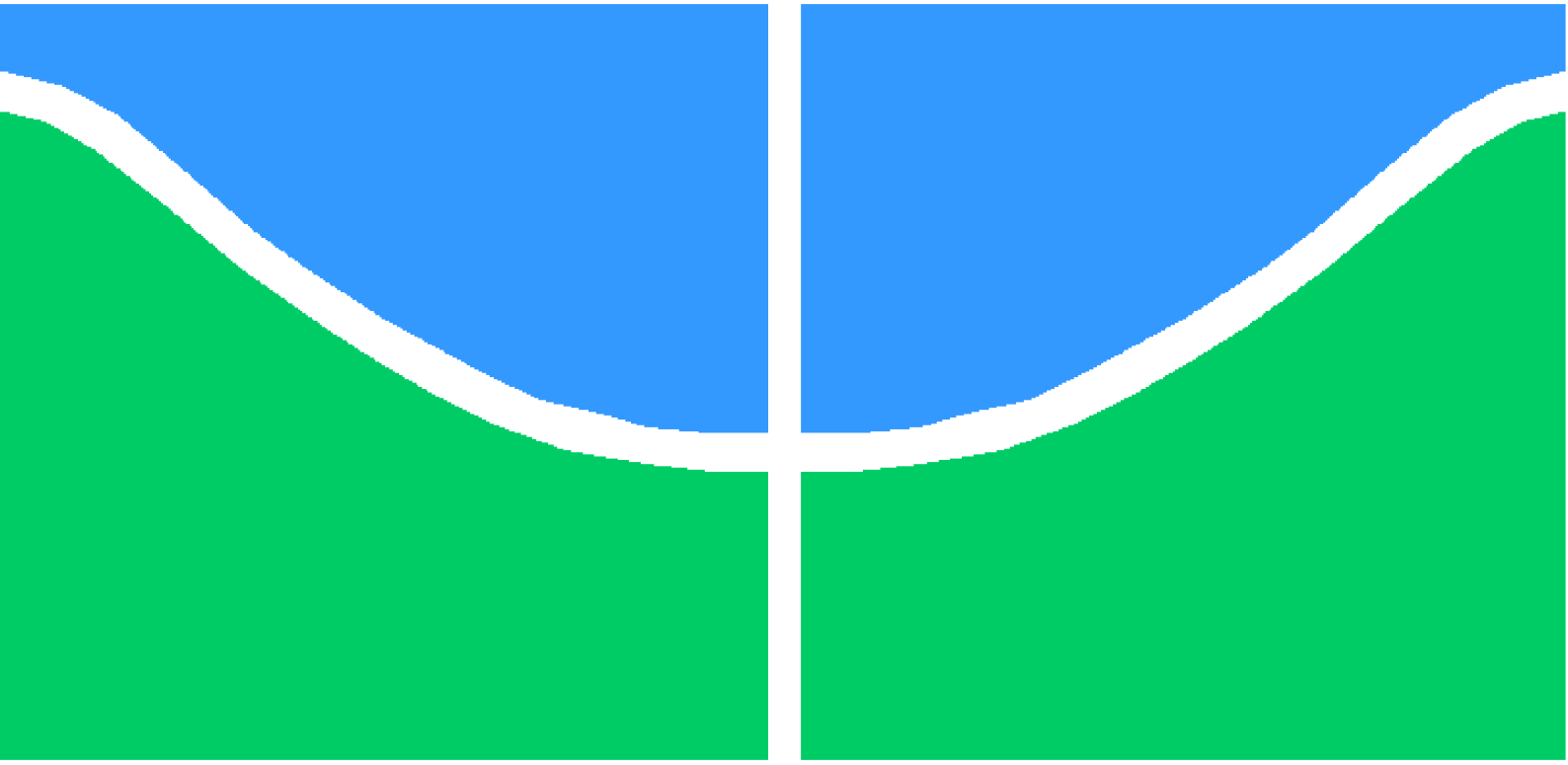


Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Um estudo empírico sobre métricas de código fonte do Android API Framework

Autor: Victor Henrique Magalhães Fernandes
Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF
2018

Victor Henrique Magalhães Fernandes

Um estudo empírico sobre métricas de código fonte do Android API Framework

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2018

Victor Henrique Magalhães Fernandes
Um estudo empírico sobre métricas de código fonte do Android API Framework/ Victor Henrique Magalhães Fernandes. – Brasília, DF, 2018-
64 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. Android. 2. Métricas de código-fonte. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Um estudo empírico sobre métricas de código fonte do Android API Framework

CDU 02:141:005.6

Victor Henrique Magalhães Fernandes

Um estudo empírico sobre métricas de código fonte do Android API Framework

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, Agosto de 2018:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Dr. Gustavo Santos
Convidado 1

Me. Joenio Marques da Costa
Convidado 2

Brasília, DF
2018

Resumo

Métricas de código-fonte são comumente usadas para avaliar a qualidade interna de aplicativos de software. Para interpretar valores métricos, a literatura sugere valores limites, por exemplo, uma classe cujo valor métrico excede um dado limite é considerado como tendo problemas de manutenção. No entanto, não existe uma regra para identificar um limite que seja útil, pertinente e fácil de explicar. Neste trabalho, propomos medir a qualidade interna de um sistema quando ele faz parte de um ecossistema maior. Nós nos concentramos no ecossistema do Android. Nós computamos métricas conhecidas de código-fonte, como AMLOC e ACCM. Abordamos quatro aspectos: (i) analisamos a distribuição de valores métricos em várias versões do Android API Framework; (ii) extraímos limites de métricas com base nessas distribuições; (iii) utilizamos uma abordagem para extrair pontuações de qualidade para sistemas Android, comparando as distribuições métricas com as computadas no framework subjacente; e (iv) validamos essa abordagem para verificar se o índice de qualidade é realmente capaz de inferir problemas de manutenibilidade e design. Com isso, foi possível definir intervalos de referência com base na API do sistema Android, que podem auxiliar novos desenvolvedores de aplicativos, a encontrar possíveis problemas de manutenção.

Palavras-chaves: Engenharia de Software. Análise estática de código-fonte. Métricas de código-fonte. Android API.

Abstract

Source code metrics are commonly used to evaluate internal quality of software applications. To interpret metric values, the literature suggests thresholds, e.g., a class whose metric value exceeds a given threshold is considered to have maintenance problems. However, there is no rule of thumb to identify a threshold that is useful, pertinent, and easy to explain. In this paper, we propose to measure the internal quality of a system when it is part of a larger ecosystem. We focus on the Android ecosystem. We compute well-known source code metrics, such as AMLOC and ACCM. We cover four aspects: (i) we analyze the distribution of metric values in several versions of Android API Framework; (ii) we extract metric thresholds based on these distributions; (iii) we propose an approach to extract quality scores for Android systems, by comparing the metric distributions with the ones computed in the underlying framework; and (iv) we validate this approach to check whether the quality score is indeed able to infer maintainability and design problems. This enabled we to set reference ranges based on the Android system API, which can help new Application developers find possible maintenance issues.

Key-words: Software Engineering. Source Code Static Analysis. Source Code Metrics. Android API.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AOSP	Android Open Source Project
API	Application Programming Interface
UnB	Universidade de Brasília
LOC	Lines of Code
AMLOC	Average Method Lines of Code
ACCM	Average Cyclomatic Complexity per Method
ANPM	Average Number of Parameters per Method
ACC	Afferent Connections per Class
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
LCOM4	Lack of Cohesion in Methods
MMLOC	Max Method LOC
NOA	Number of Attributes;
NOC	Number of Children
NOM	Number of Methods
NPA	Number of Public Attributes
NPM	Number of Public Methods
RFC	Response For a Class
SC	Structural Complexity

Sumário

1 Introdução

Os sistemas de software hoje em dia são frequentemente parte de ecossistemas maiores (LUNGU, 2009). Esses ecossistemas geralmente existem em grandes organizações, universidades ou comunidades de código aberto, como Eclipse, Android, Apache, distribuições Linux, entre outras. Portanto, assumimos que os principais desenvolvedores seguem um conjunto de padrões de codificação que refletem como os componentes principais do ecossistema devem ser usados pelos projetos do cliente, além de como esses componentes devem atender a fatores internos de qualidade, como modularidade e complexidade.

No entanto, é difícil prever como um ecossistema de software será usado por seus clientes (HORA, 2014). Por exemplo, os padrões de codificação definidos pela equipe de projeto do Android podem não ser garantidos para todos os aplicativos Android. Com projetos de clientes crescendo constantemente em número, complexidade e tamanho, é necessário um melhor suporte para medir e controlar a qualidade do software. Métricas de código-fonte são úteis para medir fatores internos de qualidade, como modularidade, acoplamento, coesão, tamanho, herança e complexidade.

Ainda assim, na prática, as métricas do código fonte raramente são usadas para controlar a qualidade do software de maneira efetiva (FENTON; NEIL, 2000); (OLIVEIRA et al., 2015). Mais especificamente, várias pesquisas foram conduzidas para estabelecer limites métricos, conforme definidos por Lorenz e Kidd (LORENZ; KIDD, 1994b), como valores heurísticos usados para definir faixas de valores métricos desejáveis e indesejáveis para o software medido (ALVES; YPMA; VISSER, 2010); (FERREIRA et al., 2012); (SHATNAWI, 2015). A definição de intervalos de referência confiáveis para métricas de código fonte que poderiam ser generalizados para vários sistemas de software provou ser um desafio. Consequentemente, avaliar um sistema de software apenas com base em valores métricos não é uma tarefa trivial.

Apesar de o Android ser popular e as métricas do código fonte serem propostas pela literatura, há uma falta de compreensão profunda sobre o ecossistema do Android, especialmente sobre as métricas do código-fonte. Com a evolução dos ecossistemas, como o Android, algumas diretrizes específicas do sistema são necessárias para realizar análises contínuas e controlar a qualidade interna e, em última instância, beneficiar tanto os principais desenvolvedores quanto os aplicativos clientes.

Este trabalho apresenta um estudo sobre a qualidade interna do Android API Framework, ou seja, seus principais componentes, bem como alguns de seus aplicativos clientes, usando métricas de código-fonte existentes. Nosso objetivo geral é avaliar os

aplicativos clientes do Android no contexto dos principais componentes do Android. Mais especificamente, a distribuição de valores métricos entre os componentes do Android é usada como referência para avaliar a qualidade interna de seus aplicativos clientes.

Neste estudo, primeiramente investigamos a evolução da distribuição dos valores métricos em várias versões do Android, tanto para o Android API Framework quanto para alguns de seus aplicativos clientes. E para isso, calculamos métricas estáticas de código-fonte, como linhas de código e complexidade ciclomática. Segundamente, usamos uma abordagem existente proposta por (MEIRELLES, 2013), para extrair limiares métricos do Android API Framework, que separa os valores métricos em vários percentis e comparamos com a abordagem proposta por (LANZA; MARINESCU, 2006) que utiliza a média e desvio padrão para analisar a distribuição dos valores métricos. E por fim, utilizamos a abordagem proposta por (JUNIOR, 2015) para calcular a similiaridade entre (i) a distribuição de valores métricos para um aplicativo cliente Android e (ii) a distribuição de valores métricos que calculamos na etapa anterior para o Android API Framework.

O restante deste trabalho está organizado da seguinte maneira: O Capítulo 2 introduz o ecossistema Android e apresenta um histórico sobre métricas de código-fonte. O capítulo 3 descreve a metodologia da pesquisa incluindo coleta e análise de dados. O capítulo 4 apresenta e discute a análise da distribuição de valores métricos no Android API Framework. O capítulo 5 apresenta e discute os resultados da análise dos aplicativos cliente Android. Por fim, o capítulo 6 conclui o estudo, destacando suas principais contribuições e apontando para possibilidades de trabalhos futuros.

2 Fundamentação Teórica

Este capítulo contextualiza o presente trabalho, apresentando conceitos que serão discutidos em cápitulos posteriores. Primeiramente, são apresentados conceitos sobre o ecossistema Android. Em seguida, definimos o conceito de análise estática e são apresentadas as métricas de código-fonte que serão utilizadas neste trabalho. Por fim, descrevemos os trabalhos relacionados investigados nesta pesquisa.

2.1 Android

O Android é um sistema operacional para dispositivos móveis baseado em uma versão modificada do kernel do Linux. Atualmente, o sistema não se concentra apenas no desenvolvimento de smartphones e tablets, mas também em Smart TVs, carros, PCs, relógios de pulso e outros dispositivos eletrônicos. Desde 2011, o Android se tornou o sistema operacional mais vendido no mundo inteiro em smartphones. Esse mercado enorme, incluindo bilhões de aplicativos baixados e milhares de diferentes marcas de dispositivos móveis, atraiu milhares de desenvolvedores (CRAVENS, 2012) ; (MOJICA et al., 2014).

No que diz respeito à manutenção de software, vários desafios estão especificamente associados ao desenvolvimento móvel (SYER et al., 2011). O hardware está em constante evolução: os dispositivos móveis vêm em diferentes formas e tamanhos, diferentes fabricantes, diferentes recursos de CPU e memória e todos os tipos de sensores. Consequentemente, o software também está evoluindo rapidamente. O sistema operacional em si é considerado fragmentado: enquanto novos dispositivos são construídos com a atualização mais recente do sistema operacional, dispositivos antigos ainda ativos ainda executam versões anteriores do sistema. Portanto, os desenvolvedores devem estar cientes das diferentes configurações de dispositivos em que seus aplicativos podem ser enfrentados e também de todas as principais atualizações do sistema operacional.

Além das preocupações do mercado, também é importante que os aplicativos do Android, como qualquer software, sigam algumas diretrizes de qualidade de software. Entender o código-fonte do ecossistema Android beneficia os desenvolvedores durante o desenvolvimento e a manutenção de aplicativos clientes. Uma prática típica para executar a análise de qualidade é realizar uma análise estática para calcular e avaliar as métricas do código-fonte. Na Subseção 2.1.1, introduzimos conceitos básicos sobre a arquitetura de uma aplicação Android e na Seção 2.2, descrevemos a análise estática de código-fonte bem como as métricas que calculamos em nosso estudo.

2.1.1 Arquitetura de uma aplicação Android

Aplicações no Android são construídas a partir de quatro tipos de componentes principais: *Activities*, *Services*, *Broadcast Receivers*, e *Content Providers* [1].

1. Uma *Activity* é basicamente o código para uma tarefa bem específica a ser realizada pelo usuário, e apresenta uma interface gráfica(*Graphic User Interface*) para a realização dessa tarefa.

2. *Services* são tarefas que são executadas em background, sem interação com o usuário. *Services* podem funcionar no processo principal de uma aplicação ou no seu próprio processo. Um bom exemplo de services são os tocadores de músicas. Mesmo que sua interface gráfica não esteja mais visível, é esperado que a música continue a tocar, mesmo se o usuário estiver interagindo com outro aplicativo.

3. *Broadcast Receiver* é um componente que é chamado quando um *Intent* é criado e enviado via *broadcast* por alguma aplicação ou pelo sistema. *Intents* são mecanismos para comunicação entre processos, podendo informar algum evento, ou transmitir dados de um para o outro. Um aplicativo pode receber um *Intent* criado por outro aplicativo, ou mesmo receber intents do próprio sistema, como por exemplo informação de que a bateria está fraca ou de que uma busca por dispositivos bluetooth previamente requisitada foi concluída.

4. *Content Providers* são componentes que gerenciam o acesso a um conjunto de dados. São utilizados para criar um ponto de acesso a determinada informação para outras aplicações. Para os contatos do sistema, por exemplo, existe um *Content Provider* responsável por gerenciar leitura e escrita desses contatos.

Cada um desses componentes pode funcionar independente dos demais. O sistema Android foi desenvolvido dessa forma para que uma tarefa mais complexa seja concluída com a ajuda e interação de vários desses componentes independente da aplicação a qual eles pertencem, não necessitando que um desenvolvedor crie mecanismos para todas as etapas de uma atividade mais longa do usuário.

Para executar a tarefa de ler um email, por exemplo, um usuário instala um aplicativo de gerenciamento de emails. Então ele deseja abrir um anexo de um email que está em formato PDF. O aplicativo de email não precisa necessariamente prover um leitor de PDF para que o usuário consiga ter acesso a esse anexo. Ele pode mandar ao sistema a intenção de abrir um arquivo PDF a partir de um *Intent*, e então o sistema encontra um outro componente que pode fazer isso, e o instancia. Caso mais de um seja encontrado, o

[1] Esta Seção é baseada na documentação oficial do Android disponível em <http://developer.android.com/develop/index.html>

sistema pergunta para o usuário qual é o componente que ele deseja utilizar. O sistema então invoca uma *Activity* de um outro aplicativo para abrir esse arquivo. Continuando no mesmo exemplo, o usuário clica em um link dentro do arquivo pdf. Esse aplicativo, por sua vez, pode enviar ao sistema a intenção de abrir um endereço web, que mais uma vez encontra um aplicativo capaz de o fazer.

Aplicativos para o sistema Android são construídos em módulos, utilizando os componentes da API, embora possam ser criadas classes em Java puro sem a utilização de nenhum recurso da API do sistema, e utilizá-las nos componentes assim como em uma aplicação Java padrão desenvolvida para desktop.

De forma geral, para desenhar uma boa arquitetura para sistema Android deve-se levar em conta todos esses componentes e conhecer bem sua aplicabilidade e a comunicação entre os mesmos, de modo que isso possa refletir na qualidade do código.

2.2 Análise Estática de Código-fonte

Grande parte das falhas no desenvolvimento de um software é dada por erros humanos (TERRA; BIGONHA, 2008). O desenvolvimento de software depende principalmente da capacidade técnica das pessoas que os desenvolvem, e por isso está sujeito a erros. Sendo assim, torna-se imprescindível a utilização de mecanismos de revisão de código-fonte que facilitem a descoberta desses erros (TEIXEIRA; ANTUNES; NEVES, 2007).

Um dos primeiros passos para detecção e correção de vulnerabilidades era realizado através da revisão manual do código fonte, onde auditores revisavam o código minuciosamente afim de identificar possíveis vulnerabilidades. Porém, esse processo acaba por ser muito lento e caro, surgindo então, mecanismos de revisão computacionais como a análise estática (TEIXEIRA; ANTUNES; NEVES, 2007).

Análise estática de código é o processo de avaliação de um sistema ou componente baseado na sua forma, estrutura, conteúdo ou documentação. Da perspectiva de análise de software, a análise estática permite localizar fraquezas no código de programas, que podem levar a uma vulnerabilidade. Ela pode ser feita de forma manual, como a inspeção de um código, por exemplo, ou automatizada, através do uso de uma ou mais ferramentas.

Apesar de não ser uma nova tecnologia, a análise estática de código-fonte vem sendo crucial em todo o ciclo de vida do desenvolvimento de software, auxiliando na identificação de pontos de baixa qualidade, confibialidade e segurança do software (KANNAVARA, 2012).

2.2.1 Ferramentas de análise estática de código-fonte

Uma ferramenta de análise estática, analisa um programa à procura de possíveis problemas no código-fonte, de forma automatizada, sem a necessidade que esse programa seja executado (KANNAVARA, 2015). Ao final dessa análise, essas ferramentas emitem um relatório que descrevem os potenciais problemas que podem vir a existir. Logo após, é necessário apenas que o programador interprete os resultados a partir do relatório emitido (TEIXEIRA; ANTUNES; NEVES, 2007). Essas ferramentas provaram ser um dos primeiros passos para identificação de erros e são fundamentais para o sucesso total de um software (KANNAVARA, 2012).

Existem diversas ferramentas de análise estática, e para este estudo selecionamos a ferramenta *Analizo*. Decidimos utilizar a *Analizo* por ela ser mantida constantemente, possuir desenvolvedores ativos que conhecem a arquitetura da ferramenta e tem a capacidade de fornecer atualizações e corrigir problemas. Além disso, a *Analizo* é capaz de manipular o código-fonte que não pode ser compilado. Por exemplo, o código pode ter erros de sintaxe, as bibliotecas usadas podem não estar mais disponíveis, ou ainda a API das bibliotecas utilizadas podem ter mudado. Isso é importante para sermos capazes de analisar o código-fonte legado em estudos de evolução do software. No caso do Android, podemos analisar várias versões diferentes incluindo as que não são mais mantidas.

2.2.1.1 Analizo

O *Analizo* é um kit de ferramentas de código aberto multilinguagem, que executa análise estática para softwares desenvolvidos em C, C ++ e Java. A arquitetura do Analizo, pode ser representada pela Figura 1, onde cada camada no diagrama usa apenas os serviços fornecidos pelas camadas diretamente abaixo dela (TERCEIRO et al., 2010).

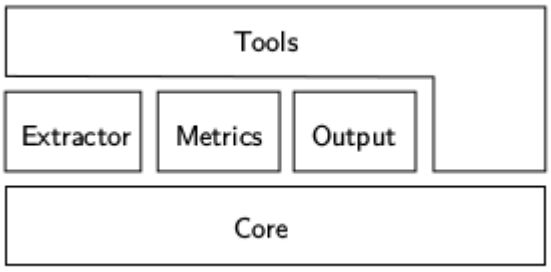


Figura 1 – Arquitetura do Analizo.

A camada *Core* contém as estruturas de dados usadas para armazenar informações relativas ao código-fonte que está sendo analisado, como a lista de módulos existentes, elementos dentro de cada módulo (atributos, variáveis ou métodos, funções) e informações de dependência (chamada, herança, etc). Essa camada implementa a maior parte da lógica de negócios do Analizo e não depende de nenhuma outra camada.

A camada *Extractors* compreende as diferentes estratégias de extração de informações de código-fonte construídas no Analizo. Os extratores obtêm informações do código-fonte e as armazenam nas estruturas de dados da camada *Core*. Atualmente, existem dois extratores. Ambos são interfaces para ferramentas externas de análise de código-fonte:

- Analizo::Extractors::Doxyparse é uma interface para o Doxyparse uma analisador de código- fonte para C, C++ e Java desenvolvido e mantido pelo grupo de colaboradores da Analizo (COSTA, 2009). Doxyparse é baseado no Doxygen [2], uma ferramenta livre e multi-linguagem para a documentação de código-fonte. Ela contém um analisador que atendeu nossas demandas neste trabalho.
- Analizo::Extractors::Sloccount é uma interface para o Sloccount [3], uma ferramenta que calcula o número de linhas de código efetivas, ou seja, sem comentários e linhas em branco.

As outras camadas intermediárias são *Metrics* e *Output*. A camada *Metrics* processa as estruturas de dados principais para calcular as métricas. A camada *Output* é responsável por lidar com diferentes formatos de arquivo.

A camada *Tools* compreende um conjunto de ferramentas de linha de comando que constituem a interface Analizo para usuários e aplicativos de nível superior. Essas ferramentas usam serviços fornecidos pelas outras camadas.

Atualmente, o Analizo calcula métricas tanto em nível de projeto, que são calculadas a partir de um valor que agrega dados de todos os arquivos do projeto, quanto métricas no nível de classe. Neste trabalho, estamos interessados principalmente nas métricas no nível de classe, que são:

- **LOC** (*Lines of Code* - Número de Linhas de Código);
- **AMLOC** (*Average Method Lines of Code* - Média do número de linhas de código por método);
- **ACCM** (*Average Cyclomatic Complexity per Method* - Média da Complexidade Ciclomática por Método);
- **ANPM** (*Average Number of Parameters per Method* - Média do Número de Parâmetros por Método);
- **ACC** (*Afferent Connections per Class* - Conexões aferentes de uma classe);
- **CBO** (*Coupling Between Objects* - Acoplamento entre objetos);

[2] <doxygen.org/>
[3] <dwheeler.com/sloccount/>

- **DIT** (*Depth of Inheritance Tree* - Profundidade da árvore de herança;
- **LCOM4** (*Lack of Cohesion in Methods* - Ausência de coesão em métodos);
- **MMLOC** (*Max Method LOC* - Número de linhas de código do maior método da classe);
- **NOA** (*Number of Attributes* - Número de atributos);
- **NOC** (*Number of Children* - Número de filhos);
- **NOM** (*Number of Methods* - Número de métodos);
- **NPA** (*Number of Public Attributes* - Número de atributos públicos);
- **NPM** (*Number of Public Methods* - Número de métodos públicos);
- **RFC** (*Response For a Class* - Respostas para uma classe);
- **SC** (*Structural Complexity* - Complexidade Estrutural).

O módulo de cálculo de métricas da Analizo possui um comando para o processamento em lote de métricas. Na maioria dos estudos quantitativos sobre engenharia de software envolvendo a aquisição de métricas de código-fonte em um grande número de projetos, processar cada projeto individualmente é impraticável, propenso a erros e difícil de repetir. O Analizo pode processar vários projetos em lote e produzir um arquivo de dados de métricas de valores separados por vírgula (CSV) para cada projeto, bem como um arquivo de dados CSV de resumo com métricas no nível do projeto para todos os projetos. Esses arquivos de dados podem ser facilmente importados em ferramentas estatísticas ou em software de planilhas para análise posterior. Isso também pode ser usado para analisar várias versões do mesmo projeto, em estudos de evolução de software. Com esse recurso, foi possível analisar as diferentes versões do Android.

2.2.2 Métricas de código-fonte

As informações extraídas do códido-fonte a partir das ferramentas de análise estática são usadas para medir métricas de código-fonte. As métricas do código-fonte medem as propriedades internas do código-fonte de um sistema de software.

Existem diferentes tipos de métricas e nesse trabalho estamos interessados em métricas de design, principalmente as métricas projetadas para sistemas orientados a objetos, como métricas de tamanho, complexidade, coesão e acomplamento. Para tal, foram selecionadas as métricas no nível de classe suportadas pela ferramenta *Analizo*.

O estudo de (MEIRELLES, 2013), mostra que as métricas possuem uma autocorrelação estatística entre si, ou seja, uma determinada métrica explica outra do ponto de

vista estatístico, não sendo necessário coletar e usar todas as correlatas. Logo, entre as métricas citadas na Seção 2.2.1.1, selecionamos as métricas mais difundidas na literatura e as classificamos em três classes: Métricas de Tamanho, Métricas de Complexidade e Métricas de Coesão e Acoplamento.

O código a seguir será usado como exemplo para demonstar como são calculadas as métricas.

```
1  public class HelloWorld {
2  // This comment will not count
3  public static void main(String args[]){
4  Printer printer = new HelloWorldPrinter();
5  printer.print();
6  }
7  }
8
9  class Printer {
10 private String message;
11
12 public Printer(String msg){
13 message = msg;
14 }
15
16 public void print(){
17 System.out.println(message);
18 }
19 }
20
21 class HelloWorldPrinter extends Printer {
22 public HelloWorldPrinter(){
23 super("Hello World!");
24 }
25
26 public void doNothing(){
27 // a method
28 }
29 }
```

Listing 2.1 – Exemplo código em Java.

2.2.2.1 Métricas de Tamanhho

Métricas de tamanho foram desenvolvidas para tentar quantificar o tamanho e auxiliar medições na fase de concepção de software. Nesta Seção, temos as seguintes

métricas de tamanho:

- **AMLOC** (*Average Method Lines of Code* - Média do número de linhas de código por método). Indica se o código está bem distribuído entre os métodos. Quanto maior, mais pesados são os métodos. É preferível ter muitas operações pequenas e de fácil entendimento que poucas operações grandes e complexas. No exemplo, a classe ***Printer*** possui AMLOC = 3, pois possui dois métodos e 6 linhas de código.

- **ANPM** (*Average Number of Parameters per Method* - Média do Número de Parâmetros por Método). Calcula a média de parâmetros dos métodos da classe. Seu valor mínimo é zero e não existe um limite máximo, porém, valores altos podem indicar que o método possui muitas responsabilidades (BANSIYA; DAVI., 1997). No exemplo, a classe ***Printer*** possui ANPM = 1, pois temos dois métodos e apenas um parâmetro, logo o valor é arredondado para 1.

- **NOA** (*Number of Attributes* - Número de atributos). Calcula o número de atributos de uma classe. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado. Uma classe com muitos atributos pode indicar que ela tem muitas responsabilidades e apresentar pouca coesão. No exemplo, a classe ***Printer*** possui NOA = 1, pois possui apenas um atributo.

- **NOM** (*Number of Methods* - Número de métodos). É usado para medir o tamanho das classes em termos das suas operações implementadas. Esta métrica pode identificar o potencial de reúso de uma classe. Em geral, as classes com um grande número de métodos são mais difíceis de serem reutilizadas, pois elas são propensas a serem menos coesas (LORENZ; KIDD, 1994a). No exemplo, a classe ***HelloWorldPrinter*** possui NOM = 2, pois possui 2 métodos.

2.2.2.2 Métricas de Complexidade

Métricas de complexidade medem a complexidade de uma parte do software. A alta complexidade deve ser evitada pois prejudica a compreensão do software e o torna mais suscetível a erros. Nesta Seção, temos a seguinte métrica de complexidade:

- **ACCM** (*Average Cyclomatic Complexity per Method* - Média da Complexidade Ciclomática por Método). Mede a complexidade do programa e pode ser representada através de um grafo de fluxo de controle (MCCABE, 1976). O uso de estruturas de controle, como, *if*, *else* e *for* aumentam a complexidade ciclomática de um método. No exemplo, o método ***print()*** na classe ***Printer*** possui ACCM = 1, pois apresenta apenas um fluxo de execução.

2.2.2.3 Métricas de Coesão e Acoplamento

Pode-se definir coesão de um sistema de software como a relação existente entre as responsabilidades de uma classe e cada um de seus métodos. Altos valores de coesão para uma classe, indicam que essa possui responsabilidades em um único aspecto do sistema. Por outro lado, uma classe com baixa coesão, indica que essa possui muitas responsabilidades no sistema.

O acoplamento mede o grau de interdependência entre as classes de um software. Altos valores de acoplamento para uma classe, indica uma maior dificuldade a mudanças, visto que todas as classes acopladas a esta classe sofrem impacto.

Nesta Seção, temos as seguintes métricas de coesão e acoplamento:

- **DIT** (*Depth of Inheritance Tree* - Profundidade da árvore de herança). É o número de superclasses ou classes ancestrais da classe sendo analisada (CHIDAMBER; KEMERER, 1994). São contabilizadas apenas as superclasses do sistema, ou seja, as classes de bibliotecas não são contabilizadas. Nos casos onde herança múltipla é permitida, considera-se o maior caminho da classe até uma das raízes da hierarquia. Quanto maior for o valor DIT, maior é o número de atributos e métodos herdados, e portanto maior é o acoplamento. Entretanto, valores baixos indicam pouco reuso de código via herança (MEIRELLES, 2013). No exemplo, a classe ***HelloWorldPrinter*** possui DIT = 1, pois ela herda os métodos e atributos da classe ***Printer***. O restante das classes possui DIT = 0, pois não possui classes ancestrais.

- **CBO** (*Coupling Between Objects* - Acoplamento entre objetos). Calcula as conexões de entrada e de saída de uma classe (CHIDAMBER; KEMERER, 1994), isto é, para uma determinada classe **C**, são contabilizadas classes que utilizam algum método ou variável de **C**, como também todas as classes que **C** referencia. No exemplo, a classe ***HelloWorld*** possui CBO = 1, pois ela faz referência a classe ***HelloWorldPrinter***. Já a classe ***HelloWorldPrinter*** possui CBO = 2, pois ela está acoplada a classe ***Printer*** e também é referenciada pela classe ***HelloWorld***.

- **LCOM4** (*Lack of Cohesion in Methods* - Ausência de coesão em métodos). Foi originalmente proposta por (CHIDAMBER; KEMERER, 1994) como LCOM, porém, após o recebimento de críticas e revisões, foi definida uma nova versão conhecida como LCOM4 (HITZ; MONTAZERI., 1995). Para calcular LCOM4 de um módulo, é necessário construir um gráfico não-orientado em que os nós são os métodos e atributos de uma classe. Para cada método, deve haver uma aresta entre ele e um outro método ou variável que ele usa. O valor da LCOM4 é o número de componentes fracamente conectados nesse gráfico. Coesão entre os métodos de uma classe é uma propriedade desejável, portanto o valor ideal dessa métrica é 1. Se uma classe tem

diferentes conjuntos de métodos não relacionados entre si, é um indício de que a classe deveria ser quebrada em classes menores e mais coesas (MEIRELLES, 2013). No exemplo, a classe ***Printer*** tem LCOM4 = 1, pois todos os métodos acessam a variável ***message***. Já ***HelloWorldPrinter*** tem LCOM4 = 2, pois o construtor e o método ***doNothing()*** não estão relacionados.

3 Metodologia

Neste estudo, argumentamos que há uma falta de compreensão das métricas do código-fonte especificamente em torno do ecossistema Android. O principal objetivo deste estudo é avaliar o ecossistema Android, ou seja, seus componentes principais e seus aplicativos clientes, para fins de avaliação em relação à qualidade interna, do ponto de vista das métricas do código-fonte e no contexto de diferentes versões do Android. Este Capítulo apresenta como esta pesquisa foi projetada, descrevendo os passos que adotamos para este estudo.

3.1 Questões de Pesquisa

Com base em nosso objetivo principal, propomos as seguintes questões de pesquisa.

- **QP1** – As distribuições de valores métricos do ecossistema Android diferem dos sistemas de software tradicionais?

  Os limites métricos geralmente são usados para inferir a qualidade interna, ou seja, os componentes com valores métricos que excedem um determinado limite são considerados mal projetados ou merecedores de esforços de manutenção. Em seguida, é considerado aceitável que alguns componentes possam exceder os valores da métrica. Para responder à QP1, calculamos as distribuições dos valores da métrica do código-fonte nos componentes principais do Android, bem como em seus aplicativos clientes e comparamos com a distribuição dos valores métricos de softwares tradicionais.

- **QP2** – Para cada métrica, é possível definir limiares de referência que podem ser usados para classificar seus valores, no contexto do ecossistema Android?

  Em um contexto específico, como o ecossistema Android, as métricas orientadas a objeto podem se comportar de acordo com uma distribuição exponencial. Para responder à QP2 e obter valores limiares adequados para este ecossistema, realizamos análises empíricas usando duas abordagens: a primeira é baseada em valores de média e desvio padrão (LANZA; MARINESCU, 2006) e a segunda é baseada nos valores percentis (MEIRELLES, 2013).

- **QP3** – Os valores das métricas do Android API Framework podem ser usados como parâmetros para avaliar a qualidade interna de seus aplicativos clientes?

  De acordo com a distribuição de métricas que calculamos no Android API Framework, usamos um índice de qualidade proposto por (JUNIOR, 2015) que calcula

a similaridade entre essa distribuição e a de um aplicativo cliente. Para responder à QP3, avaliamos se podemos chegar a conclusões sobre como as métricas são percebidas nesse ecossistema.

3.2 Método de Pesquisa

Esta seção descreve as etapas que realizamos para responder às questões de pesquisa propostas.

3.2.1 Seleção das métricas de código-fonte

Primeiro, selecionamos um conjunto de métricas bem conhecidas que serão computadas em nosso estudo. Essas métricas são essencialmente projetadas para sistemas orientados a objetos, como os que analisamos, e incluem métricas de tamanho, coesão e acoplamento, e complexidade. Nós descrevemos anteriormente cada métrica na Seção 2.2.2.

3.2.2 Seleção dos projetos

Para permitir a análise de projetos específicos para o ecossistema Android, primeiro selecionamos as principais versões do Android API Framework até outubro de 2015. Em seguida, selecionamos os principais aplicativos clientes presentes em todas as plataformas Android em análise.

Selecionamos um total de nove versões do Android, do Android *Donut* (versão 1.6) ao Android *Marshmallow* (versão 6.0). Selecionamos a versão mais recente de cada lançamento principal, que geralmente inclui alterações nas funcionalidades da primeira versão e também melhorias de estabilidade e segurança. A Tabela 1 apresenta as versões do Android em análise.

Nome da Versão	**Número da Versão**	**Data de Lançamento**
Donut	1.6	Setembro de 2009
Eclair	2.1	Outubro de 2009
Froyo	2.2.3	Maio de 2010
Gingerbread	2.3.7	Dezembro de 2010
Ice Cream Sandwich	4.0.4	Outubro de 2011
Jelly Bean	4.3.1	Julho de 2012
KitKat	4.4.4	Outubro de 2013
Lollipop	5.1.1	November de 2014
Marshmallow	6.0.1	Outubro de 2015

Tabela 1 – Versões selecionadas do Android

A princípio, pensamos em selecionar 11 versões que incluiria o Android *Nugget* (versão 7.0) e o Android *Oreo* (versão 8.0) que foi a última versão lançada até outubro de 2017, porém, não foi possível efetuar a coleta para essas versões com a ferramenta *Analizo*, que apresentou problemas específicos com estas versões[1].

Para cada versão do Android, selecionamos um total de onze aplicativos cliente. Foram selecionados os aplicativos que foram mantidos em todas as versões selecionadas. A tabela 2 apresenta os aplicativos cliente.

Nome do Aplicativo
Browser
Calculator
Calendar
Camera
Contacts
Email
HTMLViewer
PackageInstaller
Settings
SoundRecorder
Stk

Tabela 2 – Aplicativos Selecionados

3.2.3 Coleta dos dados

O projeto Android Open Source (AOSP) fornece um repositório para desenvolvedores para atender aos requisitos de compatibilidade para diferentes versões do Android. Usamos a ferramenta *repo* fornecida por este projeto para recuperar o código-fonte de cada versão em análise, bem como uma versão de seus aplicativos clientes compatíveis para cada versão do Android.

Calculamos as métricas do código-fonte de 9 versões do Android e 11 aplicativos clientes selecionados para cada versão. Usamos a ferramenta Analizo (TERCEIRO et al., 2010), que foi explorada no Capítulo 2, para calcular as métricas do código fonte para um total de 68091 arquivos, incluindo Java, C e C ++. Dentre as funcionalidades existentes do *Analizo*, utilizamos o *analizo metrics-batch*. Essa funcionalidade nos permite uma análise simultânea de vários projetos, ou diferentes versões de um mesmo projeto.

No final da coleta de dados, a ferramenta gera um arquivo CSV por projeto (ou versão do Android). Cada linha deste arquivo CSV representa uma classe no projeto em análise, e cada coluna representa o valor de uma determinada métrica de código-fonte para essa classe. No total, foram coletadas 68091 classes, onde 59.42% são Java, 23.43%

[1] <https://github.com/analizo/analizo/issues/142>

são C e 17.15% são C++. Relatamos neste estudo os resultados das métricas aplicadas às classes Java, pois o Java é a linguagem predominante entre os projetos que selecionamos.

3.2.4 Processamento dos dados

Calculamos a distribuição dos valores das métricas dos principais componentes do Android de acordo com as versões selecionadas na segunda etapa. Nós usamos scripts R, para processar os arquivos CSV gerados pelo *Analizo*, realizar cálculos estatísticos e analizar essas distribuições. Um gráfico de distribuição foi gerado para cada métrica ao longo das diferentes versões do Android.

Estamos interessados em calcular a distribuição de valores métricos para responder à QP1. As características dessas distribuições serão discutidas no Capítulo 4. Além disso, também estamos interessados em analisar a evolução dos valores métricos de cada versão do Android. Para tal, geramos gráficos para cada uma das métricas.

3.3 Definição dos intervalos de referência

Analisamos a distribuição dos valores métricos entre todas as versões do Android API Framework selecionadas, a fim de verificar se o valor médio é uma medida significativa para definirmos intervalos de referência. Para tal, fazemos uma análise empírica comparando a abordagem proposta por (LANZA; MARINESCU, 2006), que utiliza a média e o desvio padrão para determinar os intervalos de referência, com a abordagem proposta por (MEIRELLES, 2013), que utiliza classificação de percentis. Com base nessa análise, podemos identificar valores representativos para as métricas. Em suma, nesta etapa, estamos interessados em analsiar a distribuição de valores métricos entre todas as versões do Android em análise, bem como verificar se a média é representativa para extrair valores limites dessas distribuições.

A abordagem proposta por (LANZA; MARINESCU, 2006) propõe regiões de referência baseadas em média e desvio padrão. Eles consideram então, que o valor médio é significativo. Neste estudo, usamos a seguinte nomenclatura:

- Lanza-Low = média - desvio padrão
- Lanza-Medium = média
- Lanza-High = média + desvio padrão

De acordo com essas regiões, o valor da métrica para uma determinada classe ou método é considerado um outlier se o valor for 50% maior que o limite máximo (Lanza-High).

No entanto, pesquisas anteriores já mostraram que as métricas orientadas a objetos seguem uma distribuição com cauda longa. Consequentemente, média e desvio padrão não são significativos para estabelecer valores de referência desta distribuição. Utilizamos então, uma abordagem alternativa, proposta por (MEIRELLES, 2013), para analisar esse tipo de distribuição.

Esta abordagem primeiramente separa os valores das métricas em vários percentis: 25%, 50%, 75%, 90%, 95%, e 99%.

Por exemplo, o percentil 90 com um valor de 500 para uma determinada métrica, mostra que 90% das classes deste projeto foram medidas pela métrica com um valor menor ou igual a 500. Da mesma forma, recuperamos a mediana nessa distribuição no percentil 50. A abordagem então calcula o valor que representa cada percentil. (MEIRELLES, 2013) mostrou que, para a maioria das métricas orientadas a objeto, o percentual de referência é 75, e essa questão é melhor detalhada na análise individual de cada métrica no Capítulo 4. O percentil analisado neste estudo é dividido em três regiões de referência:

- Muito Frequente = até o percentil 75
- Frequente = entre o percentil 75 e 90
- Pouco Frequente = entre o percentil 90 e 95

Usando a abordagem de (LANZA; MARINESCU, 2006), assumimos que cada métrica se comportará como uma distribuição normal, o que não é verdade como apresentado no Capítulo 4. Algumas distribuições métricas têm crescimento exponencial nos percentis finais, geralmente no percentil 75. Portanto, a análise percentil propõe limiares mais realistas com base no comportamento da distribuição.

3.3.1 Exemplos de Uso

A partir da distribuição dos valores métricos das versões do Android API Framework que calculamos na etapa anterior, usamos a abordagem proposta por (JUNIOR, 2015) para calcular a similaridade entre essas distribuições e as distribuições dos aplicativos clientes selecionados. Com base nesse cálculo de similaridade, podemos analisar se os valores das métricas que encontramos para o Android API Framework, podem ser usados como referência para desenvolvedores de aplicativos cliente. Apresentamos o calcúlo de similiaridade e os resultados deste exemplo de uso dos valores de referência para aplicações Android no Capítulo 5.

4 Definição de Intervalos de Referência

Neste Capítulo, apresentamos a análise da distribuição dos valores métricos extraídos das versões do Android listadas na Seção 3.2.2. Esta análise nos ajudará a responder à QP1, considerando que as distribuições dos valores métricos do ecossistema Android diferem dos sistemas de software tradicionais. Pesquisas anteriores mostraram que as métricas orientadas a objeto, como as que selecionamos, seguem uma distribuição com cauda longa, ou seja, alguns elementos podem ter valores de métrica extremamente altos de acordo com os valores de referência recomendados para essa métrica. Nesses casos, os valores da média e desvio padrão não são representativos.

Coletamos valores de métrica de 40460 classes Java de várias versões do Android, que fornecem dados suficientes para uma avaliação completa. De acordo com a distribuição dessas métricas, avaliaremos a abordagem mais adequada para derivar os valores limites para cada métrica. Consequentemente responderemos à QP2. Vale a pena notar que os valores limites não são a verdade absoluta para identificar elementos que estão se comportando de maneira anormal de acordo com alguma métrica. No entanto, eles ajudam os desenvolvedores a direcionar sua atenção para outliers que possam revelar um problema de design.

Nas próximas Seções, vamos nos concentrar na análise de distribuição de cada métrica. Para cada métrica, apresentamos uma tabela contendo os valores de classificação percentual, variando de 25% a 99%; esses valores são necessários para a abordagem de (MEIRELLES, 2013). Cada tabela também apresenta os valores de média e desvio padrão, que são necessários para a abordagem de (LANZA; MARINESCU, 2006). Em seguida, uma segunda tabela fornece os valores limites que obtemos usando os dois métodos, separadamente.

4.1 Métricas de Tamanho

4.1.1 AMLOC

A métrica AMLOC nos fornece a média do número de linhas de código por método em uma classe. A tabela 3 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Observe que o percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 4 apresenta a comparação entre os valores limites calculados a partir dos valores percentuais, ou seja, a abordagem de (MEIRELLES, 2013) e os valores limi-

tes calculados a partir dos valores de média e desvio padrão, ou seja, a abordagem de (LANZA; MARINESCU, 2006). Para a abordagem dos percentis, temos AMLOC com o valor 12 para "Muito Frequente", 25 para "Frequente", e 41 para "Pouco Frequente". Na abordagem de (LANZA; MARINESCU, 2006), temos AMLOC com o valor -13 para "Lanza-Low", 11 para "Lanza-Medium" e 36 para "Lanza-High". Note, que obtemos um valor negativo para "Lanza-Low", indicando que o desvio padrão está relativamente alto e consequentemente "Lanza-Low" não nos fornece informações relevantes para que se possa analisar AMLOC em um cenário real. Portanto, a abordagem dos percentis nos fornece valores mais plausíveis. Além disso, a tabela 4 apresenta classes outliers com AMLOC igual a 1034; essas classes foram econtradas nas versões 2.2.3, 2.3.7 e 4.0.4 do Android.

A Figura 2 apresenta a distribuição dos valores de AMLOC por versão do Android. A partir desta figura, podemos observar uma distribuição com cauda longa. As distribuições por versão não variam muito, pois os gráficos são sobrepostos para a maioria das versões em análise. Nós vemos apenas algumas variações após o percentil 95, onde estão os valores outliers.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	1.00	5.00	11.00	20.65	29.00	66.09	312.00	9.02	14.55
android-2.1_r2.1p2	1.00	5.00	11.00	21.00	29.79	68.02	360.00	9.22	15.91
android-2.2.3_r2.1	1.00	5.20	12.00	25.96	44.25	154.34	1034.00	12.88	32.56
android-2.3.7_r1	1.00	5.25	13.08	29.99	54.00	161.57	1034.00	14.00	34.04
android-4.0.4_r2.1	1.00	5.32	13.43	30.61	55.00	161.67	1034.00	14.07	33.49
android-4.3.1_r1	1.00	5.48	12.50	27.00	46.58	137.00	913.00	12.67	28.62
android-4.4.4_r2.0.1	0.00	4.00	10.33	22.14	32.36	94.04	232.00	9.16	17.06
android-5.1.1_r38	1.00	5.64	12.50	27.00	46.34	137.00	1004.00	12.74	28.94
android-6.0.1_r81	1.00	4.00	10.48	22.33	32.32	90.28	232.00	9.11	16.80

Tabela 3 – Percentis da métrica AMLOC nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
AMLOC	75	12	25	41	1034	-13	11	36

Tabela 4 – Valores Limites para AMLOC.

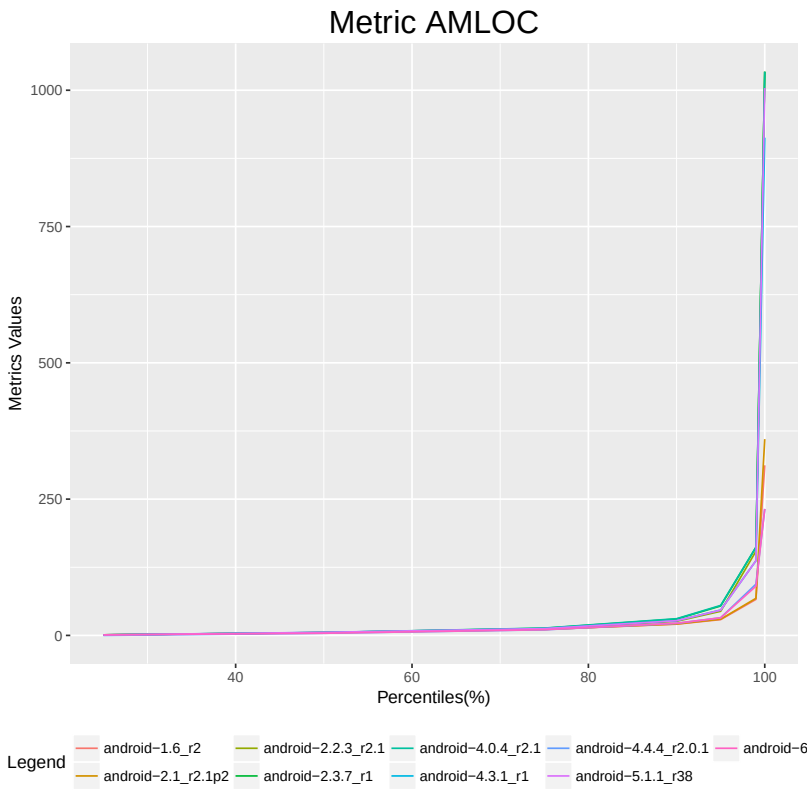


Figura 2 – Distribuição dos valores de AMLOC nas versões do Android.

4.1.2 ANPM

A métrica ANPM nos fornece a média do número de parâmetros por método em uma classe. A tabela 5 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 6 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos ANPM com o valor 2 para “Muito Frequente”, 3 para “Frequente” e 4 para “Menos Frequente”, já na abordagem de (LANZA; MARINESCU, 2006), temos ANPM com o valor 0 para “Lanza-Low”, 1 para “Lanza-Medium” e 2 para “Lanza-High”. Consideramos que a abordagem dos percentis nos forneceu limites mais razoáveis, visto que apenas um paramêtro como indica “Lanza-Medium”, não condiz com um cenário real de um método Java.

A Figura 3 apresenta a distribuição dos valores de ANPM por versão do Android. A maioria dos valores não varia muito acima de 3 parâmetros, e só observamos variância distinta após o percentil 95. A partir desta figura e da tabela 6, observamos as classes outliers nas versões 2.2.3, 2.3.7 e 4.0.4 do Android, com um número médio de 20 parâmetros

por método.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	0.00	0.82	1.40	2.25	3.00	4.82	12.00	0.98	1.10
android-2.1_r2.1p2	0.00	0.83	1.40	2.25	3.00	4.67	12.00	0.99	1.09
android-2.2.3_r2.1	0.00	0.88	1.50	2.71	3.50	6.84	20.00	1.12	1.41
android-2.3.7_r1	0.00	0.80	1.62	3.00	4.00	6.17	20.00	1.12	1.44
android-4.0.4_r2.1	0.00	0.86	1.64	3.00	4.00	7.00	20.00	1.15	1.47
android-4.3.1_r1	0.00	1.00	1.60	2.80	3.67	6.33	19.00	1.16	1.38
android-4.4.4_r2.0.1	0.00	0.80	1.50	2.21	3.00	5.00	10.00	0.98	1.10
android-5.1.1_r38	0.00	1.00	1.60	2.87	3.75	6.29	19.00	1.17	1.38
android-6.0.1_r81	0.00	0.80	1.50	2.00	3.00	4.85	10.00	0.97	1.07

Tabela 5 – Percentis da métrica ANPM nas versões do Android..

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
ANPM	75	2	3	4	20	0	1	2

Tabela 6 – Valores Limites para ANPM.

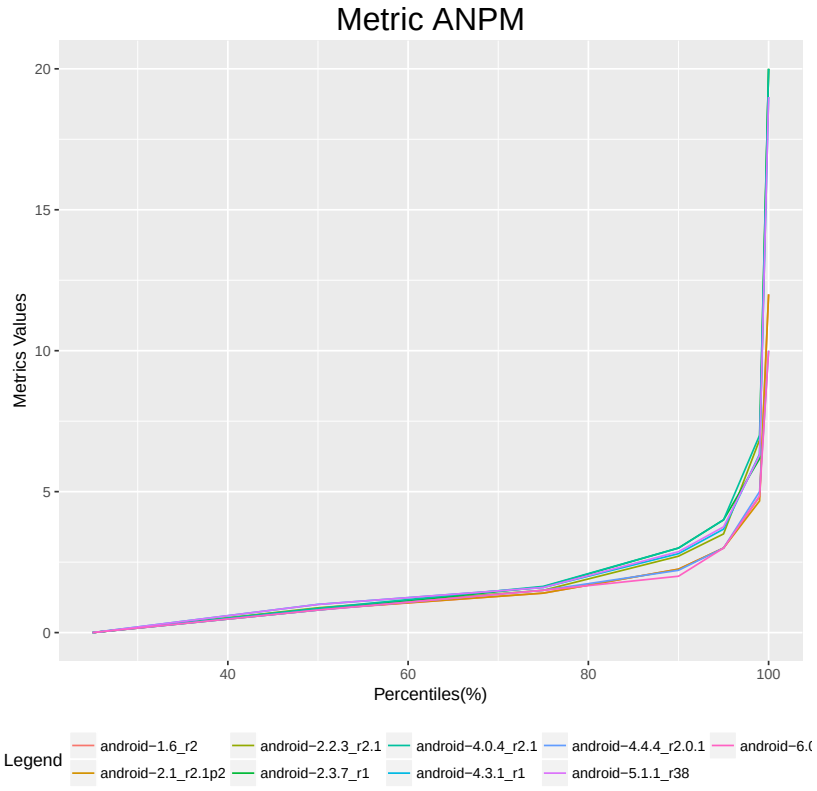


Figura 3 – Distribuição dos valores de ANPM nas versões do Android.

4.1.3 NOA

A métrica NOA mede o número de atributos de uma classe. A tabela 7 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android.

Além disso, assim como AMLOC e ANPM o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 8 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos NOA com o valor 5 para "Muito Frequente", 11 para "Frequente" e 19 para "Menos Frequente", já na abordagem de (LANZA; MARINESCU, 2006), temos NOA com o valor -8 para "Lanza-Low", 5 para "Lanza-Medium" e 17 para "Lanza-High". Podemos observar que, obtemos um valor negativo para "Lanza-Low", indicando que o desvio padrão está relativamente alto e consequentemente essa abordagem não nos fornece informações relevantes para que se possa analisar NOA em um cenário real. Portanto, a abordagem dos percentis nos fornece valores mais razoáveis.

A Figura 4 apresenta a distribuição dos valores de NOA por versão do Android. Podemos observar o comportamento da distribuição com cauda longa e especificamente para NOA, os valores aumentam excepcionalmente mais próximos do percentil 99. Também podemos notar a partir desta figura e da tabela 8, uma classe outlier encontrada na versão 4.4.4, com 611 atributos.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	0.00	1.00	5.00	11.00	19.00	47.53	241.00	4.65	11.88
android-2.1_r2.1p2	0.00	2.00	5.00	11.00	20.00	50.00	241.00	4.78	12.11
android-2.2.3_r2.1	0.00	1.00	5.00	11.00	19.00	49.00	302.00	4.64	12.33
android-2.3.7_r1	0.00	2.00	5.00	11.00	19.00	48.00	302.00	4.75	11.84
android-4.0.4_r2.1	0.00	1.00	5.00	12.00	19.00	50.00	302.00	4.80	12.87
android-4.3.1_r1	0.00	1.00	5.00	12.00	20.00	50.00	305.00	4.93	13.22
android-4.4.4_r2.0.1	0.00	2.00	5.00	11.00	17.00	40.52	611.00	4.45	13.32
android-5.1.1_r38	0.00	2.00	5.00	12.00	21.00	53.00	305.00	5.11	13.48
android-6.0.1_r81	0.00	1.00	4.00	10.00	16.00	40.00	611.00	4.18	12.59

Tabela 7 – Percentis da métrica NOA nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
NOA	75	5	11	19	611	-8	5	17

Tabela 8 – Valores Limites para NOA.

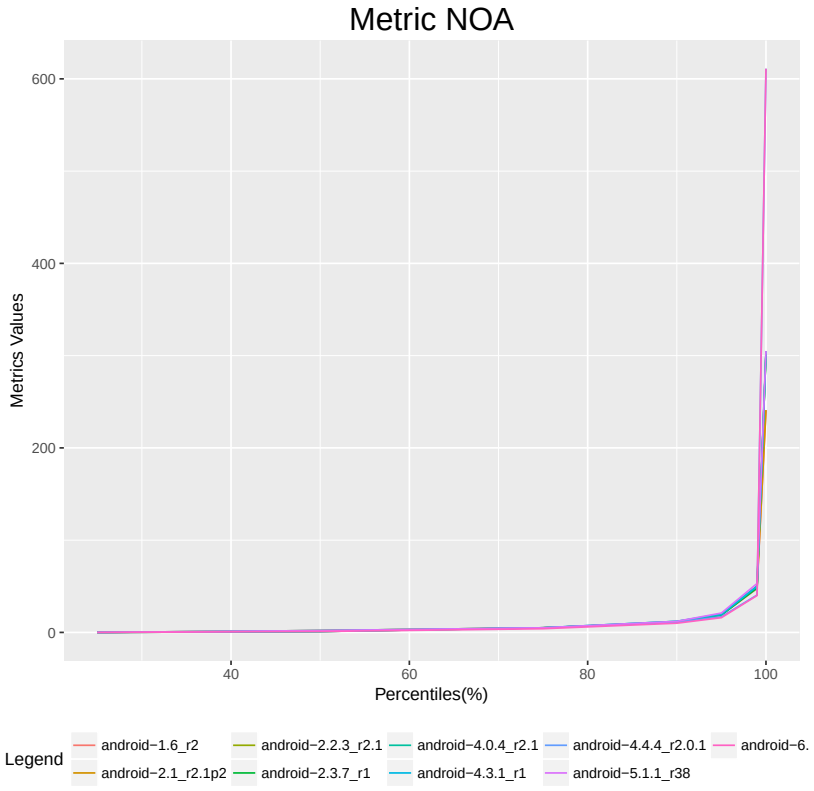


Figura 4 – Distribuição dos valores de NOA nas versões do Android.

4.1.4 NOM

A métrica NOM mede o número de métodos de uma classe. A tabela 9 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 10 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos NOM com o valor 7 para “Muito Frequente”, 17 para “Frequente” e 29 para “Menos Frequente“, já na abordagem de (LANZA; MARINESCU, 2006), temos NOM com o valor -9 para “Lanza-Low”, 7 para “Lanza-Medium” e 24 para “Lanza-High”. Assim como a métrica NOA, devido à natureza da distribuição, o desvio padrão é muito alto e a abordagem de (LANZA; MARINESCU, 2006) nos fornece valores limites que não coincidem com um cenário real. Sendo assim, consideramos que para NOM a abordagem de percentis nos fornece valores mais realistas.

A Figura 5 apresenta a distribuição dos valores de NOM por versão do Android. Assim como NOA, podemos notar o comportamento da distribuição com cauda longa, e o aumentos dos valores mais próximos do percentil 99. Também podemos notar a par-

tir desta figura e da tabela 10, uma classe outlier encontrada na versão 4.3.1, com 569 métodos.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	1.00	3.00	8.00	20.00	31.00	73.53	302.00	8.11	16.74
android-2.1_r2.1p2	1.00	3.00	8.00	19.00	31.00	73.35	310.00	8.05	16.67
android-2.2.3_r2.1	1.00	3.00	8.00	18.00	30.00	74.34	321.00	7.62	16.69
android-2.3.7_r1	1.00	2.00	7.00	17.00	27.00	72.00	329.00	6.99	15.99
android-4.0.4_r2.1	1.00	2.00	7.00	17.00	28.00	72.00	474.00	6.93	16.21
android-4.3.1_r1	1.00	3.00	7.00	18.00	28.05	72.00	569.00	7.39	16.62
android-4.4.4_r2.0.1	0.00	2.00	7.00	15.20	27.00	65.52	541.00	6.75	16.87
android-5.1.1_r38	1.00	3.00	7.00	18.00	29.00	72.00	587.00	7.45	16.50
android-6.0.1_r81	1.00	2.00	7.00	15.00	26.00	66.00	541.00	6.58	16.12

Tabela 9 – Percentis da métrica NOM nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
NOM	75	7	17	29	569	-9	7	24

Tabela 10 – Valores Limites para NOM

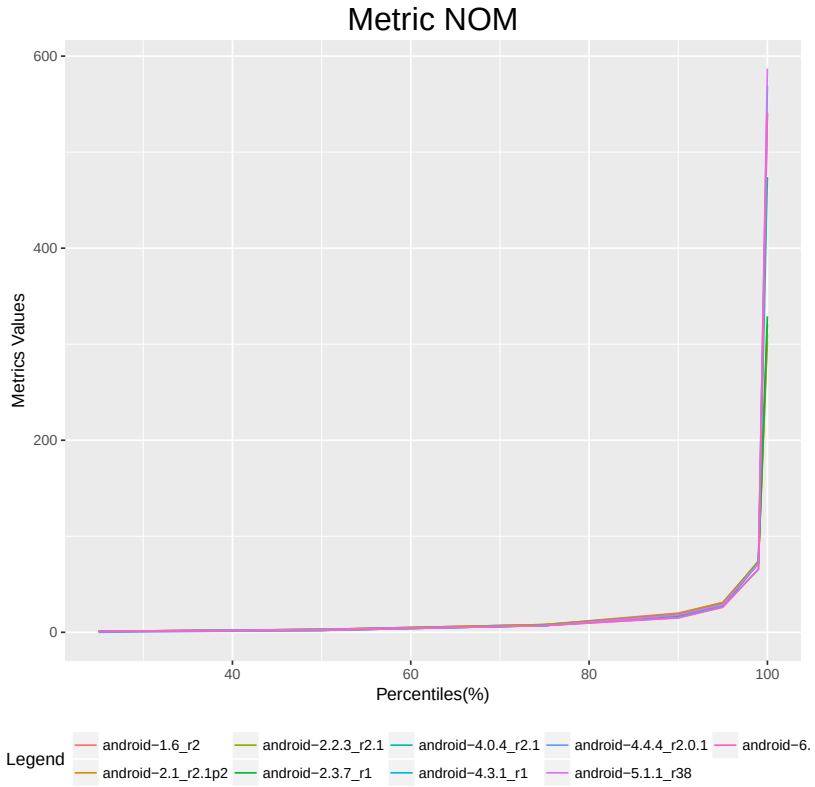


Figura 5 – Distribuição dos valores de NOM nas versões do Android.

4.2 Métricas de Complexidade

4.2.1 ACCM

A métrica ACCM nos fornece a média da complexidade ciclomática por método em uma classe. A tabela 11 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 12 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos ACCM com o valor 2 para "Muito Frequente", 4 para "Frequente" e 5 para "Menos Frequente", já na abordagem de (LANZA; MARINESCU, 2006), temos ACCM com o valor -3 para "Lanza-Low", 2 para "Lanza-Medium" e 7 para "Lanza-High". Os valores limites obtidos por ambas as abordagens são semelhantes, o que indica que o valor médio é uma medida relevante para as distribuições de ACCM. No entanto, semelhante às métricas de tamanho, a abordagem de média e desvio padrão calculou um valor negativo, o que não é compatível com a análise de um cenário real. Nesse caso, consideramos que a abordagem de percentis fornece valores mais significativos.

A Figura 6 apresenta a distribuição dos valores de ACCM por versão do Android. Para ACCM, as distribuições também não variam muito entre as versões do Android, exceto após o percentil 99. Também podemos notar a partir desta figura e da tabela 12, uma classe outlier encontrada na versão 5.1.1, com valor de ACCM igual a 681.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	1.00	1.00	1.94	3.36	5.00	12.53	143.00	1.82	3.99
android-2.1_r2.1p2	1.00	1.00	2.00	3.44	5.00	12.00	209.00	1.86	4.85
android-2.2.3_r2.1	1.00	1.00	2.00	3.67	5.67	14.63	286.00	1.97	5.63
android-2.3.7_r1	1.00	1.00	2.00	3.80	6.00	14.18	299.00	1.93	5.65
android-4.0.4_r2.1	1.00	1.00	2.00	3.87	6.00	18.00	636.00	2.01	7.23
android-4.3.1_r1	1.00	1.00	2.00	3.50	5.12	15.00	655.00	1.89	7.10
android-4.4.4_r2.0.1	0.00	1.00	1.74	3.33	5.00	11.43	120.40	1.57	3.09
android-5.1.1_r38	1.00	1.00	2.00	3.54	5.20	15.00	681.00	1.93	7.44
android-6.0.1_r81	1.00	1.00	1.70	3.21	5.00	11.33	120.40	1.57	2.94

Tabela 11 – Percentis da métrica ACCM nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
ACCM	75	2	4	5	681	-3	2	7

Tabela 12 – Valores Limites para ACCM.

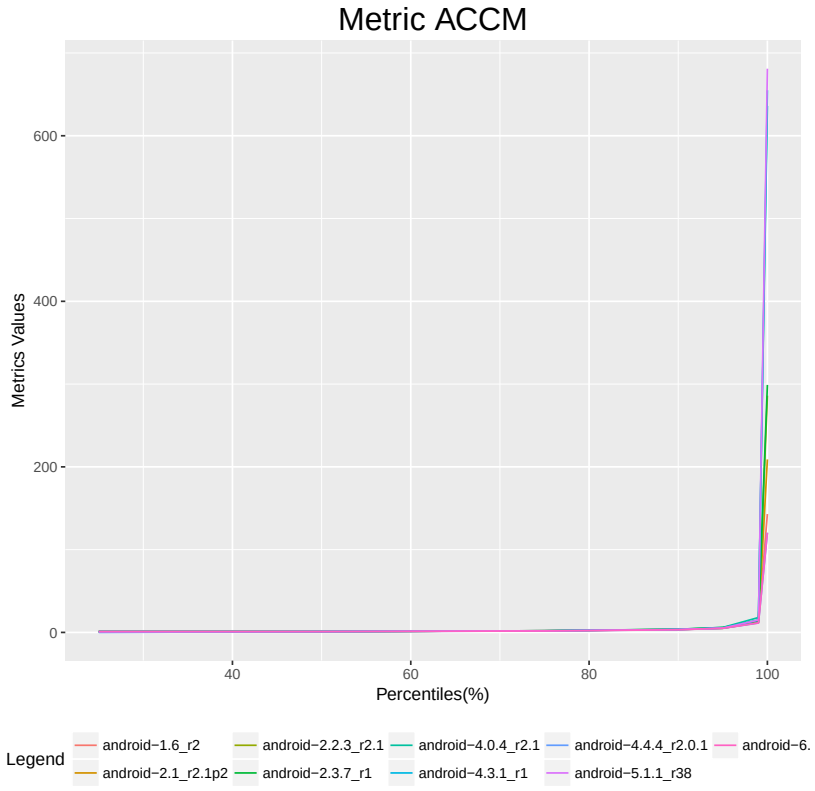


Figura 6 – Distribuição dos valores de ACCM nas versões do Android.

4.3 Métricas de Coesão e Acoplamento

4.3.1 DIT

A métrica DIT mede o número de superclasses de uma classe. A tabela 13 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 14 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos DIT com o valor 1 para “Muito Frequente”, 3 para “Frequente” e 4 para “Menos Frequente“, já na abordagem de (LANZA; MARINESCU, 2006), temos DIT com o valor 0 para “Lanza-Low”, 1 para “Lanza-Medium” e 2 para “Lanza-High”. Consideramos que nesta distribuição, o valor médio é significativo para obter valores limites. No entanto, a abordagem de percentil forneceu valores mais realistas, já que é esperado que uma classe estenda outra classe como um comportamento “Muito Frequente”.

A Figura 7 apresenta a distribuição dos valores de DIT por versão do Android. Note que a curva é mais suave do que as outras métricas que apresentamos até agora. Além

disso, há pouca variação entre diferentes versões, mesmo nas regiões outliers. Observamos uma classe outlier na versão 4.0.4 com DIT igual a 9.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	0.00	1.00	2.00	3.00	4.00	5.00	7.00	1.09	1.39
android-2.1_r2.1p2	0.00	1.00	1.00	3.00	4.00	5.00	7.00	1.07	1.37
android-2.2.3_r2.1	0.00	0.00	1.00	3.00	4.00	5.00	7.00	0.97	1.34
android-2.3.7_r1	0.00	0.00	1.00	3.00	4.00	5.00	7.00	0.88	1.31
android-4.0.4_r2.1	0.00	0.00	1.00	3.00	4.00	5.00	9.00	0.87	1.35
android-4.3.1_r1	0.00	0.00	1.00	3.00	4.00	5.00	7.00	0.97	1.34
android-4.4.4_r2.0.1	0.00	0.00	1.00	2.00	3.00	5.00	8.00	0.66	1.07
android-5.1.1_r38	0.00	0.00	1.00	3.00	4.00	5.00	7.00	1.00	1.35
android-6.0.1_r81	0.00	0.00	1.00	2.00	3.00	4.00	8.00	0.65	1.04

Tabela 13 – Percentis da métrica DIT nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
DIT	75	1	3	4	9	0	1	2

Tabela 14 – Valores Limites para DIT.

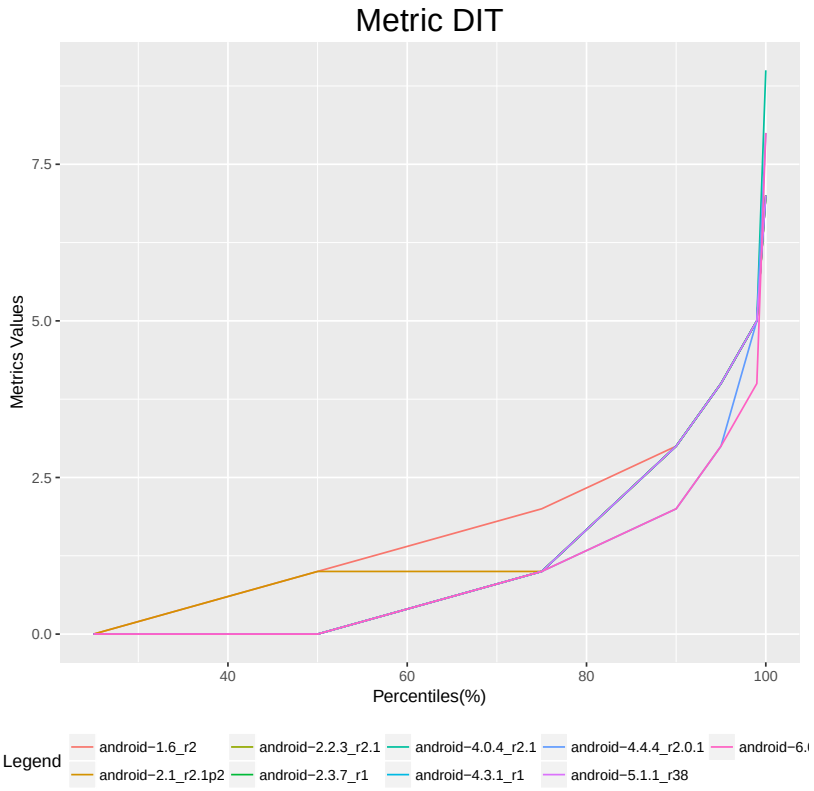


Figura 7 – Distribuição dos valores de DIT nas versões do Android.

4.3.2 CBO

A métrica CBO mede o acoplamento entre os objetos. A tabela 15 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além

disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 16 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos CBO com o valor 4 para "Muito Frequente", 8 para "Frequente" e 12 para "Menos Frequente". A abordagem de (LANZA; MARINESCU, 2006), apresenta valores similiares para CBO, com valor 3 para "Lanza-Medium" e 8 para "Lanza-High". Por outro lado, essa abordagem fornece um valor negativo para "Lanza-Low". Sendo assim a abordagem de percentis nos fornece valores mais realistas.

A Figura 8 apresenta a distribuição dos valores de CBO por versão do Android. Para CBO, as distribuições também não variam muito entre as versões do Android, excedendo os valores limites propostos perto do percentil 90. Observamos uma classe outlier na versão 5.1.1 com CBO igual a 121 classes.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	0.00	1.00	4.00	8.00	11.00	22.00	83.00	2.79	4.96
android-2.1_r2.1p2	0.00	1.00	4.00	8.00	11.00	22.00	87.00	2.78	4.93
android-2.2.3_r2.1	0.00	1.00	4.00	8.00	12.00	24.00	86.00	2.84	5.08
android-2.3.7_r1	0.00	1.00	4.00	8.00	12.00	25.00	88.00	2.93	5.32
android-4.0.4_r2.1	0.00	1.00	4.00	9.00	14.00	27.00	101.00	3.18	5.95
android-4.3.1_r1	0.00	1.00	4.00	9.00	14.00	28.00	109.00	3.35	6.28
android-4.4.4_r2.0.1	0.00	0.00	3.00	7.00	11.00	24.00	61.00	2.55	4.95
android-5.1.1_r38	0.00	1.00	4.00	9.00	14.00	29.00	121.00	3.42	6.28
android-6.0.1_r81	0.00	0.00	3.00	8.00	11.00	24.00	61.00	2.53	4.98

Tabela 15 – Percentis da métrica CBO nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
CBO	75	4	8	12	121	-2	3	8

Tabela 16 – Valores Limites para CBO.

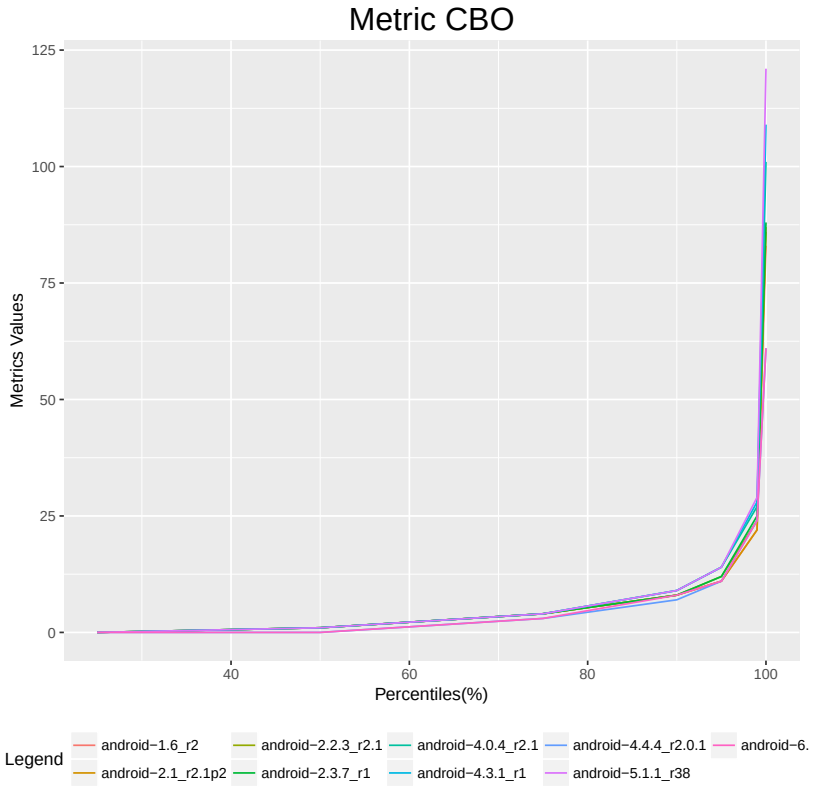


Figura 8 – Distribuição dos valores de CBO nas versões do Android.

4.3.3 LCOM4

A métrica LCOM4 mede a falta de coesão entre os métodos de uma classe. A tabela 17 mostra a distribuição dos valores desta métrica nos percentis nas diferentes versões do Android. Além disso, o valor do percentil 75 está mais próximo do valor médio, ou seja, as medidas de média e mediana não coincidem. Portanto, também usaremos o valor do percentil 75 como referência para a abordagem dos percentis.

A Tabela 18 apresenta a comparação entre os valores limites calculados a partir da abordagem dos percentis e da abordagem da média e desvio padrão. Para a abordagem dos percentis, temos LCOM4 com o valor 4 para “Muito Frequente”, 8 para “Frequente” e 13 para “Menos Frequente“. A abordagem de (LANZA; MARINESCU, 2006), apresenta valores similiares para LCOM4, com valor 4 para “Lanza-Medium” e 14 para “Lanza-High”. Por outro lado, essa abordagem fornece um valor negativo para “Lanza-Low”. Sendo assim a abordagem de percentis nos fornece valores mais realistas.

A Figura 9 apresenta a distribuição dos valores de LCOM4 por versão do Android. Para LCOM4, as distribuições também não variam muito entre as versões do Android, excedendo os valores limites propostos perto do percentil 95. Observamos uma classe outlier na versão 4.4.4 com LCOM4 igual a 541.

Projects	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
android-1.6_r2	1.00	2.00	4.00	9.00	15.00	37.00	267.00	4.12	9.63
android-2.1_r2.1p2	1.00	2.00	4.00	9.00	14.00	36.35	267.00	4.08	9.56
android-2.2.3_r2.1	1.00	2.00	4.00	8.00	14.00	36.00	279.00	3.88	9.61
android-2.3.7_r1	1.00	2.00	3.00	7.00	12.00	35.39	279.00	3.60	9.47
android-4.0.4_r2.1	1.00	1.00	3.00	7.00	13.00	38.00	318.00	3.55	9.34
android-4.3.1_r1	1.00	2.00	4.00	7.00	13.00	37.00	350.00	3.69	9.32
android-4.4.4_r2.0.1	0.00	2.00	4.00	8.00	14.00	42.04	541.00	3.97	13.85
android-5.1.1_r38	1.00	2.00	4.00	7.00	13.00	36.00	318.00	3.68	8.88
android-6.0.1_r81	1.00	2.00	4.00	8.00	13.00	39.38	541.00	3.84	12.82

Tabela 17 – Percentis da métrica LCOM4 nas versões do Android.

Métrica	Percentil de Referência	Muito Frequente	Frequente	Pouco Frequente	Valor Outlier	Lanza-Low	Lanza-Medium	Lanza-High
LCOM4	75	4	8	13	541	-6	4	14

Tabela 18 – Valores Limites para LCOM4.

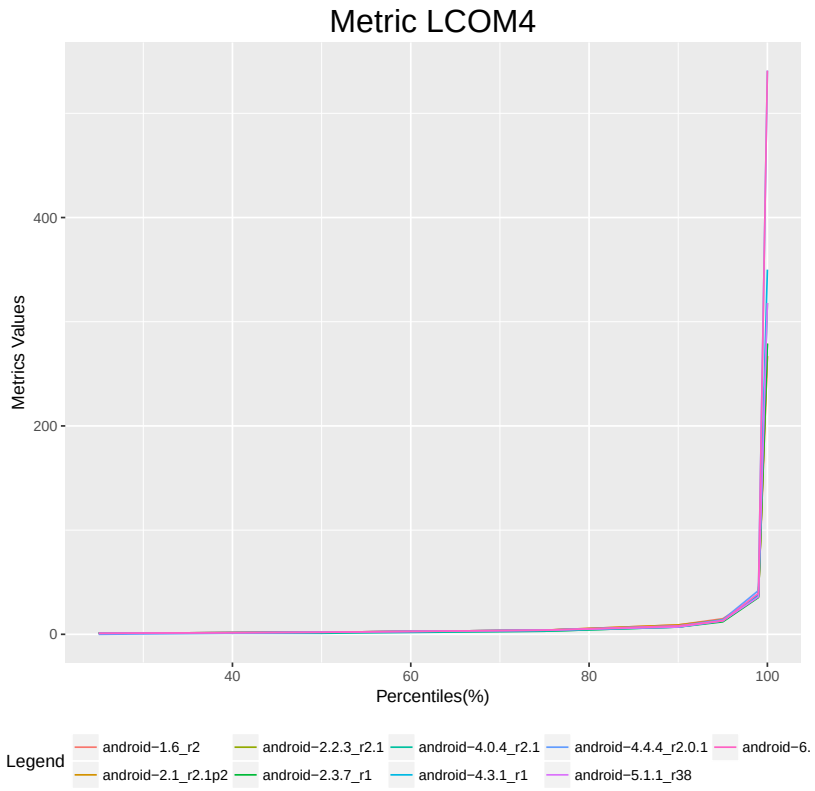


Figura 9 – Distribuição dos valores de LCOM4 nas versões do Android.

4.3.4 Valores de Referência com base na API do Android

A tabela 19 apresenta um resumo dos intervalos de referência baseados na API do sistema Android que foram discutidos durante todo o Capítulo.

Métrica	Intervalos	Rótulo
AMLOC	[0, 12]	Muito Frequente
	[12, 25]	Frequente
	[25, 41]	Pouco Frequente
	[41, ...]	Não Frequente
ANPM	[0, 2]	Muito Frequente
	[2, 3]	Frequente
	[3, 4]	Pouco Frequente
	[4, ...]	Não Frequente
NOA	[0, 5]	Muito Frequente
	[5, 11]	Frequente
	[11, 19]	Pouco Frequente
	[19, ...]	Não Frequente
NOM	[0, 7]	Muito Frequente
	[7, 17]	Frequente
	[17, 29]	Pouco Frequente
	[29, ...]	Não Frequente
ACCM	[0, 2]	Muito Frequente
	[2, 4]	Frequente
	[4, 5]	Pouco Frequente
	[5, ...]	Não Frequente
DIT	[0, 1]	Muito Frequente
	[1, 3]	Frequente
	[3, 4]	Pouco Frequente
	[4, ...]	Não Frequente
CBO	[0, 4]	Muito Frequente
	[4, 8]	Frequente
	[8, 12]	Pouco Frequente
	[12, ...]	Não Frequente
LCOM4	[0, 4]	Muito Frequente
	[4, 8]	Frequente
	[8, 13]	Pouco Frequente
	[13, ...]	Não Frequente

Tabela 19 – Intervalos definidos para o sistema Android.

5 Exemplos de Uso em Aplicativos Clientes

Neste capítulo apresentamos um exemplo de uso para analisar se os valores das métricas de código-fonte para o Android API Framework, podem ser usados como referência para avaliar a qualidade interna de seus aplicativos clientes e assim responder QP3. Valores para as métricas semelhantes aos do sistema, podem sem considerados bons, visto que há um certa dependência dos aplicativos em relação ao sistema. Logo, verificar o valor das métricas de forma relativa ao sistema, ajuda a verificar se esse valor está bom ou ruim, mesmo sem conhecer quais os valores bons ou ruins nesse contexto de desenvolvimento de aplicativos Android.

5.1 Cálculo de similiaridade

Para verificar o valor das métricas de forma relativa ao sistema, calculamos a distribuição dos valores métricos para os aplicativos clientes selecionados, e então usamos o índice de qualidade proposto por (JUNIOR, 2015) para calcular a similiaridade entre as distribuições do Android API Framework e as distribuições dos aplicativos. O cálculo de similiaridade da API então verifica as diferenças entre cada percentil do aplicativo e cada percentil da API, normalizados, e então unifica as diferenças entre as métricas em um só valor.

Obter um valor apenas de similiaridade para a API não é eficaz visto que as métricas tem diferentes grandezas. Sendo assim, (JUNIOR, 2015) propôs um trabalho de normalização dos valores das métricas para deixá-las em uma mesma variação. Uma normalização nada mais é do que a relativização da métrica em relação a um valor válido em sua escala, então, como não nos importamos com a grandeza do valor, pode-se utilizar até mesmo o valor muito frequente de cada métrica, representado pelo percentil 75, como referência. Assim, as distâncias serão calculadas em termos de quantas vezes o valor do percentil 75 de referência para cada métrica foi incrementado em um determinado valor de métrica. Por exemplo, um valor igual a 6 para ANPM, seria “normalizado” para 3 com esse cálculo, sendo interpretado então como 3 vezes o valor muito frequente da API, que é igual 2, como mostra a tabela 19. Da mesma forma, o valor 4 para DIT, seria “normalizado” para 4, sendo interpretado como 4 vezes o valor muito frequente da API, que é igual a 1. As grandezas estão mais equivalentes quando calculadas dessa forma.

(JUNIOR, 2015) propõe o cálculo de similiaridade como sendo uma medida positiva ou negativa, assim podemos perceber se o valor está menor ou maior que o do sistema, fazendo uma comparação entre ambos. É importante ressaltar que uma métrica com distância positiva mascara uma variação negativa de outra métrica com grandeza similar.

Entretanto o objetivo aqui é encontrar similaridade para o aplicativo como um todo, e não métricas isoladas, e geralmente um valor positivo ou negativo se sobressai sobre o oposto, pois boas arquiteturas tem valores bons em várias métricas, ficando com score negativo, e arquiteturas ruins tem valores ruins em algumas ou várias métricas, tendendo o valor para lado positivo.

Para o cálculo das distâncias, utilizou-se pesos em relação a importância das métricas. Assim, métricas mais importantes teriam mais impacto para informar que um valor está melhor ou pior em relação a API. As métricas de complexidade e métricas de acoplamento e coesão são evidenciadas com peso maior por terem mais impacto na qualidade do software, sendo assim, possuem peso 2. Já as métricas de tamanho possuem peso 1.

Os valores para comparação com o sistema são os valores indicados na Tabela 19. Para ACCM, por exemplo, os valores de comparação são 2, 4 e 5, para os percentis 75, 90 e 95, respectivamente.

É importante ressaltar que os percentis tem representatividade diferente. O intervalo que representa os valores muito frequentes, representado pelo percentil 75, possui 75% da amostra, enquanto os valores frequentes representado pelo percentil 90, possui 15%, e os valores pouco frequentes representado pelo percentil 95, possui 5%. Sendo assim, (JUNIOR, 2015) estabeleceu um peso relacionado a representiviade do percentil nas amostras, sendo então, os pesos 75,15 e 5 para os percentis 75, 90 e 95, respectivamente. O resultado final é somado e dividido pelo peso total multiplicado (95). Assim, uma variação de 1 em um percentil 75, que representa a maioria das amostras, tem mais impacto que uma variação de 1 no percentil 90, por exemplo.

O cálculo da distância de cada métrica é então dado por:

$$d = \frac{\sum_i^n \frac{(Mapi_i - Mapp_i)}{Mapi_{75}} \cdot W_i}{95} \tag{5.1}$$

Onde:

- i varia entre os percentis 75, 90 e 95.
- W_i é o peso do percentil i;
- $Mapi_i$ é o valor da métrica para a API no percentil i;
- $Mapp_i$ é o valor da métrica para o *app* no percentil i;
- $Mapp_{75}$ é o valor da métrica para o *app* no percentil 75, utilizado para "normalização";

E o valor da distância total do aplicativo em relação a API é dado por:

$$\frac{\sum_i^n d_i \cdot W_i}{\sum_i^n W_i} \tag{5.2}$$

Onde:

- i varia entre as métricas;
- d_i é a distância da métrica i calculada na Equação 5.1;
- W_i é o peso da métrica i;
- $\sum_i^n W_i$ é a soma dos pesos das métricas.

5.2 Análise dos aplicativos clientes do Android

HTMLViewer	-62
PackageInstaller	-44
Camera	-42
Stk	-40
Calculator	-24
Contacts	-24
Email	-12
Settings	-10
Browser	-10
SoundRecorder	-8
Calendar	-8

Tabela 20 – *Scores* de similaridade.

Calculamos a distribuição dos valores métricos para todos aplicativos selecionados, utilizando a abordagem dos percentis, proposta por (MEIRELLES, 2013), e então aplicamos o cálculo de similiaridade descrito na seção anterior. O resultado então foi multiplicado por 100 para levar os valores para uma grandeza maior.

A tabela 20 mostra o ranking dos aplicativos em relação a distância da API do Android. O aplicativo *HTMLViewer* com *score* de -62 é o mais distante da API e o que possui, no geral, os melhores valores para as métricas analisadas, já os aplicativos *Calendar* e *SoundRecorder* com *score* -8, são os que possuem valores métricos mais próximos ao da API.

Scores negativos se mostram melhores que o sistema, *scores* próximos de 0, são bem próximos à API e *scores* positivos podem ser preocupantes. Note, que todos os aplicativos apresentam *scores* com valores negativos, isso significa que todos apresentam melhores valores para as métricas analisadas, em relação a API, que é algo a se esperar, visto que os aplicativos analisados são softwares consideravelmente menos complexos do que a API, e tendem a possuir melhores valores para as métricas analisadas. De certa forma, o cálculo de similiaridade utilizado, pode servir como um indicador de certos problemas arquiteturais, quando o mesmo for positivo, e assim auxiliar novos desenvolvedores Android que não possuem conhecimento direto sobre métricas de código-fonte.

Nas próximas subseções selecionamos os aplicativos com melhor e pior *score* de acordo com a tabela 20 para fazermos uma análise individual da distribuição de seus valores métricos para as diferentes métricas selecionadas neste estudo.

5.2.1 Exemplo de uso 1 - Aplicativo HTMLViewer

O aplicativo *HTMLViewer* em sua última versão analisada(Android 6.0), possui apenas 3 classes, portanto, é um aplicativo bastante simples e possuir valores métricos melhores que o sistema, para a maioria das métricas, é algo esperado.

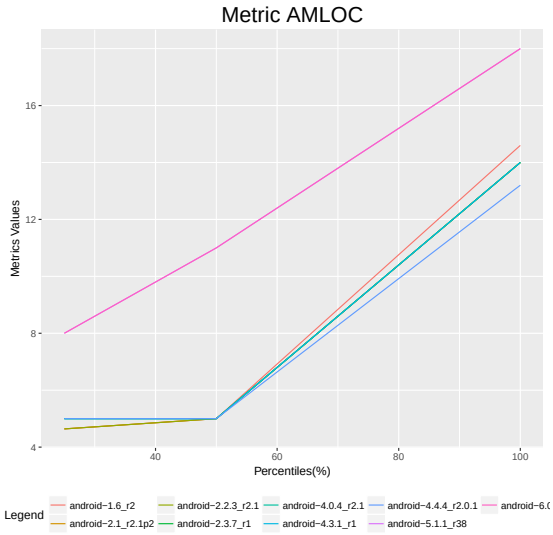
5.2.1.1 Métricas de tamanho

Para AMLOC, como podemos observar na figura 10a, até a versão 5.1.1, os valores métricos são bem parecidos, se distinguindo apenas nos últimos percentis. Na versão 6.0, os valores são bem maiores, mas ainda sim, continuam melhores que o sistema, onde os valores referência para esta métrica são 12, 25, e 41 para os percentis 75, 90 e 95, respectivamente.

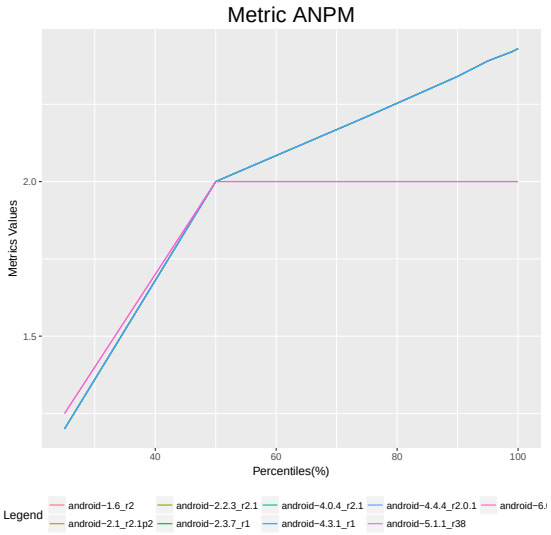
Para ANPM, podemos observar na figura 10b que os valores métricos se mantém semelhantes ao decorrer das versões analisadas com uma média de 2 parâmetros por método, que é bem proxímo ao intervalo de referência “Muito Frequente“ do sistema que varia de [0,2], conforme a tabela 19.

A figura 10c mostra que mesmo nos últimos percentis, o valor máximo para NOA é 3, apresentando melhores valores que o sistema, onde os valores referência para esta métrica são 5, 11, e 19 para os percentis 75, 90 e 95, respectivamente.

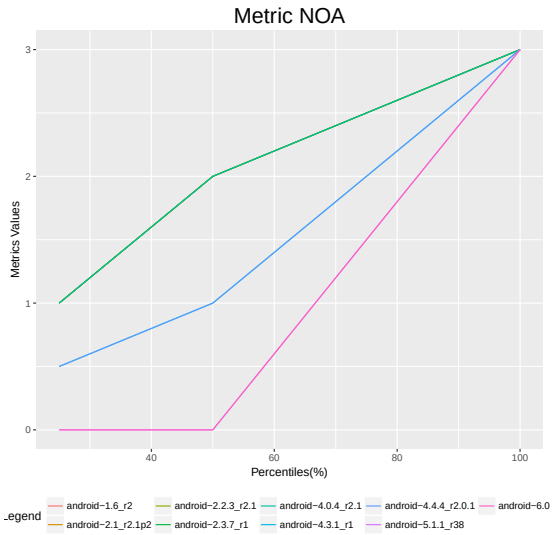
Para NOM, podemos observar na figura 10d, que até a versão 4.4.4, os valores métricos seguem semelhantes e nas últimas versões, é notável uma melhora. Mas ainda sim, estão bem proxímos ao intervalo de referência “Muito Frequente“ do sistema, que varia de [0,7], conforme a tabela 19.



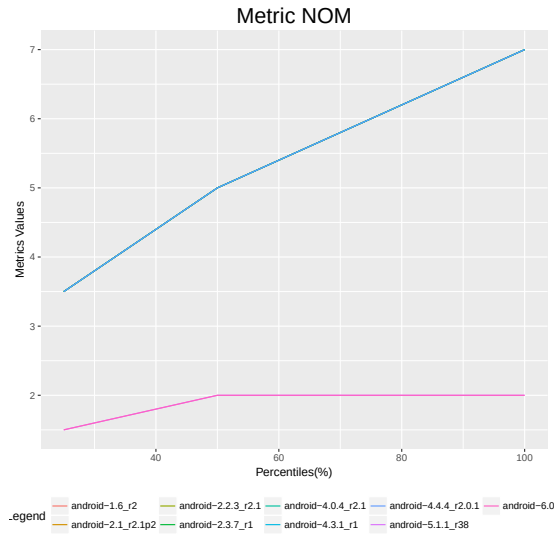
(a) Distribuição dos valores para AMLOC.



(b) Distribuição dos valores para ANPM.



(c) Distribuição dos valores para NOA.



(d) Distribuição dos valores para NOM.

Figura 10 – Distribuição dos valores das métricas de Tamanho no aplicativo HTMLViewer.

5.2.1.2 Métricas de Complexidade

Para ACCM, podemos observar na figura 11 que até a versão 4.4.4, os valores se mantiveram abaixo de 2, e apenas nas últimas versões, os valores mudam. Também podemos notar, que esses valores estão bem proxímos ao intervalo de referência “Muito Frequente“ do sistema, que varia de [0,2], conforme a tabela 19.

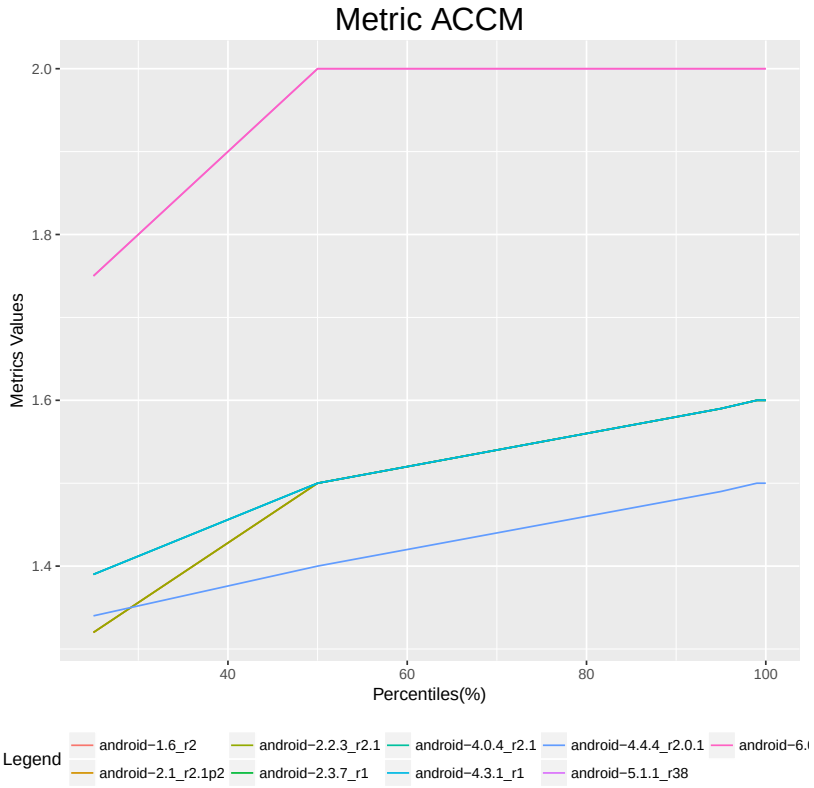


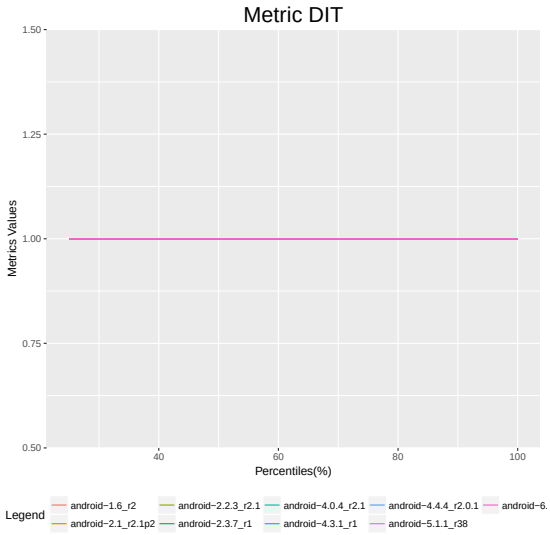
Figura 11 – Distribuição dos valores de ACCM no aplicativo HTMLViewer.

5.2.1.3 Métricas de Coesão e Acoplamento

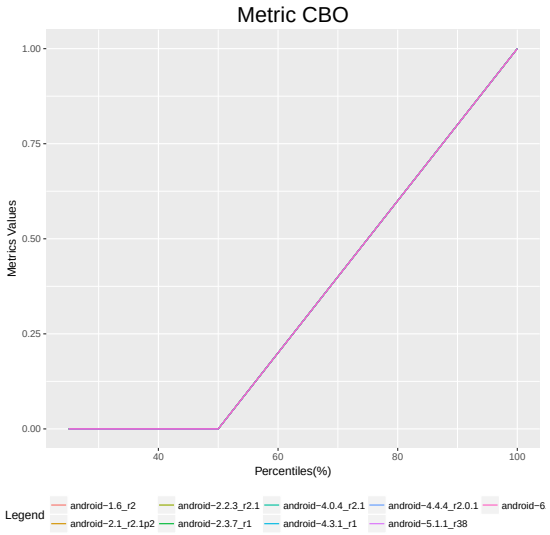
Na figura 12a, podemos notar que DIT se manteve constante em todas as versões com valor 1. Isto ocorre, devido ao fato do aplicativo *HTMLViewer* ser menos complexo, e não possuir grande alterações em suas classes ao decorrer da evolução de suas versões. Também é importante ressaltar, que para DIT o aplicativo está bem próximo ao intervalo de referência "Muito Frequente" do sistema, que varia de [0,1], conforme a tabela 19.

Para CBO, podemos obsevar na figura 12b que manteve-se os valores abaixo de 1, para todas as versões analisadas e que o aplicativo possui valores melhores que os do sistema, onde os valores referência para esta métrica são 4, 8, e 12 para os percentis 75, 90 e 95, respectivamente. Isto é esperado, pelo fato do sistema possuir um número bem maior de classes do que o aplicativo *HTMLViewer*, sendo assim é aceitável que haja maior acoplamento entre as classes no sistema.

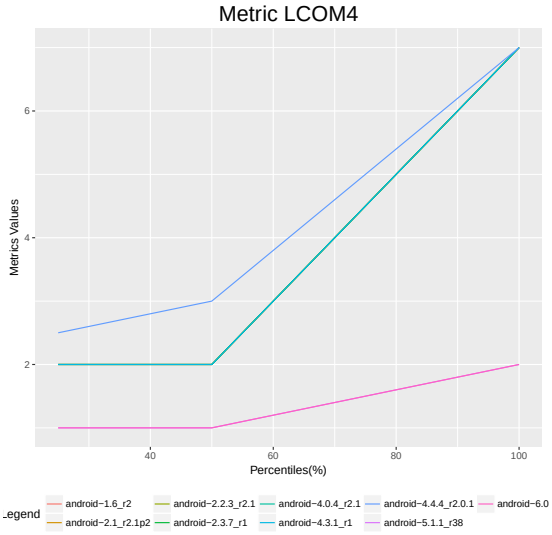
Na figura 12c, podemos observar que, até a versão 4.4.4, os valores de LCOM4 chegavam um pouco acima de 6, e estavam bem próximos ao sistema, onde os valores referência para esta métrica são 4, 8, e 13 para os percentis 75, 90 e 95, respectivamente. Já nas últimas versões, podemos notar uma melhora para esta métrica, que apresentou valores melhores que o sistema e consequentemente classes mais coesas.



(a) Distribuição dos valores para DIT.



(b) Distribuição dos valores para CBO.



(c) Distribuição dos valores para LCOM4.

Figura 12 – Distribuição dos valores das métricas de Coesão e Acoplamento no aplicativo HTMLViewer.

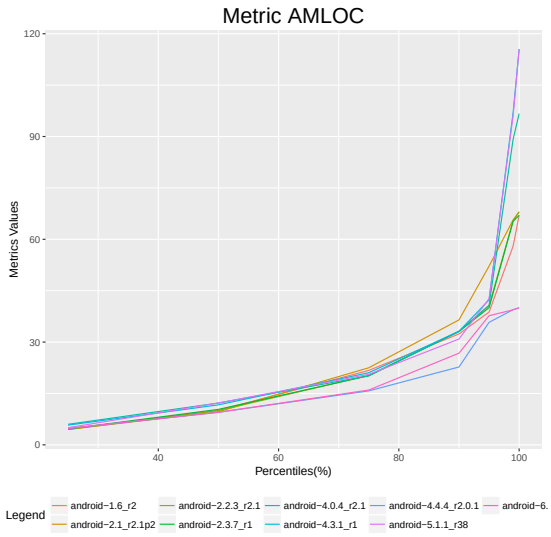
5.2.2 Exemplo de uso 2 - Aplicativo Calendar

O aplicativo *Calendar*, apesar de possuir o pior *score* conforme a tabela 20, está bem próximo ao sistema e possui bons valores para as métricas analisadas. Este aplicativo é bem mais complexo, se compararmos ao *HTMLViewer* e possui grandes mudanças em suas classes ao decorrer da evolução de suas versões.

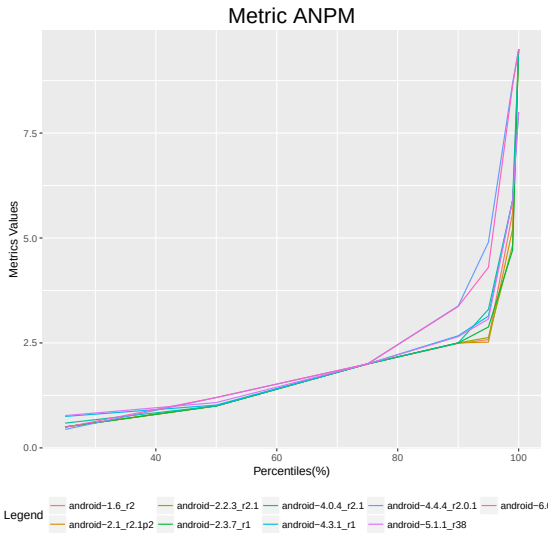
5.2.2.1 Métricas de tamanho

Ao analisarmos os gráficos na figura 13, podemos notar que no geral, para as métricas de tamanhho, os valores das métricas do Aplicativo *Calendar* crescem nos últimos

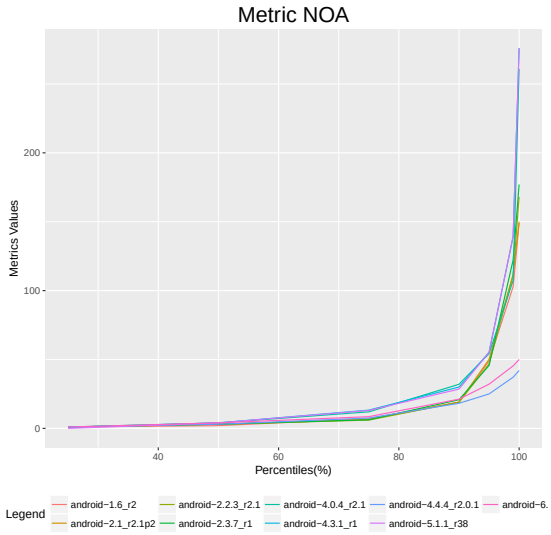
percentis e os valores estão bem próximos ao sistema, conforme mostra a tabela 19. Além disso, exceto para ANPM as distribuições melhoram bastante na última versão.



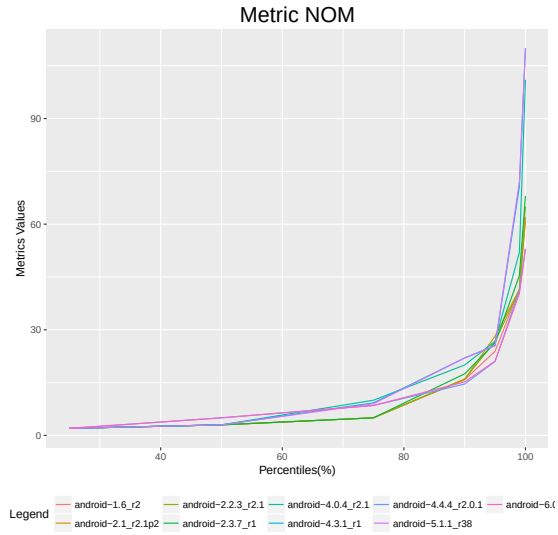
(a) Distribuição dos valores para AMLOC.



(b) Distribuição dos valores para ANPM.



(c) Distribuição dos valores para NOA.



(d) Distribuição dos valores para NOM.

Figura 13 – Distribuição dos valores das métricas de Tamanho no aplicativo Calendar.

5.2.2.2 Métricas de Complexidade

Na figura 14 podemos observar que assim como as métricas de tamanho, ACCM tem uma maior curva de crescimento nos últimos percentis. Os valores no geral, se aproximam do sistema até o percentil 90, que possui valores 0, 2 e 4 para os percentis 75, 90 e 95, respectivamente, sendo que nos últimos percentis se encontram as classes outliers. No entanto, podemos observar que, mesmo nos últimos percentis, há uma melhora nos valores nas últimas versões, logo, é notável que essas classes outliers receberam esforços de manutenção.

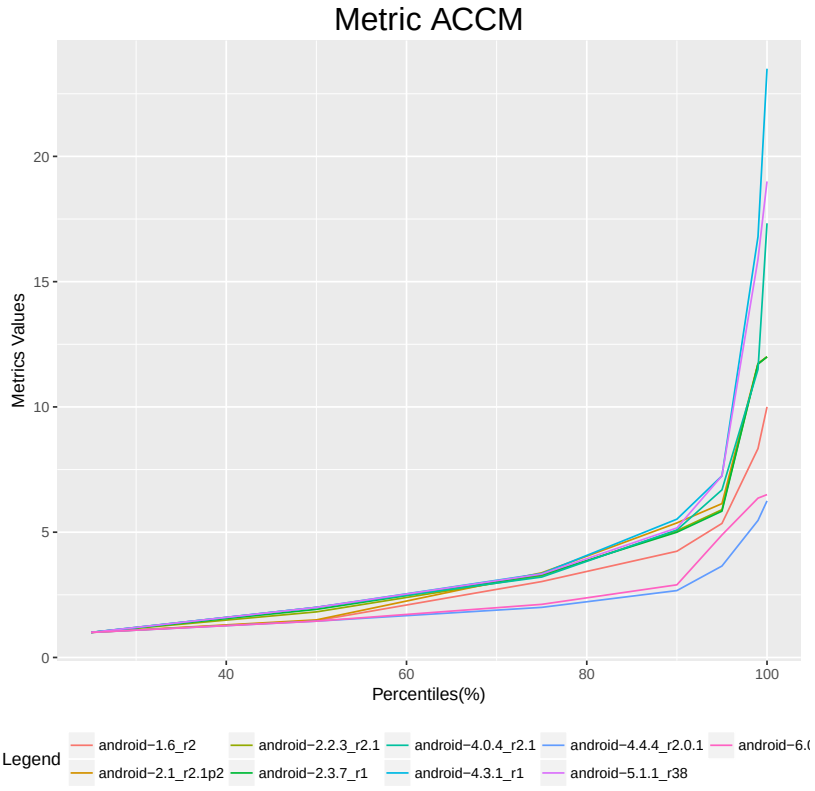


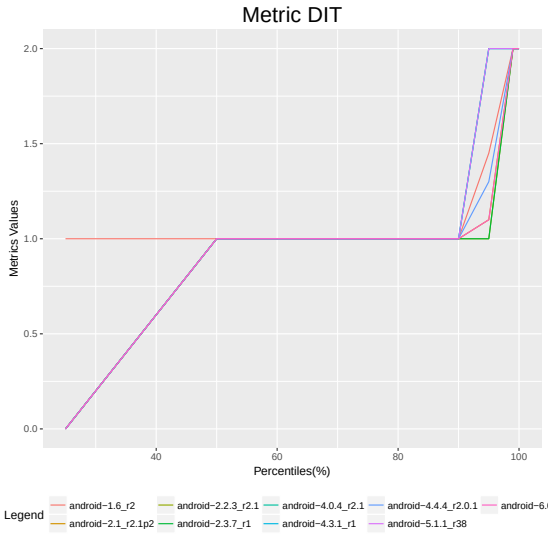
Figura 14 – Distribuição dos valores de ACCM no aplicativo Calendar.

5.2.2.3 Métricas de Coesão e Acoplamento

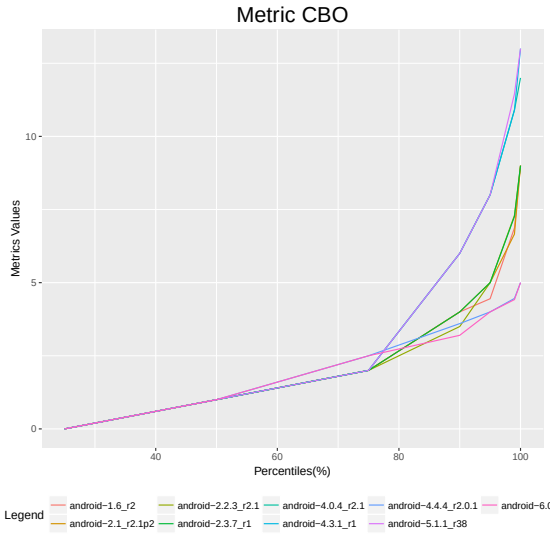
A figura 15a mostra que mesmo nos últimos percentis, o valor máximo para DIT no aplicativo *Calendar* é 2, que é um valor bem próximo ao sistema, onde os valores referência para esta métrica são 1, 3, e 4 para os percentis 75, 90 e 95, respectivamente.

Para CBO, podemos observar na figura 15b que mesmo nos últimos percentis, os valores métricos se aproximam do sistema, onde os valores referência para esta métrica são 4, 8, e 12 para os percentis 75, 90 e 95, respectivamente. No entanto podemos notar que há uma melhora para esta métrica na versão 6.0, ficando mais próximo do intervalo "Muito Frequente" do sistema, e consequentemente apresentando classes mais coesas.

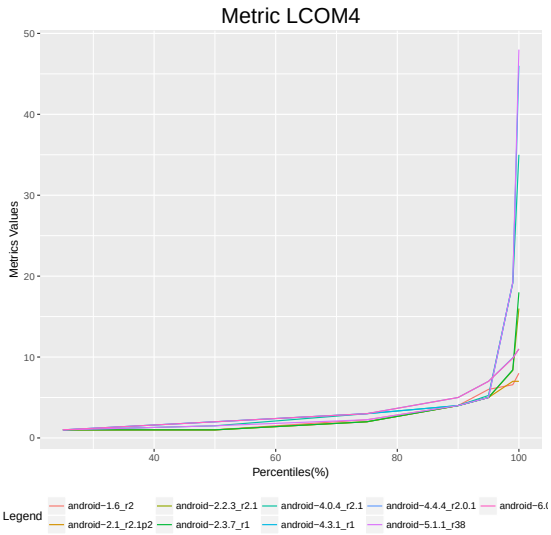
Para LCOM4, podemos observar na figura 15c que nos últimos percentis, os valores métricos são piores que o do sistema, onde os valores referência para esta métrica são 4, 8, e 13 para os percentis 75, 90 e 95, respectivamente. No entanto. podemos notar que há uma melhora para esta métrica na versão 6.0, ficando mais próxima do sistema.



(a) Distribuição dos valores para DIT.



(b) Distribuição dos valores para CBO.



(c) Distribuição dos valores para LCOM4.

Figura 15 – Distribuição dos valores das métricas de Coesão e Acoplamento no aplicativo Calendar.

6 Conclusão

O objetivo principal desse trabalho foi o monitoramento de métricas de código-fonte em torno do ecossistema Android, especificamente métricas para sistemas orientadas a objetos. Analisamos a distribuição dos valores métricos entre várias versões da API do Android, e encontramos valores de referência para cada uma das métricas selecionadas. Então, comparamos com a distribuição dos valores métricos de alguns aplicativos cliente do Android e verificamos a possibilidade de utilizar os resultados obtidos para auxiliar no desenvolvimento de novos aplicativos Android.

Comparando com alguns trabalhos relacionados, (SYER et al., 2011) mede o tamanho e o número de dependências em três aplicativos: WordPress, Google Authenticator e Facebook. Dependências incluem invocações para métodos, acesso a variáveis e herança, por exemplo. Os autores compararam os valores das métricas nesses sistemas com suas versões equivalentes de recursos na plataforma BlackBerry. Esses aplicativos Android têm de duas a seis vezes menos linhas de código do que sua versão BlackBerry. Esses aplicativos também têm menos código (46%) dependendo de bibliotecas de terceiros, significativamente mais dependências (74%) para os principais componentes do Android, e apenas um aplicativo tinha dependências para bibliotecas de terceiros, em comparação com aplicativos BlackBerry. Isto mostra, que o desenvolvimento de aplicativos para Android é altamente dependente da API subjacente, o que justifica nosso estudo sobre avaliação de qualidade com base na qualidade interna da API do Android.

(MINELLI; LANZA, 2013) tiveram conclusões semelhantes em um estudo com 20 aplicativos do F-droid, um repositório público de aplicativos Android de código aberto. Por exemplo, os autores descobriram que mais de (75%) das invocações de métodos nesses aplicativos representavam chamadas externas. Em comparação com o estudo de (SYER et al., 2011), as chamadas externas incluem as chamadas para a API do Android e as eventuais bibliotecas de terceiros. Além disso, o tamanho e a complexidade ciclomática cresceram em correlação com a adição de invocações de métodos externos. Os autores também concluíram que esse ecossistema difere dos sistemas de software tradicionais e, portanto, exigem abordagens específicas para mantê-lo e compreendê-lo. Nosso trabalho difere dos dois estudos porque (i) calculamos métricas de código-fonte mais complexas, (ii) analisamos vários aplicativos Android e várias versões de cada um deles , (iii) definimos intervalos de referência para as métricas a partir da API do Android.

Podemos observar que vários trabalhos reforçam a ideia de dependência dos aplicativos Android em relação a API Android. O que motivou a análise das métricas do sistema a serem comparadas com as métricas dos aplicativos e assim obter resultados que

possam ser úteis para desenvolvedores Android.

Em relação as questões de pesquisa levantadas no Capítulo 3, temos as seguintes observações:

- **QP1** – As distribuições de valores métricos do ecossistema Android diferem dos sistemas de software tradicionais?

  Ao compararmos os intervalos de referência definidos para o ecossitema Android, a partir da tabela 19 com outros intervalos de referência definidos a partir do estudo de (MEIRELLES, 2013), para projetos predominantemente em Java, como Eclipse e OPEN JDK8, podemos observar que para algumas métricas, como ACCM, os diferentes ecossistemas apresentam valores métricos bem semelhantes. Já em outras métricas, como CBO, o Eclipse e OPEN JDK8 apresentam melhores valores que o do ecossistema Android. Em DIT, temos o caso contrário, onde o Android apresenta melhores valores métricos que os outros ecossistemas. Logo, podemos concluir que cada ecossistema tem sua individualidade, e por mais que algumas métricas apresentem valores semelhantes, a maioria das métricas diferem entre os ecossistemas. Sendo assim, as distribuições dos valores métricos do ecossitema Android diferem de outros ecossistemas.

- **QP2** – Para cada métrica, é possível definir limiares de referência que podem ser usados para classificar seus valores?

  Na análise feita no capítulo 4, onde comparamos duas abordagens para analisar a distribuição dos valores métricos, podemos notar que, métricas orientadas a objetos seguem uma distribuição com crescimento exponencial nos percentis finais, logo, a média e o desvio padrão não foram significativos para estabelecer intervalos de referência. Entretanto, a abordagem dos percentis nos forneceu um cenário mais realista, e foi possível definir intervalos de referência com base nos percentis, 75, 90 e 95 respectivamente, como mostra a tabela 19. Vale a pena ressaltar, que os valores referência não são uma verdade absoluta para identificar problemas arquiteturais de acordo com alguma métrica, no entanto, são diretrizes que podem ajudar novos desenvolvedores.

- **QP3** – Os valores das métricas do Android API Framework podem ser usados como parâmetros para avaliar a qualidade interna de seus aplicativos clientes?

  Na análise feita no capítulo 5, onde calculamos um score de similiaridade entre a distribuição dos valores métricos do sistema e de cada aplicativo cliente, podemos notar que, no geral, os aplicativos clientes possuem valores métricos bem próximos ao sistema e podemos concluir que os intervalos de referência do sistema, também podem ser usados para avaliar seus aplicativos clientes, e assim auxiliar novos desenvolvedores Android.

6.1 Limitações

Uma das limitações econtradas nesse trabalho foi o fato de não conseguirmos analisar todas as versões do Android, devido uma limitação para coletar as últimas versões com a ferramenta que utilizamos. Apesar de demonstrarmos que, na evolução das versões do Android, os valores métricos possuem pouca variação, uma análise das distribuições até a versão mais recente nos traria melhores resultados.

Outra limitação encontrada seria a ferramenta que utilizamos para coleta, bem como as métricas que ela oferece. Devido ao escopo limitado deste trabalho, selecionamos algumas das métricas mais difundidas na literatura para sistemas orientados à objetos, porém, essas métricas podem não ser suficiente para avaliar a qualidade da arquitetura do sistema Android.

Também temos uma limitação em relação aos aplicativos selecionados, onde selecionamos apenas aplicativos padrões do Android. Existem diversos outros aplicativos que poderíamos analisar, para melhor consolidar os intervalos de referência definidos a partir da API do Android.

6.2 Trabalhos futuros

Primeiramente, seria interessante coletarmos as métricas para mais versões do Android, assim como coletarmos mais métricas, de modo que possamos melhorar os resultados obtidos neste trabalho.

A utilização do mapa de *Kohonem* para auxiliar na identificação de clusters para classificação das métricas, semelhantes ao estudo realizado por (LIMA; GUERRA; MEIRELLES, 2017) com métricas de anotações, seria interessante para definirmos novos intervalos de referência para as métricas e compararmos com as abordagens propostas neste trabalho.

Por fim, propomos uma análise dos valores métricos para mais aplicativos Android, focando no F-Droid que é um repositório mantido pela comunidade para o sistema Android, semelhante à loja do Google Play, que contém apenas aplicativos gratuitos de software livre.

Referências

ALVES, T. L.; YPMA, C.; VISSER, J. Deriving metric thresholds from benchmark data. In: *26th International Conference on Software Maintenance*. [S.l.: s.n.], 2010. p. 1–10. Citado na página 17.

BANSIYA, J.; DAVI., C. Automated metrics and object-oriented development: Using qmood++ for object-oriented metric. 1997. Citado na página 26.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object-oriented design. 1994. Citado na página 27.

COSTA, J. *Extração de Informações de Dependência entre Módulos de Programas C/C++*. [S.l.], 2009. Disponível em: <http://wiki.dcc.ufba.br/Aside/ProjetoFinalJoenioCosta>. Citado na página 23.

CRAVENS, A. A demographic and business model analysis of today's app developer. *GigaOM Pro*, 2012. Citado na página 19.

FENTON, N. E.; NEIL, M. Software Metrics: Roadmap. In: *22nd International Conference on on Software Engineering, Future of Software Engineering Track*. [S.l.: s.n.], 2000. p. 357–370. Citado na página 17.

FERREIRA, K. A. et al. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, Elsevier, v. 85, n. 2, p. 244–257, 2012. Citado na página 17.

HITZ, M.; MONTAZERI., B. Measuring coupling and cohesion in object-proceedings of international symposium on applied corporate computing. 1995. Citado na página 27.

HORA, A. C. *Assessing and Improving Rules to Support Software Evolution*. Tese (Doutorado) — University of Lille, 2014. Citado na página 17.

JUNIOR, M. R. P. Estudo de métricas de código fonte no sistema android e seus aplicativos. In: . [S.l.: s.n.], 2015. Citado 5 vezes nas páginas 18, 29, 33, 49 e 50.

KANNAVARA, R. Securing opensource code via static analysis. 2012. Citado 2 vezes nas páginas 21 e 22.

KANNAVARA, R. Challenges and opportunities with concolic testing. Portland, 2015. Citado na página 22.

LANZA, M.; MARINESCU, R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. [S.l.]: Springer Publishing Company, Incorporated, 2006. ISBN 978-3-540-39538-6. Citado 13 vezes nas páginas 18, 29, 32, 33, 35, 36, 37, 39, 40, 42, 43, 45 e 46.

LIMA, P.; GUERRA, E.; MEIRELLES, P. Definição de clusters para classificação do uso de anotações em código java. Instituto Nacional de Pesquisas Espaciais (INPE), 2017. Citado na página 61.

LORENZ, M.; KIDD, J. Object-oriented software metrics. 1994. Citado na página 26.

LORENZ, M.; KIDD, J. *Object-Oriented Software Metrics: A Practical Guide*. [S.l.]: Prentice Hall, 1994. Citado na página 17.

LUNGU, M. *Reverse Engineering Software Ecosystems*. Tese (Doutorado) — University of Lugano, 2009. Citado na página 17.

MCCABE, T. J. A complexity measure. 1976. Citado na página 26.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. 2013. Citado 9 vezes nas páginas 18, 24, 27, 29, 32, 33, 35, 51 e 60.

MINELLI, R.; LANZA, M. Software analytics for mobile applications–insights & lessons learned. In: *17th European Conference on Software Maintenance and Reengineering*. [S.l.: s.n.], 2013. p. 144–153. Citado na página 59.

MOJICA, I. J. et al. A large-scale empirical study on software reuse in mobile apps. *IEEE software*, v. 31, n. 2, p. 78–86, 2014. Citado na página 19.

OLIVEIRA, P. et al. Validating metric thresholds with developers: An early result. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. [S.l.: s.n.], 2015. p. 546–550. Citado na página 17.

SHATNAWI, R. Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process*, v. 27, n. 2, p. 95–113, 2015. Citado na página 17.

SYER, M. D. et al. Exploring the development of micro-apps: A case study on the BlackBerry and Android platforms. In: *11th International Working Conference on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2011. p. 55–64. Citado 2 vezes nas páginas 19 e 59.

TEIXEIRA, E.; ANTUNES, J.; NEVES, N. Avaliação de ferramentas de análise estática de código para detecção de vulnerabilidades. Faculdade de Ciências da Universidade de Lisboa, Departamento de Informática, 2007. Citado 2 vezes nas páginas 21 e 22.

TERCEIRO, A. et al. Analizo: an extensible multi-language source code analysis and visualization toolkit. 2010. Citado 2 vezes nas páginas 22 e 31.

TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. Departamento de Ciência da Computação UFMG, Belo Horizonte, Brasil, 2008. Citado na página 21.