relatório com o resultado

4.1 ASTAH E CÓDIGO JAVA

Os diagramas de classe utilizados para teste do programa foram gerados por meio da ferramenta Astah, a qual permite a exportação dos diagramas em formato XML, seguindo o padrão UML. A partir dessa exportação, é possível realizar uma análise programática dos diagramas. Os códigos Java foram implementados com base nos diagramas de classe gerados. Dentre as versões implementadas, algumas estavam corretas, implementando fielmente o que foi descrito nos diagramas, enquanto outras foram propositalmente criadas com erros, com o intuito de demonstrar a capacidade da ferramenta em detectá-los.

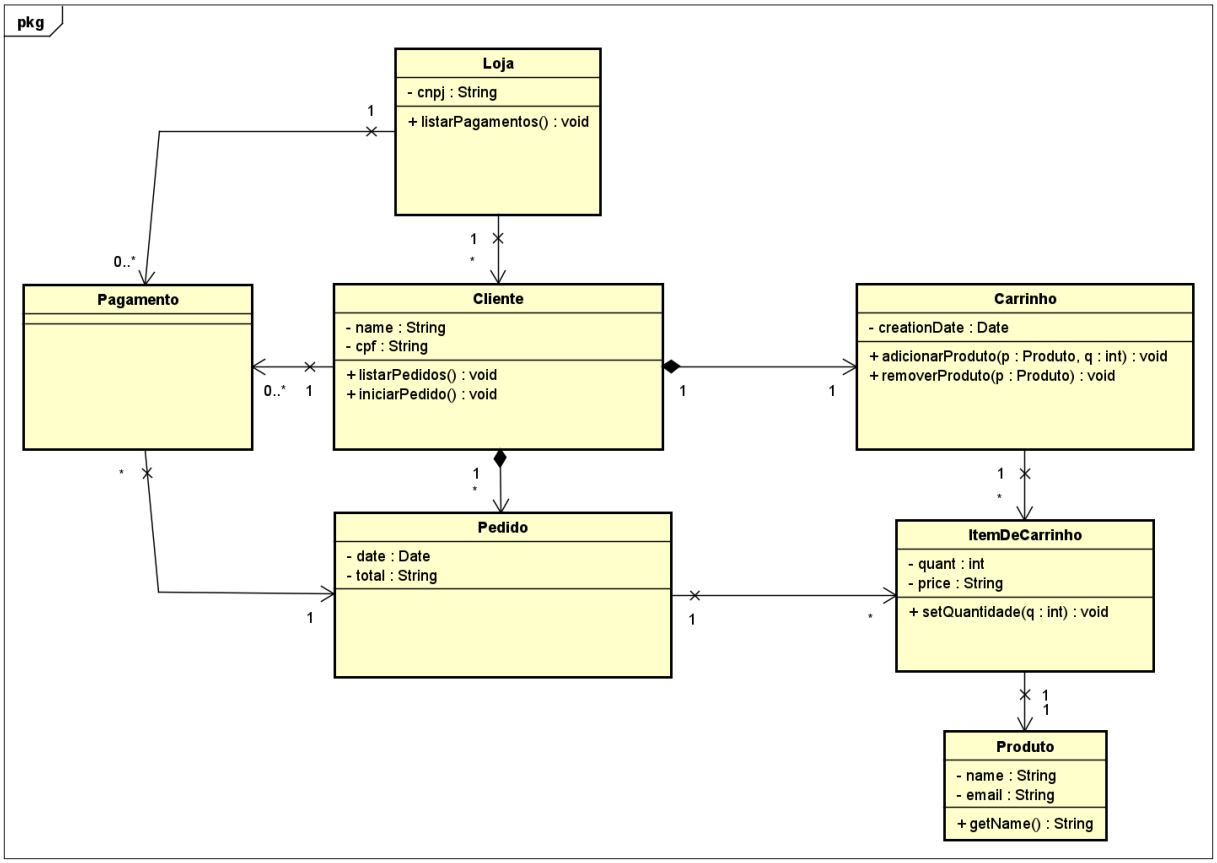


Figura 4.2: Exemplo de diagrama de classe.

4.2 XML

O XML gerado pelo Astah contém tags que representam as diferentes partes do diagrama. Cada uma delas possui propriedades para descrever o elemento específico que representa, como

name (o nome do elemento), *xmi:id* (id único do elemento) ou *xmi:type* (tipo do elemento no diagrama, pode ser uma string como *uml:Class* ou o id único de outro elemento). Tags do tipo *packagedElement* geralmente representam classes e possuem outras tags aninhadas para representar elementos que pertencem a ela. Atributos da classe são representados pela tag *ownedAttribute*, enquanto operações usam *ownedOperation* que, por sua vez, possui tags aninhadas para representar os parâmetros da operação *ownedParameter*.

```
<packagedElement
   name="ItemDeCarrinho" xmi:id="001" xmi:type="uml:Class"
>
   <ownedAttribute
      name="quant" type="_I46HoxU0Ee6PAM8hniUwVQ"
      visibility="private" xmi:id="002" xmi:type="uml:Property"
   />
   ...
   <ownedOperation
      name="setQuantidade" visibility="public" xmi:id="003"
      xmi:type="uml:Operation"
   >
      <ownedParameter
         direction="return" type="_I46HpBU0Ee6PAM8hniUwVQ" xmi:id="004"
         type="uml:Parameter"
      />
   ...
   </ownedOperation>
</packagedElement>
```

Código 4.3: Exemplo de XML parcial de uma classe

4.3 CONVERSÃO EM JSON

Embora o metamodelo que gera o XML proporcione uma estrutura adequada para a ferramenta sendo desenvolvida, a forma em que são tratados os tipos dos membros do diagrama foi considerada muito complexa e difícil de trabalhar. No XML, tipos são definidos como seus próprios p*packagedElements*, e são associados a outros membros do diagrama através de seu ID. Diante disso, optou-se pela alteração do metamodelo, introduzindo tipos diretamente na definição dos membros relevantes. Além disso, optou-se pela tradução do XML em JSON, uma vez que a ferramenta foi desenvolvida em Typescript e esse formato é suportado nativamente.

4.4 METAMODELO

O metamodelo utilizado para realizar a análise consiste em um conjunto de classes que representam as diversas partes comuns tanto ao código Java quanto ao Diagrama de Classes. As propriedades dessas classes são derivadas diretamente do JSON, conforme definido a seguir.

4.4.1 Classes

Classes são o nível mais alto representado no metamodelo. Cada uma contém listas para armazenar seus atributos e operações, além de atributos que definem, seu nome e visibilidade, definida como *package* por padrão (Oracle, sd).

```
export class MetaClass {
  "name": string;
```

```
3   "id": string;
4   "ownedAttribute": MetaAttribute[] | MetaAssociationAttribute[] = [];
5   "ownedOperation": MetaOperation[] = [];
6   "visibility": string = "package";
7 }
```

Código 4.4: Definição de classe no metamodelo

4.4.2 Atributos

Atributos são os membros de uma classe que armazenam dados e definem o estado de um objeto. O metamodelo contém propriedades como o nome dos atributos, seu tipo e sua visibilidade.

```
1 export class MetaAttribute {
2   aggregation?: string;
3   name?: string;
4   type?: string;
5   visibility?: string = "package";
6   id?: string;
7   xmitype?: string;
8 }
```

Código 4.5: Definição de atributo no metamodelo

4.4.3 Associações

Associações representam a relação entre classes em um sistema orientado a objetos. Uma associação pode ser unidirecional ou bidirecional, caso ela seja definida em uma ou ambas as direções entre as classes. No caso de uma associação bidirecional, cada classe terá uma referência à outra classe, enquanto em uma associação unidirecional, apenas uma das classes contém a referência para a outra. No metamodelo, as associações são uma extensão de um atributo comum, com a inclusão de um atributo *association*, que indica ser uma associação comum, *aggregation*, indicando uma agregação e os objetos *lowerValue* e *upperValue*, que definem a multiplicidade da associação.

```
1  export class MetaAssociationAttribute extends MetaAttribute {
2    constructor() {
3      super();
4      this.lowerValue = { value: '' };
5      this.upperValue = { value: '' };
6    }
7    aggregation?: string;
8    association?: string;
9    lowerValue?: {
10     value?: string;
11   };
12   upperValue?: {
13     value?: string;
14   };
15 }
```

Código 4.6: Definição de associação no metamodelo

4.4.4 Operações

Operações definem o comportamento de uma classe, incluindo as ações que uma classe pode fazer e seus efeitos. O metamodelo contém propriedades como nome, tipo de retorno e definição de parâmetros.

```
1 export class MetaOperation {
2   "name": string;
3   "returnType": string;
4   "visibility": string;
5   "id": string;
6   "xmitype": string;
7   "ownedParameter": MetaParameter[] = [];
8 }
```

Código 4.7: Definição de operação no metamodelo

4.4.5 Parâmetros

Parâmetros são os dados fornecidos a uma operação para que essa possa ser executada. O metamodelo define seu nome, tipo e direção, podendo ser um parâmetro de entrada ou de retorno.

```
1 export class MetaParameter {
2   direction?: string;
3   name?: string;
4   type?: string;
5   id?: string;
6   xmitype?: string;
7 }
```

Código 4.8: Definição de parâmetro no metamodelo

4.5 COMPARAÇÃO

A primeira etapa da comparação é verificar se todos os elementos incluídos no diagrama de classes e definidos no metamodelo estão presentes no código Java. Para cada elemento adquirido da análise do diagrama, é realizada uma busca no modelo gerado da análise do código para verificar a existência do seu correspondente de mesmo nome. Com essa busca também é possível determinar se existem elementos presentes no código que não foram definidos no diagrama. Após determinada a existência de um elemento é possível analisá-lo mais profundamente. Nas sessões a seguir serão definidas as diferentes categorias de erros que a ferramenta é capaz de detectar, além de como os elementos são analisados com base nelas.

4.5.1 Erros de Associação

Associações entre classes são uma parte essencial de um diagrama UML, e erros de associação podem trazer problemas significativos na implementação. A ferramenta é capaz de detectar quatro tipos de erro de associação, sendo eles:

- **Falta ou Excesso de Associação**: detecta e reporta casos em que falta associação entre duas classes que deveriam estar associadas, ou caso existam associações no código Java que não foram previstas no diagrama.

- **Erro de Multiplicidade**: detecta e reporta casos em que a multiplicidade de uma associação entre classes está incorreta. Multiplicidade se refere ao número de objetos que podem estar relacionados entre as classes. (i.e. Relações *1:1, 1:*, *:**)

- **Erro de navegabilidade**: detecta e reporta casos em que a navegabilidade da associação entre duas classes está incorreta. Navegabilidade se refere a habilidade de "atravessar" uma associação em uma direção em particular. No Astah, a navegabilidade é representada por uma flecha na direção navegável, e a não navegabilidade é representada por um "x" (Figura 4.3).

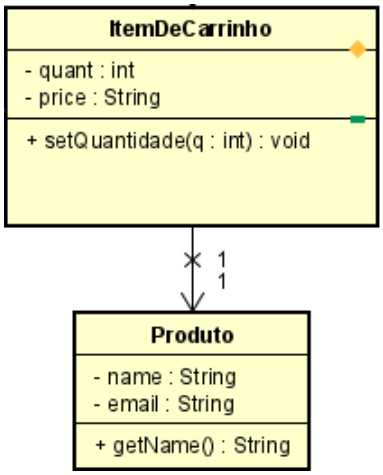


Figura 4.3: Exemplo de navegabilidade.

- **Tipo de Associação**: detecta e reporta erros de tipo de associação para associações comuns e agregações. Para exemplificar a detecção dessas falhas vamos usar a seção do diagrama e do código Java (com erros introduzidos) a seguir:

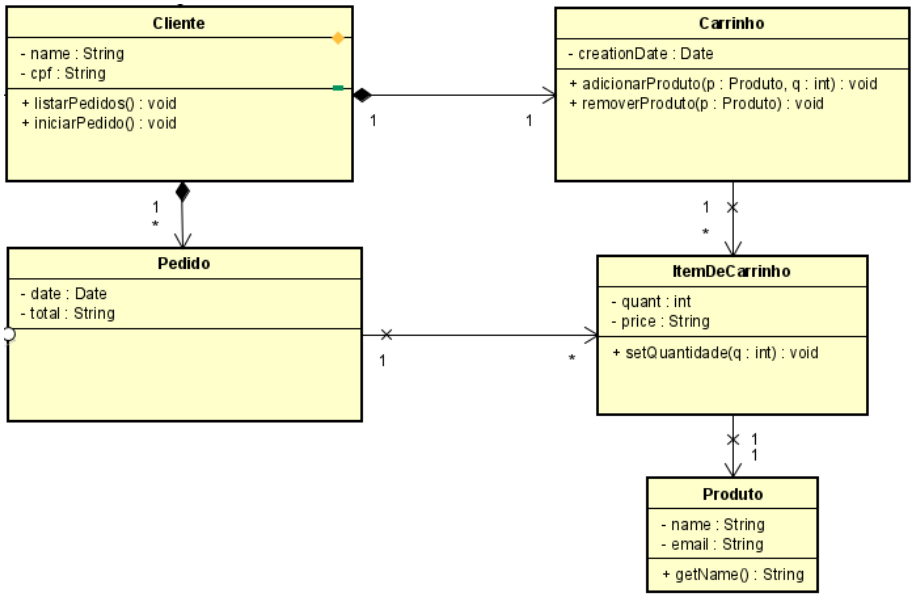


Figura 4.4: Exemplo de comparação: associação.

A classe *Carrinho* possui uma associação *1..** com a classe *ItemDeCarrinho*, mas no código vamos representá-la como *1..1*.

```
1  public class Carrinho {
2     private Date creationDate;
3     //Expected array: multiplicity error
4     private ItemDeCarrinho itemDeCarrinho;
5     public void adicionarProduto(Produto p, int q) {
6     }
7     public void removerProduto(Produto p) {
8     }}
```

Na classe Cliente vamos remover o atributo que representa a agregação com a classe Pedido.

```
1  public class Cliente {
2     private String name;
3     private String cpf;
4     private Carrinho carrinho;
5     private Pagamento[] pagamento;
6     // private Pedido[] pedido; // Missing aggregation
7     public void listarPedidos() {
8     }
9     public void iniciarPedido() {
10    }
11 }
```

Finalmente, na classe *Produto* vamos introduzir uma associação que não existe no diagrama, com a classe *Pedido*, além de violar a navegabilidade da associação com a classe *ItemDeCarrinho*.

```
1  public class Produto {
2     private String name;
3     private String email;
4     private Pedido pedido; // Extra association
5     //Extra association: violates navegability
6     private ItemDeCarrinho[] itemDeCarrinho;
7     public String getName() {
8    return null;
9     }
10 }
```

O primeiro passo é usar os membros do modelo gerado a partir do diagrama para encontrar equivalentes no modelo gerado do código Java.

```
1  Analysing associations Diagram -> Code...
2     Association Cliente --> Carrinho FOUND as carrinho: Carrinho[]
3     Association Cliente --> Pagamento FOUND as pagamento: Pagamento[]
4     Association Cliente --> Pedido NOT FOUND
5     Aggregation* Carrinho --> Cliente FOUND as carrinho: Carrinho
6     Association Carrinho --> ItemDeCarrinho Multiplicity mismatch:
7        Expected * but found 1
8     Association Pagamento --> Pedido FOUND as pedido: Pedido[]
9     Association Loja --> Cliente FOUND as cliente: Cliente[]
10    Association Loja --> Pagamento FOUND as pagamento: Pagamento[]
11    Association ItemDeCarrinho --> Produto FOUND as produto: Produto[]
12    Association Pedido --> ItemDeCarrinho FOUND
13       as itemDeCarrinho: ItemDeCarrinho[]
14    Aggregation* Pedido --> Cliente NOT FOUND (missing on parent class)
```

Com esses resultados podemos ver que a ferramenta identificou dois dos quatro problemas introduzidos. Na linha 6 foi identificado o erro de multiplicidade, incluindo o valor esperado pelo diagrama. Nas linhas 3 e 12 foi detectada a falta de agregação entre as classes *Cliente* e *Pedido*.

O segundo passo é fazer a comparação reversa, usando o código Java para verificar se existem membros extra nele quando comparado ao diagrama.

```
1  Analysing associations Code -> Diagram...
2        Association Carrinho --> ItemDeCarrinho FOUND as : ItemDeCarrinho[]
3        Association Cliente --> Carrinho FOUND as : Carrinho[]
4        Association Cliente --> Pagamento FOUND as : Pagamento[]
5        Association ItemDeCarrinho --> Produto FOUND as : Produto[]
6        Association Loja --> Cliente FOUND as : Cliente[]
7        Association Loja --> Pagamento FOUND as : Pagamento[]
8        Association Pagamento --> Pedido FOUND as : Pedido[]
9        Association Pedido --> ItemDeCarrinho FOUND as : ItemDeCarrinho[]
10       Association Produto --> Pedido NOT FOUND
11       Association Produto --> ItemDeCarrinho NOT FOUND
```

Neste caso, nas linhas 10 e 11, a ferramenta indica que não encontrou duas associações no diagrama que foram definidas no código. Com o metamodelo utilizado pela ferramenta não é possível explicitar a diferença entre uma associação extra e uma violação de navegabilidade, apenas reportar sua existência.

4.5.2 Erros de Falta/Excesso e Nomenclatura

A ferramenta é capaz de detectar e reportar inconsistências nas convenções de nomenclatura estabelecidas pelo diagrama, além da falta ou excesso de classes, operações ou atributos no código Java. Por exemplo, na classe *Loja* à seguir, definimos apenas um atributo, *cnpj* e duas associações. Na sua representação em Java temos uma propriedade extra, *nome*, além de um parâmetro a mais na operação.

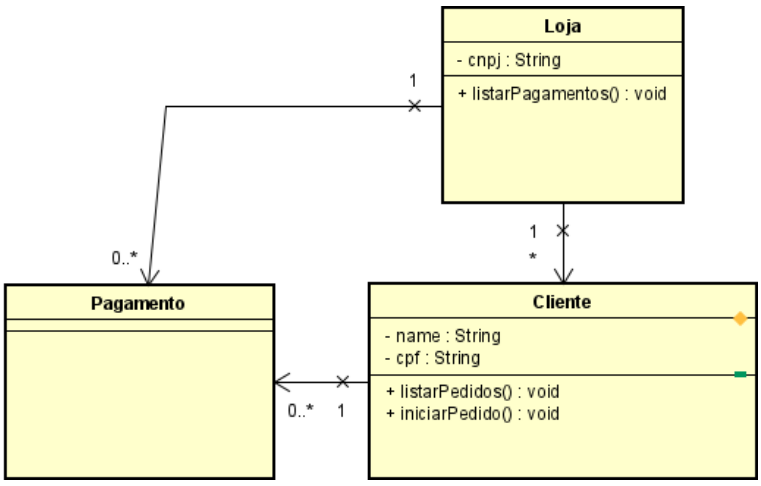


Figura 4.5: Exemplo de comparação: nomenclatura.

```
1 import java.util.ArrayList;
2 public class Loja {
3    private String cnpj;
4    private String nome; // Extra attribute
5    private Cliente[] cliente;
6    private Pagamento[] pagamento;
7    public void listarPagamentos(int a) { // extra parameter
8    }
9 }
```

A ferramenta então verifica se todos os membros da classe *Loja* do diagrama estão presentes no código Java: **Segunda categoria de testes - Convenções de Nomenclatura:**

```
1 Analysing class Loja...
2    Matching class FOUND
3    Comparing attributes...
4       Attribute cnpj FOUND
5    Comparing operations...
6       Operation listarPagamentos FOUND
7          Parameter mismatch: Expected 0 parameters, but found 1
```

E é determinado que tudo que foi definido no diagrama está presente em código, mas que o número de parâmetros na operação *listarPagamentos* é diferente do esperado. Em seguida é verificado se existem membros em excesso no código:

```
1 Analysing class Loja...
2    Matching class FOUND
3    Comparing attributes...
4       Attribute cnpj FOUND
5       Attribute nome NOT FOUND
6    Comparing operations...
7       Operation listarPagamentos FOUND
```

Onde é determinado que falta um atributo “nome” no diagrama.

4.5.3 Erros de Visibilidade

Inconsistências de visibilidade também podem ser detectadas em classes, operações ou atributos. Por exemplo, na classe *ItemDeCarrinho* (Figura 4.3), definimos dois atributos privados, *quant* e *price*, e uma operação pública, *setQuantidade*. Na sua representação em Java introduzimos *price* como *public*.

```
1 public class ItemDeCarrinho {
2    private int quant;
3    public String price;
4    private Produto produto;
5    public void setQuantidade(int q) {
6    }
7 }
```

A ferramenta então analisa a visibilidade de cada membro da classe, e compara com o valor esperado no diagrama.

```
1 Analysing class ItemDeCarrinho...
2    Matching class FOUND
3    Comparing attributes...
4       Attribute quant FOUND
5       Attribute price FOUND
6          Visibility mismatch: Expected public but found private
7    Comparing operations...
8       Operation setQuantidade FOUND
```

Neste exemplo é identificado o problema de visibilidade, e valor esperado é indicado.

4.5.4 Erros de Tipo

Outro aspecto importante para a validação do código Java é a verificação dos tipos dos membros das classes. Na classe “Pedido”, o “total” é definido como uma string. Para este exemplo ele é alterado para boolean no código Java.

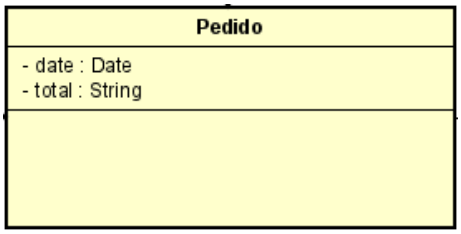


Figura 4.6: Exemplo de comparação: tipo.

```
1 public class Pedido {
2    private Date date;
3    private boolean total;
4    private ItemDeCarrinho[] itemDeCarrinho;
5 }
```

O tipo de cada atributo é analisado e comparado com o valor esperado no diagrama.

```
1 Analysing class Pedido...
2    Matching class FOUND
3    Comparing attributes...
4       Attribute date FOUND
5       Attribute total FOUND
6          Parameter mismatch: Expected String but found boolean
```

A ferramenta reporta que o atributo foi encontrado, mas possui uma discrepância com o que foi definido no diagrama, explicitando o tipo esperado e o encontrado.

5 TESTES

O principal objetivo da ferramenta é detectar inconsistência entre o código Java e o diagrama UML que ele implementa. Para garantir a precisão e completude da ferramenta, vários testes foram criados para sua validação. Esses testes utilizam um diagrama UML e o código Java gerado automaticamente pelo Astah. Falhas são então introduzidas no código para simular erros que um programador pode cometer durante a implementação. Em seguida, a ferramenta é usada para detectar e reportar essas falhas.

Para cada categoria de erro, definidas na seção 4.7, foram introduzidas mutações no código Java pertinentes à categoria sendo avaliada. Em seguida, o código foi avaliado pela ferramenta e os resultados foram analisados.

5.1 ERROS DE ASSOCIAÇÃO

Para cada subcategoria de erro de associação foram introduzidas cinco mutações. A ferramenta foi capaz de detectar todos os problemas referentes a falta ou excesso de associação, violações de navegabilidade, multiplicidade e de tipos de associações (limitados a associações comuns e agregações). Além disso, a ferramenta também reportou todas as ocorrências em que esperava uma associação “para muitos” (0..*, 1..*, etc), mas falhou em reportar erros quando esperava um associação “para um” (1..1, *..1, etc). Essa falha ocorre pois a multiplicidade da classe de destino é representada em uma propriedade separada que não foi tratada, e a ferramenta assuma "*"como valor padrão.

5.2 ERROS DE FALTA/EXCESSO E NOMENCLATURA

Foram introduzidos um total de 20 mutações para esta seção, em duas versões do código, com 10 erros cada, abrangendo tanto problemas com nomenclatura, quanto a falta ou excesso de atributos, classes, operações e parâmetros. Na primeira versão foram detectados 9/10 erros, enquanto na segunda versão foram detectados 8/10 erros. Mediante análise, foi determinado que os erros não detectados estavam presentes dentro de outros membros com mutações. A ferramenta depende da existência de um membro em ambos modelos para realizar a validação de seus sub-membros. Portanto, não é possível detectar a falta de um atributo em uma classe que não existe no diagrama, ou em um parâmetro que pertence a uma operação com algum erro de nomenclatura, nesse caso, gerando apenas um aviso sobre a falta de classe e atributo.

5.3 ERROS DE VISIBILIDADE

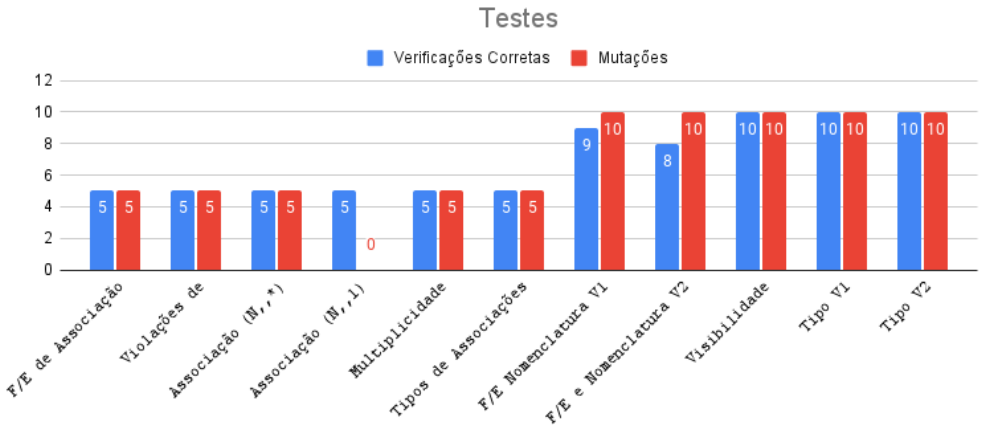
Foram introduzidas 10 mutações para esta categoria, incluindo alterações de visibilidade de classes, operações e atributos. Além disso, foi testado se a ferramenta é capaz de detectar a visibilidade padrão de um atributo ou método em Java. Um atributo ou método que não possui uma visibilidade explícita no código assume visibilidade “package private” (ou “package” na terminologia do Astah). Em todos os casos, a ferramenta foi capaz de identificar o erro e reportar o valor esperado para a visibilidade do membro em questão.

5.4 ERROS DE TIPO

Para esta categoria novamente foram introduzidas 20 mutações, em duas versões do código, com 10 erros cada, com erros de tipo em atributos e parâmetros de operações. Vale ressaltar que atributos que representam associações não foram alterados para esta categoria, uma vez que a ferramenta reportaria eles como erro de associação, não de tipo. A ferramenta foi capaz de detectar todos os erros de tipo para atributos, mas falhou em detectar problemas com os tipos dos parâmetros de operações em todos os casos introduzidos, sendo um na primeira versão e quatro na segunda.

5.5 CONCLUSÃO

A principal meta é verificar a consistência entre o código Java e o diagrama UML correspondente. Para garantir a precisão da ferramenta, foram realizados diversos testes usando um diagrama UML e códigos Java criado pelos autores. Falhas foram introduzidas no código para simular erros comuns na implementação. A ferramenta foi utilizada para detectar e reportar essas falhas. Foram realizados testes para diferentes categorias de erros, como erros de associação, falta/excesso de membros e a corretude de sua nomenclatura, visibilidade e tipo. A ferramenta mostrou um bom desempenho na detecção de erros de associação, falta/excesso e nomenclatura, e visibilidade. No entanto, houve algumas falhas na detecção de erros de tipo nos parâmetros das operações. No geral, os testes validaram a eficácia e a abrangência da ferramenta na detecção de inconsistências entre o código Java e o diagrama UML.



6 CONCLUSÃO

A ferramenta de comparação e análise de código Java e diagramas de classe oferece uma abordagem eficiente para identificar erros e inconsistências entre esses dois artefatos. Com base em um metamodelo definido, a ferramenta é capaz de realizar comparações, destacando problemas como erros de associação, erros de nomenclatura, erros de visibilidade e erros de tipo.

Ao analisar o código Java e o diagrama de classe, a ferramenta identifica diferenças e semelhanças entre eles, fornecendo uma visão abrangente das discrepâncias existentes. Isso ajuda os desenvolvedores a garantir que a implementação corresponda às especificações do diagrama e vice-versa, facilitando a detecção e correção de erros antes que eles se tornem problemas maiores.

Além disso, a ferramenta também é capaz de lidar com casos em que elementos estão ausentes em um dos artefatos, permitindo que os desenvolvedores identifiquem falhas e incompatibilidades na modelagem ou implementação. Isso ajuda a melhorar a consistência e a qualidade do código, alinhando-o com a estrutura e os requisitos estabelecidos pelo diagrama de classe.

Com uma abordagem automatizada e sistemática, a ferramenta agiliza o processo de verificação e validação, economizando tempo e esforço dos desenvolvedores e professores para correção de códigos. Conforme explicado nos testes, a ferramenta proporciona uma análise que atualmente não está 100% precisa, mas com a evolução do código de "parsing", visa-se a precisão máxima.

Em suma, a ferramenta de comparação e análise de código Java e diagramas de classe é uma ferramenta valiosa para melhorar a consistência e a qualidade do software. Ao identificar e relatar discrepâncias entre o código e o diagrama, ela ajuda a garantir a conformidade com os requisitos e a facilitar a manutenção e evolução do sistema. Tendo em vista os objetivos previstos para a ferramenta, todos os objetivos foram cumpridos, com algumas notas sobre eles. A comparação automática não está 100% precisa, o relatório gerado atualmente é um relatória visual simples, no terminal.

REFERÊNCIAS

Araujo, J. e Moreira, A. (2005). Integrating uml activity diagrams with temporal logic expressions. Em *Proc. of the 10th International Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'05)*, páginas 91–98.

Astah (s.d.). Astah. https://astah.net/. Acessado em 23/6/2023.

Baresi, L. e Pezzè, M. (2001). On formalizing uml with high-level petri nets. Em *Lecture Notes in Computer Science*, volume 2001, páginas 276–304.

Brambilla, M., Cabot, J. e Wimmer, M. (2017). *Model-Driven Software Engineering in Practice (2nd ed.)*. Morgan & Claypool Publishers.

Bray, T., Paoli, J. e Yergeau, F. (2013). Xml. `https://www.w3.org/XML/`. Acessado em 02/06/2023.

France, R. e Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. Em *Future of Software Engineering (FOSE '07)*, página 37–54.

Gasheva, T., Vlasov, D., Otinov, A. e Datsun, N. (2021). Validation automation of uml diagrams created by students. Em *Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 4*, páginas 7–18.

IEEE (1990). Ieee standard glossary of software engineering terminology.

Jouault, F. e Kurtev, I. (2006). Transforming models with atl. Em *Satellite Events at the MoDELS 2005 Conference*, páginas 128–138.

JSON (s.d.). Json. https://json.org/. Acessado em 23/6/2023.

Object Management Group (2011). Omg® unified modeling language®. `https://www.omg.org/spec/UML/2.5.1/PDF`. Acessado em 23/6/2023.

Object Management Group (2017). Unified modeling language specification. `http://www.omg.org/spec/UML`. Acessado em 23/6/2023.

Oracle (s.d.). Oracle documentation. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html. Acessado em 01/7/2023.

Richters, M. (2002). Em *A precise approach to validating UML models and OCL constraints*.

Sommerville, I. (2011). *Software Engineering (9th ed.)*. Pearson Education.

TypeScript (s.d.). Typescript. https://www.typescriptlang.org/. Acessado em 23/6/2023.

Usman, M. e Nadeem, A. (2009). Automatic generation of java code from uml diagrams using ujector. Graduação, Mohammad Ali Jinnah University, Islamabad, Paquistão.