UNIVERSIDADE CESUMAR – UNICESUMAR
CENTRO DE CIÊNCIAS EXATAS TECNOLÓGICAS E AGRÁRIAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

UTILIZANDO INSTRUMENTAÇÃO NA EXTRAÇÃO DE MÉTRICAS DE APLICAÇÕES JAVA EM TEMPO DE EXECUÇÃO

GABRIEL GASPAR HEMPEL

MARINGÁ – PR
2020

GABRIEL GASPAR HEMPEL

UTILIZANDO INSTRUMENTAÇÃO NA EXTRAÇÃO DE MÉTRICAS DE APLICAÇÕES JAVA EM TEMPO DE EXECUÇÃO

Artigo apresentado ao Curso de Graduação em Engenharia de Software da Universidade Cesumar – UNICESUMAR como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software sob a orientação do Prof. Me. Arthur Cattaneo Zavadski.

MARINGÁ – PR

2020

FOLHA DE APROVAÇÃO

GABRIEL GASPAR HEMPEL

UTILIZANDO INSTRUMENTAÇÃO NA EXTRAÇÃO DE MÉTRICAS DE APLICAÇÕES JAVA EM TEMPO DE EXECUÇÃO

Artigo apresentado ao Curso de Graduação em Engenharia de Software da Universidade Cesumar – UNICESUMAR como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software sob a orientação do Prof. Me. Arthur Cattaneo Zavadski.

Aprovado em: ____ de dezembro de 2020.

BANCA EXAMINADORA

_____

Prof. Me. Arthur Cattaneo Zavadski (Orientador)

_____

Prof. Dr. Nelson Nunes Tenório Júnior (UNICESUMAR)

UTILIZANDO INSTRUMENTAÇÃO NA EXTRAÇÃO DE MÉTRICAS DE APLICAÇÕES JAVA EM TEMPO DE EXECUÇÃO

Gabriel Gaspar Hempel

RESUMO

As métricas de código são fundamentais no desenvolvimento de um projeto de software. A coesão e o acoplamento, são uma das principais métricas de código-fonte, a coesão é a medida que os elementos de um módulo trabalham em conjunto por um mesmo objetivo. O acoplamento, é a comunicação entre os módulos de um sistema. Outro fator de qualidade de software são os testes de software que são feitos para garantir a correta implementação de uma funcionalidade e com o intuito de encontrar bugs. Existem diversas ferramentas que executam os testes de software e calculam a cobertura do código. Este trabalho tem como objetivo investigar como é feita a cobertura de código a partir de uma ferramenta coverage e utilizar o mesmo conceito para calcular uma métrica de coesão a partir de um código Java que será analisado.

Palavras-chave: Acoplamento. Cobertura de Testes. Coesão.

USING INSTRUMENTATION TO EXTRACT METRICS OF JAVA APPLICATIONS IN TIME EXECUTION

ABSTRACT

Code metrics are fundamental in the development of a software project. Cohesion and coupling are one of the main metrics of source code, cohesion is the measure that the elements of a module work together for the same objective. Coupling is the communication between the modules of a system. Another factor of software quality is the software tests that are done to ensure the correct implementation of a feature and in order to find bugs. There are a number of tools that perform software testing and calculate code coverage. This work aims to investigate how code coverage is made from a coverage tool and use the same concept to calculate a cohesion metric from a Java code that will be analyzed.

Keywords: Cohesion. Coupling. Test Coverage.

1 INTRODUÇÃO

Em um projeto de software, independentemente da metodologia adotada, a qualidade de software é fundamental. Portanto, a análise do código-fonte é indispensável, uma vez que, dentre as incontáveis características de um software, algumas são exclusivas do código-fonte. Sendo assim, o monitoramento de coesão e acoplamento são essenciais durante o desenvolvimento de um software (MEIRELLES, 2013).

A coesão é a proximidade dos elementos (atributos e métodos) de um módulo. Em outros termos, quanto mais um módulo utiliza seus elementos focado em um único objetivo, mais coeso ele é. Por outro lado, o acoplamento é a interdependência entre os módulos de um sistema. O baixo acoplamento significa minimizar as dependências (comunicação e compartilhamento de informações com módulos externos) entre os módulos (HERZIG; VARZINCZAK, 2005).

Outro fator fundamental, para a qualidade de software, são os testes de software que tem como objetivo evidenciar que determinada implementação está de acordo com o seu propósito, além de buscar por falhas que podem ocorrer antes do software ser posto em uso (SOMMERVILLE, 2011). Posto isto, a cobertura de teste é uma métrica de software que avalia a efetividade dos testes e o quão exaustivamente eles são executados (HARON; SYED-MOHAMAD, 2015). Dentre as inúmeras ferramentas de cobertura de testes, será realizada uma análise da ferramenta JaCoCo que é uma biblioteca de cobertura de código gratuita para Java que foi criada pela equipe EclEmma.

Nesse contexto, o problema de pesquisa deste trabalho é “Qual a importância da coesão e acoplamento no código, e se é possível, a partir de um código extrair uma métrica de coesão utilizando o mesmo conceito que uma ferramenta coverage utiliza para realizar a cobertura do código?”.

Para tanto, será realizada uma pesquisa bibliográfica sobre coesão e acoplamento. E será feita uma análise sobre como a ferramenta de cobertura de testes JaCoCo realiza a cobertura de código e verificar se é possível aplicar o mesmo conceito para extrair uma métrica de coesão de um código analisado.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Utilizar o conceito de ferramentas de cobertura de testes e criar um algoritmo que apresenta as métricas de coesão do código-fonte analisado.

2.2 OBJETIVOS ESPECÍFICOS

1 – Descrever e analisar as métricas de acoplamento e coesão do código-fonte.

2 – Descrever e analisar uma ferramenta de cobertura de testes funciona de modo a utilizar a mesma estratégia para coesão e acoplamento.

3 – Desenvolver um algoritmo que seja capaz de informar, a partir de um código-fonte, a coesão do código-fonte analisado.

3 REFERENCIAL TEÓRICO

3.1 ACOPLAMENTO

O acoplamento é a medida da força de associação definida pelas conexões que um módulo por ter com outros módulos (CONSTANTINE; STEVENS; MYERS, 1974). Sendo assim, qualquer indício de determinado módulo utilizando métodos ou instância de variáveis de outros módulos, constitui acoplamento (CHIDAMBER; KEMERER, 1991).

O acoplamento é, também, um atributo natural e fundamental de um software, devido a sua importância para a estruturação lógica de uma aplicação, visto que as interações entre os módulos são primordiais para o desenvolvimento de funcionalidades de um software (DEL ESPOSTE; BEZERRA, 2014).

Nesse contexto, os módulos são partes de um sistema que se comunicam entre si, para atender as regras de negócio de um software (PRESSMAN, 2001). Dessa forma, a modularização do software possibilita que alterações no código-fonte sejam executadas com mais facilidade sem ocasionar erros em outras partes do sistema (LINO, 2011).

Os módulos devem ser independentes, uma vez que resolver problemas mais rápido e fácil quando o problema está fragmentado em partes que podem ser examinadas separadamente (CONSTANTINE; STEVENS; MYERS, 1974).

De acordo com os autores estudados, quanto maior o número de módulos, maior serão o número de conexões necessárias para realizar a integração entre os elementos do sistema (PRESSMAN, 2001). Consequentemente, maior será o nível de acoplamento, uma vez que o número de conexões entre os módulos será maior, como resultado, sistemas muito modularizados acabam dificultando o entendimento dos módulos individualmente (DARCY et al., 2005).

Por conseguinte, quanto maior o grau de acoplamento entre os módulos, menos flexível e mais complexo o sistema fica, dessa maneira, modificar e entender o que um módulo faz, fica bem mais complexo (MEIRELLES, 2013). Outro aspecto a ser observado é que altos níveis de acoplamento também podem aumentar a propagação de erros, causando um “ripple effect”. Um “ripple effect”, é um efeito cascata que acontece quando um erro se espalha por toda a aplicação (PRESSMAN, 2001).

Sistemas devem ser projetados com baixos níveis de acoplamento, uma vez que resolver problemas, fica bem mais difícil quando temos que levar em conta todos os aspectos ao mesmo tempo. Em outras palavras, é bem mais complexo resolver um problema, quando uma alteração no código pode causar danos em outras partes. Isto é, quanto menor e mais simples forem os números de conexões entre os módulos, mais fácil será de entendê-lo (CONSTANTINE; STEVENS; MYERS, 1974).

Além disso, o baixo acoplamento entre os módulos é mais suscetível a mudanças, pois reduz a possibilidade de mudanças ou erros em cascata ocorrerem e afetarem uma parte maior do sistema (BECK; DIEHL, 2011). Em outros termos, reduz o risco de acontecer um “ripple effect”.

Sendo assim, uma característica essencial para garantir o baixo acoplamento, é o encapsulamento dos atributos do módulo. Quando um módulo externo consegue acessar e modificar um atributo interno de outro módulo, a integridade do atributo é corrompida, provocando uma forte dependência entre os módulos que fazem a utilização deste atributo (FERREIRA, 2011).

Da mesma maneira que o encapsulamento, o Test Driven Development (TDD), também promove o baixo acoplamento. O TDD é a prática de escrever os testes antes de desenvolver o código que implementa a funcionalidade. O ato de escrever os testes primeiro,

promove a escrita de um código que precisa ser chamado e testável, e sendo assim, ele também deve ser desacoplado. Quanto mais testável um módulo for, menor será o seu acoplamento. Logo, a execução do TDD força a escrita de um software desacoplado (MARTIN, 2003).

Para concluir, o acoplamento entre módulos pode ser minimizado de duas formas, uma delas é reduzir o número de associações que um módulo tem com outras partes do sistema. A outra maneira é aumentar a relação entre os elementos de um mesmo módulo, deixando o módulo mais coeso (CONSTANTINE; STEVENS; MYERS, 1974).

3.2 COESÃO

A coesão é a medida da força relativa entre os elementos de um mesmo módulo. Esses elementos podem ser atributos ou métodos que uma classe possui e como eles interagem mutuamente (PRESSMAN, 2001). Então, a coesão é o grau de diversidade de assuntos que uma classe ou módulo é responsável (MEIRELLES, 2013).

A coesão é uma medida inversa da complexidade, quanto mais coeso é um módulo de programa, menos complexo esse módulo é considerado, e como efeito, menor será o esforço necessário para mantê-lo (DARCY et al., 2005). Devemos sempre buscar por alta coesão, uma vez que elementos de um módulo trabalhando em conjunto por um mesmo objetivo é melhor do que operando por uma variedade de objetivos diversos (WOODWARD, 1993). Além disso, a alta coesão é um dos fundamentos da Orientação a Objetos, visto que facilita a compreensão do código, sua reutilização, manutenção e seu teste (JAIN; GUPTA, 2015).

Isto posto, a alta coesão ainda promove alguns princípios de design de software, como o SRP – The Single-Responsibility Principle (O Princípio de Responsabilidade Única), onde uma classe deveria ter apenas uma razão para ser alterada. Classes que possuem muitas responsabilidades, tornam essas responsabilidades associadas, criando frágeis arquiteturas que quebram de maneiras inesperadas quando alteradas. Dessa maneira, um módulo coeso deveria executar uma única tarefa em um sistema, possuindo uma única responsabilidade (MARTIN, 2003).

Outro aspecto a ser destacado é que a alta coesão também promove encapsulamento. O encapsulamento ocorre quando os elementos de um módulo são inacessíveis para outros módulos que não necessitam dessas informações. Logo, erros ou bugs que ocorrem durante a

manutenção, têm menores probabilidades de se propagar por todo o sistema (PRESSMAN, 2001).

A baixa coesão ocorre quando elementos de um módulo são responsáveis por diferentes tarefas, concentrando informações diversificadas. Isso pode ocasionar aumento na complexidade do código e dificuldade de manutenções futuras (DEL ESPOSTE; BEZERRA, 2014). Em vista disso, a incorreta atribuição de responsabilidades a uma classe ou módulo durante a etapa de design de um software pode causar a baixa coesão e consequentemente a criação de classes mal projetadas, dificultando a manutenibilidade do software (BADRI; BADRI; TOURE, 2010).

A busca pela alta coesão em um projeto de software ainda garante outros benefícios, como o baixo acoplamento entre as classes e a diminuição no número de linhas de código. Como no estudo realizado por Badri, Badri e Toure (2010) que demonstra que quanto maior o acoplamento entre dois módulos e maior o seu número de linhas, provavelmente menor será a coesão da classe.

Em um projeto de software, não é necessário determinar o nível preciso de coesão de um módulo, mas sempre buscar por alta coesão e reconhecer a baixa coesão, para que a mudança na arquitetura possa ser realizada em busca de métodos independentes e coesos (PRESSMAN, 2001). Dessa forma, a busca pela alta coesão pode ser feita através do uso de métricas existentes que tem como objetivo realizar a medida de coesão de um módulo ou classe. Grande parte dessas métricas de coesão são baseadas no grau de similaridade entre os métodos de um módulo e na conectividade que os atributos que esse módulo possui. Porém, existem algumas diferenças em suas definições (BADRI; BADRI; TOURE, 2010).

3.3 MÉTRICAS NA ENGENHARIA DE SOFTWARE

Segundo o IEEE (1990) (Glossário Padrão de Termos de Engenharia de Software) métrica é definida como “A medida quantitativa do grau que um sistema, componente ou processo possui sobre um determinado atributo”. Assim, a métrica de software é o conjunto de regras que tem como propósito a avaliação de um atributo relacionado ao software. E através dessa medida, um engenheiro de software é capaz de corrigir ou melhorar o produto, projeto ou processo (FERREIRA, 2011).

Sendo assim, uma métrica de software é uma característica de um sistema, sua documentação ou seu processo de desenvolvimento que pode ser medida objetivamente

(SOMMERVILLE, 2011). Isto é, métricas de software são concebidas para alcançar medidas reproduzíveis e objetivas, com a intenção de assegurar a qualidade do software (PRESSMAN, 2001).

Em contrapartida, são incontáveis o número de métricas de software existentes, uma vez que, software é um artefato complexo e abrangente e para determinar a sua qualidade, é fundamental a utilização de diferentes métricas. Então, o resultado de uma métrica, não pode definir a qualidade do projeto como um todo (MIDDING, 2016).

Posto isto, a qualidade do software não corresponde apenas a implementação de uma funcionalidade do sistema, mas também, aos requisitos não funcionais de um software (SOMMERVILLE, 2011). Portanto, como descrito na ISO/IEC9126-1 (ABNT, 2003), alguns fatores de qualidade de software são:

- Funcionalidade: O nível em que o software satisfaz os requisitos, para atender as necessidades declaradas.
- Confiabilidade: A quantidade de tempo que o software fica à disposição para a sua utilização.
- Usabilidade: O grau de facilidade do uso do software.
- Eficiência: A capacidade do software de utilizar os recursos do computador de maneira proveitosa.
- Capacidade de Manutenção: A simplicidade que um reparo pode ser realizado no software.
- Portabilidade: A capacidade do software de ser deslocado de um ambiente para outro.

As métricas de software referem-se a uma variedade de medidas que podem ser aplicadas em um software, com a intenção de melhorá-lo continuamente e produzir sistemas com alta qualidade (PRESSMAN, 2001). Portanto, seja qual for a metodologia de desenvolvimento adotada, supervisionar a qualidade do software é indispensável. E dentre as inúmeras métricas de software que existem, algumas são exclusivas do código-fonte, visto que, a partir dele podemos analisar a organização, legibilidade, manutenibilidade, modularidade e simplicidade do código (MEIRELLES, 2013).

O acompanhamento da qualidade do código-fonte, colabora para a aplicação de estratégias de desenvolvimento que buscam o aprimoramento constante do código (DEL

ESPOSTE; BEZERRA, 2014). Consequentemente, o desenvolvimento de um software pode enfrentar problemas, caso o mesmo evolua sem observar algumas características do código-fonte, como por exemplo: a coesão e o acoplamento (MEIRELLES, 2013).

3.3.1 Métricas de Coesão

- Lack of Cohesion in Methods (LCOM): É o número de pares de métodos que não possuem relacionamentos com atributos em comum em uma classe (CHIDAMBER; KEMERER, 1991).
- Lack of Cohesion in Methods 2 (LCOM2): É o número de pares de métodos que não compartilham atributos menos o número de pares de métodos que compartilham atributos (KEMERER; CHIDAMBER, 1994).
- Lack of Cohesion in Methods 3 (LCOM3): É o número de pares de métodos que não compartilham atributos menos o número de pares de métodos que compartilham atributos (HENRY; LI, 1993).
- Lack of Cohesion in Methods 4 (LCOM4): A LCOM4 é similar a LCOM3, na qual uma borda é adicionada ao grafo, essa borda representa a invocação dos métodos (HITZ; MONTAZERI, 1995).
- Lack of Cohesion in Methods 5 (LCOM5): É o cálculo entre o número de atributos, métodos e o número de métodos referenciando cada atributo de uma classe. (HENDERSON-SELLERS, 1996).
- Tight Class Cohesion (TCC): É o número relativo de métodos diretamente conectados. Em outras palavras, TCC é a medida em porcentagem, da quantidade de pares de métodos públicos que utilizam um atributo em comum em uma classe (BIEMAN; KANG, 1995).
- Loose Class Cohesion (LCC): É o número relativo de métodos diretamente ou indiretamente conectados. Dois métodos são diretamente ou indiretamente conectados quando estiverem conectados a um atributo (BIEMAN; KANG, 1995).
- Class Cohesion (CC): A similaridade entre métodos é o cálculo de número de atributos compartilhados e o número de atributos diferentes que dois métodos possuem. Class Cohesion (CC) é a razão da soma de similaridade entre os pares de métodos com o número total de métodos. Sendo assim, se uma classe tem

apenas um método, não possui atributos ou não possui nenhum método, ela é considerada altamente coesa para esta métrica (BONJA; KIDANMARIAM, 2006).

3.3.2 Métricas de Acoplamento

- Response For Class (RFC): É o conjunto de métodos de uma classe que pode ser executado em resposta a uma mensagem que foi recebida por outro objeto. Dessa forma, quanto maior o número de métodos capazes de serem invocados fora da classe, maior será a complexidade da classe (KEMERER; CHIDAMBER, 1994).
- Coupling Between Objects (CBO): É a contagem do número de relacionamentos que uma classe possui com outra. Isto é, como dito anteriormente, uma classe é acoplada se usa um método ou uma instância de variável declarada em outra classe (KEMERER; CHIDAMBER, 1994).
- Message Passing Coupling (MPC): É a contagem do número de invocações que uma classe faz a um método. Se o número de chamadas feitas por uma classe a um método externo for muito alto, isso pode indicar um alto nível de dependência entre a classe e o método (HENRY; LI, 1993).
- Data Abstraction Coupling (DAC): É o número de atributos de uma classe das quais os tipos dos atributos pertencem a outra classe. Dessa maneira, quanto maior for o número de atributos que uma classe possui com o tipo deste atributo sendo de outra classe, maior será o acoplamento (HENRY; LI, 1993).
- Afferent Coupling (Ca): É o número de classes fora de um pacote (package) que dependem de classes que estão dentro de um pacote (MARTIN, 2003).
- Efferent Coupling (Ce): É o número de classes dentro de um pacote que dependem de classes que estão fora deste pacote (MARTIN, 2003).

3.4 FERRAMENTAS E MÉTRICAS DE COBERTURA DE TESTES

Segundo o IEEE (1990), a cobertura de testes é o grau em que um determinado teste ou conjunto de testes atende a todos os requisitos especificados para um determinado sistema ou componente. Em outros termos, a cobertura de teste é uma métrica de software que avalia

a efetividade dos testes e o quão exaustivamente eles são executados (HARON; SYED-MOHAMAD, 2015).

Testes de software tem como objetivo evidenciar que determinada implementação está de acordo com o seu propósito, além de buscar por falhas que podem ocorrer antes do software ser posto em uso (SOMMERVILLE, 2011). Dessa maneira, o teste de software comprova que as funcionalidades desenvolvidas foram atendidas e estão de acordo com as especificações, com os requisitos comportamentais e de desempenho. Logo, a execução dos testes e a criação de novos, indicam uma boa confiabilidade e consequentemente uma garantia de qualidade de software (PRESSMAN, 2001).

Sommerville (2011) define três tipos de testes que podem ser realizados durante o desenvolvimento de software:

- Teste de Unidade: Tem como objetivo o teste unitário das funcionalidades implementadas nas classes e métodos da aplicação.
- Teste de Componente: Tem como propósito o teste das interfaces do sistema, observando as unidades individuais e criando módulos compostos.
- Teste de Sistema: É o teste realizado sobre as integrações entre os módulos que foram criadas, no qual o software é testado como um todo.

Neste contexto, a cobertura de testes é a medida da efetividade de testes de software. Diante disso, a cobertura de testes aumenta quando mais casos de testes são criados sem repetir os casos já existentes, e se a cobertura completa dos testes ainda não tenha sido alcançada (MALAIYA et al., 2002).

Posto isto, algumas das métricas de cobertura de testes são Statement Coverage (Cobertura de Linhas de Código), Branch Coverage (Cobertura de Ramos) e Path Coverage (Cobertura de Caminhos). Segundo Zhu, Hall e May (1997), essas métricas são definidas como:

- Statement Coverage: Ocorre quando todas as instruções do código são executadas pelo menos uma vez.
- Branch Coverage: Acontece quando todos os caminhos ou condições possíveis do código são executados.

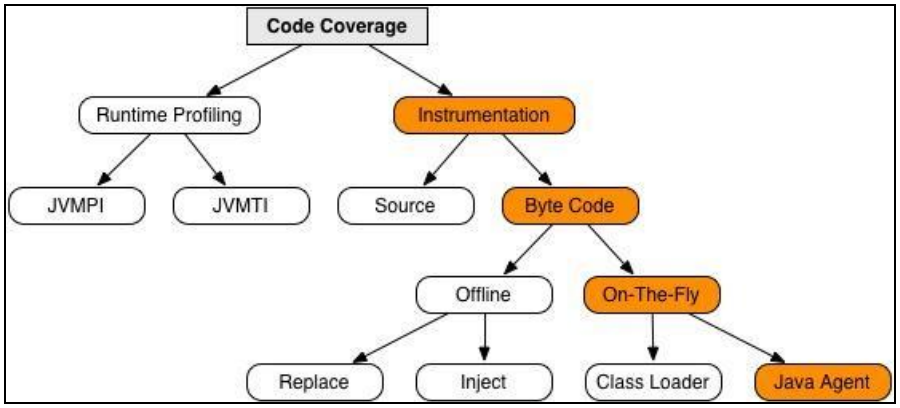
- Path Coverage: É a execução de todos os caminhos possíveis do programa. Realiza-se quando todas as instruções e condições do código são executadas.

Dessa maneira, a análise de cobertura de código pode ser dividida em três funções. A primeira é a instrumentação do código, que é a adição de algum código para calcular os resultados da cobertura. Em seguida, temos a coleta de dados, que consiste no armazenamento de dados retirados da cobertura. Por último temos a análise dos dados que tem como propósito criar procedimentos para modificar ou criar testes (HARON; SYED-MOHAMAD, 2015).

No contexto deste trabalho, veremos como a ferramenta JaCoCo realiza a instrumentação do código para calcular os resultados da cobertura. O JaCoCo é uma biblioteca de cobertura de código gratuita para Java, que foi criada pela equipe EclEmma.

No JaCoCo, a coleta das informações de uma cobertura deve ser realizada em tempo de execução. Logo, existem diferentes maneiras para realizar a coleta dessas informações e diferentes técnicas de implementação são adotadas. A figura 1 apresenta uma visão geral sobre essas abordagens, em destaque a abordagem utilizada pelo JaCoCo (JACOCO, 2017).

Figura 1 – Abordagens de Coleta de Informações



Fonte: JaCoCo (2017).

Para realizar a coleta de informações sobre uma cobertura, o JaCoCo cria versões instrumentadas das definições das classes originais. Esse processo é chamado de instrumentação e utilizam os Java Agents e ocorre em tempo real durante o carregamento da classe. Posto isto, a instrumentação do código requer mecanismos para modificar e gerar Java Byte Code. Para esse propósito existem diferentes bibliotecas que facilitam essa manipulação. O JaCoCo utiliza a ASM Library que é uma ferramenta de análise e manipulação de Java Byte Code (JACOCO, 2017).

3.5 INSTRUMENTAÇÃO DO CÓDIGO EM JAVA

A linguagem Java fornece uma API para a instrumentação de programas em execução na JVM (Java Virtual Machine). A instrumentação é a modificação dos bytes do código de uma aplicação Java. Sendo assim, para realizar a instrumentação é necessário a implementação de um Java Agente. Um Java Agente é um tipo especial de classe que utiliza a API de instrumentação fornecida pelo Java para interceptar e modificar o bytecode das aplicações que estão em execução na JVM (BALAKRISHAN, 2020).

Um Java Agente é implementado como um arquivo JAR, e pode ser feito de duas maneiras:

1. Através de linhas de comando de uma interface;
2. Após a inicialização da VM (Virtual Machine).

A implementação de um Java Agente utilizando uma interface de linhas de comando é feita a partir da execução do comando "-javaagent:'jar-path' [=options]". No qual, jar-path é o caminho para o arquivo JAR em que os Agentes se encontram. Options são as opções de agentes que podem ser carregados em uma mesma linha de comando, criando múltiplos agentes. Cada Agente implementa o método premain que funciona como um interceptador e é executado antes da execução do método main de uma aplicação Java. Dessa maneira, após a inicialização da JVM, cada premain método será executado na ordem em que os Agentes foram especificados e em seguida o método main da aplicação será chamado (JAVA™, 2015).

A segunda maneira de implementar um Java Agente é após a inicialização da VM (Virtual Machine), ou seja, após a execução do método main de uma aplicação Java. Para isso, a classe do Agente deve implementar o método agentmain. O carregador de classes (System Class Loader) de um programa Java que carrega o método main, deve incorporar o Agente JAR no caminho de classes do sistema. Dessa maneira, a classe do Agente é carregada e a JVM chama o método agentmain (JAVA™, 2015).

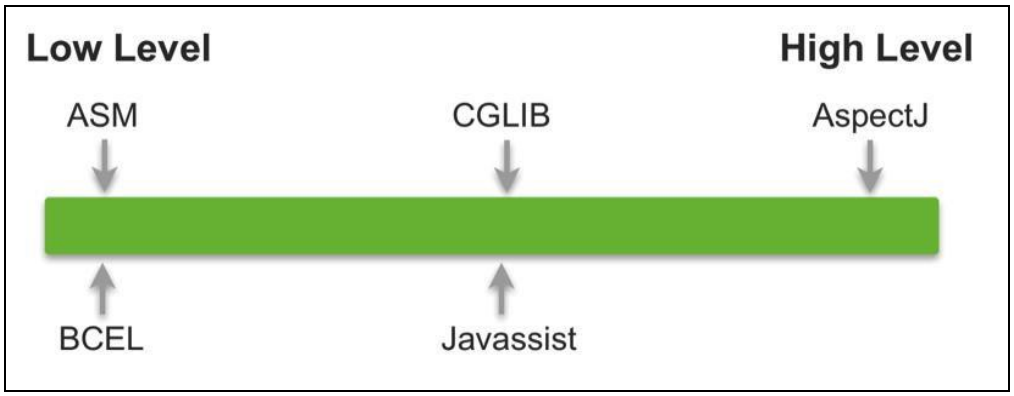
As duas maneiras descritas, precisam estar presentes no arquivo de manifesto de um JAR. Para Agentes executados antes da chamada do método main, o atributo do manifesto Premain-Class deve conter o valor do nome da classe do Agente que implementa o método premain. Para os agentes executados depois da chamada do método main, o atributo do

manifesto Agent-Class deve conter a classe que implementa o método agentmain (JAVA™, 2015).

Nesse contexto, existem diversas bibliotecas que auxiliam na manipulação do bytecode Java, a maneira que essas bibliotecas manipulam o bytecode variam, entre High Level, na qual podemos realizar a manipulação com puro código Java (AspectJ), e Low Level, na qual precisamos trabalhar com o código no nível de bytecode (ASM). A figura a seguir demonstra algumas bibliotecas e seus níveis (PULS, 2014).

No contexto deste trabalho, a ferramenta escolhida para auxiliar na instrumentação do código, será a mesma que é utilizada pelo JaCoCo.

Figura 2 – Bibliotecas de manipulação de bytecode.



Fonte: Puls (2014).

3.6 ASM

ASM é uma biblioteca de análise e manipulação de bytecode Java. Pode ser usada para manipular classes existentes ou para alterá-las dinamicamente. A biblioteca ainda fornece algoritmos de análise a partir de transformações complexas e personalizadas, cujas ferramentas de análise de código podem ser construídas (ASM, 2020).

A arquitetura implementada na ASM para gerar e transformar bytecodes é baseada na classe abstrata ClassVisitor. Onde cada método do ClassVisitor é correspondente a um membro de uma estrutura de uma classe compilada. Como demonstrado na figura 3 (ASM, 2020).

Figura 3 – Estrutura de uma Classe Compilada.

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Fonte: ASM (2020).

Nesse cenário, implementamos o método visitMethod da classe abstrata ClassVisitor, para visitar cada método de uma classe. Porém, os métodos do ClassVisitor precisam ser chamados em uma determinada ordem, que está representada na próxima figura (ASM, 2020).

Figura 4 – Estrutura de Visitors

```
visit visitSource? visitOuterClass? ( visitAnnotation | visitAttribute )*
( visitInnerClass | visitField | visitMethod )*
visitEnd
```

Fonte: ASM (2020).

Para realizar a instrumentação de uma classe, o ASM fornece a seguinte estrutura de classes:

1. ClassReader: Realiza a leitura dos bytes, analisando a classe compilada e faz a chamada para os métodos do ClassVisitor.
2. ClassVisitor: É a classe que fornece os métodos responsáveis por realizar a manipulação do bytecode, após a leitura dos bytes.
3. ClassWriter: É classe responsável pela saída final da manipulação do bytecode, é responsável por construir classes compiladas diretamente na forma binária e retornar os bytes manipulados, após sua transformação.

Posto isto, para realizar a instrumentação do código em Java com a ajuda do ASM, utilizamos sua estrutura que facilita a leitura e a modificação de um Bytecode de uma classe em Java (ASM, 2020).

4 METODOLOGIA

Neste trabalho foi realizada uma pesquisa básica estratégica, envolvendo os conceitos e as métricas de acoplamento e coesão. Além disso, também foi realizada uma pesquisa sobre as métricas de software, testes de software, cobertura de testes e como a ferramenta JaCoCo utiliza a API de instrumentação do Java para realizar a cobertura do código.

Quanto ao objetivo, é classificado como descritivo e foi realizado um estudo em livros e artigos científicos buscando as definições de acoplamento, coesão, métricas de software e testes. Depois, foi realizado um pequeno experimento, no qual foi criado um algoritmo que calcula a métrica de coesão LCOM5 de uma classe Java.

O método adotado é o hipotético-dedutivo, cuja hipótese levantada foi de como uma ferramenta de cobertura de testes funciona e a partir da compreensão de sua implementação, retirar métrica de coesão de uma classe em Java.

Para alcançar o objetivo, foi necessário o entendimento da API de instrumentação do Java, biblioteca de manipulação de bytecode ASM e também a aplicação da fórmula da LCOM5. Dessa maneira, adotando uma abordagem quantitativa e os seguintes procedimentos:

1. Bibliográfico: Levantamento sobre: coesão, acoplamento, métricas de software e teste de software.
2. Documental: Estudo da biblioteca JaCoCo, a API de instrumentação do Java e a biblioteca ASM.
3. Experimental: Um algoritmo capaz de extrair a métrica de LCOM5 de uma classe em Java, utilizando a API de instrumentação do Java e a biblioteca de manipulação de bytecode ASM.

5 DESENVOLVIMENTO

Um dos fatores de qualidade de um software é a sua manutenibilidade. Ou seja, a forma como escrevemos o código é muito importante para a vida de um projeto. O código, além de atender ao requisito especificado, precisa ser de fácil entendimento para que outra pessoa seja capaz de ler, entender e, caso necessário, modificá-lo.

A busca pelo baixo acoplamento e alta coesão em uma arquitetura de software são uns dos fatores que garantem a escrita de um código simples e legível, facilitando o seu entendimento e sua manutenção futura.

A coesão é a medida de como os métodos e atributos de uma classe interagem mutuamente. Em outras palavras, a coesão observa as responsabilidades e assuntos que uma classe trata. Sendo assim, quanto menor o número de assuntos e responsabilidades uma classe possui, maior será a coesão da classe.

Em contrapartida, o acoplamento é medida de como as classes interagem e trocam informações entre si, dessa maneira, quanto maior o número de conexões que uma classe possui com outras, maior será a dificuldade de compreender e modificar o código dessa classe.

Nesse contexto, existem diversas métricas de coesão e acoplamento que tem como objetivo apresentar uma perspectiva do quanto uma classe é coesa, e respectivamente, acoplada. Cada métrica oferece maneiras diferentes de calcular a coesão e acoplamento, porém o objetivo é sempre de identificar a baixa coesão e alto acoplamento em um projeto, para que a correção seja aplicada.

Neste trabalho, foi criado um algoritmo que calcula o valor da métrica LCOM5, que é uma métrica de coesão, aplicando o conceito de instrumentação do código. A ideia de utilizar a instrumentação veio a partir do estudo da ferramenta JaCoCo, que utiliza a API de instrumentação do Java e a biblioteca de manipulação de bytecode ASM, para realizar a cobertura de código.

5.1 ALGORITMO CAPAZ DE CALCULAR LCOM5

O algoritmo desenvolvido calcula a métrica de coesão LCOM5, utilizando sua fórmula LCOM5 = (a - ml)/(h - mh). Onde m representa o número de métodos, h representa o

número de atributos e por fim, a representa o número de atributos distintos acessados nos métodos de uma classe (HENDERSON-SELLERS, 1996).

Para criar um Java Agent, é necessária a criação de um projeto Java e a implementação do método premain ou agentmain, dependendo da forma escolhida para realizar a instrumentação. Além disso, para utilizar a biblioteca ASM é necessário adicionar suas dependências ao projeto do Java Agent, como na figura a seguir:

Figura 5 – Dependências da ASM

```
<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm</artifactId>
  <version>9.0</version>
</dependency>

<dependency>
  <groupId>org.ow2.asm</groupId>
  <artifactId>asm-util</artifactId>
  <version>9.0</version>
</dependency>
```

Fonte: O autor (2020).

Posto isto, a utilização da API de instrumentação do Java será através da interface de linha de comando do Windows. Dessa maneira, será necessário a implementação do método premain na classe do Java Agente.

A classe que representa o Java Agente será denominada de Agent. O método premain que a classe Agent implementa recebe dois parâmetros. O primeiro, representa os argumentos que podem ser passados pela linha de comando, quando vamos executar o Java Agente. O segundo parâmetro, é a classe Instrumentation, que é a classe que fornece os serviços necessários para realizar a instrumentação. No corpo do método, a classe Transformer é instanciada e em seguida é registrada como uma classe transformadora através do método addTransformer que pertence à classe Instrumentation.

Figura 6 – Classe Agent

```
public class Agent {

      public static void premain(String agentArgs, Instrumentation inst) {

        System.out.println("\n");
        System.out.println("Starting Instrumentation.....");
        System.out.println("\n");

        Transformer transformer = new Transformer(agentArgs);
        inst.addTransformer(transformer);

      }

}
```

Fonte: O autor (2020).

A classe Transformer é a classe responsável por transformar os arquivos de uma classe, por este motivo, a classe Transformer implementa a interface ClassFileTransformer e o método transform. Além disso, a classe Transformer possui o atributo targetClass, esse atributo representa o nome da classe alvo, em outras palavras, a classe escolhida para a aplicação da métrica LCOM5.

Figura 7 – Classe Transformer

```
public class Transformer implements ClassFileTransformer {

    private String targetClass;

    public Transformer(String targetClass) {
        this.targetClass = targetClass;
    }

    public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
            ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassFormatException {


        if(className.contains(targetClass)) {

            LCOM5 lcom = new LCOM5();

            ClassReader classReader = new ClassReader(classfileBuffer);
            ClassWriter classWriter = new ClassWriter(classReader, 0);
            ClassVisitor classVisitor = new CustomClassVisitor(classWriter, lcom, className);

            classReader.accept(classVisitor, 0);

            printResultsLCOM(lcom, className);
        }

        return null;
    }
```

Fonte: O autor (2020).

O método transform é chamado para cada nova definição ou redefinição de classe e recebe os seguintes parâmetros:

- loader: o carregador de definição da classe a ser transformada.

- className: o FQL(Fully Qualified Name) da classe.
- classBeingRedefined: caso acionado, por uma redefinição ou transformação, a classe sendo redefinida ou transformada.
- protectionDomain: o domínio de proteção da classe sendo definida ou redefinida.
- classfileBuffer: os bytes de entrada no formato de arquivo da classe.

Em sua implementação, é feita uma validação para que a instrumentação só ocorra para a classe alvo, que será definida por parâmetro da linha de comando responsável por executar o Java Agente. Então, a biblioteca ASM é acionada e utilizamos a classe CustomClassVisitor para calcular a métrica LCOM5.

A classe CustomClassVisitor estende da classe ClassVisitor e tem como atributos: className que representa o nome da classe que está sendo analisada, e o atributo lcom, que é a classe responsável por calcular a métrica de coesão.

Figura 8 – Classe CustomClassVisitor

```
public class CustomClassVisitor extends ClassVisitor {

    private LCOM5 lcom;
    private String className;

    public CustomClassVisitor(ClassVisitor cv, LCOM5 lcom, String className) {
        super(Opcodes.ASM5, cv);
        this.lcom = lcom;
        this.className = className;
    }
```

Fonte: O autor (2020).

A classe também sobrescreve os métodos visitField e visitMethod. O primeiro método passa por todos os atributos de uma classe, e o segundo, passa por todos os métodos de uma classe. Em ambos os métodos, as informações de descrição, nome, assinatura, acesso e exceções que o método ou atributo possui são passados por parâmetro e podem ser acessados no corpo do método.

A implementação do método visitField é simples. O método é responsável apenas por somar o número de atributos de uma classe e informar esse número para a classe LCOM5.

Figura 9 – Método visitField da Classe CustomClassVisitor

```
@Override
public FieldVisitor visitField(int access, String name, String descriptor, String signature, Object value) {
    lcom.addNumberOfAtributes();
    return super.visitField(access, name, descriptor, signature, value);
}
```

Fonte: O autor (2020).

O método visitMethod, tem uma implementação similar ao método visitField. Porém, existe uma validação para ignorar o construtor da classe, para isso olhamos para o nome do método e esperamos que seja diferente de <init>. Em seguida, a mesma lógica implementada, porém, em vez de realizar uma contagem no número de atributos, é feita uma contagem no número de métodos de uma classe.

Ainda no método visitMethod, é feita a soma do número de atributos distintos que um método de uma classe acessa. Isso ocorre, no retorno do visitMethod onde é possível acessar os atributos de um método. Sendo assim, uma validação é feita sobre cada atributo do método, verificando se o mesmo pertence à mesma classe do método. Em outros termos, a soma de atributos distintos acessados por um método é feita apenas caso o método que está sob análise, utilize algum atributo da classe.

Figura 10 – Método visitMethod da Classe CustomClassVisitor

```
@Override
public MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions) {

    if(!name.equals("<init>")) {
        lcom.addNumberOfMethods();
        return new MethodVisitor(Opcodes.ASM5) {

            @Override
            public void visitFieldInsn(int opcode, String owner, String name, String descriptor) {

                if(owner.equals(className)) {
                    lcom.addNumberOfAttrAccessed();
                }

                super.visitFieldInsn(opcode, owner, name, descriptor);
            }

        };
    }

    return super.visitMethod(access, name, desc, signature, exceptions);
}
```

Fonte: O autor (2020).

Por último, a classe LCOM5, é responsável por calcular a soma de atributos, métodos e atributos distintos acessados por um método. Cada um desses cálculos é armazenado em um atributo da classe.

Figura 11 – Classe que cálculo a métrica LCOM5

```
public class LCOM5 implements Metric {

    private BigDecimal numberOfAtributes = BigDecimal.ZERO;
    private BigDecimal numberOfMethods = BigDecimal.ZERO;
    private BigDecimal numberOfAttrAccessed = BigDecimal.ZERO;

    public void addNumberOfAtributes() {
        this.numberOfAtributes = this.numberOfAtributes.add(BigDecimal.ONE);
    }

    public void addNumberOfMethods() {
        this.numberOfMethods = this.numberOfMethods.add(BigDecimal.ONE);
    }

    public void addNumberOfAttrAccessed() {
        this.numberOfAttrAccessed = this.numberOfAttrAccessed.add(BigDecimal.ONE);
    }
```

Fonte: O autor (2020).

Com o valor de cada atributo, é possível calcular o valor da LCOM5 da classe que está sendo instrumentada. Logo, a fórmula da LCOM5 é resolvida através do método getValue(). Esse método, aplica a fórmula descrita anteriormente, e retorna o valor da métrica.

Figura 12 – Método Responsável por calcular a fórmula de LCOM5

```
/***
 * LCOM5 = (a-mh)(h-mh)
 *
 * h = number of attributes
 * m = number of methods
 * a = summation of the number of definite attributes that are accessed by each method in a class.
 *
 * @return
 */
public BigDecimal getValue() {

    BigDecimal calc1 = numberOfAttrAccessed.subtract(numberOfMethods.multiply(numberOfAtributes));
    BigDecimal calc2 = numberOfAtributes.subtract(numberOfMethods.multiply(numberOfAtributes));

    return calc1.divide(calc2, RoundingMode.HALF_EVEN);
}
```

Fonte: O autor (2020).

Após a implementação da classe que calcula a métrica, é necessário informar no manifesto do projeto, o atributo que indica a classe responsável por implementar o método premain. Deste modo, após criar o arquivo .jar do Java Agente com o manifesto indicando a classe do Agente que implementará o método premain, podemos executar o Agente pela interface de linhas de comandos do Windows. Para isto, o seguinte comando é executado:

Figura 13 – Comando para executar o Java Agent

Fonte: O autor (2020).

O parâmetro informado por -javaagent aponta para o diretório onde se encontra o arquivo .jar do Java Agente criado, em seguida passamos como argumento o package das classes que serão instrumentadas. Em seguida, é informado o jar executável das classes que serão analisadas. A classe que implementa o método main do arquivo App.jar, possui a seguinte estrutura:

Figura 14 – Classe App

```
public class App {

    public static void main(String[] args){

        Trabalho t1 = new Trabalho("Programador", BigDecimal.TEN);
        Pessoa p1 = new Pessoa("Hempel", 20L);

        System.out.println("-----------------------------------------");
        System.out.println("Nome: " + p1.getNome());
        System.out.println("Idade: " + p1.getIdade());
        System.out.println("Trabalho: " + t1.getDescricao());
        System.out.println("-----------------------------------------");

    }

}
```

Fonte: O autor (2020).

A classe Pessoa, possui a seguinte estrutura:

Figura 15 – Classe Pessoa

```
public class Pessoa {

    private String nome;
    private Long idade;
    private String endereco;
    private String sexo;
    private String cpf;

    public Pessoa(String nome, Long idade) {
        this.nome = nome;
        this.idade = idade;
    }

    public String getNome() {
        return nome;
    }

    public Long getIdade() {
        return idade;
    }

    public Trabalho getTrabalho() {
        return new Trabalho();
    }

    public void doNothing() {

    }

    public void acelerar() {
        Carro carro = new Carro();
        carro.acelerar();
    }

}
```

Fonte: O autor (2020).

A classe Trabalho, possui a seguinte estrutura:

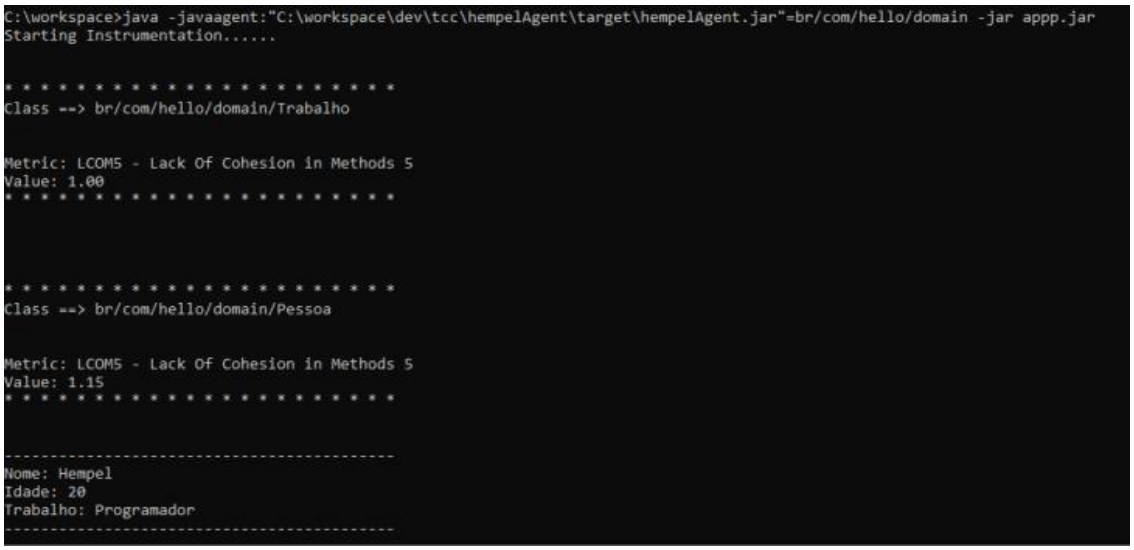
Figura 16 – Classe Trabalho

```
public class Trabalho {

    private String descricao;
    private BigDecimal salario;

    public Trabalho() {}

    public Trabalho(String descricao, BigDecimal salario) {
        this.descricao = descricao;
        this.salario = salario;
    }

    public String getDescricao() {
        return descricao;
    }

    public BigDecimal getSalario() {
        return salario;
    }

    public void trabalhar() {}

    public Long getAnosDeTrabaho() {
        return 1L;
    }

}
```

Fonte: O autor (2020).

Por conseguinte, após executar o comando, o seguinte resultado é obtido:

Figura 17 – Resultado da Instrumentação



Fonte: O autor (2020).

A primeira informação é de que a instrumentação começou, em seguida temos os nomes das classes, o nome da métrica e seu valor. Desta maneira o algoritmo foi capaz de extrair o valor de LCOM5 das classes Pessoa e Trabalho. Após o fim do método premain e o fim da instrumentação, o programa Java é executado normalmente com a chamada do método main.

6 CONCLUSÃO

O presente trabalho demonstrou a importância das métricas de código-fonte, mais especificamente, as métricas de coesão e acoplamento, e como a utilização dessas métricas em um projeto de software podem simplificar o desenvolvimento da aplicação.

Com o objetivo de identificar a baixa coesão em um projeto, foi criado um algoritmo capaz de extrair a métrica de coesão de um código analisado, com a finalidade de que o programador consiga identificar a baixa coesão e aplicar a correção necessária para garantir um código legível e de fácil manutenibilidade.

Foi realizada uma pesquisa sobre como uma ferramenta de cobertura de testes realiza a análise do código com a finalidade de aplicar o mesmo conceito para extrair a métrica de

coesão LCOM5 de um código-fonte. Portanto, assim como no JaCoCo, foi utilizada a API de instrumentação do Java em conjunto com a biblioteca de manipulação de bytecode ASM no algoritmo capaz de calcular a LCOM5.

Nesse contexto, este trabalho pode servir de inspiração para futuras pesquisas e projetos relacionados às métricas de coesão e acoplamento e como realizar o cálculo dessas métricas utilizando a instrumentação do código, que é uma ferramenta extraordinária e oferece outros mecanismos de instrumentação. Sendo assim, como trabalho futuro, pode-se aplicar a API de instrumentação do Java para o cálculo de outras métricas, como a de acoplamento, utilizando outros recursos que a API oferece, como por exemplo, executar um Java Agent, após a inicialização da JVM.

REFERÊNCIAS

ABNT. Associação Brasileira de Normas Técnicas. **NBR ISO/IEC9126-1**. Engenharia de software. Qualidade de produto: Parte 1. Rio de Janeiro, 2003.

ASM. **A Java bytecode engineering library.** 2020. Disponível em: <https://asm.ow2.io/index.html>. Acesso em: 18 out. 2020.

BADRI, Linda; BADRI, Mourad; TOURE, Fadel. Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems. **Conferência Internacional sobre Engenharia de Software Avançada e Suas Aplicações,** ASEA 2010, Ilha de Jeju, Coréia, v. 117, dez. 2010.

BALAKRISHAN, Sathiyakugan. **Understanding Java Agents.** 2020. Disponível em: <https://dzone.com/articles/java-agent-1>. Acesso em: 17 out. 2020.

BECK, Fabian; DIEHL, Stephan. **On the congruence of modularity and code coupling.** New York, NY, USA: Association for Computing Machinery, 2011. p. 354-364.

BIEMAN, James M.; KANG, Byung-Kyoo. **Cohesion and reuse in an object-oriented system**. New York, NY, USA: Association for Computing Machinery, 1995. v. 20, p. 259-262.

BONJA, Challa; KIDANMARIAM, Eyob. **Metrics for class cohesion and similarity between methods.** New York, NY, USA: Association for Computing Machinery, 2006. p. 91-95.

CHIDAMBER, Shyam R.; KEMERER, Chris F. **A metrics suite for object oriented design.** IEEE Transactions on Software Engineering, 1994. v. 20, p. 476-493.

CHIDAMBER, Shyam R.; KEMERER, Chris F. **Towards a metrics suite for object oriented design.** New York, NY, USA: Association for Computing Machinery, 1991. p. 197-211.

CONSTANTINE, L.; STEVENS, W.; MYERS, G. Structured design. **IBM Systems Journal,** 1974. v. 13, p. 115-139.

DARCY, David P. et al. The structural complexity of software an experimental test. **IEEE Transactions on Software Engineering**, 2005. v. 31, p. 982-995.

DEL ESPOSTE, Arthur de Moura; BEZERRA, Carlos Filipe Lima. **Cenário de decisões baseado em métricas de software:** definição e implementação de cenários a partir de métricas de design e de vulnerabilidade para tomada de decisão. 2014. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software)– Universidade de Brasília, Brasília, 2014.

DOMINGOS, Samuel António Miquirice. **Study on the relationships between cohesion and coupling metrics on fault prediction in object oriented systems**. 2018. Dissertação (Mestrado em Ciências da Computação)– Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2018.

FERREIRA, Kecia Aline Marques. **Um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos.** 2011. 224 f. Tese (Doutorado em Engenharia de software)– Universidade Federal de Minas Gerais, Belo Horizonte, 2011.

HARON, Nur Hafizah; SYED-MOHAMAD, Sharifah Mashita. Test and Defect Coverage Analytics Model for the Assessment of Software Test Adequacy. **Kuala Lumpur**, Malaysia: 9th Malaysian Software Engineering Conference, 2015. P. 13–18.

HENDERSON-SELLERS, B. **Software metrics.** Hemel Hempstaed, UK: Prentice Hall, 1996.

HENRY, Sallie; LI, Wei. **Maintenance metric for the object oriented paradigm.** Baltimore, MD, USA: Proceedings First International Software Metrics Symposium, 1993. P. 52–60.

HERZIG, Andreas; VARZINCZAK, Ivan. Cohesion, coupling and the meta-theory of actions. **Proceedings of the 19th International Joint Conference on Artificial Intelligence,** Edinburgh, Scotland: 2005. p. 442-447.

HITZ, Martin; MONTAZERI, Behzad. Measuring coupling and cohesion in object-oriented systems. **Proc. Int. Symposium on Applied Corporate Computing,** Monterrey, Mexico, 1995.

IEEE. **Standard Glossary of Software Engineering Terminology**. IEEE Std 610.12-1990, p. 1-84, 1990.

JACOCO. **Java Code Coverage Library**. 2017. Disponível em: <https://www.eclemma.org/jacoco/>. Acesso em: 3 out. 2020.

JAIN, Vaibhav; GUPTA, Arpit. **Lack of conceptual cohesion of methods: a new alternative to lack of cohesion of methods.** New York, NY, USA: Association for Computing Machinery, 2015.

JAVA™. **Platform, Standard Edition.** 7 API Specification. Disponível em: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>. Acesso em: 17 out. 2020.

LINO, Carlos Eduardo. **Reestruturação de software com adoção de padrões de projeto para a melhoria da manutenibilidade**. 2011. 68f. Trabalho de Conclusão de Curso (Graduação em Sistema de Informação)– Universidade Federal de Lavras, Lavras-MG, 2011.

MALAIYA, Y. K. et al. Software reliability growth with test coverage. **IEEE Transactions on Reliability**, 2002. v. 51, p. 420-426.

MARTIN, Robert C. **Agile software development, principles, patterns, and practices**. USA: Pearson Education, 2003. v. 2.

MEIRELLES, Paulo Roberto Miranda. **Monitoramento de métricas de código-fonte em projetos de software livre.** 2013. Tese (Doutorado em Ciência da Computação)– Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013.

MIDDING, Rogerio Cristiano. **Avaliação da qualidade interna de softwares desenvolvidos com a técnica TDD.** Trabalho de Conclusão de Curso (Graduação)– Universidade Tecnológica Federal do Paraná, Toledo, 2016.

PRESSMAN, Roger S. **Software engineering:** a practitioner's approach. New York: McGraw Hill, 2001. v. 5.

PULS, Ashley. **Diving into bytecode manipulation:** creating an audit log with asm and javassist. 2014. Disponível em: <https://blog.newrelic.com/engineering/diving-bytecode-manipulation-creating-audit-log-asm-javassist/>. Acesso em: 18 out. 2020.

SOMMERVILLE, Ian. **Software engineering.** New Jersey: Pearson Prentice Hall, 2011, v. 1, p. 1-500.

WOODWARD, M. R. Difficulties using cohesion and coupling as quality indicators. [S.l.]: **Software Quality Journal** 2, 1993. p. 109-127.

ZHU, Hong; HALL, Patrick A. V.; MAY, John H. R. **Software unit test coverage and adequacy.** New York, NY, USA: Association for Computing Machinery, 1997. v. 29, p. 366–427.