CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
Recredenciado pela Portaria Ministerial nº 1.162, de 13/10/16, D.O.U nº 198, de 14/10/2016
ASSOCIAÇÃO EDUCACIONAL LUTERANA DO BRASIL
Glaycow Silveira Silva e Souza

DESENVOLVIMENTO DE UMA FERRAMENTA PARA ANÁLISE E EXTRAÇÃO DE MÉTRICAS DE QUALIDADE DE DIAGRAMAS DE CLASSE UML BASEADO NO QUALITY MODEL OBJECT ORIENTED DESIGN

Palmas – TO

2018

Glaycow Silveira Silva e Souza

DESENVOLVIMENTO DE UMA FERRAMENTA PARA ANÁLISE E EXTRAÇÃO DE MÉTRICAS DE QUALIDADE DE DIAGRAMAS DE CLASSE UML BASEADO NO QUALITY MODEL OBJECT ORIENTED DESIGN

Trabalho de Conclusão de Curso (TCC) II elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M. Sc. Jackson Gomes de Souza

Palmas – TO

2018

Glaycow Silveira Silva e Souza

DESENVOLVIMENTO DE UMA FERRAMENTA PARA ANÁLISE E EXTRAÇÃO DE MÉTRICAS DE QUALIDADE DE DIAGRAMAS DE CLASSE UML BASEADO NO QUALITY MODEL OBJECT ORIENTED DESIGN

Trabalho de Conclusão de Curso (TCC) II elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M. Sc. Jackson Gomes de Souza

Aprovado em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. M. Sc. Jackson Gomes de Souza

Orientador

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. Heloise Acco Tives Leão

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. Madianita Bogo

Centro Universitário Luterano de Palmas – CEULP

Palmas – TO

2018

DEDICATÓRIA

AGRADECIMENTOS

RESUMO

O presente trabalho tem o objetivo o desenvolvimento de uma ferramenta para extração de métricas de qualidade de diagramas de classe UML baseados no *Quality Model Object Oriented Design* QMOOD. As extrações das informações baseiam-se na análise de diagramas de classe definidos na linguagem *PlantUML*. No referencial teórico são apresentados os conceitos de design de *software* orientados a objetos, conceitos de orientação a objeto, conceitos básicos de UML que envolve os diagramas de classe. Também são apresentados os conceitos do QMOOD e suas respectivas propriedades e métricas. Levando em consideração esses conceitos a metodologia deste trabalho é de caráter exploratório e qualitativa. Com intuito de atingir o objetivo, foi desenvolvido um software para análise dos diagramas, extração e quantificação das métricas baseadas no QMOOD.

PALAVRAS-CHAVE: Métricas, Diagramas de Classe, UML, QMOOD.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A década de 1960 foi marcada por um período intitulado como a “crise do *software*” (PRESSMAN, 2011). Esse termo foi utilizado para representar os problemas com a qualidade de software inicialmente descobertos na década de 1960 com o desenvolvimento do primeiro grande sistema de software e continuaram a incomodar a engenharia de software ao longo do século XX (SOMMERVILLE, 2011). Esses problemas estão relacionados a falta de técnicas de desenvolvimento de software, prazos dos projetos estourados e a baixa qualidade dos *softwares* entregues (PRESSMAN, 2011).

Segundo Sommerville (2011), o software entregue era lento e pouco confiável, difícil de manter e de reusar. Em resposta à insatisfação dos clientes com essa situação, passaram a ser adotadas técnicas e métodos de engenharia de *software* foram desenvolvidos, as quais foram desenvolvidas a partir dos métodos usados na indústria manufatureira (SOMMERVILLE, 2011; PRESSMAN, 2011). Essas técnicas de gerenciamento de qualidade, em conjunto com novas tecnologias de software e melhores testes de *software*, conduziram a melhorias significativas no nível geral da qualidade dos *softwares* desenvolvidos (SOMMERVILLE, 2011).

Segundo Koscianski e Soares (2007), qualidade de um software envolve todo um processo de como deve montado o software dando ênfase no ’como’ deve ser elaborado o seu desenvolvimento. Dando ênfase no ’como’ desenvolver *softwares* com qualidade é imposto a utilização de modelos de boas práticas como o CMMI (2018) e as normas ISO/IEC 12207 (2008), ISO/IEC 9126 (1991), entre outras normas (KOSCIANSKI; SOARES, 2007). A qualidade de software ainda depende principalmente da utilização de metodologias de boas práticas de forma correta (KOSCIANSKI; SOARES, 2007).

Segundo Bansiya e Davis (2002), o modelo de qualidade QMOOD (Quality Model Object Oriented Design), por se basear nas características de design de OO, tem a vantagem de definir seis altos níveis de atributos de qualidade de projetos orientados a objetos (Reutilização, Flexibilidade, Compreensibilidade, Funcionalidade, Extensibilidade e Eficácia). Esses seis atributos de qualidade podem ser aplicados tanto no início quanto no fim do desenvolvimento de um *software.* Para a identificação das métricas de qualidade na fase inicial do projeto pode-se utilizar diagramas de classe da UML como fonte de análise.

Esses diagramas podem ser concebidos, por exemplo, com a linguagem PlantUML. Essa linguagem permite criar diagramas de classes, utilizando uma definição de linguagem simples

e legível (PlantUML, 2018). A análise de digramas de classe pode ser realizada por um interpretador, que divide uma informação em partes menores, de acordo com as definições especificadas, a fim de facilitar a tradução em outro idioma. Esse interpretador, também conhecido como *parser*, apresenta a divisão dessa informação em por meio de uma estrutura de dados na forma de uma árvore (PEREIRA, 2008).

O objetivo deste trabalho é o desenvolvimento de um software para extração de métricas de qualidade de diagramas de classe UML utilizando o modelo QMOOD. Para alcançar este objetivo foi desenvolvido um *parser* para a sintaxe do PlantUML, um padrão textual para representação de diagramas da UML, que extrai informações do diagrama e as utiliza para avaliar a sua qualidade conforme as métricas do QMOOD.

Esta monografia está organizada da seguinte forma: o capítulo 2 apresenta o Referencial Teórico, contendo tópicos como o modelo QMOOD; o capítulo 3 apresenta os materiais utilizados no desenvolvimento do trabalho e os procedimentos metodológicos; o capítulo 4 descreve o software desenvolvido e apresenta o seu funcionamento; por fim, o capítulo 5 apresenta as conclusões sobre o resultado obtido.

2 REFERENCIAL TEÓRICO

Está seção abordará os temas Design de *Software* Orientados a Objetos, Conceitos de orientação a objeto e *Quality Model Object Oriented* Design, podendo assim ter uma visão conceitual e detalhadas de como e o processo de extração de métricas de qualidade em diagrama de classe UML.

2.1 DESIGN DE SOFTWARE ORIENTADOS A OBJETOS

O Projeto Orientado a Objetos (OOD, do inglês *Object Oriented Design*) procura entender como os sistemas devem ser projetados utilizando uma estratégia na qual os engenheiros/arquitetos de sistemas pensam em como serão os principais componentes estruturais de um sistema e as suas respectivas relações (SOMMERVILLE, 2010). A saída do processo de design é um modelo de arquitetura que descreve como o sistema será organizado, todos os seus componentes e suas estruturas. A execução do sistema é feita por meio da interação dos objetos definidos no modelo que evidenciem a execução e informações do código gerado (SOMMERVILLE, 2010).

2.2 CONCEITOS DE ORIENTAÇÃO A OBJETO

2.2.1 UML

A UML é a sigla de *Unified Modelling Language*, que pode ser traduzida por Linguagem de Modelagem Unificada (GUEDES, 2011). A UML é uma família de notações gráficas, constituído por um metamodelo único, que a ajuda na modelagem de software, particularmente em software que serão construídos utilizando o estilo de orientação a objeto (OO) (FOWLER, 2005). A UML tornou-se a linguagem padrão de modelagem adotada internacionalmente pela indústria de engenharia de software. Em decorrência disso, existe hoje uma grande valorização e procura no mercado por profissionais que dominem essa linguagem (NUNES; O'NEILL, 2004).

Segundo (GUEDES, 2011) "A UML surgiu da união de três métodos de modelagem: o método de Booch, método OMT (*Object Modelling Technique*) de Jacobson e o método OOSE (*Object-Oriented Software Engineering*) de Rumbaugh". Esses eram os métodos da década de 1990 que eram mais utilizados por profissionais da área de desenvolvimento de *software*. A união desses métodos contou com o amplo apoio da *Rational Software*, que incentivou e financiou o projeto (GUEDES, 2011).

Segundo Nunes e O'Neill (2004) Pela abrangência e simplicidade dos conceitos utilizados, a UML facilita o desenvolvimento de *software*, permitindo integrar os aspectos de natureza organizacional que constituem o negócio e os elemento de natureza tecnológica, que irá

constitui no software. Segundo Guedes (2011) é importante ressaltar que UML não é uma linguagem de programação, e sim uma linguagem de modelagem, cujo seu principal objetivo é auxiliar os usuários a definirem as características do sistema, tais como seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos e até mesmo suas necessidades físicas em relação aos equipamentos em que o sistema será implantado. Com essas características deixa claro que UML não é um processo de desenvolvimento de *software* e tampouco está ligada a uma forma exclusiva de modelagem, sendo totalmente independente (GUEDES, 2011).

A UML possui vários diagramas com o objetivo de fornecer múltiplas visões do *software* a ser modelado. Analisar e modelar sob diversos aspectos diferentes, procura-se atingir a completude da modelagem, permitindo que cada diagrama complemente os outros diagramas. Em cada um dos diagramas da UML é utilizando símbolos que representam os elementos de sistema que estão a ser modelados e linhas que representam os relacionamentos (NUNES; O'NEILL, 2004; GUEDES, 2011).

2.2.2 DIAGRAMA DE CLASSE

O diagrama de classe é um dos mais importantes e mais utilizados da UML (GUEDES, 2011). Tendo como seu principal enfoque permitir a visualização das classes que compõem o sistema, seus respectivos atributos e métodos, bem como demonstrar como as classes do diagrama se relacionam, se complementam e transmitem informações entre si. Esse diagrama é uma descrição formal da estrutura do objeto de um sistema. Para cada objeto é descrita a sua identidade, os vários tipos de relacionamentos estáticos exististes com os outros objetos, as restrições, os seus atributos e suas operações (NUNES; O'NEILL, 2004; GUEDES, 2011).

A criação de um modelo de classe resulta de um processo de abstração através do qual se identificam os objetos (entidades e conceito) que são relevantes para o contexto que compõem o sistema que se pretende modelar. Procura descrever as características comuns em propriedades (atributos) e de comportamento (operações). A essa descrição genérica dá-se o nome de classe. Assim, as classes descrevem os objetos com seus atributos e suas operações 6 comuns (métodos), e servem a dois propósitos: permitem compreender o mundo real naquilo que é relevante para o sistema de informação que se pretende desenvolver e fornecem uma base prática para a implementação de *softwares* (NUNES; O'NEILL, 2004). O diagrama de classe tem como recomendação a sua utilização ainda durante a fase de análise após a coleta dos requisitos, produzindo um modelo conceitual a respeito das informações necessárias ao software. No modelo conceitual, o engenheiro preocupa-se apenas em representar as informações necessárias que o software deve dispor em termo de classes seus atributos, bem

com as associações entre as classes. Não sendo necessário modelar as características que o software poderá conter nessa etapa, como os métodos que as classes irão conter por conta que os métodos já fazem parte do "como" o software será desenvolvido. Somente na fase de projeto toma-se o modelo conceitual do diagrama de classe e é produzido o modelo de domínio, já pensando na solução do software (GUEDES, 2011).

2.2.3 TIPOS DE DADOS BÁSICOS

Para cada atributo do diagrama pode-se identificar o seu tipo de dado. Este tipo caracteriza a informação que o atributo irá conter. Os tipos de dados disponíveis dependem diretamente da linguagem que o *software* será desenvolvido. Contudo, é possível restringir os seguintes conjuntos de tipos de dados básicos: *Integer* (*number*) representa um número inteiro, *Double* representa os números reais, *String* representa textos, *Date* representa data, *Boolean* representa um valor lógico verdadeiro ou falso (NUNES; O'NEILL, 2004).

Tantos atributos como operações podem ser visíveis ou não por outras classes. Esta propriedade denomina-se visibilidade e assume 3 níveis: *Public* (Público) qualquer classe tem acesso ao elemento; Protect (Protegido) qualquer descendente da classe pode acessar o elemento; e *Private* (Privado) apenas própria classe tem acesso ao elemento (NUNES; O'NEILL, 2004).

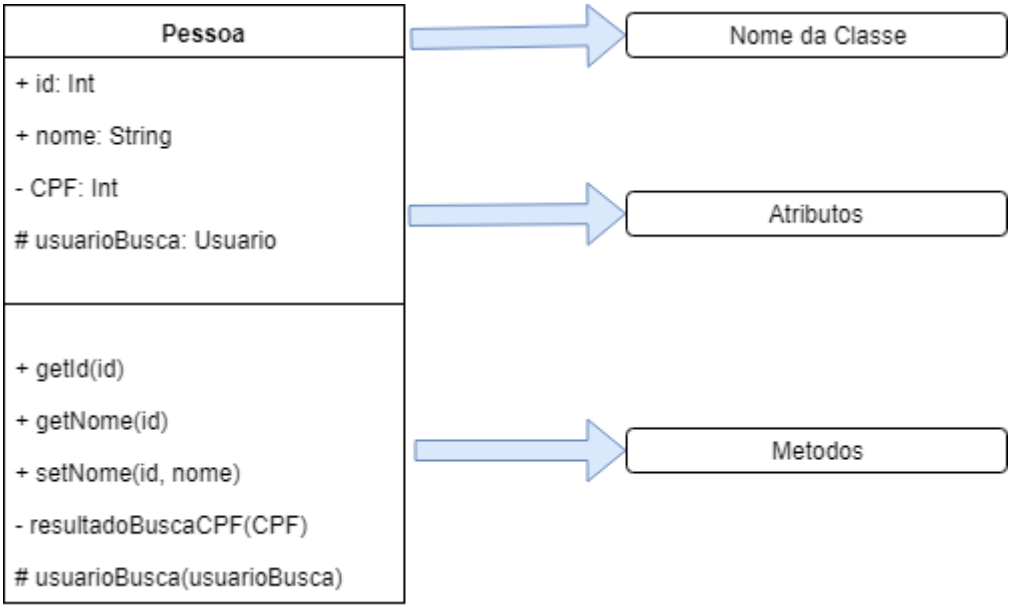
2.2.4 OBJETOS

Objeto é uma entidade ou conceito existente no contexto de modelagem do mundo real. É relevante incorporar os objetos no modelo de informações. É caracterizado por um conjunto de propriedades, comportamentos e identidade. As propriedades são as características que definem o objeto, transportadas para um conjunto de atributos cujos valores estabelecem o estado em que o objeto terá. O comportamento é definido como as operações que os objetos podem 7 efetuar. A identidade permite identificar um objeto em particular como único num conjunto de objetos semelhantes (NUNES; O'NEILL, 2004).

2.2.5 CLASSES

As classes são representações abstratas de um conjunto de objetos que compartilham uma mesma estrutura e comportamento. Na prática, um objeto é um caso particular de uma classe que também pode ser referido como uma instância da classe. A representação de uma classe na UML contém duas linhas que separam 3 partes, conforme a Figura 1. A primeira contém o nome da classe, a segunda os atributos da classe e a última os métodos da mesma (NUNES; O'NEILL, 2004; GUEDES, 2011).

Figura 1 Representação de uma Classe.



Na Figura 1 é possível observar o formato da representação de uma classe, sendo dividida em três partes: seu nome (Pessoa), seus atributos com tipo de visibilidade pública (id e nome) representado pelo símbolo "+", o atributo CPF com o tipo visibilidade privada representado pelo atributo "-" e o atributo (usuarioBusca) com o tipo de visibilidade protegida representada pelo atributo "#". Os seus métodos getId(id), setNome(id, nome) e getNome(nome) com tipo de visibilidade pública, o método resultadoBuscaCPF(CPF) com o tipo de visibilidade privado e o método usuarioBusca(usuarioBusca) com o tipo de visibilidade protegida.

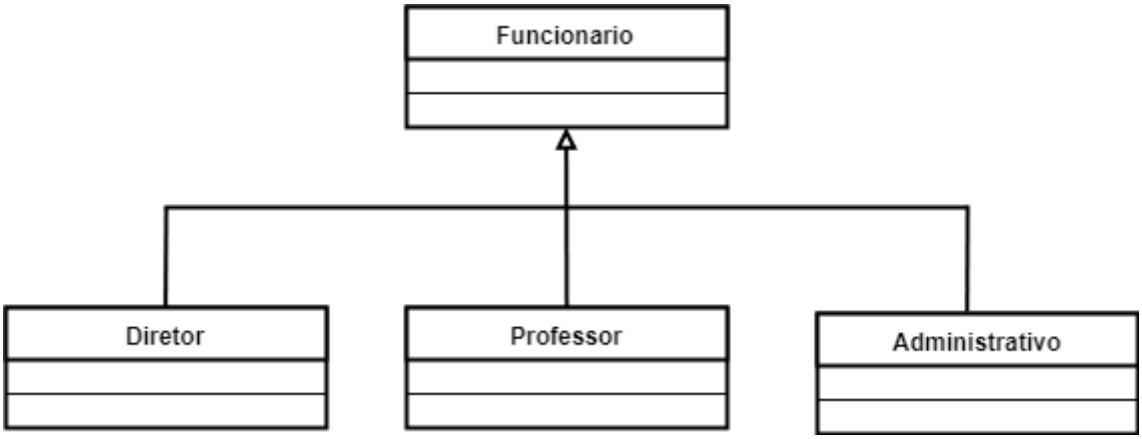
Em uma classe não é realmente obrigatório conter as três divisões, pois pode haver classes que não tenham atributos ou que não contenham métodos. Pode acontecer ainda de seus atributos e métodos não serem apresentados no diagrama. Segundo Guedes (2011) é recomendável apresentar somente os atributos relevantes ao diagrama para evitar que o diagrama fique cheio de informações que podem ser modificadas ou retiradas futuramente. Assim, é possível encontrar classes com somente duas divisões ou mesmo com apenas uma, no caso, aquela que contém a descrição da classe, por que esta é obrigatória (GUEDES, 2011).

2.2.6 HERANÇA

A herança é uma das características mais poderosas e importantes do design orientada a orientação a objetos. Isso é devido ao fato de a herança permitir o reaproveitamento de atributos e de métodos, otimizando o tempo de desenvolvimento de um *software*, além de permitir a diminuição das linhas de código produzidas pelo fato do reaproveitamento de código. Tal

característica facilita a manutenção dos códigos futuramente (GUEDES, 2011; NUNES; O'NEILL, 2004). Herança trabalha com os conceitos de superclasses e subclasses. Uma superclasse, também chamada de classe pai (ou classe mãe dependendo dos autores), é uma classe que contém classes derivadas a partir dela. Estas classes derivadas são chamadas de subclasses, ou também conhecidas como classes filhas conforme a Figura 2. As subclasses herdam todas as características da classe superior, ou seja, os atributos e métodos (GUEDES, 2011).

Figura 2 Representação de uma Herança.



Na Figura 2 apresenta um exemplo do relacionamento do tipo herança onde as classes Diretor, Professor e Administrativo herdando as informações da classe Funcionário.

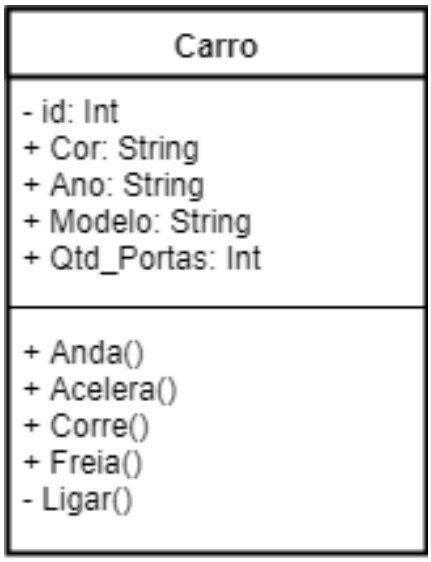
A herança permite declarar uma classe com atributos e métodos específicos e, a partir disso, derivar uma subclasse sem a necessidade de declarar os atributos e métodos previamente definidos. A subclasse herda os atributos e métodos automaticamente. Desta forma destaca-se a reutilização do código já declarado na superclasse. Assim, na subclasse só se deve preocupar em declarar os atributos ou métodos exclusivos da subclasse, tornando o processo de codificação do *software* mais ágil (GUEDES, 2011).

A herança permite também trabalhar com especializações. Podemos criar classes gerais, como características específicas compartilhadas por muitas classes, mas que tenham pequenas diferenças em sua estrutura. Assim, poderemos criar uma classe geral com as características comuns a todas as classes e diversas subclasses a partir da estrutura da subclasse, detalhando somente os comportamentos exclusivos das subclasses. Uma subclasse pode se tornar uma superclasse a qualquer momento, necessitando somente que uma subclasse derive seus atributos e métodos a partir da outra subclasse (GUEDES, 2011; FOWLER, 2005).

2.2.7 ATRIBUTOS E MÉTODOS

Classes costumam ter atributos que armazenam os dados dos objetos da classe. Também possuem métodos ou operações que são funções que uma instância da classe pode executar. Os valores dos atributos podem variar de uma instância para outra. Por conta dessa característica, é possível identificar cada objeto individualmente ao passo que os métodos são idênticos para todas as instâncias de uma classe conforme a Figura 3 (GUEDES, 2011).

Figura 3 Representação de Atributos e Métodos.



Na Figura 3 pode se observar como é representado os atributos e métodos e uma classe, onde se tem uma classe com o nome carro e seus atributos id, Cor, Ano, Modelo, Qtd_Portas. O atributo id tem visibilidade privada para as demais classes, o que é representado pelo símbolo de menos, e os demais atributos têm visibilidade pública para as demais classes, o que é representado pelo símbolo de mais. Os métodos Andar(), Acelerar(), Freia(), Ligar(). Sendo têm visibilidade pública e o método Ligar() tem sua visibilidade privada para as demais classes.

Embora os métodos sejam declarados no diagrama de classe identificando os possíveis parâmetros que são por eles recebidos e os possíveis valores por ele retornados, o diagrama de classe não se preocupa em definir em quais etapas e em quais métodos deverão percorrer quando forem chamados, sendo esta uma função atribuída a outros diagramas da UML (FOWLER, 2005; GUEDES, 2011).

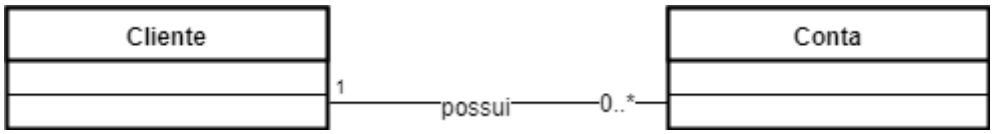
Métodos podem receber valores como parâmetros e retornar outros valores como resultado de sua execução. Segundo (GUEDES, 2011) "por padrão retorno de métodos podem ser feitos com valores de sucesso ou falso", por exemplo, uma classe conta executa o método de abertura de conta, se o valor retornado pelo método um objeto com os valores *true* e uma mensagem de sucesso significará que o método foi concluído com sucesso e que uma nova

conta foi aberta. Já se o retorno foi igual um objeto com valores *false* e uma mensagem de negação, sabemos que o método não foi concluído da forma esperada e que algum problema ocorreu no processo de abertura da conta (GUEDES, 2011).

2.2.8 RELACIONAMENTOS OU ASSOCIAÇÃO

As classes, em uma modelagem orientada a objetos, costumam ter relacionamento entre si (GUEDES, 2011). Tais relacionamentos são chamados de associações e permitem que as classes compartilhem informações e colaborarem para a execução dos processos de software (GUEDES, 2011). Uma associação é representa no diagrama de classe por uma linha ligando classes às quais pertencem os objetos relacionados (BEZERRA, 2015). As associações são representadas por linhas ligando uma classe a outra classe que está envolvida no relacionamento (GUEDES, 2011). Essas linhas podem ter nomes ou títulos para auxiliar a compreensão do tipo de vínculo estabelecido entre os objetos das classes envolvidas nas associações conforme a Figura 4 (GUEDES, 2011).

Figura 4 Representação de como e um Relacionamento ou Associação.



A Figura 4 representa como pode ser representado um relacionamento ou associação entre classe. É importante observar que, embora as associações sejam representadas entre classes do diagrama, elas representam na verdade ligações possíveis entre objetos das classes envolvidas (BEZERRA, 2015). Por exemplo, quando ligamos a classe Cliente e Conta, na Figura 4, isso significa que durante a execução do sistema, haverá a possibilidade de troca de mensagens entre os objetos destas classes (BEZERRA, 2015). Assim, os objetos dessas classes podem colaborar para execução de funções no software (BEZERRA, 2015).

2.2.9 MULTIPLICIDADES

Segundo Bezerra (2015) às associações permitem a representação da informação dos limites inferiores e superiores da quantidade de objetos aos quais os outros objetos possam estar associados a uma determinada classe. Esses limites são chamados de multiplicidade na linguagem UML (BEZERRA, 2015). Cada associação em um diagrama de classe possui duas multiplicidades, uma em cada extremo da linha que a representa em sua associação (BEZERRA, 2015). A multiplicidade pode ser representada por símbolos conforme apresentados na Tabela 1. Simbologia para representar multiplicidades (NUNES; O'NEILL, 2004).

Tabela 1. Simbologia para representar multiplicidades

Nome	Simbologia
Apenas Um	1
Zero ou Muitos	0..*
Um ou Muitos	1..*
Zero ou Um	0..1

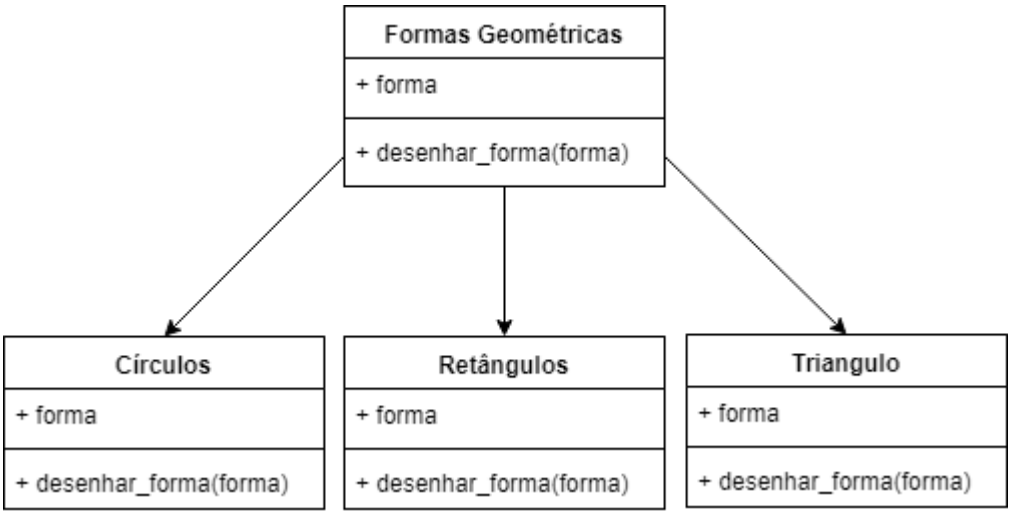
Fonte: adaptado de (BEZERRA, 2015).

Podemos observar na tabela 1 que a UML define mais de um símbolo para representar a mesma multiplicidade (BEZERRA, 2015). O símbolo "1" é equivalente à utilização do símbolo "1..1". Outra equivalência ocorre entre os símbolos "*" e "0..*" (BEZERRA, 2015). A multiplicidade representada pelo símbolo "1" significa que apenas um objeto de uma classe poderá se relacionar com os outros objetos da outra classe (GUEDES, 2011). A multiplicidade representada pelo símbolo "*" significa que todo o objeto de uma classe poderá se relacionar com os outros objetos da outra classe que o relacionamento está firmado (BEZERRA, 2015; GUEDES, 2011).

2.2.10 POLIMORFISMO

O conceito de polimorfismo está associado à herança (GUEDES, 2011). O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface (BEZERRA, 2015). O polimorfismo se refere à capacidade de duas ou mais classes de objetos responderem à mesma mensagem, cada qual de seu próprio modo (BEZERRA, 2015). O exemplo clássico do polimorfismo são as formas geométricas conforme exemplificado na Figura 5 Exemplificação do Polimorfismo. (BEZERRA, 2015).

Figura 5 Exemplificação do Polimorfismo.



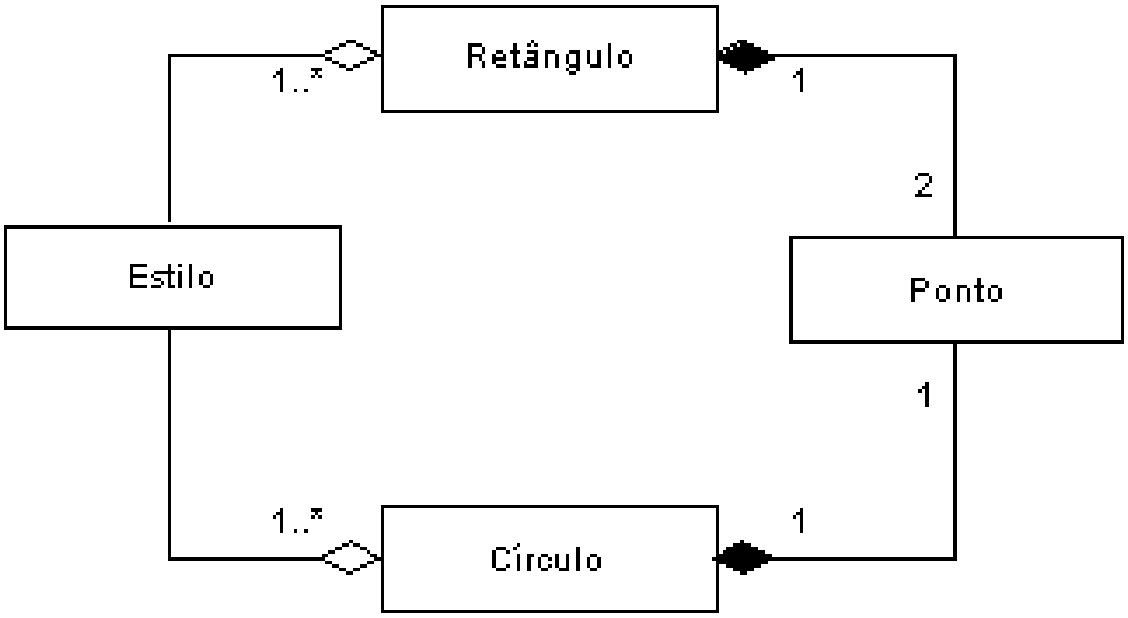
Na Figura 5 é possível observar como é a estrutura de um polimorfismo em uma classe de formas geométricas. Pensando em uma coleção de formas geométricas que contenha círculos, retângulos entre outras formas específicas (BEZERRA, 2015). Pelo princípio do polimorfismo, quando um trecho de código precisa desenhar os elementos daquela classe, essa classe não deve precisar conhecer os tipos específicos de figuras existentes; basta que cada elemento da classe receba uma mensagem solicitando que desenhe a si próprio (BEZERRA, 2015). Isso é possível porque esse trecho de código não precisa conhecer o tipo de cada figura. Ao mesmo tempo, essa região de código não precisa ser alterada quando, por exemplo, uma classe correspondente a um novo tipo de forma geométrica (uma reta, por exemplo) tiver que ser adicionada (BEZERRA, 2015).

Ainda segundo Bezerra (2015) podemos notar mais uma vez que, assim como no caso do encapsulamento, a abstração também é aplicada para obter o polimorfismo: um objeto pode enviar a mesma mensagem para objetos semelhantes, mas que implementam a sua interface de formas diferentes.

2.2.11 AGREGAÇÃO E COMPOSIÇÃO

A agregação no diagrama de classes pretende demonstrar o fato que um todo é composto por parte. Tornando a agregação um tipo especial de associação, onde se tenta demonstrar que as informações de um objeto, precisam ser complementadas pelas informações contidas em um ou mais objetos de outras classes. Esse tipo de associação pretende demonstrar uma relação todo/parte entre os objetos associados. E o símbolo de agregação difere do de associação por conter um losango na extremidade da classe que contém os objetos-todo conforme na Figura 6 Representação de uma Agregação e Composição.(NUNES; O'NEILL, 2004).

Figura 6 Representação de uma Agregação e Composição.



Fonte: Repositório Digital da UFCG

A composição é uma agregação com um significado mais forte existindo uma dependência direta entre as duas classes (se a parte deixa de existir, o todo também deixará de existir). Normalmente, a multiplicidade no lado do todo não ultrapassa o 1, o que não acontece com a agregação (NUNES; O'NEILL, 2004). Uma composição constitui-se em uma variação da agregação, onde é apresentado um vínculo mais forte entre os objetos-todo e os objetos-parte, procurando demonstrar que os objetos- partes tem de estar associados a um único objeto-todo. Em uma composição os objetos-partes não podem ser destruídos por um objeto diferente do objeto-todo ao qual está relacionado. O símbolo de composição diferencia-se graficamente do símbolo de agregação por utilizar um losango preenchido conforme na Figura5 (GUEDES, 2011).

2.2.12 ENCAPSULAMENTO

Segundo Hamilton e Miles (2006) o encapsulamento de operações e dados dentro de um objeto é provavelmente a parte mais poderosa e útil da abordagem orientada a objetos para o design de softwares. O mecanismo de encapsulamento é uma forma de restringir o acesso ao comportamento interno de um objeto (BEZERRA, 2015).

Na orientação a objetos temos o conceito que um objeto contém ou encapsula seus dados e os métodos trabalham em cima destes dados (HAMILTON; MILES, 2006). Referindo-se à essa analogia uma classe fictícia cafeteira, pode encapsular os métodos nivel_agua, ligar, desligar, preparar_cafe e, provavelmente, alguns elementos elétricos que ninguém deveria mexer (HAMILTON; MILES, 2006). Os métodos desta classe, como o nivel_agua, são acessíveis a todas as outras classes e outros atributos, como os elétricos estão escondidos para

as demais classes (HAMILTON; MILES, 2006). Além desses atributos essa classe cafeeira poderá conter métodos que permitirá outras classes realizam operação em cima desses atributos elétricos, contendo no mínimo uma classe prepararCafe (HAMILTON; MILES, 2006).

2.3 Quality Model Object Oriented Design

Qualidade de software é um termo complexo e muito abrangente. A avaliação da qualidade de software varia de pessoa para pessoa, dependendo do ponto de vista de cada uma. Os pontos de podem ser descritas por cincos pontos de vistas diferentes. Visão transcendental, visão do usuário, visão do fabricante, visão do produto e visão baseada em valor. A visão transcendental sugere que a qualidade é algo que se reconhece imediatamente, mas não consegue se definir. A visão do usuário visa a qualidade tendo como objetivo atender a um usuário final. A visão do fabricante vê a qualidade como o grau de atendimento às especificações do produto. A visão do produto está ligada às características do produto. E a visão baseada em valor vê a qualidade como um valor que o cliente estaria disposto a pagar por um produto. Onde cada uma das visões descreve um tipo de qualidade diferente para um software (PRESSMAN, 2011).

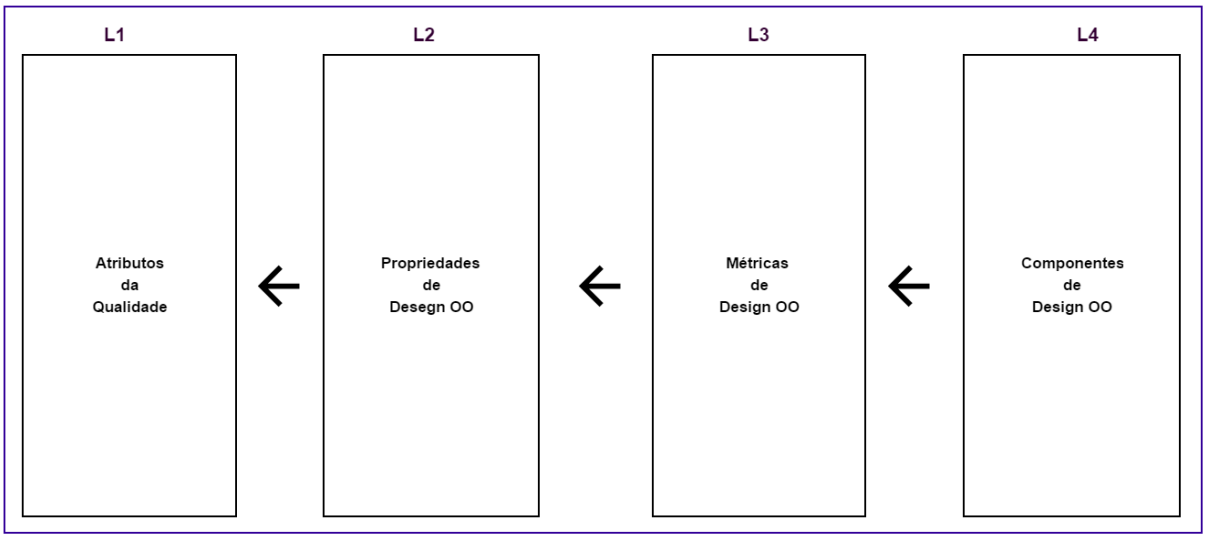
A medição de qualidade de um software está ligada ao valor da qualidade do componente, sistema ou processo de software. Ao comparar esses valores pode se tirar conclusões de qualidade do que se está sendo desenvolvido. O objetivo a longo prazo do processo de medições/métricas e substituir as revisões dos produtos/serviços realizados. Utilizando uma ferramenta que poderá identificar, avaliar os produtos/serviços produzidos usando uma série de métricas de qualidade podendo inferir o nível de qualidade dos produtos/serviços realizados (SOMMERVILLE, 2010).

Muitas das métricas e modelos de qualidade de softwares orientados a objetos disponíveis na literatura podem ser aplicada somente em produtos que estarão no estágio final ou quase final. Isso por que as métricas dependem de informações extraídas da implementação do produto. Isso fornece as informações muito tarde para ajudar a melhorar a características internas dos produtos gerados antes mesmo da sua conclusão. Assim, existe a necessidade de gerar métricas de qualidades dos modelos que possam ser aplicados no início do projeto (JABANGWE et al., 2015; BANSIYA; DAVIS, 2002).

Segundo BERTRAN (2009) O modelo QMOOD Bansiya e Davis (2002) é uma abordagem para estimativa e avaliação dos atributos de qualidade de *design* OO. A estrutura pressente neste modelo e igual à maioria dos modelos da qualidade já desenvolvidos, tendo sua forma de hierarquia ou de árvore, os níveis superiores representam os atributos externos da qualidade do *design* que se deseja medir, como a funcionalidade e compressibilidade. Estes atributos externos são compostos por um ou mais atributos internos, os quais, no contexto do modelo QMOOD

representam as propriedades do *design* OO. O modelo QMOOD é composto em quatro níveis (L1 a L4) os quais são apresentados na Figura 7 Níveis e relações no QMOOD..

Figura 7 Níveis e relações no QMOOD.



Fonte: Adaptado de Bansiya (2002).

2.3.1 Atributos da qualidade (L1)

No nível inicial (L1) estão os atributos externos da qualidade considerada no modelo QMOOD. Estes atributos se baseiam no conjunto de atributos da norma (ISO/IEC 9126, 1991) que são "Funcionalidade", "Confiabilidade", "Eficiência", "Usabilidade", "Manutenibilidade" e "Portabilidade". Como esta é uma norma definida para avaliar a qualidade de um *software*, alguns de seus atributos, pela sua natureza, não podem ser quantificados utilizando apenas as informações disponíveis no *design* (BERTRAN, 2009).

Tabela 2. Definição dos Atributos de Qualidade.

Atributo de qualidade	Definição
Reutilização	Reflete a replicação de um componente, script ou método de uma parte de um software ou outro produto possa ser utilizando em outro software.
Flexibilidade	A facilidade com que um sistema ou componente possa ser modificado para uso em aplicações ou ambientes diferentes daquelas para o qual foi especificadamente projeto.
Compreensibilidade	A facilidade com que as propriedades sejam facilmente entendidas e compreendidas. Isso está diretamente atrelado a complexidade da estrutura do projeto.
Funcionalidade	Refere- se as responsabilidades atribuídas às classes do projeto, que são disponibilizadas pelas classes através de suas interfaces públicas.

Extensibilidade	Refere-se a capacidade de um software ou propriedade de um produto existente permita-se incorporar em novos requisitos no projeto.
Eficácia	Refere-se o grau de capacidade em que o design é capaz de atingir o funcionamento e comportamento desejados usando conceitos de design OO e suas técnicas.

Fonte: Adaptado de Bansiya (2002); Bertrán (2009).

Em seu artigo Bansiya e Davis (2002) substituíram os termos "Portabilidade por "extensibilidade", que define melhor essa característica nos projetos. Da mesma forma o termo "eficiência" foi substituído por "Eficácia", que descreve melhor essa qualidade para projetos. O termo "sustentabilidade" também implica a existência de um produto de *software* e foi substituído por "Compreensibilidade", que se concentra mais nas características do projeto. Então os atributos de qualidade ficaram os seguintes: funcionalidade, eficácia, facilidade de compreensão, facilidade de extensão, usabilidade e flexibilidade apresentados na Tabela 3. Descrição das Métricas de Design. (BANSIYA; DAVIS, 2002).

2.3.2 Propriedade de *Design* OO (L2)

As propriedades de *design* são conceitos tangíveis que podem ser avaliados diretamente examinando-se a estrutura interna e externa do *design* e as relações dos componentes de *design* como: classes, métodos e seus respectivos atributos (BERTRAN, 2009). As propriedades abstrações, encapsulamento, acoplamento, coesão, complexidade e tamanho do projeto são usadas livremente como representativos das características de qualidade do projeto tanto no desenvolvimento estrutural quanto no orientado a objetos (BANSIYA; DAVIS, 2002; BERTRAN, 2009). As hierarquias de mensagens, polimorfismo, herança e composição. Sendo assim, as propriedades do *design* OO utilizadas no QMOOD são as seguintes: tamanho do design, hierarquia, coesão, abstração, acoplamento, herança, polimorfismo, composição, troca de mensagens entre classe e complexidade apresentados na tabela 3 (SHATNAWI; LI, 2011; BANSIYA; DAVIS, 2002).

2.3.3 Métricas de *Design* OO (L3)

Cada uma das propriedades de design identificadas no modelo QMOOD de Bansiya e Davis (2002) representa um atributo ou característica de um *design* que podem ser objetivamente quantificadas mediante o uso de métricas de *design* durante a fase desenho do *software*. Para projetos que são orientados a objetos, essas informações devem incluir a definição de classes, hierarquias, todos as categorias de parâmetros e declarações de atributos(BERTRAN, 2009; BANSIYA; DAVIS, 2002).

Tabela 3. Descrição das Métricas de Design.

Métrica	Nome	Descrição
DSC	Número de classes no design	Conta o número total de classe no design.
NOH	Número de Hierarquias	Conta a quantidade de hierarquias de classes no design.
ANA	Média de Ancestrais	Representa a média de classes das quais a classe avaliada herda informação. É calculado utilizando o número total de classes na estrutura de herança.
DAM	Métrica de acesso a dados	Calcula a proporção que representam os atributos privados (protegidos) do total de atributos da classe (faixa 0 - 1).
DCC	Acoplamento direto classes	Conta a quantidade de classes com as quais a classe avaliada está diretamente relacionada. Na contagem são incluídas as declarações de atributos e parâmetros.
CAM	Coesão entre os métodos da classe	Essa métrica calcula a relação entre os métodos de uma classe com base na lista de parâmetros dos métodos da classe Bansiya et al. (1999). Primeiramente é calculado o conjunto máximo independente dos tipos dos parâmetros de todos os métodos. Depois é computada a soma da interseção dos tipos dos parâmetros de cada um dos métodos com o conjunto independente. Finalmente, a soma é dívida pelo número total de parâmetros multiplicado pela quantidade de métodos. O valor obtido e um intervalor entre 0 e 1, o valor 1 representa a coesão máxima e 0 representa uma classe completamente não coesiva.
MOA	Medida de agregação	Representa a composição de uma classe utilizando os atributos declarados. É computada contanto o número de atributos cujo o tipo é definido pelo usuário.
MFA	Abstração funcional	Calcula à proporção que representam os métodos herdados do total de métodos da classe (faixa 0 – 1).
NOP	Número de métodos polimórficos	Conta a quantidade de métodos que podem ter comportamentos polimórficos.
CIS	Tamanho de uma classe	Conta a quantidade métodos públicos que a classe possui. Não são considerados nesta métrica os atributos públicos.
NOM	Quantidade de métodos	Conta a quantidade de métodos declarados na classe.

Fonte: Adaptado de Bansiya (2002); Bertrán (2009)

Uma pesquisa das métricas de projeto existentes Bansiya (1997) revelou que existem várias métricas que podem ser modificadas e usadas na avaliação de algumas propriedades do projeto, como abstração, mensagens e herança. No entanto, existem várias outras propriedades de design, como encapsulamento e composição, para as quais não existem métricas de design

orientadas a objeto. Além disso, embora as métricas para avaliar a complexidade, a coesão e o acoplamento já tenham sido definidas, essas métricas exigem uma implementação quase por completas das classes do projeto antes de poder ser calculadas, portanto, não podem ser extraídas na fase inicial utilizando o QMOOD. Isso levou à definição de cinco métricas: métrica de acesso a dados (DAM), métrica de acoplamento de classe direta (DCC), coesão entre métodos de métrica de classe (CAM), métrica de agregação (MOA) e métrica de abstração funcional (MLA) apresentadas na Tabela 3 (BANSIYA; DAVIS, 2002; BERTRAN, 2009).

2.3.4 Componente de *Design* OO (L4)

Os componentes de *design* OO do modelo QMOOD que definem a arquitetura de um *design* orientado a objeto são os seguintes: classe, atributo, conjunto de operações (métodos), as relações entre classe composição, herança e as hierarquias de classe. O modelo QMOOD torna explicitas as relações entre cada um dos 4 níveis, oferecendo tabelas de mapeamento (Tabela 3 e equações (Tabela 4)Cada um dos níveis é interligado entre si. Onde cada propriedade de design do segundo nível e relacionada com as métricas do terceiro nível assim podendo quantificá-las (BANSIYA; DAVIS, 2002).

Tabela 4. Métricas de Design para propriedades de Design.

Propriedades do *Design*	**Métricas do Design**
Tamanho do *design*	Número de classes no *design* (DSC)
Hierarquias	Números de hierarquias de classes (NOH)
Abstração	Média de ancestrais ou superclasses (ANA)
Encapsulamento	Métrica de acesso a dados (DAM)
Acoplamento	Acoplamento direto entre as classes (DCC)
Coesão	Coesão entre os métodos da classe (CAM)
Composição	Medida da agregação (MOA)
Herança	Medida de abstração funcional (MFA)
Polimorfismo	Número de métodos polimórficos (NOP)
Troca de mensagens	Tamanho da interface de uma classe (CIS)
Complexidade	Número de métodos (NOM)

Fonte: Adaptado de Bansiya (2002).

A correspondência entre o primeiro e o segundo nível se torna mais explicita ainda, pois, cada um dos atributos da qualidade é o resultado da soma do valor de cada uma das propriedades do design associados a ele. Os pesos podem assumir valores negativos ou positivos dependendo da relevância das propriedades do *design* em relação ao atributo de qualidade (BERTRAN, 2009).

Tabela 5. Fórmulas para Calcular os Atributos de Qualidade.

Atributos de Qualidade	Equação de Cálculo do Índice
Reutilização	((-0.25 * acoplamento) + (0.25 * coesão) + (0.5 * mensagens) + (0.5 * tamanho do design))
Flexibilidade	((0.25 * encapsulamento) - (0.25 * acoplamento) + (0.5 * composição) + (0.5 * polimorfismo))
Compreensibilidade	((-0.33 * abstração) + (0.33 * encapsulamento) – (0.33 * acoplamento) + (0.33 * coesão) – (0.33 * polimorfismo) – (0.33 complexidade) – (0.33 * tamanho do design))
Funcionalidade	((0.12 * coesão) + (0.22 * polimorfismo) – (0.22 * mensagens) + (0.22 * tamanho do design) + (0.22 * hierarquias))
Extensibilidade	((0.5 * abstração) + (0.5 * encapsulamento) + (0.5 * composição) + (0.5 herança) + (0.5 * polimorfismo))
Eficácia	((0.2 * abstração) + (0.2 * encapsulamento) + (0.2 * composição) + (0.2 * herança) + (0.2 * polimorfismo))

Fonte: Adaptado de Bansiya (2002).

O QMOOD é um modelo hierárquico para avaliação da qualidade de atributos em projetos que utiliza como modelo a orientação a objeto, como reutilização, flexibilidade, compreensibilidade, funcionalidade, extensibilidade e eficácia. Os valores obtidos após os cálculos dos atributos são obtidos de acordo com as equações apresentadas na Tabela 5.As equações consideram um conjunto de 11 propriedades de *design* que são avaliadas usando algumas métricas de qualidade de acordo com a tabela 3(MARIANI; VERGILIO, 2017).

3 MATERIAIS E MÉTODOS

Esta seção apresenta os materiais e métodos essenciais para o desenvolvimento deste trabalho.

3.1 Materiais

Para elaboração do referencial teórico deste trabalho foram utilizados como fontes bibliográficas: artigos, dissertações, livros, manuais técnicos e páginas web de ferramentas de desenvolvimento.

Para o desenvolvimento do *parser* foi utilizada a ferramenta *Jison*, que é uma API *open source* para criação de *parser* em JavaScript. Essa ferramenta utiliza uma gramática livre de contexto como entrada e gera um arquivo JavaScript capaz de analisar a linguagem descrita por essa gramática (CARTER, 2009). No âmbito deste trabalho, o *parser* foi responsável por realizar a leitura de um diagrama de classe no formato do PlantUML e retornar a estrutura desse diagrama em um formato específico. O PlantUML é uma linguagem de desenho de diagramas UML (PlantUML, 2018), e foi responsável pela criação dos diagramas de classe utilizados neste trabalho para geração das métricas

Com intuito de desenvolver o *front-end* (sistema no qual um usuário poderá inserir um diagrama de classes para análise de métricas de qualidade) foram utilizas a ferramentas de desenvolvimento web: HTML5, CSS3 e JavaScript.

3.2 Metodologia

Para o desenvolvimento deste trabalho foram realizadas quatro etapas, sendo elas: levantamento de requisitos, desenvolvimento do *parser*, desenvolvimento do software e validação das métricas e dos cálculos obtidos pela ferramenta. A Figura 8 Metodologia.ilustra a metodologia utilizada no trabalho.

Figura 8 Metodologia.

Inicialmente, foi realizado o levantamento de requisitos, que consistiu na análise do processo de obtenção de métricas de qualidade no modelo QMOOD conforme foi apresentado nas Tabela 3, Tabela 4 e Tabela 5. A partir da análise realizada foram obtidos quatorze requisitos (Tabela 6).

Posteriormente, foi iniciado o desenvolvimento do *parser*. O desenvolvimento do *parser* consistiu na definição da gramática para realizar a análise do diagrama de classe passado pelo usuário.

Após o desenvolvimento do *parser*, foi desenvolvido o software de acordo com suas funcionalidades, sendo elas: análise dos dados do *parser*, geração das métricas de qualidade e cálculo das métricas.

A última etapa envolveu a realização de testes das métricas fornecidas pelo software. Para isso foi criado um diagrama de classes especificamente para este fim, que possui representações de situações de interesse, como herança. Este diagrama foi analisado manualmente, realizando o mesmo procedimento implementado no software para obtenção das métricas. Por fim, os resultados foram comparados para verificar se o software estaria gerando resultados corretamente.

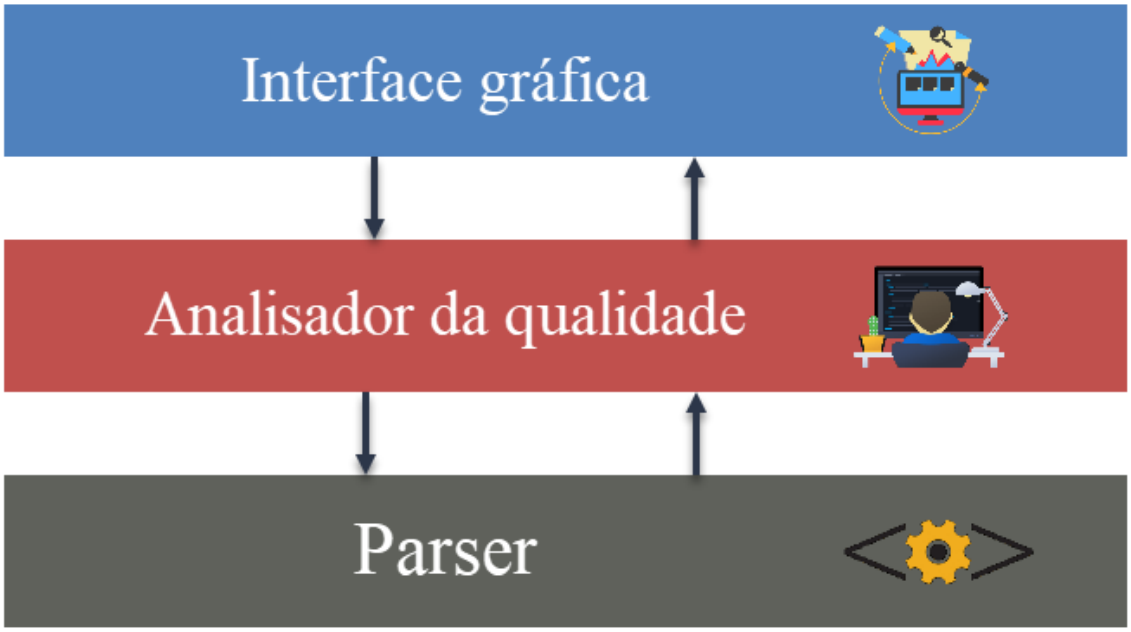
4 RESULTADOS

Essa sessão apresenta os resultados obtidos no desenvolvimento do software proposto neste trabalho.

4.1 Visão Geral

A arquitetura do *software* desenvolvido nesse trabalho é dividida em três etapas, conforme apresentas na Figura 9.

Figura 9 Arquitetura do software



A **interface gráfica** disponibiliza um editor de texto para o usuário informar o diagrama de classe em formato do PlanUTML e realiza a apresentação das métricas de qualidade obtidas a partir da análise do diagrama.

O **analisador da qualidade** tem seu comportamento dividido em duas etapas. Na primeira etapa o analisador recebe o diagrama de classe fornecido pelo usuário na interface gráfica e transmite a informação ao *parser*. Na segunda etapa gera as onze métricas de qualidade do modelo QMOOD, a partir dos dados recebidos do *parser*, realiza os cálculos das métricas de qualidade e, por fim, apresenta o resultado na interface gráfica.

O ***parser*** recebe o diagrama do **analisador da qualidade**, verifica se está conforme o padrão da linguagem *PlantUML*, o traduz em um modelo de objeto que representa o diagrama e, por fim, retorna o modelo de objeto para o **analisador da qualidade**.

As seções a seguir apresentam resultados do desenvolvimento da solução proposta, começando pelos artefatos de desenvolvimento de software.

4.2 Artefatos do desenvolvimento do software

Esse capítulo apresenta os artefatos do desenvolvimento do software deste trabalho.

4.2.1 Listas de requisitos

A Tabela 6. Requisitos representa a lista de requisitos do software desenvolvido no presente trabalho.

Tabela 6. Requisitos

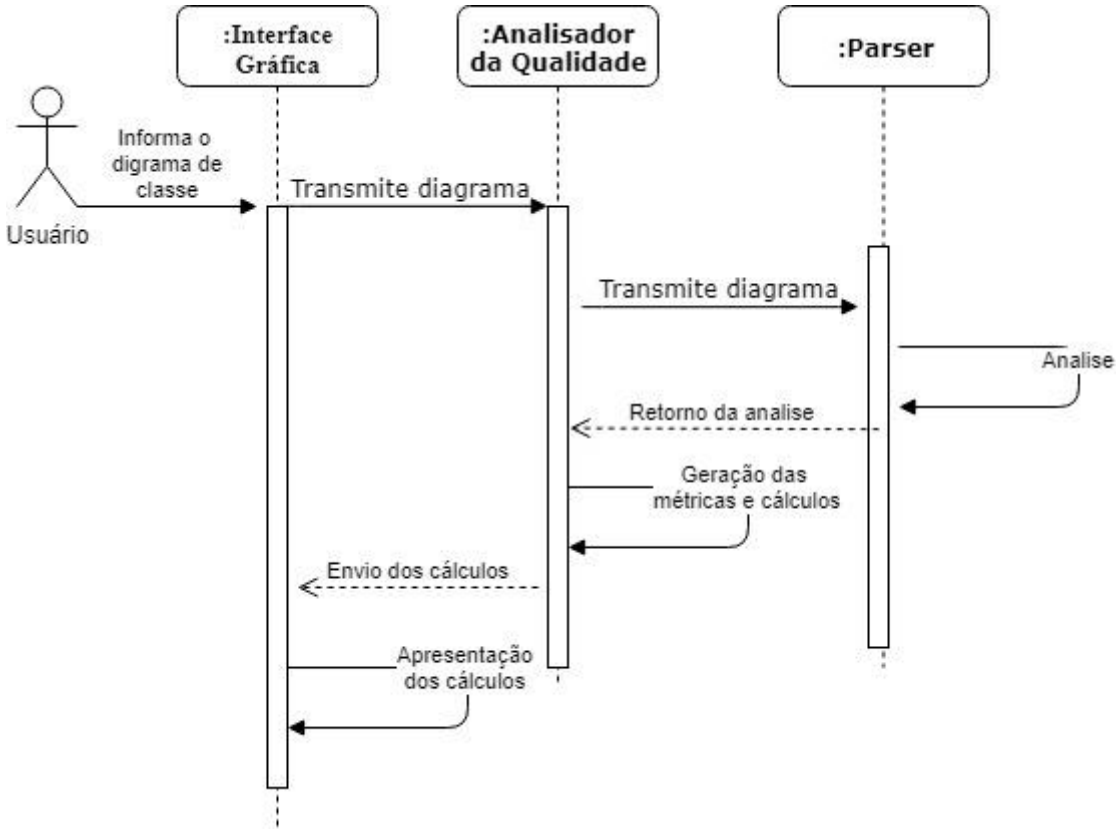
Número	Requisitos
R1	Obter a quantidade de classes
R2	Obter a quantidade de relacionamentos do tipo herança
R3	Para cada classe, obter a quantidade de superclasses (herança)
R4	Para cada classe, obter a lista de superclasses
R5	Para cada classe, obter a quantidade de atributos
R6	Para cada classe, obter a quantidade de atributos protegidos
R7	Para cada classe, obter a quantidade de atributos públicos
R8	Para cada classe, obter a quantidade de atributos privados
R9	Para cada classe, obter a quantidade de métodos
R10	Para cada classe, obter a quantidade de métodos protegidos
R11	Para cada classe, obter a quantidade de métodos públicos
R12	Para cada classe, obter a quantidade de métodos privados
R13	Para cada classe, obter a quantidade de métodos herdados (considerando toda a hierarquia de herança a partir da classe)
R14	Obter a quantidade de métodos polimórficos

Estes requisitos foram implementados para alcançar os objetivos propostos para o trabalho.

4.2.2 Diagrama de Sequencia

Esse tópico apresenta a sequência de execução do software deste trabalho representado na Figura 10.

Figura 10 Diagrama de sequência do software



A execução do software tem o início pelo pela informação do diagrama de classe pelo usuário na interface gráfica. A interface gráfica transmite o diagrama para o analisador de qualidade, o analisador de qualidade recebe o diagrama e enviar para o *parser*. O *parser* analisa as informações do diagrama e retorna um modelo de objetos para o analisador de qualidade.

O analisador de qualidade gera as métricas de qualidade do diagrama conforme o modelo QMOOD, com o termino das gerações da métricas de qualidade, realizado os cálculos das métricas e por fim envia o resultado dos cálculos para a interface gráfica. A interface gráfica apresenta o resultado dos cálculos das métricas de qualidade do diagrama informado pelo usuário.

4.3 Parser para o PlantUML

O PlantUML representa o diagrama de classe da UML de uma forma textual, utilizando determinadas sintaxes para cada recurso:

- classe, seus atributos e métodos;
- atributo, com visibilidade, nome e tipo;
- método, com visibilidade, lista de parâmetros (com nome e tipo) e tipo de retorno;
- diferentes tipos de relações entre classes (herança, agregação, composição e associação)

A sintaxe para representar uma classe é:

```
class <Nome>
```

onde **<Nome>** é o nome da classe. Exemplo:

```
class Pessoa
```

Os atributos e métodos da classe estão entre os símbolos **{** e **}**.

A sintaxe para representar atributo de uma classe é:

```
<Visibilidade><Tipo> <Nome>
```

onde:

- **<Visibilidade>** é a visibilidade do atributo (e também vale para os métodos), da seguinte forma: **-** (sinal de subtração) representa atributo privado; **#** (sinal de cerquilha) representa atributo protegido; **~** (sinal de til) representa atributo privado no pacote; **+** (sinal de adição) representa atributo público;
- **<Tipo>** é seu tipo de dados – não há um conjunto específico de tipos, os quais podem ser definidos conforme cada contexto; e
- **<Nome>** é o nome do atributo.

Exemplo:

```
+nome : String
```

A sintaxe para representar método de uma classe é:

```
<Visibilidade><Tipo> <Nome>(<Parâmetros>)
```

onde:

- <**Visibilidade**> é a visibilidade do método (idem visibilidade do atributo);
- <**Tipo**> é o tipo de retorno do método;
- <**Nome**> é o nome do método;
- <**Parâmetros**> é a lista dos parâmetros do método, no formato **<Tipo> <Nome>**.

Exemplo:

```
+string getNome()
```

A sintaxe para representar herança entre duas classes é:

```
<Superclasse> <|-- <Subclasse>[ : <Rótulo>]
```

onde:

- **<Subclasse>** representa o nome da subclasse (que herda da superclasse);
- **<Superclasse>** representa o nome da superclasse (classe que é estendida); e
- **[: <Rótulo>]** é opcional e representa o rótulo da herança.

Exemplo:

```
Pessoa <|-- Funcionario : é
```

A sintaxe para representar composição entre duas classes é:

```
<Todo> [<CardinalidadeTodo>] *-- [<CardinalidadeParte>] <Parte>[ : <Rótulo>]
```

onde:

<Todo> representa a classe do lado "todo" da composição;

<Parte> representa a classe do lado "parte" da composição;

[<CardinalidadeTodo>] é opcional e representa a cardinalidade do relacionamento no lado "todo"; o valor deve estar entre aspas duplas;

[<CardinalidadeParte>] é opcional e representa a cardinalidade do relacionamento no lado "parte"; o valor deve estar entre aspas duplas; e

[: <Rótulo>] é opcional e representa o rótulo da composição.

Exemplo:

```
Pessoa "1" *-- "1" Cabeca : tem
```

A sintaxe para representar agregação entre duas classes é:

```
<Todo> [<CardinalidadeTodo>] o-- [<CardinalidadeParte>] <Parte>[ : <Rótulo>]
```

onde **<Todo>**, **<Parte>**, **[<CardinalidadeTodo>]**, **[<CardinalidadeParte>]** e **[: <Rótulo>]** representam o mesmo que na composição – com a particularidade do uso de cada um deles no contexto.

Exemplos:

```
Veiculo "1" o-- "1" Motor : tem
Veiculo "1" o-- "*" Porta : tem
```

A sintaxe para representar associação entre duas classes é:

```
<ClasseA> [<CardinalidadeClasseA>] -- [<CardinalidadeClasseB>] <ClasseB>[ :
<Rótulo>]
```

onde:

- **<ClasseA>**: representa a classe do lado esquerdo do relacionamento;
- **[<CardinalidadeClasseA>]** é opcional e representa a cardinalidade no lado esquerdo do relacionamento;
- **[<CardinalidadeClasseB>]** é opcional e representa a cardinalidade no lado direito do relacionamento;
- **<ClasseB>** representa a classe do lado direito do relacionamento; e
- **[: <Rótulo>]** é opcional e representa o rótulo do relacionamento.

Exemplos:

```
Veiculo -- Vendedor : vendido por
Veiculo -- Cliente : adquirido por
```

Com base nestas informações de sintaxe para utilizar os recursos do PlantUML a gramática do *parser* foi definida para representar as regras léxicas (para reconhecer símbolos) e sintáticas (para reconhecer sintaxe). A Figura 11 ilustra parte da gramática do *parser*.

Figura 11 Trecho da Gramática do Parser

```
classe_conteudo
    : classe_atributo
    | classe_metodo
    | {{$$ = [];}}
    ;

modificador_visibilidade
    : PRIVATE
      {{$$ = 'private';}}
    | PROTECTED
      {{$$ = 'protected';}}
    | PUBLIC
      {{$$ = 'public';}}
    |
      {{$$ = 'public';}}
    ;

classe_atributo
    : modificador_visibilidade ID ID (NEWLINE | NL) classe_conteudo
      {{$$ = prependChild($5, {nodeType:'classe_atributo',tipo:$2,nome:$3,visibilidade: $1});}}
    | ID DOISPONTOS ID (NEWLINE | NL) classe_conteudo
      {{$$ = prependChild($5, {nodeType:'classe_atributo',tipo:$1,nome:$3});}}
    ;
```

A Figura 11 apresenta parte do arquivo JISON para representar a gramática do *parser*. O código ilustrado representa a sintaxe da classe, do modificador de visibilidade e do atributo de classe:

- **classe**: contém atributos, métodos ou pode estar vazia;
- **modificador de visibilidade**: pode ser PRIVATE (símbolo -), PROTECTED (símbolo #), PUBLIC (símbolo +), sendo PUBLIC, por padrão

- **atributo de classe**: contém modificador de visibilidade, tipo, identificador (nome) seguido ou não de quebra de linha

O *parser* analisa a estrutura de uma entrada (no formato do PlantUML) e extrai informações importantes para a geração das métricas do diagrama de classes. Para exemplificar, a Figura 12 apresenta um exemplo de diagrama de classe em PlantUML.

Figura 12 Exemplo de Classe do PlantUML

```
class Dummy {
 +String data
 -Int idade
}
```

O *parser* analisa o texto do diagrama de classes da Figura 12 e gera as seguintes informações sobre o código:

- classe: **Dummy**
- atributos:
  - **data**: tipo **String**, visibilidade **public** (**+**)
  - **idade**: tipo **Int**, visibilidade **private** (**-**)

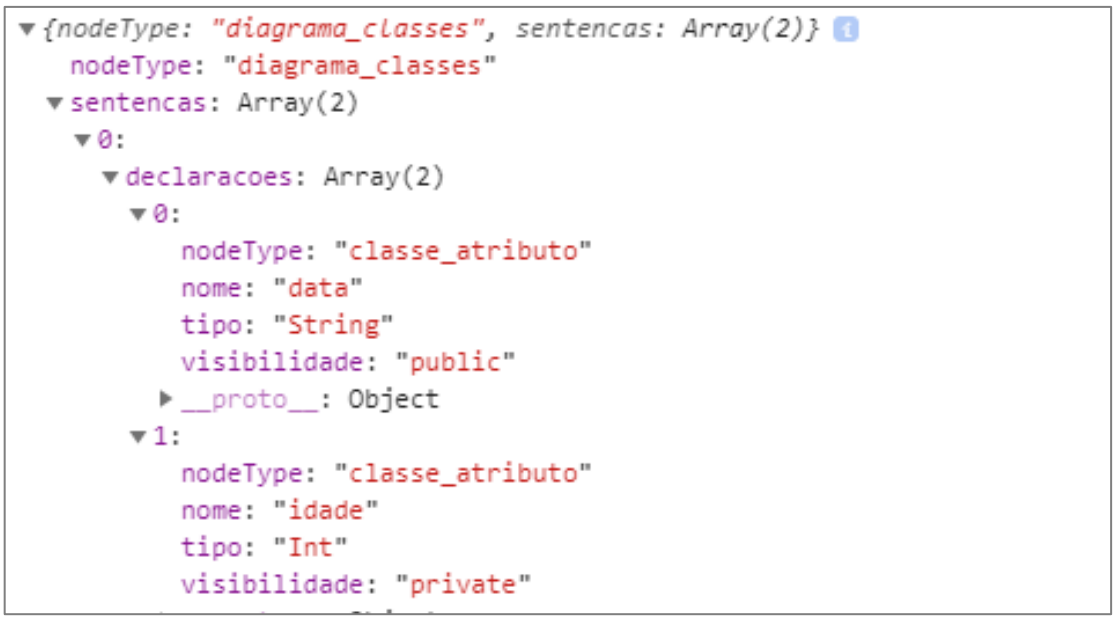
O *parser* organiza as informações sobre o diagrama na forma de um modelo de objetos que representa uma estrutura de dados em árvore, composta por nós. Cada nó é um objeto com atributos específicos conforme o dado que está sendo representado, conforme apresenta a Tabela 7.

Tabela 7. Exemplo do retorno do parser

Nó	**Atributos**
Classe	nome, tipo e corpo
Corpo	Nome, tipo atributo, visibilidade e tipo conteúdo da classe
Nome	Nome que o usuário informou para o atributo
Tipo atributo	Tipo do atributo informado pelo usuário
Visibilidade	Visibilidade que o usuário informou para o atributo podendo ser: **público**, **privado** e **protegido**
Tipo conteúdo da classe	Conteúdo da classe podendo ser **atributo** ou **método**, conforme informado pelo usuário

A Figura 13 representa o retorno do *parser* para o código apresentado na Figura 12

Figura 13 Retorno do parser



O *parser* retorna ao analisador de qualidade o modelo de objetos conforme apresentado na Figura 13. O objeto retornado tem a seguinte estrutura:

- Tipo do diagrama: sendo diagrama de classes.
- Declarações: estrutura do corpo da classe contendo atributos e métodos.

O array de declarações contém o corpo da classe representada na Figura 12 que contém dois atributos que ação referenciados pela varia **nodeType** dentro do array de declarações conjunto ao tipo do dado referenciado pelo variável **tipo**, o nome da classe representados pelo variável **nome** e o tipo de visibilidade do atributo representado pela variável **visibilidade**.

4.4 Analisador da Qualidade

O analisador da qualidade realiza sua função em duas etapas. Na primeira etapa ele recebe o diagrama informado pelo usuário transforma o diagrama no formato de objeto e encaminha para o *parser*. Na segunda etapa o analisador recebe o resultado gerado pelo *parser* e dá o início da geração as onze métricas de qualidade do modelo QMOOD. Para realizar a geração das métricas de qualidade foram criadas funções para a realização das mesmas, como apresentam as seções a seguir.

4.4.1 Métrica de acesso a dados (DAM)

O Código-fonte 1 representa a métrica DAM do modelo QMOOD.

Código-fonte 1: Trecho do método DAM

```
1  function DAM(dados) {
2      var atributos_classe = [];
```

```
3       for (var i = 0; i < dados.sentencas.length; i++) {
4           if(typeof dados.sentencas[i] === 'object') {
5               if(dados.sentencas[i].declaracoes !== undefined) {
6                   atributos_classe.push(dados.sentencas[i].declaracoes);
7               }
8           }
9       }
10      for( var i = 0; i < atributos_classe.length; i++) {
11         if(atributos_classe[i].length !== 0) {
12             for (var x = 0; x < atributos_classe[i].length; x++) {
13                 if(atributos_classe[i][x].nodeType === 'classe_atributo') {
14                     if(atributos_classe[i][x].visibilidade === 'public') {
15                         qtdAtibutosPublicos++
16                 } else if (atributos_classe[i][x].visibilidade ==='private')
17                      {
18                          qtdAtibutosPrivados++;
19                      }
20                      qtdAtibutos++;
21                  }
22              }

23          } else {
24              return 0;
25          }
26      }
27      return (qtdAtibutosPrivados / qtdAtibutos);
28  }
```

A realização da função DAM se dá pelo início da criação da variável **atributo_classe** na linha 2 que recebe um array de dados vazio, em sequência realiza um *loop* na linha 3 do array de dados passados pela a função que o chamou, verificando se os dados são do tipo objeto e se são diferentes de indefinidos e após essa análise adiciona o objeto que passou por todas essas verificação para a variável **atributo_classe** na linha 6.

O próximo passo é a realização do *loop* na linha 10 dos dados da variável **atributo_classe** onde são realizadas verificações para a realização do cálculo, validando-se a quantidade de dados dentro da variável **atributo_classe** é diferente de zero após isso é realizado outro *loop* na linha 12 utilizando a variável **atributo_classe** para assim ser possível acessar os dados da classe do diagrama que estão em nível mais abaixo na estrutura de dados retornado pelo *parser*.

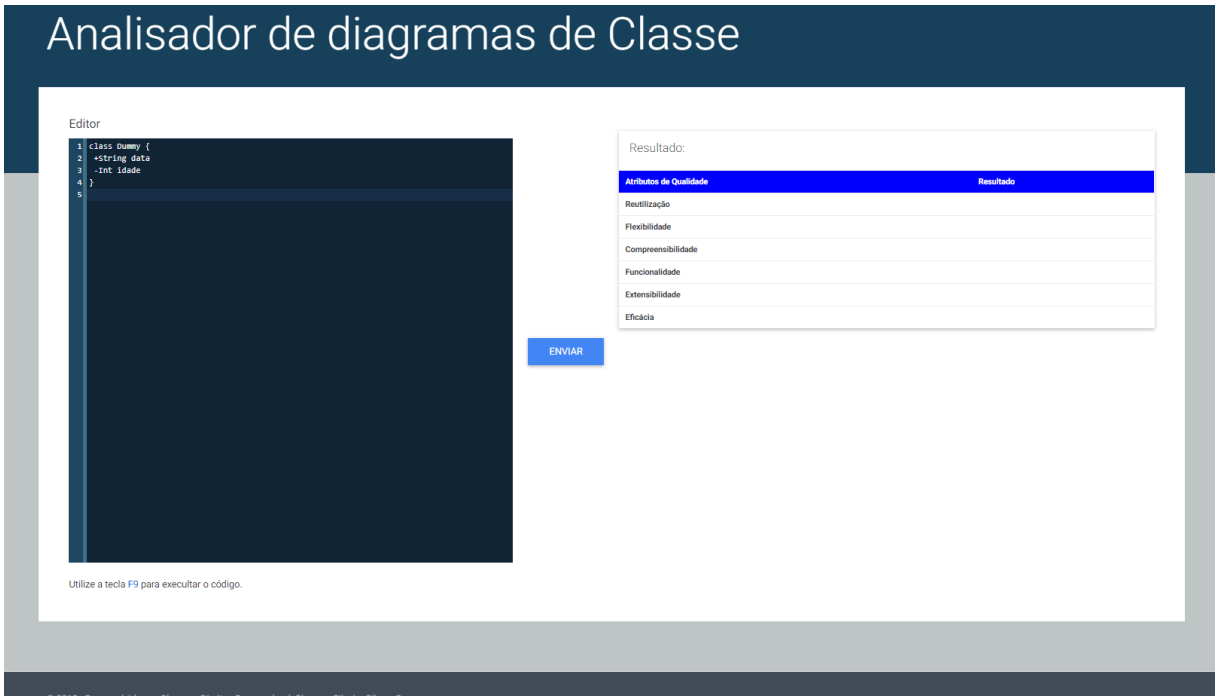
Após a realização do segundo *loop*, é validado se o tipo de visibilidade é do tipo pública e privada nas linhas 14 e 16 é assim realizada a soma de cada tipo de visibilidade em sua respectiva variável linhas 15 e 18. Logo depois de realizar essas verificações, é somada na variável **qtdAtributos** na linha 20 a quantidade de atributos dentro das classes.

Com o termino destas validações, é retornado para a classe que chamou o método CAM o cálculo da proporção que representam os atributos privados do total de atributos da classe utilizando a formula da linha 27.

4.5 Interface Gráfica

A interface gráfica fornece um editor de texto para o usuário informar o seu diagrama de classe conforme o modelo da linguagem PlantUML e resultado após o cálculo das métricas de qualidade representado pela Figura 14.

Figura 14 Interfase Gráfica



A Figura 14 apresenta a interface gráfica do software desenvolvido neste trabalho. A tela foi dividida em duas partes, uma contendo o editor e a outra o resultado do cálculo das métricas de qualidade do modelo QMOOD. Somente é apresentado ao usuário o resultado dos cálculos das métricas de qualidade após o mesmo selecionar a opção **enviar**.

5 CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo identificar os atributos, métodos e relacionamentos de um diagrama de classe, assim como, quantificar o diagrama de classe utilizando o modelo QMOOD. O *software* desenvolvido nesta monografia é capaz de identificar as métricas a partir do digrama de classe definido na linguagem PlantUML.

Para o processo de desenvolvimento do *software* foi utilizada a definição da linguagem PlantUML de diagramas de classe, com intuito de desenvolver um *parser* conforme o padrão da linguagem. Desta forma, foi possível realizar o processo de identificação do diagrama de classe que é feito através do *parser* desenvolvido nesse trabalho. O *parser* analisa o diagrama de classe organiza as informações sobre o diagrama na forma de um modelo de objetos que representa uma estrutura de dados em árvore, composta por nós. Cada nó é um objeto com atributos específicos conforme é apresentado na seção 4.3

Depois do tratamento feito pelo *parser* o analisador recebe o resultado e dá início ao processo de geração das onze métricas de qualidade (DSC, NOH, ANA, DAM, DCC, CAM, MOA, MFA, NOP, CIS e NOM) e seus respectivos cálculos do modelo QMOOD (Reutilização, Flexibilidade, Compreensibilidade, Funcionalidade, Extensibilidade e Eficácia). Por fim, a interface gráfica desenvolvida fornece um editor de texto para o usuário informar o seu diagrama de classe conforme o modelo da linguagem PlantUML e o resultado após o cálculo das métricas de qualidade, como foi apresentado na seção 4.4.

Para trabalhos futuros, propõe-se o desenvolvimento de uma área administrativa, para visualização dos diagramas gerados pelos usuários. Outro ponto para o aperfeiçoamento do *software* seria permitir aos usuários o compartilhamento dos diagramas e as métricas entres eles, por fim, trabalhos voltados para o armazenamento dos diagramas e métricas seriam de grande valia, permitindo que o software apresente melhores resultados.

REFERÊNCIAS

ALBEDALI, A. et al. **Toward Software Measurement and Quality Analysis of MARF and GIPSY Case Studies a Team 13 SOEN6611-S14 Project Report**. CoRR, abs/1407.0063, 2014. Disponível em: < http://arxiv.org/abs/1407.0063>. Acesso em: 10/04/2018.

BERTRAN, I. M. **AVALIAÇÃO DA QUALIDADE DE SOFTWARE COM BASE EM MODELOS UML**. 2009. Dissertação (Mestrado) — PUC-RIO, RIO DE JANEIRO. Disponível em: < http:// www.maxwell.vrac.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=13748@1>. Acesso em: 06/04/2018.

BANSIYA, J.; DAVIS, C. G. **A Hierarchical Model for Object-Oriented Design Quality As- sessment**. IEEE Trans. Software Eng., v. 28, n. 1, p. 4 – 17, 2002. Disponível em: <http: //doi.ieeecomputersociety.org/10.1109/32.979986>. Acesso em: 13/02/2018.

CARTER, Zachary. **Jison An API for creating parsers in JavaScript**. 2009. Disponível em: <https://zaa.ch/jison/>. Acesso em: 31/10/2018.

CMMI. **About CMMI® Institute**. 2018. Disponível em: <http://cmmiinstitute.com/about-cmmi- institute>. Acesso em: 06/01/2018.

ISO/IEC 12207. **International Standards through**. 2008. Disponível em: <https://www.iso.org/ obp/ui/#iso:std:iso-iec:12207:ed-2:v1:en>. Acesso em: 17/02/2018.

ISO/IEC 9126. Software enginnering – Product quality. Dezembro 1991. Disponível em: <https: //www.iso.org/standard/16722.html>. Acesso em: 08/04/2018.

KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de Software aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. 2. ed. São Paulo: Novatec, 2007. ISBN 978-85-7522-112-9.

PRESSMAN, R. S. **Engenharia de Software - Uma abordagem profissional**. 7. ed. Porto Alegre: AMGH Editora LTDA, 2011. 780 p. Tradução Ariovaldo Griesi, Mário Moro Fecchio.

SOMMERVILLE, I. Engenharia de Sofrware. 9. ed. São Paulo: PEARSON, 2011. ISBN 8579361087.

RIBEIRO, D. D. C. OSTRA: **UM ESTUDO DO HISTÓRICO DA QUALIDADE DO SOFTWARE ATRAVÉS DE REGRAS DE ASSOCIAÇÃO DE MÉTRICAS**. 2012. 175 p. Monografia (Engenharia de Software) — Universidade Federal do Fluminense. Disponível em:<https://docgo.net/philosophy-of-money.html?utm_source=daniel-drumond-castellani-ribeiro-ostra-um-estudo-do-historico-da-qualidade-do-software-atraves-de-regras-de-associacao-de-metricas>. Acesso em: 13/02/2018.

PEREIRA, Bruno Ricardo da Silva. **Implementação de um parser HL7 para integração do ALERT com aplicações terceiras.** 2008. 134 f. Dissertação (Mestrado) - Curso de Mestrado Integrado em Engenharia Informática e Computação, Faculdade de Engenharia, Universidade do Porto, Porto, 2008.

PlantUML, PlantUML in a nutshell. Disponível em: <http://plantuml.com/> Acesso em: 31/10/2018.