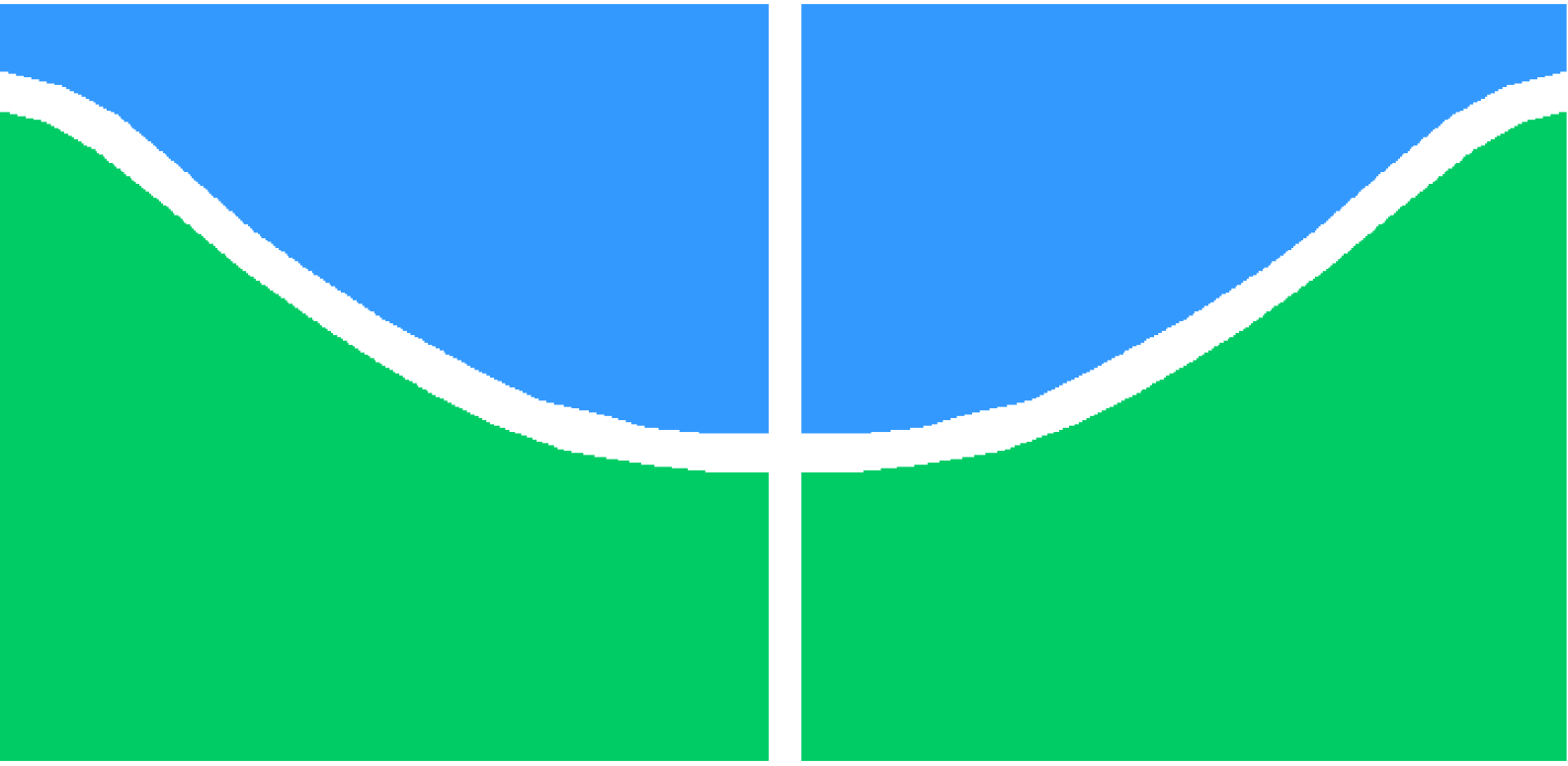

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

# Estudo de métricas de código fonte no sistema Android e seus aplicativos

Autores: Marcos Ronaldo Pereira Júnior
Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF
2015

Marcos Ronaldo Pereira Júnior

# Estudo de métricas de código fonte no sistema Android e seus aplicativos

Monografia submetida ao curso de graduação em Engenharia de Softwareda 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

2015

Marcos Ronaldo Pereira Júnior
Estudo de métricas de código fonte no sistema Android e seus aplicativos/ Marcos Ronaldo Pereira Júnior. – Brasília, DF, 2015-
80 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 , 2015.

1. Engenharia de Software. 2. Métricas de código. 3. Android. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Estudo de métricas de código fonte no sistema Android e seus aplicativos

CDU

Marcos Ronaldo Pereira Júnior

# Estudo de métricas de código fonte no sistema Android e seus aplicativos

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

Trabalho aprovado. Brasília, DF, Julho de 2015:

---

**Prof. Dr. Paulo Roberto Miranda Meirelles**
Orientador

---

**Prof. Fabrício Braz**
Convidado 1

---

**Prof. Renato Sampaio**
Convidado 2

Brasília, DF
2015

# Agradecimentos

Agradeço principalmente a Deus por me acompanhar em cada etapa da minha vida e à minha família e amigos por ajudarem a me tornar a pessoa que eu sou. Agradeço também aos professores que se empenham em ensinar, e às oportunidades que me foram oferecidas dentro da Universidade.

# Resumo

Nos últimos anos, a utilização de dispositivos móveis inteligentes se mostra cada vez mais presente no dia a dia das pessoas, ao ponto de substituir computadores na maioria das funções básicas. Com isso, vem surgindo projetos cada vez mais inovadores que aproveitam da presença constante desses dispositivos, modificando a forma que vemos e executamos várias tarefas cotidianas. Vem crescendo também a utilização de sistemas livres e de código aberto, a exemplo da plataforma Android que atualmente detém extensa fatia do mercado de dispositivos móveis. Ocupando papel fundamental na criação dessas novas aplicações, a engenharia de software traz conceitos de engenharia ao desenvolvimento e manutenção de produtos de software, tornando o processo de desenvolvimento mais organizado e eficiente, sempre com o objetivo de melhorar a qualidade do produto. O objetivo geral deste trabalho é o monitoramento de métricas estáticas de código fonte na API do sistema operacional Android, essencialmente métricas orientadas a objetos, e fazer um estudo da evolução de seus valores nas diferentes versões da API, estudar as semelhanças com aplicativos do sistema e então verificar a possibilidade de utilizar os dados obtidos para auxiliar no desenvolvimento de aplicativos.

**Palavras-chaves**: Engenharia de Software. Métricas de código. Android.

# Abstract

In recent years, the use of smart mobile devices have increased in everyday life, to the point of replacing computers for the most basic functions. Innovative projects that leverage the constant presence of these devices emerged by modifying the way we see and perform various daily tasks. It has also increased the use of free and open source systems, such as the Android platform which currently holds large share of the mobile device market. Occupying a key role in developing these new applications, software engineering brings engineering concepts to the development and maintenance of software products, making the development process more organized and efficient, aiming for improvement in product quality. The main objective of this study is to monitor source code metrics in the Android API, essentially object-oriented metrics, and study the evolution of its values in the different versions of the API, to verify the similarities with system applications, and then check the possibility of using the data obtained to assist in application development.

**Key-words**: Software Engineering. Source code metrics. Android.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ACC | *Afferent Connections per Class* |
| AOSP | *Android Open Source Project* |
| App | *Application* |
| API | *Application Programming Interface* |
| ACCM | *Average Cyclomatic Complexity per Method* |
| AMLOC | *Average Method Lines Of Code* |
| CSV | *Comma Separated Values* |
| CBO | *Coupling Between Objects* |
| COF | *Coupling Factor* |
| DIT | *Depth in Inheritance Tree* |
| ICS | *Ice Cream Sandwich* |
| LCOM4 | *Lack of Cohesion in Methods* |
| LOC | *Lines of Code* |
| NOC | *Number of Children* |
| NOM | *Number of Methods* |
| OO | Orientado a Objetos |
| RFC | *Response For a Class* |
| SDK | *Software Development Kit* |
| UnB | Universidade de Brasília |

# Sumário

# 1 Introdução

## 1.1 Contexto

Nos últimos anos, muitas das funções de um computador vem sendo gradativamente transferidas para dispositivos que cabem na palma da mão (MOORE, 2007). Atualmente podemos encontrar dispositivos móveis com poderosos processadores, alta capacidade de armazenamento de dados e conectividade sem fio avançada. A cada dia temos uma maior quantidade de informação concentrada de forma prática e segura, resultando em tarefas cotidianas automatizadas e controladas.

Para trabalhar nesses dispositivos, precisamos saber como eles funcionam e como podemos utilizá-los, para criar produtos de alta qualidade sem impactar no desempenho do processo de desenvolvimento. Monitorar o desenvolvimento desses produtos ao longo do seu desenvolvimento pode impactar bastante na qualidade de um produto final, e métricas de código fonte podem ser utilizadas em todas as etapas da criação de um software com esse objetivo.

Vários estudos já foram conduzidos sobre a qualidade de um produto de software. R.Basili, Briand e Melo (1995) apresenta um estudo prático demonstrando que métricas de código fonte podem ser bastante úteis como indicadores de qualidade. Xing, Guo e R.Lyu (2005) apresentam um método para prever qualidade de software em estágios iniciais de desenvolvimento baseado em métricas de complexidade de código. Vários outros estudos semelhantes já foram conduzidos nessa mesma direção. Meirelles (2013) afirma que a avaliação de qualidade de código fonte no início do desenvolvimento pode ser de grande valia para auxiliar equipes inexperientes durante as etapas seguintes do desenvolvimento de software.

Poucos trabalhos podem ser encontrados na literatura com aplicação de métricas de código fonte na API (*Application Programming Interface*) Android ou em seus aplicativos, e dado o contexto atual de disseminação de *smartphones* e *tablets*, o foco desse trabalho está em auxiliar na qualidade de produtos de software desenvolvidos para uma das maiores plataformas móveis da atualidade, o sistema operacional Android.

Neste trabalho foram levantadas informações acerca de métricas e sobre o sistema operacional Android, entretanto, para deixar o texto mais objetivo, essas informações foram retiradas e adicionadas em páginas específicas na Wikiversidade [1] sob as tags “Android” e “Métricas de código”

[1] https://pt.wikiversity.org/

## 1.2 Objetivos

O objetivo principal deste trabalho é o monitoramento de métricas de código fonte na API do sistema operacional Android, essencialmente métricas orientadas a objetos, e fazer um estudo da evolução de seus valores nas diferentes versões da API, estudar as semelhanças com aplicativos do sistema e então verificar a possibilidade de utilizar os dados obtidos para auxiliar no desenvolvimento de aplicativos. Essencialmente, será realizada uma análise exploratória de métricas de código fonte na API do sistema Android e de aplicativos. Será então apresentada uma proposta de cálculo de similaridade entre aplicativos e a API Android, utilizando como exemplo de uso desse cálculo um aplicativo parcial desenvolvido sem monitoramento de métricas.

Objetivos específicos:

- Estudo da evolução de métricas de código fonte em diversas versões da API;
- Estudo da correlação da API Android e de seus aplicativos;
- Definição e análise de intervalos de referência para valores de métricas;
- Proposta de utilização dos intervalos definidos para auxílio no desenvolvimento de aplicativos;

## 1.3 Estrutura do trabalho

Este documento está dividido em 4 capítulos. No Capítulo 2 são melhor explicados os objetivos do trabalho e os métodos para o alcance dos objetivos propostos. Em seguida, o Capítulo 3 apresenta discussões sobre análise dos dados coletados, validação dos mesmos, e uma proposta de aplicação dos valores definidos. O Capítulo 4 apresenta um aplicativo Android parcial com base em um estudo de caso específico, desenvolvido neste trabalho com o objetivo de alcançar uma boa arquitetura a partir de padrões de projeto e sem o auxílio de métricas, para base de comparação. Por fim, são apresentadas as considerações finais sobre a análise dos dados coletados e verificação de similaridade, bem como sugestões para continuidade deste estudo.

# 2 Metodologia

## 2.1 Objetivos

O objetivo geral deste trabalho é realizar uma análise de toda a API do sistema Android, a partir de uma análise estática de seu código fonte, e então avaliar a possibilidade de utilizar os resultados como referência para o desenvolvimento de aplicativos desenvolvidos para o Android, partindo da premissa que os aplicativos desenvolvidos utilizando a API do sistema são fortemente dependentes da mesma.

A partir da análise do código fonte da API Android, definir intervalos para valores da métrica que se adequam a arquitetura do sistema, assim como definir intervalos para os aplicativos do sistema e verificar o grau de aproximação que ambos encontram em seu design.

Será discutida a utilização de modelos de regressão para analisar o comportamento do sistema e utilizar como modelo preditor para valores ideais de métricas OO, utilizado como variável independente diferentes métricas de tamanho. Esses modelos de regressão tem como objetivo validar os intervalos de valores definidos manualmente neste trabalho.

Confirmando uma boa qualidade de código na API devido aos valores de métricas, propomos um fator de aproximação para aplicar a aplicativos em desenvolvimento para avaliar sua qualidade de acordo com a aproximação ao sistema, em termos de métricas estáticas de código. Essas métricas utilizadas refletem complexidade arquitetural e decisões de design, essencialmente métricas OO.

Esse fator de aproximação, que chamamos de *score* de similaridade, será centralizado em 0, onde valores positivos indicam valores das métricas maiores que os da API, e valores negativos indicam resultados menores que os da API. Quanto mais próximo de 0, mais próximo a API, e quanto menor o valor, melhores são os resultados.

Para exemplo de uso do score proposto, será apresentado o desenvolvimento do aplicativo e-lastic, desenvolvido sem o monitoramento de métricas de código fonte, para comparação de sua estrutura construída com base em padrões de projeto, com resultados esperados do monitoramento de métricas.

## 2.2 Questão de pesquisa e hipóteses

Baseando-se nas ideias apresentadas, foi levantada a seguinte questão de pesquisa (QP):

- QP - É possível monitorar métricas estáticas de código fonte de aplicativos Android de acordo com a análise de intervalos e aproximação às métricas do código do sistema Android?

Respondendo a essa pergunta podemos confirmar a efetividade de utilizar o próprio sistema como arquitetura referência em análise de aplicativos desenvolvidos para ele. Para alcançar os objetivos descritos e responder a questão de pesquisa, algumas hipóteses devem ser estudadas e avaliadas:

- H1 - É possível identificar padrões e tendências na evolução da arquitetura do sistema Android e nos aplicativos desenvolvidos para ele.
- H2 - O desenvolvimento de aplicativos Android pode ser guiado pelo resultado de uma análise evolutiva do código do próprio sistema.
- H3 - Uma grande aproximação ao sistema implica em uma boa qualidade de código.
- H4 - As decisões arquiteturais aplicadas no estudo de caso e-lastic têm resultados equivalentes à decisões arquiteturais baseadas em métricas.

A hipótese H1 será validada com a identificação de um padrão de comportamento de métricas no código fonte de diversas versões do sistema. A identificação de um padrão de permanência/aumento/diminuição ao longo da evolução do sistema nos auxilia a definir intervalos de métricas válidos para o sistema, e ter uma prévia de sua validade em versões futuras.

A hipótese H2 será validada com a identificação de semelhança entre o código fonte da API e aplicativos Android. Caso os intervalos de métricas definidos para a API no Capítulo 3 sejam obedecidos pelos aplicativos do sistema, demonstrando similaridades entre ambos, os intervalos de valores do sistema serão referência para comparação direta com aplicativos sem preocupação com a escala do projeto.

A hipótese H3 será validada pela observação dos valores de métricas encontrados na API Android em comparação com outros trabalhos com outros sistemas. Caso os valores coletados na API sejam considerados bons pela própria definição de cada métrica e por outros trabalhos, a API será avaliada com boa qualidade de código fonte. Caso tenha de fato boa qualidade, uma aproximação à API em termos de métricas implica na mesma conclusão. Como aplicação dos resultados dessa verificação, será proposto um score de similaridade com a API, que será aplicado aos aplicativos do sistema e ao aplicativo e-lastic para avaliação de qualidade.

Para a hipótese H4, serão avaliados os valores de métricas do aplicativo e-lastic, e o score de similaridade proposto neste trabalho será aplicado no mesmo. É importante

ressaltar que esse aplicativo foi desenvolvido sem o monitoramento de métricas. Valores teóricos bons, dentro dos intervalos aqui definidos para a API, indicam que os resultados do desenvolvimento com base em experiência de desenvolvedor e padrões de projeto são semelhantes aos resultados obtidos a partir do desenvolvimento com monitoramento de métricas de código fonte.

## 2.3 Trabalhos relacionados

Syer et al. (2011) realiza um estudo de dependência das APIs de desenvolvimento tanto da plataforma Android quanto da plataforma Blackberry, a fim de fazer uma comparação entre os sistemas. A partir disso, é verificado o quanto uma API influencia na quantidade de código desenvolvido, assim como é verificada a dependência de código de terceiros para o desenvolvimento de aplicativos. Em suma, o resultado comparativo demonstrou que aplicativos no sistema Android são significativamente mais dependentes da API do sistema devido a maior completude da API, reduzindo por consequência a quantidade de código de terceiros e código próprio dentro dos projetos. Embora possa tornar mais fácil o desenvolvimento, essa maior dependência em relação ao código do sistema torna o código de aplicativos desenvolvidos para Android mais difícil de ser portado para outras plataformas.

Minelli e Lanza (2013) apresenta o desenvolvimento de uma ferramenta de análise estática de código, que avalia não apenas métricas de código fonte, mas também a dependência de código de terceiros dentro do projeto de aplicativos Android. O objetivo desse trabalho não foi apenas analisar software em plataforma móvel, mas também diferenciar a abordagem quado analisando um software tradicional ou um software para dispositivo móvel, de forma a verificar a manutenibilidade desses sistemas. O estudo de Syer et al. (2011) também apresenta algumas discussões no quesito manutenibilidade em sistemas móveis. Assim como neste último, Minelli e Lanza (2013) demonstra uma alta dependência de aplicativos Android em relação ao sistema, apresentando em geral cerca de 2/3 das chamadas de método de aplicativos sendo para bibliotecas externas ao app, em sua maioria para a API Android ou métodos de bibliotecas padrão Java.

Linares-Vasquez (2014) reforça a ideia de dependência de aplicativos em relação a API Android, e fala sobre as grandes mudanças da API devido a sua rápida evolução nos últimos anos. Tenta então ajudar desenvolvedores com dicas para melhor se prepararem para mudanças na plataforma, assim como para mudanças em bibliotecas de terceiros, que podem ser bastante significativas em diversos aspectos, e consequentemente impactar negativamente no desenvolvimento de aplicativos, introduzindo mudanças bruscas e possivelmente bugs.

A dependência dos aplicativos Android em relação ao próprio sistema fica bem

clara em vários trabalhos publicados até a data de escrita deste trabalho, o que motiva bastante a coleta e análise de métricas no sistema para serem comparadas com métricas de aplicativos. Os resultados podem ser bastante úteis para novos desenvolvedores que não têm referências para basear seu desenvolvimento e poderiam guiar a evolução de seu sistema em comparação com a evolução do próprio Android.

Em se tratando de métricas, Gray e MacDonell (1997) apresenta várias abordagens para analisar métricas em software, mais a nível de projeto, e gerar modelos preditivos. Vários métodos de aprendizado de máquina são explanados e comparados em termos de modelagem relacionada a métricas em software.

Lanubile e Visaggio (1997) descreve uma comparação de várias técnicas para prever qualidade de código, classificando como alto risco, com grande probabilidade de conter erros, ou baixo risco, com probabilidade de conter poucos ou nenhum erro. Vários métodos são avaliados desde regressão até redes neurais. Embora os resultados apresentados no artigo não tenham sido muito promissores, a conclusão de que nenhum dos métodos utilizados se mostrou efetivo para separar entre componentes com ou sem erros pode ajudar a remover algumas tentativas desnecessárias em trabalhos relacionados.

Ainda tentando avaliar a probabilidade de conter erros, Gyimothy, Ferenc e Siket (2005) também utiliza técnicas de *machine learning*, utilizando como dados essencialmente métricas para sistemas orientados a objetos, de Chidamber e Kemerer. Esse estudo é conduzido em cima de software livre, utilizando como estudo de caso o Mozilla. Uma das contribuições que o artigo apresenta é relacionar classes e bugs reportados dentro do Mozilla. Além disso, verificaram que a métrica de acoplamento entre objetos (*coupling between objects* - CBO) foi a mais determinante em prever probabilidade de falha em classes, refletindo um pouco da qualidade do código escrito. Da mesma forma, a métrica linhas de código (*lines of code* - LOC) também se mostrou bastante útil, assim como a métrica de falta de coesão em métodos (*Lack of Coesion On Methods* - LCOM). Outras observações são que as métricas de profundidade de herança (*Depth In Tree* - DIT e *Number of children* - NOC) não se mostraram determinísticas, ou seja, seus valores não contribuíram na detecção de erros, ao mesmo tempo que número de classes também não teve impacto nos resultados. É importante notar que essas observações são válidas para o Mozilla, escrito em C/C++, o que não implica necessariamente que sejam válidas para todas as linguagens, embora isso seja bastante provável para outras linguagens orientadas a objetos.

Xing, Guo e R.Lyu (2005) apresenta um método utilizando *support vector machine (SVM)*, um modelo na área de *machine learning*, para prever qualidade de software em estágios iniciais de desenvolvimento baseado em métricas de complexidade de código, usando *LOC* e outras métricas como métricas de complexidade de Halstead e complexidade ciclomática.

R.Basili, Briand e Melo (1995) trabalha com métricas OO para verificação de qualidade de código. Foi observado que várias das métricas de Chidamber e Kemerer são bastante úteis para prever estados futuros já nas primeiras etapas do ciclo de vida. O estudo prático demonstrou que tais métricas podem ser bastante úteis como indicadores de qualidade.

Existem vários outros trabalhos publicados a respeito de prevenção de falhas com verificação da qualidade do código fonte, porém a principal diferenciação deste trabalho em relação a eles é o fato de a qualidade não ser medida em quantidade de bugs/erros ou modificações do código, mas sim em uma comparação com intervalos de valores ideais e como o resultado de uma comparação do código com a arquitetura da API do sistema. Entretanto, os dados utilizados neste trabalho serão essencialmente os mesmos da maioria desses trabalhos, resumindo-se basicamente em métricas OO, métricas de complexidade e de volume de código fonte.

## 2.4 Coleta de Dados

O código fonte para análise foi retirado diretamente do *Android Open Source Project*[1] (AOSP). Esse código é mantido essencialmente pela Google, com colaboração da comunidade de desenvolvedores. Essa versão é mantida e evoluída para funcionar como base para que as fabricantes de dispositivos possam manter sempre a última versão do sistema, com atualizações funcionais e de segurança.

A ferramenta *repo*[2] é utilizada para unificar os projetos internos dos componentes do sistema em seus repositórios, e um tutorial para configurar a ferramenta e fazer o download do projeto pode ser encontrado no site do AOSP.

Para a análise da API do sistema, foram escolhidas 14 versões do sistema, selecionando arbitrariamente a primeira e a última versão de cada grande *release*. Por exemplo, para o *Android Eclair*, foram pegas as versões 2.0 e 2.1, e para o Jelly Bean, as versões 4.1.1 e 4.3.1. Para o *Android Lollipop*, a última versão selecionada não será a última antes da próxima grande *release*, mas sim a última lançada até a data de início deste trabalho. Em uma análise ideal, todas as versões possíveis (ou pelo menos todas as tags oficiais) seriam levadas em consideração, mas o motivo dessa escolha de versões foi a impossibilidade de realizar uma análise do código fonte de todas as versões para este trabalho, por limitações de tempo e de recursos computacionais. Portanto, foram escolhidas as versões iniciais de cada grande *release* onde é alterado o *codename* da versão, que contém grandes mudanças e significativos avanços no sistema, assim como as versões finais de cada

[1] <http://source.android.com/>

[2] Ferramenta desenvolvida especificamente para o contexto do Android, utilizada em conjunto com o GIT

*codename*, que representam as versões mais estáveis das funcionalidades adicionadas nas versões com aquele *codename*.

Cada versão escolhida corresponde a uma *tag* no repositório oficial. Segue a listagem das *tags* escolhidas:

1. *Android Donut* 1.6 r1.2
2. *Android Donut* 1.6 r1.5
3. *Android Eclair* 2.0 r1
4. *Android Eclair* 2.1 r2.1p2
5. *Android Froyo* 2.2 r1
6. *Android Froyo* 2.2.3 r2
7. *Android Gingerbread* 2.3 r1
8. *Android Gingerbread* 2.3.7 r1
9. *Android Ice Cream Sandwich* 4.0.1 r1
10. *Android Ice Cream Sandwich* 4.0.4 r2.1
11. *Android Jelly Bean* 4.1.1 r1
12. *Android Jelly Bean* 4.3.1 r1
13. *Android Lollipop* 5.1.0 r1

Para cada *tag*, foi criado um diretório separado em um sistema Debian, onde foi executada o comando "repo -init" com um complemento específico para *setup* inicial daquela *tag*. Esse comando prepara o diretório e cria arquivos de controle da ferramenta para o repositório que está sendo iniciado. São baixados alguns arquivos, como por exemplo o arquivo em XML, chamado *manifest*, que contém os projetos ou repositórios que compõe o sistema, todos para a tag específica que foi iniciada. Antes de fazer o download do código, foram retirados do *manifest* da ferramenta todos os subprojetos que não estavam contidos dentro da pasta *frameworks*, que é o principal alvo da análise.

Dessa forma só os projetos de interesse são baixados quando o download for feito. Essa escolha se deu pelo fato de que grande parte do código Java da API do sistema utilizado no desenvolvimento de aplicativos se encontra neste local. Cyanogenmod, uma das maiores e mais conhecidas versões alternativas ao AOSP mas ainda baseada no mesmo, mantida por colaboradores voluntários em paralelo ao código da Google, cita em sua

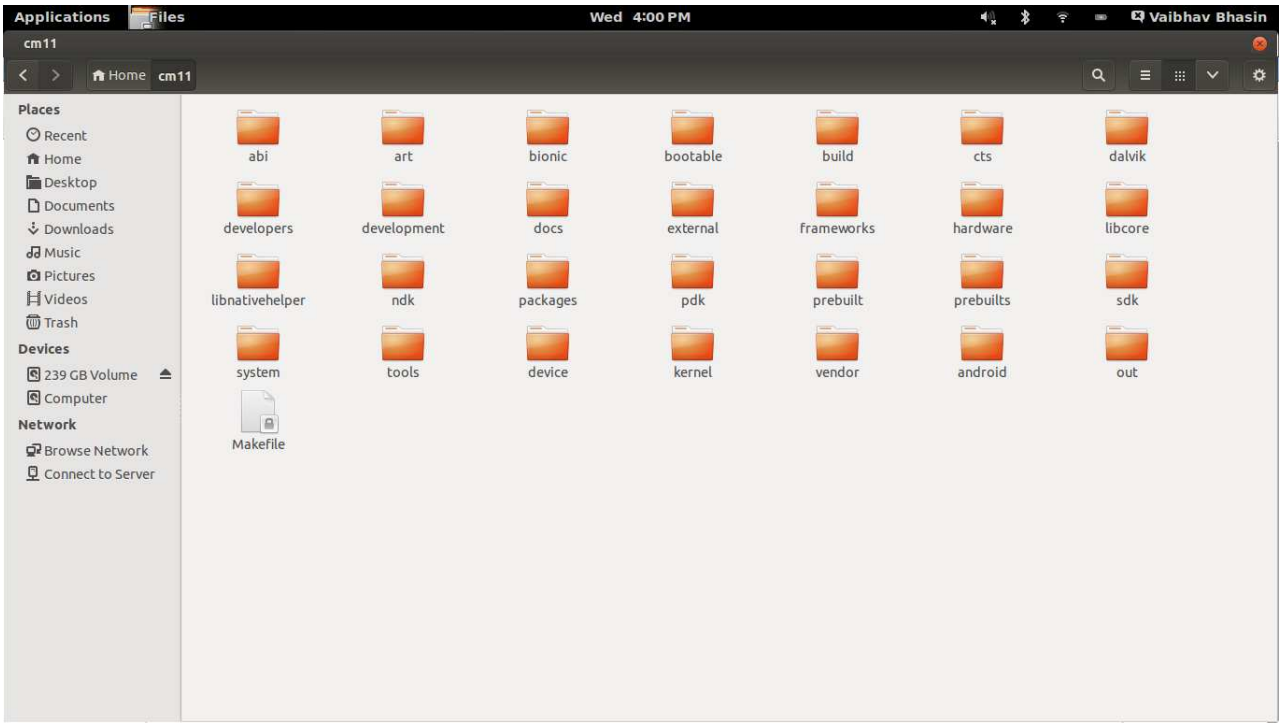


Figura 1 – Exemplo de estrutura de arquivos de diretório raiz do AOSP

página de ajuda a desenvolvedores[3] que o diretório *frameworks* é onde se encontram os "*hooks*" que programadores usam para construir seus aplicativos, ou seja, a API de construção de aplicativos em geral.

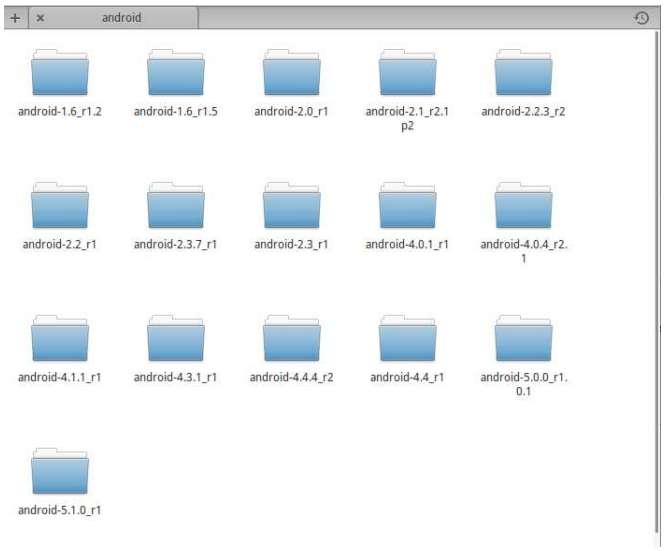


Figura 2 – Diretório onde foram armazenadas cada versão a ser analisada

O restante dos diretórios do AOSP contém desde adaptações de bibliotecas para o Android como o bionic, até código fonte para o *Android Run Time* (ART), que substituiu a dalvik nas últimas versões do sistema (especificamente desde as versões de *codename Lollipop*), e também códigos de baixo nível específicos para alguns dispositivos. Também

[3] <http://wiki.cyanogenmod.org/w/Doc:_the_cm_source>

existem diretórios para projetos externos ao Android, utilizados pelo mesmo, como o SQLite e outros projetos externos. O kernel utilizado no sistema também tem o seu diretório nessa hierarquia, assim como os aplicativos nativos. A estrutura completa do AOSP não será explorada neste trabalho, mas o conteúdo da pasta raiz pode ser visualizado na Figura 1. A estrutura de diretórios onde foram preparados os códigos para a análise pode ser vista na Figura 2.

Em seguida foi feito o download de cada *tag* em seu diretório, também utilizando a ferramenta *repo*, que realiza um *checkout* de cada subprojeto listado em seu *manifest* através do comando *sync*. O total de espaço em disco ocupado após o download de todas as *tags* foi cerca de 10 GB. É importante notar que esse espaço não corresponde apenas a arquivos de código fonte. Após o download e antes de realizar a análise, foi realizada uma filtragem de arquivos na pasta base da análise, removendo todos os arquivos que não correspondessem a código fonte C, C++ ou Java. Da mesma forma, foram removidos todos os diretórios vazios que restaram da deleção dos arquivos, deixando uma árvore de diretórios menor para ser percorrida pela ferramenta de análise, aumentando assim o desempenho da mesma. O espaço total ocupado pelo código após a filtragem foi de 1,7GB. Não realizar essa remoção não acarretaria em problemas para a análise, entretanto resultaria em um maior tempo necessário para o término da mesma.

Além de todas as versões do Android que foram analisadas, o código fonte dos aplicativos do sistema da última versão listada para esta análise (*Android Lollipop* 5.1.0 r1) também foi analisado, com o objetivo principal de comparação com o código do sistema. Aplicativos como de email, calculadora e câmera são analisados pela ferramenta Analizo assim como a API de desenvolvimento. São esperados valores relativos semelhantes aos da API do sistema Android.

No total foram coletados mais de 100 mil módulos/classes das linguagens C, C++ e Java. Java foi a linguagem mais predominante, com aproximadamente 85% das amostras, enquanto C++ ocupou pouco mais de 10% e C pouco mais de 3%. Embora seja relevante comentar as diferenciações entre as linguagens e seus paradigmas para algumas métricas, não há necessidade de separação entre os valores para linguagem C, procedural, e linguagens Java/C++, orientadas a objetos, uma vez que, dadas as proporções das mesmas apresentadas nos dados, não há relevância estatística para tal. Entretanto, em algumas métricas algumas observações teóricas podem ser ressaltadas, embora, mais uma vez, não haja implicação substancial no resultados gerais apresentados.

## 2.5 Análise de dados

Após o download de todas as versões escolhidas, foi utilizada a ferramenta Analizo para análise estática de código e coleta de métricas. Foi utilizada a funcionalidade de

*batch* do Analizo, de forma a coletar métricas de todas as *tags* de uma só vez.

O Analizo[4] é uma ferramenta livre e extensível para análise de código com suporte a várias linguagens, incluindo Java, predominante do código da API Android. Uma grande quantidade de métricas são coletadas pela ferramenta, embora apenas algumas sejam utilizadas para esta análise.

A saída da ferramenta é um arquivo CSV (*Comma-Separated Values* - valores separados por vírgula) para cada projeto ou versão a ser analisada, assim como um arquivo CSV que centraliza os valores de cada métrica em nível de projeto para cada um dos projetos/versões. Isso pode ser interessante se utilizado com o mesmo projeto em diferentes versões para verificar o avanço de algumas métricas juntamente com a evolução do sistema.

Trabalhos com análise de intervalos de métricas como as teses de Meirelles (2013) e Oliveira (2013), que utilizam a ferramenta Analizo, nos ajudam na escolha dessa ferramenta para comparação direta com as discussões nessas teses, uma vez que pode haver pequenas variações nos cálculos das métricas em diferentes ferramentas. Por exemplo, algumas ferramentas devolvem sempre um DIT (Profundidade na árvore de herança - *Depth In Tree*) mínimo de 1 para classes Java, que herdam de *Object*, enquanto o Analizo não contabiliza essa herança, como será discutido na análise da métrica DIT no Capítulo 3.

Neste trabalho, serão utilizadas métricas estáticas de código fonte para avaliar a qualidade de um produto de software. Essas métricas utilizadas, essencialmente métricas OO, refletem complexidade arquitetural e decisões de design. Métricas de tamanho também serão avaliadas, com o objetivo de relacionar as demais métricas de forma relativa em vez de uma comparação direta, se possível. Métricas de tamanho são mais simples mas ainda são úteis para encontrar problemas arquiteturais no software, como será discutido no Capítulo 3. Complexidade tem forte relação com o tamanho do código, e manter as duas é uma forma de realizar análises mais adequadas e chegar a resultados mais consistentes.

As métricas finais escolhidas estão listadas a seguir. Suas explicações e definições serão apresentadas a medida que forem exploradas no Capítulo 3. A seleção não foi feita neste trabalho, sendo utilizada a seleção feita por Meirelles (2013).

- Média de linhas de código por método - *Average Method Lines Of Code* (AMLOC)
- Média de complexidade ciclomática por método - *Average Cyclomatic Complexity per Method* (ACCM)
- Resposta para uma classe - *Response For a Class* (RFC)
- Profundidade na árvore de herança - *Depth in Inheritance Tree* (DIT)
- Número de subclasses - *Number of Children* (NOC)

[4] <http://www.analizo.org/>

- Falta de coesão em métodos - *Lack of Cohesion in Methods* (LCOM4)
- Conexões aferentes de uma classe - *Afferent Connections per Class* (ACC)
- Fator de acoplamento - *Coupling Factor* (COF)

Essas métricas foram coletadas para cada classe presente em cada versão do Android analisada. Cada uma das versões contém milhares de classes/módulos a serem computados, e essa grande quantidade de classes ajuda a compensar o pequeno número de versões do sistema que foi utilizado, pois como as métricas são calculadas por classe, foi gerada uma quantidade significativa de amostras para cada tag do sistema.

A maioria das métricas aqui utilizadas foi selecionada por ser bem difundida e discutida na literatura, levando então a ter estudos para relacionar a este. Por exemplo, a possível comparação com valores encontrados em trabalhos semelhantes, como a tese de Oliveira (2013) e o trabalho de Ferreira et al. (2009), incentiva a escolha dessas métricas. Além disso, métricas também úteis, porém não discutidas nesses estudos, como por exemplo as métricas de Halstead, não são capturadas pela ferramenta Analizo.

É importante enfatizar que inicialmente a métrica de acoplamento que seria utilizada seria CBO, a fim de comparação com complexidade estrutural apresentados em trabalhos como o de Terceiro e Chavez (2009). Entretanto, foram encontrados aqui valores muito discrepantes dos valores esperados, divergindo muito dos valores encontrados por Meirelles (2013), inclusive para o sistema Android, levando então a conclusão de que algum problema pode ter ocorrido no cálculo da ferramenta. Assim, a métrica de acoplamento utilizada neste estudo foi a métrica ACC.

Para a análise de código fonte em C, que é uma linguagem estruturada, com essas métricas orientadas a objetos, algumas observações devem ser ressaltadas: Em vez de classes, são considerados módulos, e as funções são utilizadas como métodos (TERCEIRO; CHAVEZ, 2009). Embora seja uma abordagem relativamente eficaz para o cálculo das métricas OO, os valores de algumas métricas podem ser bastante distintos das mesmas métricas calculadas para as linguagens que realmente utilizam o paradigma orientado a objetos. Entretanto, como já comentado, não há representatividade da linguagem C nos dados obtidos que motive a análise desses paradigmas isoladamente. Os valores para C++ e Java já se apresentam similares por utilizarem o mesmo paradigma e portanto terem funcionamento semelhante.

Antes de prosseguir com a utilização dos dados, foi preciso executar uma correção dos dados, pois os arquivos CSV de saída continham classes/módulos com parâmetros de tipos genéricos que utilizam vírgula em sua declaração, comprometendo a formatação correta do arquivo CSV, que utiliza vírgula como separador de valores. Esse problema fazia com que algumas linhas fossem calculadas com mais valores do que deveriam conter,

uma vez que as vírgulas adicionais empurravam os valores para a direita e os últimos eram então ignorados. A correção de dados então se resumiu em identificar essas amostras que continham vírgulas e remover as vírgulas adicionais. Esses dados corrigidos então foram armazenados em um diretório a parte para utilização nos estágios seguintes.

Após essa captura de todas as métricas, foi utilizada a linguagem R para manipulação inicial dos dados. R é uma linguagem focada em computação estatística e possui diversos módulos que facilitam análise estatística, como manipulação facilitada de tabelas e apresentação de dados na forma de gráficos.

Foram calculados, com linguagem R, os percentis de cada métrica para cada versão do sistema analisada. Cada percentil é a centésima parte dos dados ordenados de forma crescente, ou seja, cada percentil contém 1% dos dados sendo que o primeiro contém as amostras com menor valor. Esses valores representam nada mais que a frequência cumulativa para cada valor. Isso quer dizer que, para o 90º percentil com um valor de 500, por exemplo, 90% das amostras apresentam o valor menor ou igual a 500. Como essa frequência cumulativa é calculada em cima dos dados ordenados, a mediana pode ser encontrada no 50º percentil.

Os percentis que foram armazenados foram: 1, 5, 10, 25, 50, 75, 90, 95 e 99, além do menor valor e do valor máximo. Esses dados foram posteriormente reunidos em um arquivo em separado para cada métrica contendo os percentis daquela métrica calculados para cada uma das versões do sistema. A Tabela 1 contém um exemplo desse resultado demonstrado para a métrica de complexidade ciclomática coletada do código fonte do sistema Android em várias versões, e sua análise será explorada no Capítulo 3. Os dados coletados para aplicativos foram reunidos da mesma maneira, porém, como só a versão 5.1 foi analisada para os aplicativos, e em vez de versões, o primeiro valor da tabela é o nome do aplicativo do sistema de onde as métricas foram coletadas.

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.45 | 4.69 | 9.5 | 55 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.45 | 4.69 | 9.5 | 55 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.5 | 4.75 | 9.74 | 59 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 1 | 1.12 | 2 | 3.5 | 4.8 | 9.88 | 60 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 1 | 1.07 | 2 | 3.74 | 5.28 | 12 | 99 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 1 | 1.07 | 2.02 | 3.75 | 5.26 | 12 | 99 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 1 | 1 | 2.07 | 4 | 5.82 | 12.83 | 99 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 1 | 1 | 2.08 | 4 | 5.8 | 12.76 | 99 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 1 | 1 | 2.13 | 4 | 6 | 17 | 94.33 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 1 | 1 | 2.11 | 4 | 6 | 17 | 94.33 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.86 | 5.78 | 16 | 99.4 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.62 | 5.23 | 12 | 120.4 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.5 | 5 | 11 | 158.6 |

Tabela 1 – Complexidade ciclomática nas versões da API analisadas

Com esses arquivos em mãos e com os valores de vários percentis para cada métrica nas diversas versões, podem então ser gerados gráficos para melhor visualizar a evolução

de cada métrica juntamente com a evolução do sistema. Esses resultados serão discutidos no Capítulo 3.

Alguns outros gráficos também podem ser gerados a partir dos dados completos, com os valores de cada métrica para cada classe. Gráficos como de distribuição, demonstrando os valores mais frequentes, e gráfico de linha ou de área para demonstrar evolução das métricas, são bastante úteis para a análise dos dados coletados.

Assim como feito por Meirelles (2013), os resultados das análises discutidos no Capítulo 3 serão em função dos percentis 75, 90 e 95, correspondendo a valores muito frequentes, frequentes e pouco frequentes, respectivamente. A utilização de 95%, embora não inclua todas as amostras, ainda resulta em uma amostra estatística significativa, uma vez que, como os valores estão ordenados, boa parte desses 5% restantes são valores discrepantes que podem interferir negativamente na análise correta dos dados.

Este capítulo teve o intuito de descrever os procedimentos gerais para obtenção e análise dos dados, enquanto a análise em si será abordada no Capítulo 3. Todos os códigos utilizados para manipular os dados estão disponíveis em um repositório público[5] dentro do diretório "analise".

No Capítulo 3 também serão discutidos possíveis modelos de regressão, e será apresentada uma proposta a verificação de similaridade de aplicativos em relação à API, com base nos valores dessas métricas coletadas e nas distribuições dos valores a partir dos percentis 75, 90 e 95. São apresentadas discussões sobre grandeza dos dados das métricas, normalização de valores, e uma abordagem de pesos para dar às métricas diferentes valores de importância no valor final de similaridade. Uma fórmula para cálculo desse valor é apresentada para que outros projetos possam ter sua distância calculada e poder fazer comparação com os resultados dessa proposta.

[5] https://github.com/marcosronaldo/tcc-latex

# 3 Análise Exploratória

Na primeira seção deste capítulo serão discutidos os resultados da análise de distribuição dos dados coletados, definindo para a API Android valores de referência em cada métrica. Na seção seguinte, é feito um trabalho de comparação de aplicativos com a API em função dos valores de referência definidos para a API em cada métrica. A primeira seção juntamente com a proposta de cálculo de similaridade da segunda seção reúnem as principais contribuições deste trabalho.

Na terceira seção, são apresentadas discussões sobre a possibilidade de validação dos valores de referência definidos no início deste capítulo através de regressão estatística.

A conceituação dos recursos utilizados neste trabalho está sendo feita a medida que os resultados são apresentados.

## 3.1 Análise de Distribuição

Esta seção é focada na análise dos dados, tentando explicar seu comportamento com relação às características do sistema, compará-los a outros estudos, e até mesmo comparar com dados de métricas em aplicativos, utilizando os próprios aplicativos do sistema como base de comparação.

### 3.1.1 Distribuição dos dados

Os gráficos na Figura 3 apresentam as distribuições dos dados para as métricas LCOM4, ACC, ACCM e RFC. Em todos os casos, a probabilidade cumulativa, representada pelos percentis, apresenta um aumento significativo de valor apenas nos últimos percentis, demonstrando valores bem discrepantes em relação aos demais. Entretanto, devido ao aumento acontecer a partir do percentil 95, esses 5% podem ser descartados como um ruído estatístico.

Nas tabelas que serão discutidas nas seções seguintes, a mediana, no percentil 50, não representa de forma alguma os dados como faria numa distribuição normal. Para muitas métricas esse percentil apresenta nada mais que o ideal teórico que na prática não é comum, como nas métricas ACCM, NOC, LCOM4 e ACC. Assim, serão utilizados percentis 75, 90 e 95 para análise de cada métrica, assim como feito por Meirelles (2013).

Valores considerados como 0 para muitas métricas não tem valor semântico para interpretação e resulta da impossibilidade de calcular a métrica. Resultado 0 para LCOM4, ACCM e RFC, por exemplo, indicam classes sem métodos.

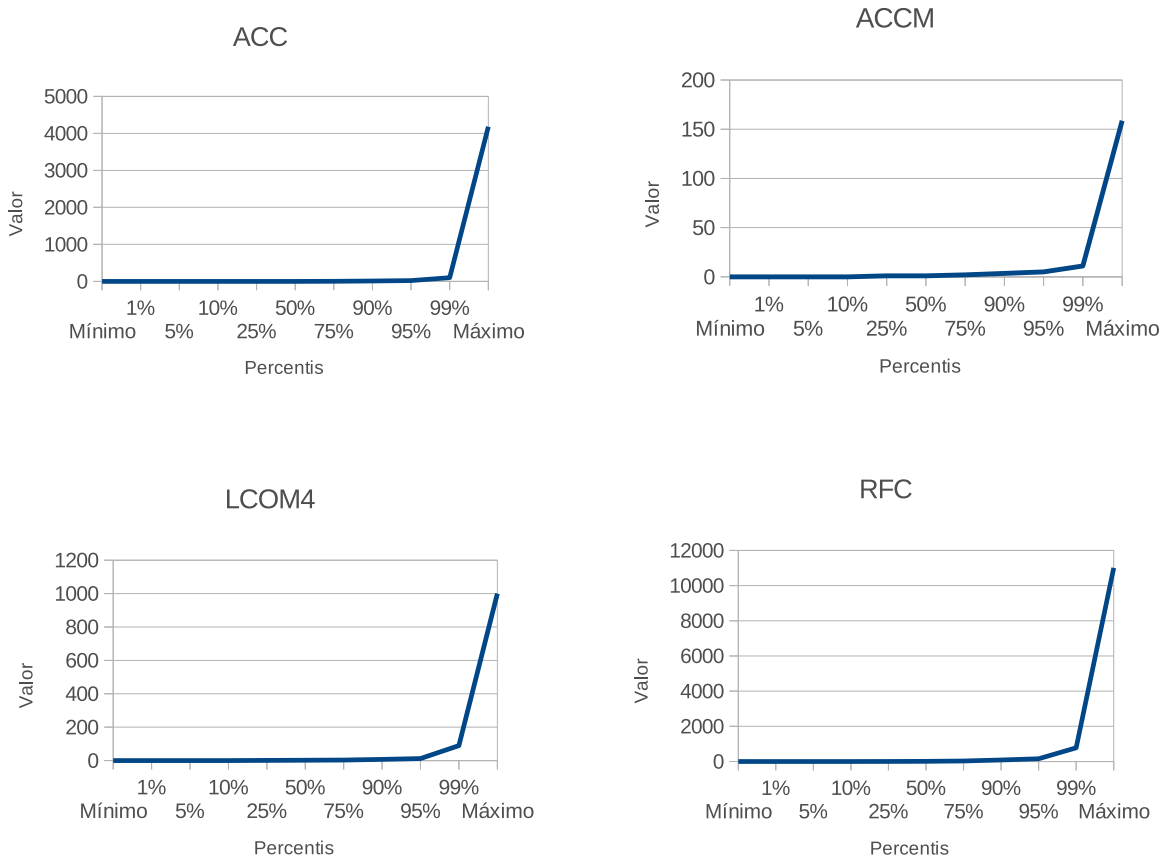


Figura 3 – Distribuição das métricas ACC, ACCM, LCOM4 e RFC na versão 5.1.0

### 3.1.2 Média de linhas de código por método (AMLOC)

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 2.33 | 5.57 | 11.5 | 21.5 | 30 | 65.87 | 312 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 2.33 | 5.57 | 11.5 | 21.5 | 30 | 65.87 | 312 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 2 | 5.56 | 11.5 | 21.85 | 30.19 | 67.81 | 390.5 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 2 | 5.63 | 11.5 | 21.86 | 30.34 | 68.4 | 395 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 1.76 | 5.8 | 12.8 | 26.5 | 44.21 | 156.66 | 1034 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 1.77 | 5.82 | 12.82 | 26.5 | 44.17 | 156.62 | 1034 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 1 | 5.8 | 13.6 | 30.18 | 55.36 | 164.77 | 1034 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 1 | 5.83 | 13.71 | 30 | 54.06 | 163.4 | 1034 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 1 | 5.86 | 14 | 31 | 54.37 | 162.42 | 1034 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 1 | 5.86 | 14 | 31 | 53.98 | 162 | 1034 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 1 | 5.5 | 13.08 | 28.96 | 51 | 151.95 | 1034 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 1 | 5.5 | 12.5 | 26.2 | 43 | 126 | 721 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 2 | 5.5 | 12 | 24 | 37.8 | 105 | 708 |

Tabela 2 – AMLOC no Android

LOC representa o número de linhas de código fonte de uma classe, enquanto AMLOC a média do número de linhas dos métodos daquela classe. O valor de AMLOC deve ser o menor possível, pois métodos grandes “abrem espaço” para problemas de complexidade excessiva.

A Tabela 2 apresenta os valores para a métrica AMLOC nas versões do Android

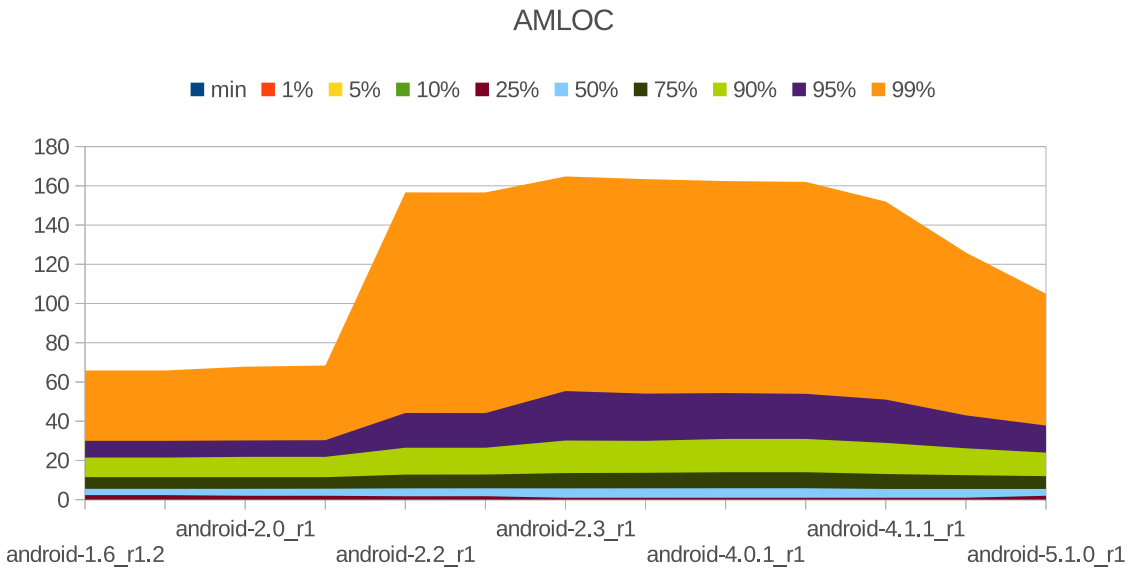


Figura 4 – Evolução da métrica AMLOC ao longo das versões da API

analisadas. É facilmente perceptível que a média de linhas de código por método não teve variação relevante. Em todas as versões analisadas, os valores muito frequentes, isto é, percentil 75, são métodos com até 14 linhas de código, enquanto de 14 a 30 aparecem como frequentes, e 31 a 55 pouco frequentes. A Figura 4 apresenta a evolução da métrica AMLOC com a API, com uma variação muito pequena de valores para os percentis 75 e 90. O percentil 99 demonstra uma variação maior, mas ele representa dados não frequentes na análise.

Esses valores estão de acordo com os apresentados em Meirelles (2013), porém levemente menores para os percentis 75 e 90, com aproximadamente 3 linhas de código a menos por método. Os valores se mostraram bem semelhantes para o projeto Android, mesmo considerando o fato que este trabalho estuda apenas a API de desenvolvimento de aplicativos, essencialmente em Java e dentro do diretório "*frameworks*" do AOSP, e Meirelles (2013) analisa todo o código fonte do sistema, que apresenta em sua totalidade uma maior proporção da linguagem C em relação as demais. Esses valores são subsídios para reafirmar que arquivos em C em geral, tem uma maior utilização de linhas de código do que arquivos em Java. Oliveira (2013) comenta que as diferenças entre as linguagens C/C++/Java para esta métrica não é significativa, uma vez que a sintaxe entre as 3 é bastante semelhante. Dada essa afirmação, comparamos os intervalos definidos por ele, chegando à conclusão de que os valores das métricas estão, para todas as versões, abaixo dos valores bom e regular para os percentis 75 e 90, o que é um bom resultado.

Os valores apresentados na análise são relativamente baixos quando comparados com outros softwares livres, como demonstrado por Meirelles (2013). Da mesma forma, os valores de aplicativos do sistema, demonstrados na Tabela 3, apresentam uma grande

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 1 | 1 | 3 | 6.33 | 10.78 | 16.9 | 24.38 | 48.36 | 57.5 |
| Settings | 722 | 0 | 0 | 0 | 0 | 3.63 | 8 | 15 | 21.47 | 28.5 | 49.4 | 80.42 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 1 | 4.5 | 9.85 | 16.09 | 21.5 | 38.28 | 66.67 |
| Bluetooth | 239 | 0 | 0 | 0 | 2.83 | 6.02 | 10.79 | 22.16 | 39.86 | 59.84 | 111.11 | 221 |
| QuickSearchBox | 196 | 0 | 0 | 1 | 1 | 3 | 4.39 | 6.17 | 10.53 | 13.15 | 22.28 | 32 |
| Calculator | 10 | 1 | 1 | 1 | 1 | 6.67 | 8.92 | 13.33 | 23.26 | 27.38 | 30.68 | 31.5 |
| Terminal | 17 | 0 | 0.15 | 0.75 | 1.85 | 3.09 | 8.12 | 15.29 | 20.92 | 29.38 | 48.27 | 53 |
| PackageInstaller | 19 | 0 | 0.68 | 3.4 | 4 | 4.63 | 6.59 | 16.98 | 18.9 | 22.89 | 31.45 | 33.59 |
| Dialer | 215 | 0 | 0 | 0 | 1 | 3 | 7 | 11.13 | 16.89 | 19.98 | 32.02 | 61.33 |
| Browser | 259 | 0 | 0 | 1 | 1 | 3.5 | 6.89 | 11 | 19 | 25.95 | 46.02 | 55.33 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 1 | 4.23 | 12 | 18.75 | 23.32 | 40.77 | 58 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0.2 | 4 | 8.64 | 15.78 | 25.47 | 32.18 | 69.42 | 112.67 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 3 | 6 | 11.5 | 17.38 | 21.67 | 44.36 | 107 |
| BasicSmsReceiver | 5 | 8.67 | 8.83 | 9.47 | 10.27 | 12.67 | 16.33 | 18.75 | 18.9 | 18.95 | 18.99 | 19 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 1 | 3 | 5 | 9.71 | 17 | 23.67 | 37.95 | 139.63 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 2.73 | 5.13 | 10.59 | 17.17 | 24.79 | 54.71 | 163.5 |
| Music | 75 | 0 | 0 | 0 | 1 | 4.1 | 9.51 | 16.89 | 21.76 | 28.0 | 48.32 | 90 |
| Camera | 253 | 0 | 0 | 0 | 1 | 3 | 7.42 | 13 | 22.37 | 31.45 | 72.23 | 112.67 |
| Email | 400 | 0 | 0 | 0 | 1 | 3.58 | 8 | 15.35 | 24.49 | 31.61 | 63.28 | 128 |
| Nfc | 178 | 0 | 0 | 0 | 1 | 3 | 9.64 | 18.5 | 31.63 | 38 | 42.48 | 70.5 |
| Gallery | 89 | 0 | 0.87 | 1 | 1 | 4 | 7.63 | 12.67 | 19.0 | 28.6 | 53.12 | 55 |
| ContactsCommon | 292 | 0 | 0 | 0 | 1 | 3.23 | 7.1 | 13 | 19 | 23.88 | 34.5 | 53.33 |
| Contacts | 265 | 0 | 0 | 0 | 1 | 3 | 6.45 | 11.5 | 18.61 | 23.72 | 63.53 | 86 |
| DeskClock | 121 | 0 | 0 | 0 | 1 | 5 | 9.16 | 15.26 | 24.02 | 27.3 | 30.71 | 40.13 |
| HTMLViewer | 4 | 5 | 5.12 | 5.6 | 6.2 | 8 | 11 | 14.5 | 16.6 | 17.3 | 17.86 | 18 |
| Calendar | 216 | 0 | 0 | 0 | 1 | 5 | 11.67 | 19.58 | 30.95 | 39.3 | 90 | 115.5 |
| Exchange | 135 | 0 | 0 | 0 | 1 | 4 | 10.01 | 17.31 | 28.41 | 34.65 | 44.41 | 51.25 |

Tabela 3 – AMLOC nos aplicativos nativos

semelhança nos resultados.

Métodos relacionados à interface gráfica tentem a ser relativamente grandes quando a interface é criada dinamicamente em Java, mas esse aumento não tem grande representatividade no código do sistema, uma vez que a maioria das partes do sistema que contém componentes gráficos estão nos aplicativos, como o launcher, settings e outros apps do sistema. Assim, aplicativos podem ter valores maiores de AMLOC, principalmente nos componentes do tipo *Activity*.

Embora alguns poucos aplicativos tenham valores mais elevados para essa métrica, os intervalos se mantém válidos para a grande maioria dos aplicativos. Esses valores de aplicativos foram retirados dos aplicativos nativos da última versão do sistema analisada (Lollipop 5.1.0), e continuam se mantendo semelhantes ao sistema, como o próprio acoplamento à API sugere.

Os aplicativos do sistema também se mantém dentro dos intervalos bom e regular definidos em Oliveira (2013). Os valores para o percentil 95 também se encontram abaixo do valor regular, na maioria dos casos.

Em suma, os valores para os aplicativos se assemelham muito com os valores para as versões da API Android analisadas, levando então à conclusão de que os mesmos intervalos são válidos para as métricas em ambos os casos, embora se possa esperar valores menores em aplicativos, porém com uma maior variância. Essa variância se dá pelo di-

ferente propósito de cada aplicativo, que utiliza pedaços variados do sistema e tem sua codificação adaptada para seu propósito.

Intervalos encontrados:

- Valores abaixo de 14 são muito frequentes nos aplicativos e na API;
- Enquanto no sistema os valores para o percentil 90 estão abaixo de 31, nos aplicativos eles alcançam esse valor em poucos casos, ficando em sua maioria abaixo de 25;
- Valores acima de 31 são pouco frequentes em ambos os casos;

### 3.1.3 Média de complexidade ciclomática por método (ACCM)

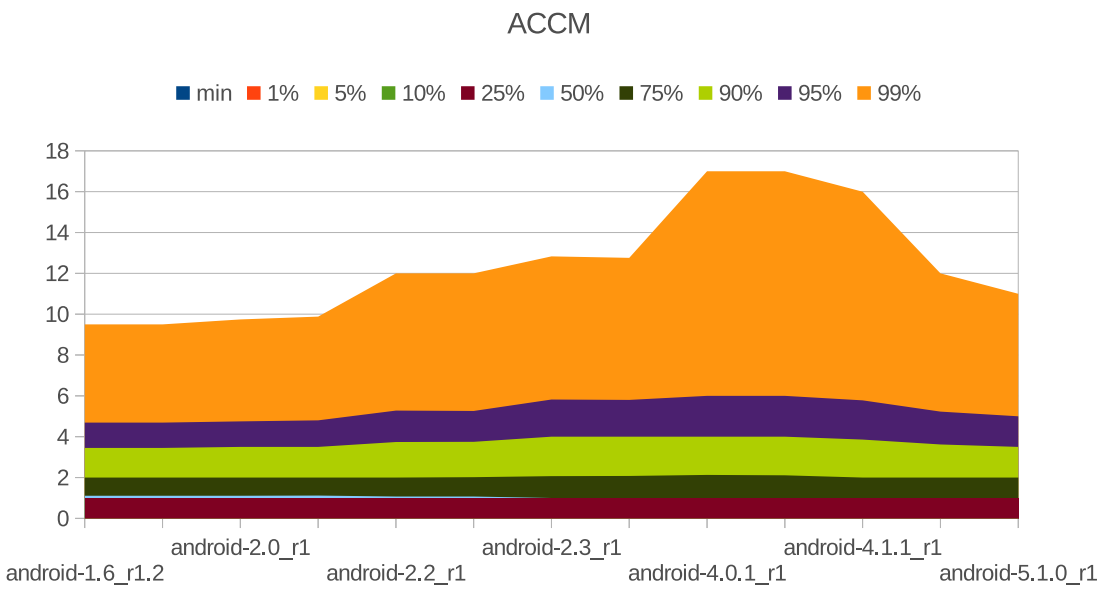


Figura 5 – Evolução da métrica ACCM ao longo das versões da API

Complexidade ciclomática contabiliza o número de caminhos independentes que um software pode seguir em sua execução, calculado a partir da representação em grafo das estruturas de controle (SHEPPERD, 1988). Na prática, cada condicional dentro do sistema incrementa o valor desta métrica em 1, uma vez que divide a execução em um caminho de execução se a expressão condicional for válida, ou um segundo caminho caso não seja. O valor teórico ideal para essa métrica é quando o software tem apenas um caminho de execução, com complexidade ciclomática igual a 1. Complexidade ciclomática é calculada em nível de método, e o valor de ACCM para uma classe corresponde a média dos valores de complexidade ciclomática de cada um dos seus métodos.

A Figura 5 apresenta a evolução de ACCM métrica com a evolução da API. Essa métrica não teve uma variação grande ao longo das versões nos percentis 75, 90 e 95, e o

pouco do valor que ganhou nas versões centrais do gráfico foi sendo novamente reduzido nas versões seguintes. Valores sinalizados com 0 ocorrem em classes que não contém métodos.

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.45 | 4.69 | 9.5 | 55 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.45 | 4.69 | 9.5 | 55 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 1 | 1.11 | 2 | 3.5 | 4.75 | 9.74 | 59 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 1 | 1.12 | 2 | 3.5 | 4.8 | 9.88 | 60 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 1 | 1.07 | 2 | 3.74 | 5.28 | 12 | 99 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 1 | 1.07 | 2.02 | 3.75 | 5.26 | 12 | 99 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 1 | 1 | 2.07 | 4 | 5.82 | 12.83 | 99 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 1 | 1 | 2.08 | 4 | 5.8 | 12.76 | 99 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 1 | 1 | 2.13 | 4 | 6 | 17 | 94.33 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 1 | 1 | 2.11 | 4 | 6 | 17 | 94.33 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.86 | 5.78 | 16 | 99.4 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.62 | 5.23 | 12 | 120.4 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3.5 | 5 | 11 | 158.6 |

Tabela 4 – ACCM no Android

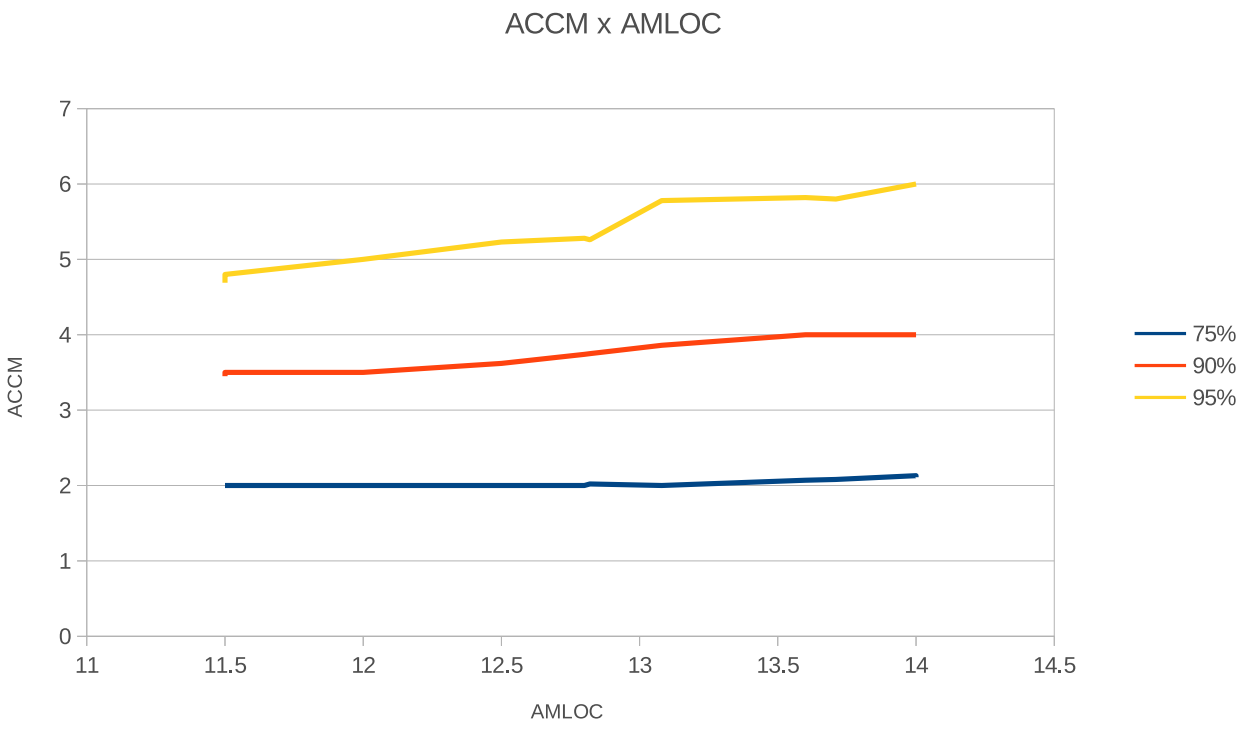


Figura 6 – ACCM por AMLOC nas Tabelas 2 e 4

Uma relação entre ACCM e AMLOC pode ser claramente vista na Tabela 4 e está melhor demonstrada no gráfico da Figura 6. Nos valores do percentil 75, que correspondem a valores muito frequentes, as versões de 2.2.3 a 4.0.4 contém os únicos valores para o sistema onde a complexidade ciclomática supera o número 2, e não por acaso são os valores com maior AMLOC nesse percentil como demonstra a Tabela 2. Essa relação direta também está presente nos percentis 90 e 95, que representam valores frequentes e pouco frequentes, respectivamente.

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 1 | 1 | 1 | 1.4 | 2.09 | 3 | 3.62 | 8.68 | 11.87 |
| Settings | 722 | 0 | 0 | 0 | 0 | 1 | 1.57 | 2.5 | 3.67 | 4.43 | 8 | 17 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 1 | 1 | 1.83 | 2.67 | 3.5 | 6 | 8.3 |
| Bluetooth | 239 | 0 | 0 | 0 | 1 | 1.33 | 2.19 | 4 | 7.95 | 10.36 | 24.29 | 49 |
| QuickSearchBox | 196 | 0 | 0 | 1 | 1 | 1 | 1.03 | 1.67 | 2.47 | 3 | 4.68 | 5 |
| Calculator | 10 | 1 | 1 | 1 | 1 | 1.33 | 1.58 | 2.67 | 3.53 | 5.27 | 6.65 | 7 |
| Terminal | 17 | 0 | 0.15 | 0.75 | 1 | 1 | 1.5 | 1.74 | 4.8 | 8.25 | 10.45 | 11 |
| PackageInstaller | 19 | 0 | 0.17 | 0.85 | 1 | 1 | 1.3 | 2.86 | 3.7 | 4.07 | 4.39 | 4.47 |
| Dialer | 215 | 0 | 0 | 0 | 1 | 1 | 1.17 | 2 | 2.77 | 3 | 3.97 | 57.83 |
| Browser | 259 | 0 | 0 | 1 | 1 | 1 | 1.5 | 2.17 | 3.27 | 4.04 | 7.03 | 8.8 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 1 | 1.06 | 2.01 | 2.76 | 4 | 6.7 | 8.33 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0.2 | 1 | 1.6 | 2.4 | 3.37 | 4.11 | 9.59 | 10 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 1 | 1.38 | 2.16 | 3 | 3.71 | 6.01 | 11 |
| BasicSmsReceiver | 5 | 1.33 | 1.34 | 1.38 | 1.43 | 1.58 | 1.71 | 1.99 | 2.43 | 2.58 | 2.7 | 2.73 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 1 | 1 | 1.17 | 1.94 | 2.83 | 3.67 | 6.69 | 53 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 1 | 1.23 | 2 | 3 | 4 | 9.87 | 30 |
| Music | 75 | 0 | 0 | 0 | 1 | 1 | 1.67 | 2.5 | 3.37 | 4.31 | 8.97 | 18 |
| Camera | 253 | 0 | 0 | 0 | 1 | 1 | 1.49 | 2.27 | 3.17 | 3.97 | 9.83 | 17 |
| Email | 400 | 0 | 0 | 0 | 1 | 1 | 1.33 | 2 | 3.02 | 4.18 | 7.51 | 19.4 |
| Nfc | 178 | 0 | 0 | 0 | 1 | 1 | 2 | 3.35 | 5.16 | 7.87 | 9.62 | 15.5 |
| Gallery | 89 | 0 | 0.87 | 1 | 1 | 1 | 1.61 | 2.5 | 3.35 | 3.91 | 7.26 | 9 |
| ContactsCommon | 292 | 0 | 0 | 0 | 1 | 1 | 1.29 | 2 | 3.44 | 4.54 | 7 | 7.5 |
| Contacts | 265 | 0 | 0 | 0 | 1 | 1 | 1.23 | 2 | 3 | 3.64 | 9.74 | 21 |
| DeskClock | 121 | 0 | 0 | 0 | 1 | 1 | 1.69 | 2.29 | 3.26 | 3.81 | 4.49 | 5.33 |
| HTMLViewer | 4 | 1.5 | 1.51 | 1.55 | 1.6 | 1.75 | 2 | 2 | 2 | 2 | 2 | 2 |
| Calendar | 216 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 4.68 | 6.33 | 14.93 | 19 |
| Exchange | 135 | 0 | 0 | 0 | 1 | 1 | 1.66 | 3.2 | 4.49 | 5.41 | 6.83 | 7.67 |

Tabela 5 – ACCM nos aplicativos nativos

Assim como na API, no geral aplicativos com maiores valores de AMLOC tendem a ter um maior valor de ACCM, embora não seja totalmente determinístico. Com exceção do *SMSReceiver*, os 5 aplicativos com maior valor de AMLOC são os que contém maior complexidade do conjunto.

Reforçando a importância dessa métrica em uma análise estática de código, Oliveira (2013) define a mesma com um peso adicional em relação a outras métricas em seu estudo. Valores de referência definidos para esse estudo foram 1 a 3, 3 a 5, e 5 a 7, para excelente, bom e regular, respectivamente. As Tabelas 4 e 5 demonstram que os valores obtidos neste trabalho estão dentro do intervalo excelente ou bom, para os percentis 75 e 90, e dentro de bom ou regular para o percentil 95, que representa valores menos frequentes. No geral, os resultados indicam que o sistema tem uma boa complexidade ciclomática e que os aplicativos desenvolvidos para o mesmo acompanham essa mesma linha. Meirelles (2013) definiu intervalos semelhantes para códigos em C, e valores um pouco reduzidos para códigos em C++ e Java (0 a 2, 2 a 4, e 4 a 6 para os percentis 75, 90 e 95 respectivamente). Os resultados encontrados para a API do sistema Android se encontram todos dentro desses intervalos, confirmando o bom resultado. Já os aplicativos tem algumas exceções que extrapolam levemente esses valores, mas continuam em sua maioria dentro desses limites.

Baseando-se nessas observações, são considerados os seguintes intervalos:

- Valores abaixo 2 são muito frequentes para os aplicativos e para a API, e até 2.5 são considerados excelentes. É importante relembrar que uma complexidade ciclomática 2 implica em afirmar que 2 testes unitários resultam em 100% de cobertura para esse trecho de código;
- ACCM é menor ou igual a 4 em todas as versões do Android e na grande maioria dos aplicativos dentro do percentil 90, sendo uma referência para um valor maior mas ainda considerado bom.
- Valores acima de 4 são considerados regulares e são pouco frequentes em ambos os casos, porém para a API do sistema o percentil 95 chegou a 6. Valores acima de 6 são bem raros e correspondem a uma quantidade estatisticamente desprezível para esta análise.

### 3.1.4 Resposta para uma classe (RFC)

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 2 | 10 | 31 | 79 | 133.8 | 357.48 | 2858 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 2 | 10 | 31 | 79 | 133.8 | 357.48 | 2858 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 2 | 10 | 31 | 79 | 131.5 | 350 | 2902 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 2 | 10 | 32 | 79 | 133.05 | 352.87 | 2923 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 2 | 9 | 30 | 77.9 | 131.45 | 372.45 | 2754 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 2 | 9 | 30 | 78 | 131.15 | 372.15 | 2754 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 1 | 8 | 27 | 76 | 129 | 358 | 2347 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 1 | 8 | 27 | 76 | 130 | 354.22 | 2347 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 1 | 7 | 28 | 82 | 140 | 388 | 2871 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 1 | 7 | 28 | 81 | 141 | 391 | 2921 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 1 | 7 | 26 | 77.6 | 136 | 363.72 | 6596 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 1 | 8 | 29 | 81 | 143 | 379 | 8279 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 2 | 9 | 30 | 84 | 156 | 775.72 | 11010 |

Tabela 6 – RFC no Android

RFC é uma métrica que conta o número de métodos que podem ser executados a partir de uma mensagem enviada a um objeto dessa classe (CHIDAMBER; KEMERER, 1994). O valor então é calculado pelo somatório de todos os métodos daquela classe, e todos os métodos chamados diretamente por essa classe. Uma classe com alto valor de RFC pode ser uma classe com um número muito grande de métodos, e/ou uma classe bastante dependente de outra(s) classe(s). Um valor alto de RFC pode indicar baixa coesão e alto acoplamento. Valores mais próximos a zero são os ideais teóricos dessa métrica.

A API do sistema Android tende a ter um valor relativamente alto de RFC devido à forma como sua arquitetura foi desenhada, como apresenta a Tabela 6. Serviços do sistema são acessados muitas vezes através de objetos do sistema, e para seu uso correto alguns métodos devem ser chamados explicitamente. Por exemplo, para acessar o *bluetooth*, não se chama diretamente um método de uma classe *BluetoothAdapter*, pois os serviços do sistema geralmente estão encapsulados e são retornados por um método *get()*, seguidos dos métodos que se deseja utilizar desse serviço. Por exemplo, para verificar dispositivos

*bluetooth* próximos, deve-se obter o *adapter* via chamada estática de método para a própria classe para obter a instância, seguida de uma chamada de método para início de *discovery* de dispositivos, e em seguida utilizar os métodos *isDiscovering()* e *cancelDiscovery()* para controlar a busca. Um acesso direto a uma variável booleana removeria a necessidade da chamada de método *isDiscovering()*, entretanto perderia seu encapsulamento. De forma geral, encapsulamento de variáveis tende a aumentar o valor de RFC, que conta apenas métodos.

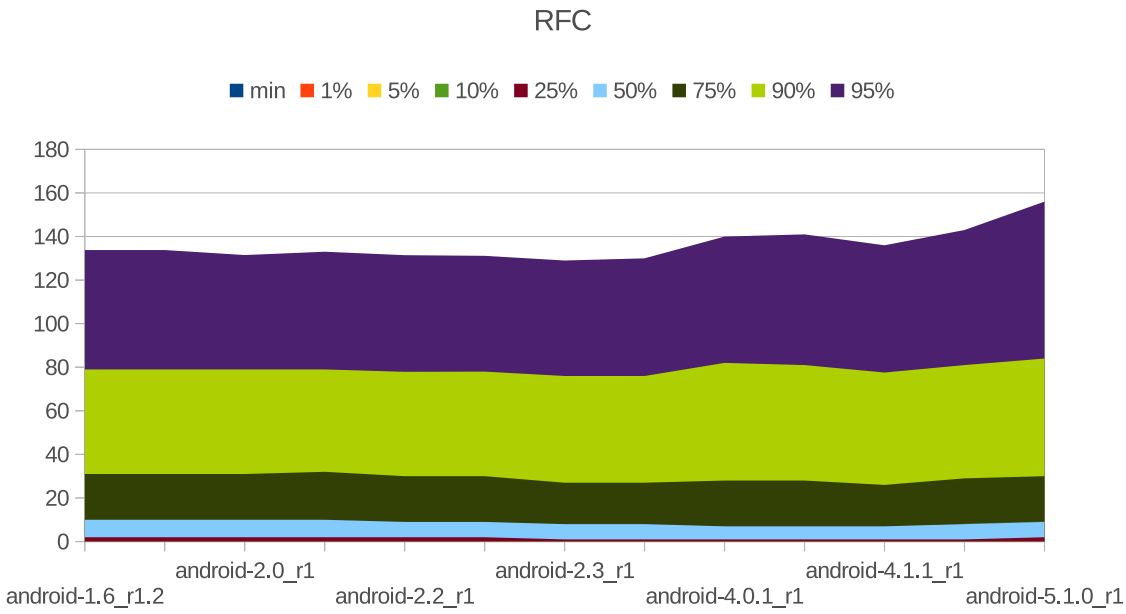


Figura 7 – Evolução da métrica RFC ao longo das versões da API

Contribuindo para o aumento, o resultado da busca de dispositivos bluetooth é realizado de forma assíncrona na forma de mensagens utilizando *intents*, então mais um método é criado dentro de um *receiver* (que pode ser a própria classe estendendo *BroadcastReceiver*) para receber essa mensagem, aumentando um pouco o valor de RFC. Uma comunicação síncrona hipotética com uma chamada estática direta como *BluetoothAdapter.discoverNearDevices()* retornando uma lista seria em teoria uma forma muito mais simples de ser utilizada, porém perderia a proteção do encapsulamento e deixaria de utilizar o comportamento em escopo de objeto para usar em escopo de classe, e também se perderia o maior controle sobre a própria busca que a API dá ao usuário com os métodos adicionais. Além disso, o encapsulamento de serviço dos sistema é um controle adicional que permite o mesmo escalonar melhor a utilização de recursos que necessitam de exclusão mútua. Por exemplo, um acesso direto à câmera dificultaria o sistema de dar acesso a 1 cliente de cada vez, pois afinal, o usuário não consegue usar a câmera, por exemplo, em dois aplicativos simultaneamente.

Comunicações assíncronas são muito usadas ao longo de todo o sistema para utili-

zação de recursos, e então é necessário ter uma forma de receber mensagens de aplicativos e do sistema, o que é feito com a classe *BroadcastReceiver* e implementando métodos específicos da mesma. Mesmo fora do contexto Android, comunicações assíncronas tendem a criar métodos adicionais de comunicação, como é visto no padrão *Observer*, que se assemelha muito a essa comunicação por *Intents*. A Figura 7 mostra que essa métrica não teve variação grande ao longo das versões do Android, tendendo inclusive a um leve aumento nas últimas versões, demonstrando que os valores altos apresentados são mesmo uma característica da arquitetura do sistema.

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 1 | 1 | 3.75 | 12.5 | 37.25 | 84.4 | 178.75 | 850.72 | 1061 |
| Settings | 722 | 0 | 0 | 0 | 0 | 3 | 10 | 31 | 69 | 112 | 229.2 | 596 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 1 | 6 | 22 | 65 | 113 | 352.2 | 752 |
| Bluetooth | 239 | 0 | 0 | 0 | 2 | 8 | 25 | 68.75 | 131.3 | 205.6 | 468.34 | 658 |
| QuickSearchBox | 196 | 0 | 0 | 1 | 1 | 3 | 8 | 19 | 30 | 57.3 | 115.12 | 213 |
| Calculator | 10 | 1 | 1 | 1 | 1 | 6 | 8 | 13 | 42.8 | 62.4 | 78.08 | 82 |
| Terminal | 17 | 0 | 0.15 | 0.75 | 1 | 3 | 13 | 47.5 | 52.5 | 56.5 | 64.9 | 67 |
| PackageInstaller | 19 | 0 | 0.17 | 0.85 | 2.4 | 3 | 10.5 | 26 | 62.5 | 128 | 155.2 | 162 |
| Dialer | 215 | 0 | 0 | 0 | 1 | 3 | 8.5 | 25 | 66.4 | 113.1 | 250.74 | 321 |
| Browser | 259 | 0 | 0 | 1 | 1 | 3.25 | 13 | 36.75 | 78.6 | 124.35 | 328.89 | 795 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 1 | 8 | 28.25 | 82.5 | 120.25 | 297.35 | 434 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0.2 | 3 | 11 | 38 | 77.6 | 139.6 | 400.76 | 742 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 3 | 12 | 33.75 | 77.7 | 110 | 312.14 | 595 |
| BasicSmsReceiver | 5 | 7 | 7.03 | 7.15 | 7.3 | 7.75 | 12.5 | 24.75 | 38.7 | 43.35 | 47.07 | 48 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 1 | 3 | 9 | 25 | 59 | 114.5 | 356.8 | 1012 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 2 | 9 | 31 | 77.8 | 144.6 | 515.64 | 1407 |
| Music | 75 | 0 | 0 | 0 | 1 | 2 | 6.5 | 19.75 | 45.6 | 95.0 | 155.5 | 192 |
| Camera | 253 | 0 | 0 | 0 | 1 | 2 | 10 | 33 | 80 | 128.5 | 307.27 | 921 |
| Email | 400 | 0 | 0 | 0 | 1 | 2.5 | 9 | 28 | 60 | 93.2 | 192.22 | 399 |
| Nfc | 178 | 0 | 0 | 0 | 1 | 3 | 16 | 37 | 93.8 | 155 | 263.44 | 306 |
| Gallery | 89 | 0 | 0.87 | 1 | 1 | 4 | 17 | 31.5 | 68.3 | 118.7 | 224.66 | 296 |
| ContactsCommon | 292 | 0 | 0 | 0 | 1 | 3 | 9 | 22 | 60 | 105.5 | 199.4 | 271 |
| Contacts | 265 | 0 | 0 | 0 | 1 | 3 | 8.5 | 23 | 59 | 96.25 | 253.46 | 463 |
| DeskClock | 121 | 0 | 0 | 0 | 1 | 4.75 | 21 | 50.25 | 121.3 | 151.35 | 230.25 | 691 |
| HTMLViewer | 4 | 1 | 1.06 | 1.3 | 1.6 | 2.5 | 4 | 4.5 | 4.8 | 4.9 | 4.98 | 5 |
| Calendar | 216 | 0 | 0 | 0 | 1 | 4 | 13 | 37.5 | 109.6 | 160 | 422.8 | 1291 |
| Exchange | 135 | 0 | 0 | 0 | 1 | 4 | 14 | 37.75 | 72.1 | 107.55 | 162.05 | 224 |

Tabela 7 – RFC nos aplicativos nativos

Em suma, na API do sistema, o valor de RFC pode ser considerado alto, porém justificável. Os componentes do Android podem ter seu valor RFC e DIT mais alto que outras classes em Java devido ao nível de herança que eles apresentam. Por exemplo, um componente gráfico em um *app* herda de *Activity*, que por sua vez herda de *Context*, que contém uma estrutura de hierarquia intermediária.

O acoplamento entre a própria API de desenvolvimento e o aplicativos fica exemplificado com o valor dessa métrica sendo alto para ambos os casos. É importante lembrar que componentes do sistema se comunicam da mesma forma com outros componentes do sistema como se comunicam com aplicativos desenvolvidos para o mesmo. A Tabela 7 demonstra os valores de RFC para aplicativos nativos.

Meirelles (2013) define como bons intervalos para projetos Java valores de 0 a 9, 10 a 26, e 27 a 59, para os percentis 75, 90 e 95, respectivamente. Os valores na análise

da API obtidos neste trabalho estão bem acima desse valor, estando em seu percentil 75 um valor perto de 30, que seria no máximo regular nessa escala.

Baseando-se em todas essas observações, consideramos os seguintes intervalos:

- Valores abaixo 31 são frequentes para API Android. Para os aplicativos do sistema, existe uma grande variância de valores, porém estando em sua grande maioria abaixo de 38 para o percentil 75.
- RFC chegou a 130 em aplicativos nativos, porém no geral não alcançam o valor 85. Esse mesmo valor é o limite para a API do sistema.
- Valores acima de 85 são valores considerados altos para a métrica RFC, e são pouco frequentes nos dados analisados. Valores acima de 140 não são frequentes.

Um intervalo de valores até 38 pode ser considerado bom para aplicativos, e regular no intervalo desse valor até o 85. Acima disso são considerados valores altos para projetos Java, mas são comuns no Android. Esses intervalos aqui encontrados estão mais próximos dos limites definidos por Meirelles (2013) para a Linguagem C do que para Java. Inclusive os valores aqui encontrados aqui para a API Android, essencialmente em Java, são semelhantes aos valores para o projeto Android como um todo, com predominância da linguagem C, encontrados por Meirelles (2013). Essa semelhança leva a conclusão de que estilos semelhantes de estruturação e design são utilizados em todo o AOSP, independente da linguagem utilizada em cada módulo.

### 3.1.5 Profundidade na árvore de herança (DIT) / Número de subclasses (NOC)

DIT é uma métrica que mede a profundidade que uma classe se encontra na árvore de herança, e caso haja herança múltipla, DIT mede a distância máxima até o nó raiz da árvore de herança (CHIDAMBER; KEMERER, 1994). Se ela não herda nada, tem DIT igual a 0. NOC mede a quantidade de filhos que uma classe tem (CHIDAMBER; KEMERER, 1994). Caso ninguém herde dela, o valor é 0. Filhos de filhos não são contabilizados em NOC. Valores mais próximos a zero são os ideais teóricos de ambas as métricas.

As Figuras 8 e 9 demonstram o comportamento de DIT e NOC com a evolução do sistema. São valores baixos e com variação muito pequena para as duas métricas. Para DIT os valores para cada percentil varia em no máximo 1 em algumas versões e depois voltam ao valor anterior. NOC também se mantém 0 ou 1 para o percentil 90 em todas as versões.

A primeira observação sobre o dados das Tabelas 8 e 9 é que elas contém um número grande de zeros até o percentil 50. Como a linguagem Java representa mais de

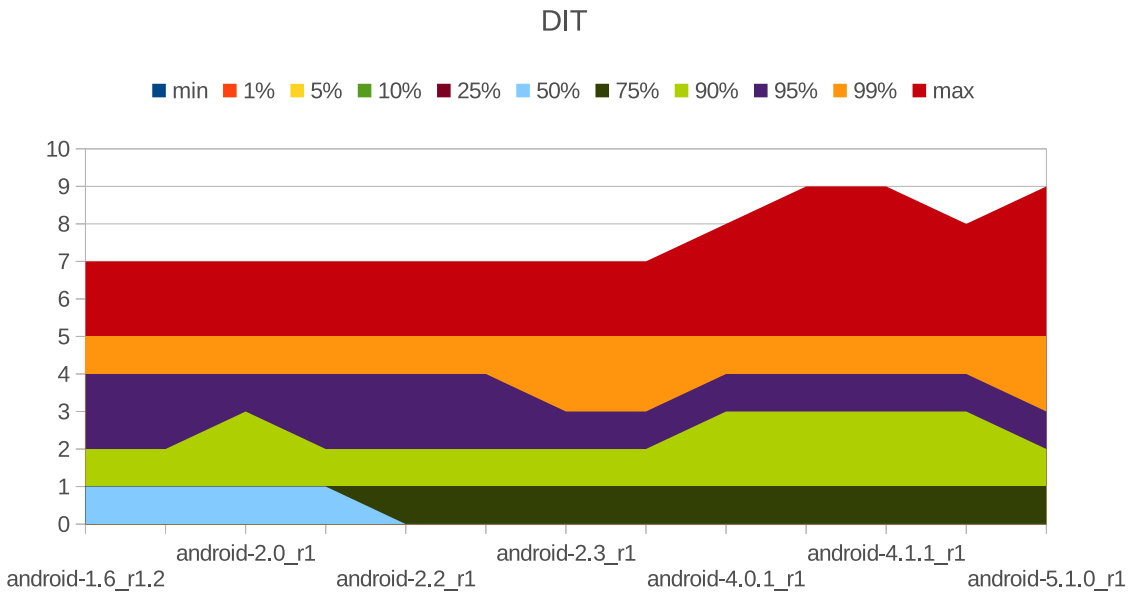


Figura 8 – Evolução da métrica DIT ao longo das versões da API

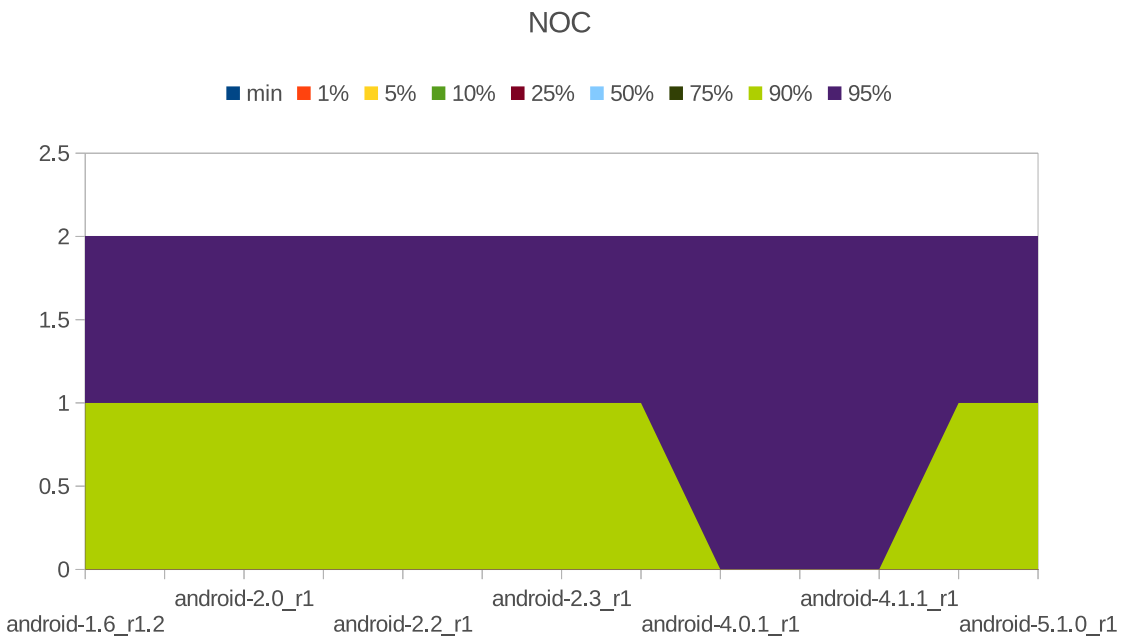


Figura 9 – Evolução da métrica NOC ao longo das versões da API

85% da amostra, vários desses zeros estão presentes também na linguagem Java. O que tiramos disso é que a ferramenta Analizo não contabiliza a classe *Object* na métrica DIT, pois caso contabilizasse o valor mínimo para o Java seria 1, visto que todo objeto Java herda de *Object*. Os valores são em geral baixos, chegando a no máximo 4 até o percentil 95 em todos as versões da API, demonstrando valores bem menores dos que os intervalos definidos por Meirelles (2013) para Java, que chegam até 2, 4 e 6 para os percentis 75, 90

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 5 | 7 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 5 | 7 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 4 | 5 | 7 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 5 | 7 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4 | 5 | 7 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4 | 5 | 7 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 5 | 7 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 5 | 7 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 4 | 5 | 8 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 4 | 5 | 9 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 4 | 5 | 9 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 4 | 5 | 8 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 5 | 9 |

Tabela 8 – DIT no Android

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2.41 | 3 |
| Settings | 722 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 4 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 3 |
| Bluetooth | 239 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| QuickSearchBox | 196 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 5 | 5 |
| Calculator | 10 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Terminal | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| PackageInstaller | 19 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Dialer | 215 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 1 | 1 | 2 | 2 |
| Browser | 259 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 4 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 4 |
| BasicSmsReceiver | 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 4 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3 | 3 |
| Music | 75 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Camera | 253 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 3 |
| Email | 400 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 3 |
| Nfc | 178 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| Gallery | 89 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| ContactsCommon | 292 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 3.1 | 5 |
| Contacts | 265 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 3 |
| DeskClock | 121 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| HTMLViewer | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Calendar | 216 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |
| Exchange | 135 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 |

Tabela 9 – DIT nos aplicativos nativos

e 95, respectivamente. Oliveira (2013) utiliza os mesmos intervalos para excelente, bom, e regular, respectivamente. Ferreira et al. (2009) não define intervalos para essa métrica, mas indica um valor 2 como referência.

A API do sistema se manteve dentro dos intervalos excelente ou bom definidos nesses outros trabalhos citados em todos os percentis analisados nesse trabalho. Os aplicativos se mantiveram dentro do intervalo excelente em todos os percentis, não ultrapassando o valor 2. Em geral, projetos mais simples tendem a fazer menos reúso de código fonte por meio de herança. Oportunidades para uma boa utilização desse recurso de orientação a objetos aparecem com o crescimento do projeto. Dessa forma, é esperado que aplicativos realmente tenham valores menores de DIT e NOC.

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 7 | 110 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 7 | 110 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 7 | 122 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 7 | 124 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 141 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 141 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 147 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 149 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 261 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 262 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 295 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 327 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 398 |

Tabela 10 – NOC no Android

A Tabela 10 apresenta os valores da métrica NOC para o sistema Android. A maioria das classes não tem filhos, tendo 0 como valor muito frequente em todas as versões da API do sistema. Da mesma forma que a métrica DIT, os valores encontrados aqui são relativamente baixos. Meirelles (2013) define para projetos Java o valor 0 como muito frequente, 1 a 2 frequente e 3 pouco frequente. O que encontramos aqui é 0 muito frequênte, 0 a 1 como frequente, e 2 como pouco frequente.

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3.41 | 5 |
| Settings | 722 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 79 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3.4 | 8 |
| Bluetooth | 239 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2.63 | 10 |
| QuickSearchBox | 196 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 3.06 | 4 |
| Calculator | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Terminal | 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PackageInstaller | 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Dialer | 215 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 3 |
| Browser | 259 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2.43 | 4 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 0 | 0 | 0.25 | 1 | 2 | 5.7 | 7 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 8 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 8 | 17 |
| BasicSmsReceiver | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 5 | 17 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3.48 | 7 |
| Music | 75 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2.7 | 10 |
| Camera | 253 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 5 |
| Email | 400 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4.06 | 9 |
| Nfc | 178 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.4 | 1 | 2 | 2 |
| Gallery | 89 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 6.39 | 9 |
| ContactsCommon | 292 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 4.3 | 15 |
| Contacts | 265 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2.37 | 9 |
| DeskClock | 121 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 5 |
| HTMLViewer | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Calendar | 216 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 8 |
| Exchange | 135 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 8.69 | 15 |

Tabela 11 – NOC nos aplicativos nativos

A Tabela 11 mostra que os mesmos valores de NOC discutidos para a API do Android são válidos para os aplicativos nativos da plataforma.

De forma geral, a complexidade da API android com relação à árvore de herança

é relativamente baixa, e isso é refletido nos seus aplicativos, que pelo seu tamanho tem complexidade de herança ainda menor. Os valores de DIT e NOC são muito bons tanto para a API do sistema quanto para os aplicativo desenvolvidos para o mesmo. Vale ressaltar que ambas as métricas são calculadas apenas para linguagem OO, sendo que para C o valor é sempre 0. Como C representa cerca de 2% das amostras, os resultados não são afetados de forma significativa.

Sobre árvore de herança, consideramos os seguintes intervalos para o Android:

- DIT até 1 e NOC igual a 0 são valores muito frequentes em todas as amostras.
- DIT até 2 e NOC igual a 1 são valores frequentes em todas as amostras.
- DIT até 4 e NOC igual a 2 são valores pouco frequentes para a API, mas para seus aplicativos os valores de DIT no percentil 95 permanecem no número 2.

### 3.1.6 Falta de coesão em métodos (LCOM4)

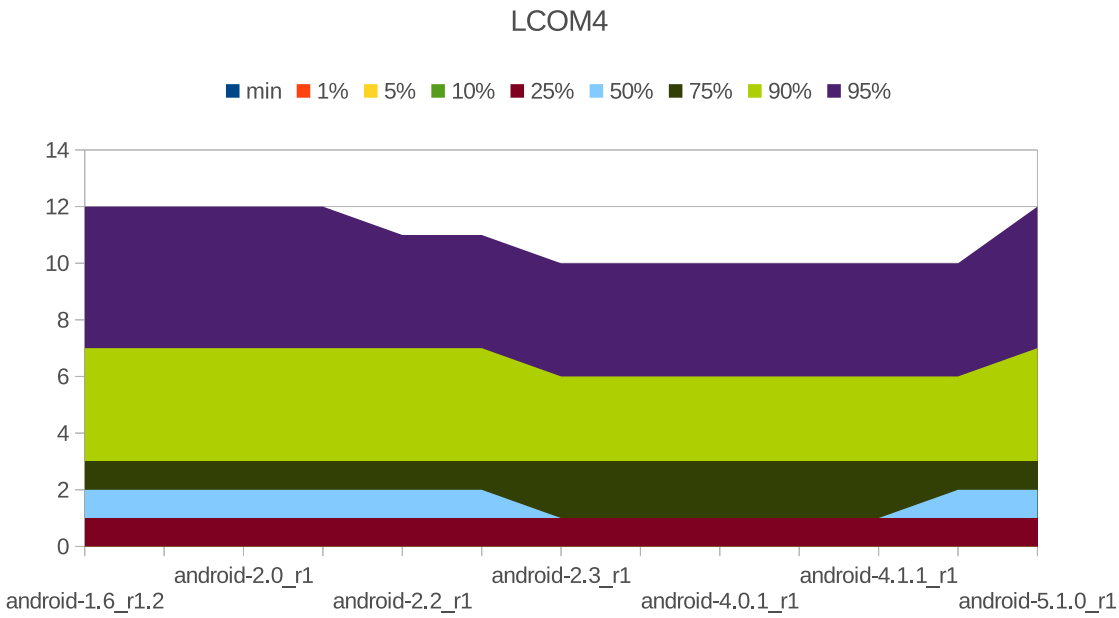


Figura 10 – Evolução da métrica LCOM4 ao longo das versões da API

LCOM4 calcula quantos conjuntos de métodos relacionados existem dentro de uma classe, isto é, métodos que compartilham utilização de algum atributo ou que se referenciam. O valor ideal teórico de LCOM4 é 1, que representa a maior coesão possível, e valores maiores que isso podem indicar que a classe está com muita responsabilidade, tentando alcançar muitos propósitos distintos.

A Figura 10 apresenta a continuidade dos valores para LCOM na API do android, que, mesmo quando variam de uma versão para outra, retornam aos valores antigos,

circulando acerca de um pequeno range de valores. Para o percentil 75, que representa valores muito frequentes, o gráfico demonstra uma linha reta horizontal em verde escuro que não desviou do valor 3 em nenhuma versão da API.

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 12 | 33 | 254 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 12 | 33 | 254 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 12 | 32 | 253 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 12 | 33 | 253 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 11 | 31 | 253 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 11 | 31 | 253 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 6 | 10 | 31 | 253 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 6 | 10 | 31 | 253 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 6 | 10 | 32.92 | 253 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 6 | 10 | 32 | 253 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 6 | 10 | 31 | 437 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 6 | 10 | 31 | 541 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 7 | 12 | 89.16 | 1000 |

Tabela 12 – LCOM4 no Android

A Tabela 12 mostra que em todas as versões da API Android o valor muito frequente é 3. De 3 a 7 são valores frequentes, e de 7 a 12 pouco frequentes. Valores acima de 12 não são frequentes no sistema.

Embora o valor ideal de LCOM4 seja 1, valores maiores que 1 não são totalmente estranhos. Muitas classes são criadas para representar alguma entidade real, e para manter seu valor semântico devem desempenhar alguns papéis distintos ao mesmo tempo. E isso é mais frequentemente visto em projetos que contém muitos dispositivos físicos acessíveis e utilizáveis no sistema.

Em dispositivos móveis, por exemplo, tarefas de tirar foto e capturar vídeo, que são bem distintas, são reunidas na classe câmera, que representa o dispositivo físico que contempla essas funcionalidades. Essa representação de hardware em uma classe específica auxilia a manter uma maior organização no código, e mesmo que sejam tarefas distintas, resultando possivelmente em um maior valor de LCOM4, as classes ainda podem ser consideradas coesas. Fazer a separação da representação de um dispositivo físico em diversas classes pode não ser tão fácil quanto em um projeto com objetos mais "abstratos", que podem ser mais facilmente separados sem prejudicar o entendimento da estrutura do sistema.

Meirelles (2013) define para projetos Java intervalos de 0 a 3, 4 a 7, e 8 a 12 para os percentis 75, 90 e 95, respectivamente. Os resultados encontrados aqui acompanharam muito bem esses intervalos.

LCOM4 também contabiliza classes de modelo, e no caso do Java, os *getters* e *setters* acarretam no aumento do valor do resultado da métrica(MEIRELLES, 2013), entretanto foi verificado, em alguns projetos utilizados como teste, que a ferramenta Analizo não contabilizou esses métodos. A Tabela 13 mostra que os intervalos 0 a 4, 4 a 7, e 7 a

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 1 | 1 | 1 | 2 | 4 | 7.1 | 10.1 | 22.23 | 42 |
| Settings | 722 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 5 | 6 | 10.8 | 44 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 7 | 11 | 22 | 135 |
| Bluetooth | 239 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 8 | 10.15 | 21.52 | 27 |
| QuickSearchBox | 196 | 0 | 0 | 1 | 1 | 1 | 2 | 3 | 6 | 13 | 21.18 | 30 |
| Calculator | 10 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 5.6 | 6.8 | 7.76 | 8 |
| Terminal | 17 | 0 | 0.15 | 0.75 | 1 | 1 | 1 | 3 | 6.5 | 10.75 | 17.35 | 19 |
| PackageInstaller | 19 | 0 | 0.17 | 0.85 | 1 | 1 | 1 | 3 | 4.3 | 5.3 | 6.66 | 7 |
| Dialer | 215 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 6 | 9 | 15.87 | 28 |
| Browser | 259 | 0 | 0 | 1 | 1 | 1 | 2 | 4 | 6 | 9 | 32.87 | 54 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 1 | 2 | 4 | 6 | 8 | 14.95 | 25 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0.2 | 1 | 2 | 3 | 6 | 7.4 | 22.52 | 47 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 6 | 9 | 17.07 | 38 |
| BasicSmsReceiver | 5 | 1 | 1 | 1 | 1 | 1 | 1 | 1.25 | 1.7 | 1.85 | 1.97 | 2 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 6 | 10.5 | 25.6 | 76 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 1 | 2 | 4 | 7 | 11 | 33.84 | 51 |
| Music | 75 | 0 | 0 | 0 | 1 | 1 | 2 | 5 | 15.7 | 19.75 | 45.08 | 48 |
| Camera | 253 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 6 | 13 | 39.86 | 65 |
| Email | 400 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 7 | 13 | 46.02 | 144 |
| Nfc | 178 | 0 | 0 | 0 | 1 | 1 | 1 | 3 | 9.4 | 18 | 25.48 | 29 |
| Gallery | 89 | 0 | 0.87 | 1 | 1 | 1 | 1.5 | 3.25 | 6 | 9 | 17.52 | 21 |
| ContactsCommon | 292 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 6 | 7.5 | 17.1 | 77 |
| Contacts | 265 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 6 | 8 | 15.11 | 21 |
| DeskClock | 121 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 5 | 5.05 | 11.24 | 24 |
| HTMLViewer | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1.5 | 1.8 | 1.9 | 1.98 | 2 |
| Calendar | 216 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 5 | 6.3 | 31.32 | 51 |
| Exchange | 135 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 6 | 12 | 21.35 | 32 |

Tabela 13 – LCOM4 nos aplicativos nativos

12 são válidos para a grande maioria dos aplicativos. Os resultados são muito parecidos com os valores para a API, sendo que a métrica só aumenta em 1 no percentil 75.

### 3.1.7 Conexões aferentes de uma classe (ACC)

| versão | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| android-1.6_r1.2 | 5745 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 12 | 26 | 137 | 1820 |
| android-1.6_r1.5 | 5745 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 12 | 26 | 137 | 1820 |
| android-2.0_r1 | 6331 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 25 | 139 | 1953 |
| android-2.1_r2.1p2 | 6360 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 25 | 139.41 | 1970 |
| android-2.2_r1 | 7352 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 12 | 25.45 | 132.98 | 2027 |
| android-2.2.3_r2 | 7358 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 12 | 26 | 132.86 | 2028 |
| android-2.3_r1 | 8093 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 25 | 114.08 | 2052 |
| android-2.3.7_r1 | 8240 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 25 | 115.22 | 2070 |
| android-4.0.1_r1 | 11709 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 24 | 122 | 2681 |
| android-4.0.4_r2.1 | 11851 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 24 | 121.5 | 2711 |
| android-4.1.1_r1 | 14115 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 24 | 117.86 | 2965 |
| android-4.3.1_r1 | 15472 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 12 | 25 | 121 | 3789 |
| android-5.1.0_r1 | 20129 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 11 | 22 | 99.72 | 4180 |

Tabela 14 – ACC no Android

ACC é um valor parcial de uma das métricas MOOD (*Metrics for Object Oriented Design*) propostas por Abreu e Carapuça (1994). É um o resultado de um cálculo intermediário para calcular o fator de acoplamento (COF).

ACC mede o nível de acoplamento de uma classe através do número de outras classes que fazem referência a ela, por meio da utilização de algum método ou atributo.

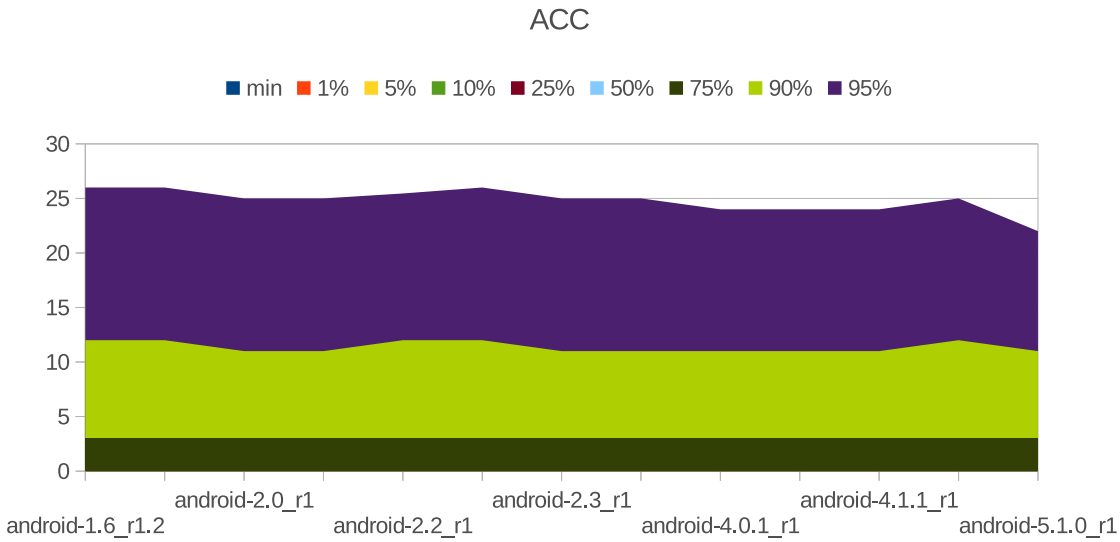


Figura 11 – Evolução da métrica ACC ao longo das versões da API

Apenas as conexões de entrada são contabilizadas, então, diferente de CBO que faz uma contagem bidirecional, ACC só contabiliza a quantidade de classes clientes de uma classe A qualquer, ou seja, que referenciam A, não importando quantas classes A referencia. O ideal teórico é ter valores de ACC tão pequenos quanto possível.

A Figura 11 e a Tabela 14 mostram que os valores da métrica ACC para os percentis 75, 90 e 95 não passam de 3, 12 e 26, respectivamente. Oliveira (2013) define os intervalos 0 a 2, 2 a 7, e 7 a 15 como excelente, bom e regular para os valores dessa métrica. Meirelles (2013) define como referência para Java os intervalos 0 a 1, 1 a 5, e 5 a 12 para os percentis 75, 90 e 95, respectivamente.

Os valores encontrados para o Android estão altos quando comparados com esses estudos, e portanto estes não são bem aplicáveis à API do Android, que, como apresenta a Figura 11, contém valores que não parecem ter tendência clara de reduzir para os intervalos considerados bons. A variância dentre as versões da API aqui analisadas é muito baixa, uma vez que quase os mesmos valores são vistos ao longo das versões. Como os intervalos encontrados são constantes ao longo das versões, refletem o design do sistema e serão utilizados como referência.

A Tabela 15 demonstra que os valores para os aplicativos do sistema também são maiores que os intervalos definidos em outros estudos. Entretanto, eles são mais parecidos com os valores encontrados dentro da própria API, mais uma vez refletindo a semelhança de aplicativos em relação a mesma, e demonstrando um certo padrão para códigos relacionados ao Android. Para a grande maioria dos aplicativos, os percentis 75, 90 e 95 não ultrapassam os valores 4, 12 e 22, respectivamente.

Classes de modelo tendem a ser utilizadas como variável de instância em diversos

| app | classes | min | 1% | 5% | 10% | 25% | 50% | 75% | 90% | 95% | 99% | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Launcher2 | 161 | 0 | 0 | 0 | 0 | 0 | 2 | 5.25 | 17.1 | 26.1 | 92.38 | 110 |
| Settings | 722 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 10 | 26.6 | 307 |
| Camera2 | 462 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 10 | 18 | 43.6 | 122 |
| Bluetooth | 239 | 0 | 0 | 0 | 0 | 0 | 1 | 5 | 16 | 31 | 81.5 | 169 |
| QuickSearchBox | 196 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 9 | 24.56 | 54 |
| Calculator | 10 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2.4 | 3.2 | 3.84 | 4 |
| Terminal | 17 | 0 | 0 | 0 | 0 | 0 | 1 | 2.25 | 5 | 8.25 | 13.65 | 15 |
| PackageInstaller | 19 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 3.3 | 4.3 | 5.66 | 6 |
| Dialer | 215 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 6.7 | 11 | 16.87 | 35 |
| Browser | 259 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 8 | 15.15 | 58.16 | 122 |
| InCallUI | 117 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 6.5 | 10.5 | 25.55 | 93 |
| LegacyCamera | 214 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 11.8 | 16 | 53.4 | 91 |
| Gallery2 | 895 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 11 | 21.35 | 75 | 150 |
| BasicSmsReceiver | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0.25 | 0.7 | 0.85 | 0.97 | 1 |
| UnifiedEmail | 872 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 10 | 16 | 53 | 160 |
| Launcher3 | 354 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 14.8 | 21.8 | 72.24 | 124 |
| Music | 75 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2.7 | 6.7 | 14 |
| Camera | 253 | 0 | 0 | 0 | 0 | 0 | 2 | 5 | 11.9 | 18.25 | 57.13 | 103 |
| Email | 400 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 6 | 18.1 | 40.22 | 201 |
| Nfc | 178 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 11.4 | 18 | 61.48 | 67 |
| Gallery | 89 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 8.3 | 13 | 49.13 | 50 |
| ContactsCommon | 292 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 7 | 15 | 69.4 | 108 |
| Contacts | 265 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 5 | 10 | 14.37 | 30 |
| DeskClock | 121 | 0 | 0 | 0 | 0 | 0 | 1 | 6 | 12 | 20 | 32.91 | 37 |
| HTMLViewer | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.6 | 1.8 | 1.96 | 2 |
| Calendar | 216 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 9 | 18 | 29.44 | 45 |
| Exchange | 135 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 5 | 10.05 | 71.54 | 108 |

Tabela 15 – ACC nos aplicativos nativos

lugares em um aplicativo, e portanto podem ter valores de ACC mais altos. Referências estáticas não são contabilizadas pela ferramenta Analizo para o cálculo de ACC. Em suma, no Analizo só aumenta o ACC em escopo de objeto, não de classe, ou seja, se a classe for utilizada como objeto em algum contexto, tendo alguma variável ou método acessado através desse objeto.

Consideramos então os seguintes intervalos para o Android:

- Para o percentil 75, ACC igual a 3 para a API e 4 para aplicativos;
- 12 é um limite recorrente para o percentil 90 para aplicativos e para a API;
- Para o percentil 95 em diante, aplicativos tem um valor médio menor que a API, estando com limites até 22 e 26, respectivamente.

### 3.1.8 Fator de acoplamento (COF)

COF é uma métrica MOOD proposta por Abreu e Carapuça (1994), e nada mais é que uma relativização do valor de ACC explicado na seção anterior para escopo de software. ACC calcula as conexões que uma classe tem, enquanto COF soma todas essas conexões de todas as classes e divide pelo total de conexões possíveis, resultando e um valor que varia de 0 a 1. O acoplamento ideal para um projeto qualquer é que o valor

| versão | classes | cof |
|---|---|---|
| android-1.6_r1.2 | 5745 | 0.0013695147 |
| android-1.6_r1.5 | 5745 | 0.001369545 |
| android-2.0_r1 | 6331 | 0.0012379467 |
| android-2.1_r2.1p2 | 6360 | 0.0012384764 |
| android-2.2_r1 | 7352 | 0.0010625123 |
| android-2.2.3_r2 | 7358 | 0.0010616483 |
| android-2.3_r1 | 8093 | 0.0009540904 |
| android-2.3.7_r1 | 8240 | 0.000937113 |
| android-4.0.1_r1 | 11709 | 0.0007400012 |
| android-4.0.4_r2.1 | 11851 | 0.0007300137 |
| android-4.1.1_r1 | 14115 | 0.0005968253 |
| android-4.3.1_r1 | 15472 | 0.0005764279 |

Tabela 16 – COF no Android

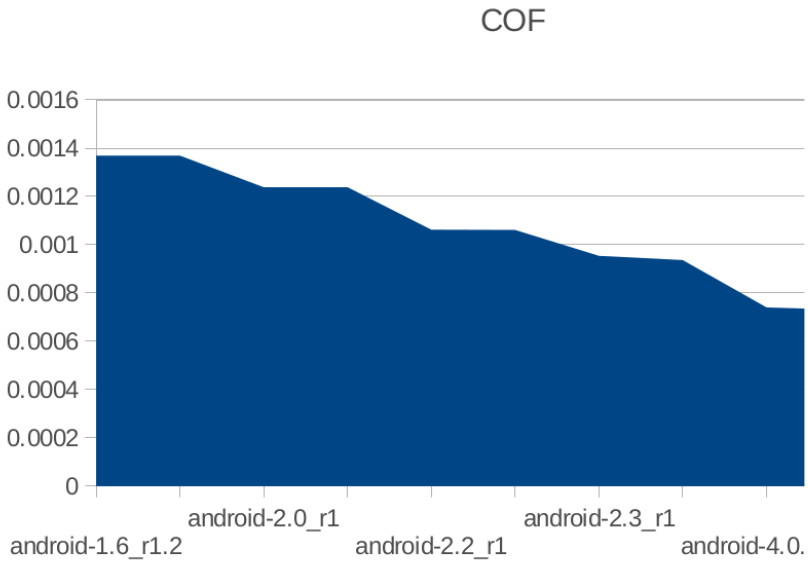


Figura 12 – Evolução da métrica COF ao longo das versões da API

de COF esteja tão próximo de zero quanto possível, indicando que as classes são mais independentes e desacopladas.

O valor de COF para a API do Android ao longo de suas versões caiu de 0.13% para 0.057%, como apresenta a Figura 12 e a Tabela 16. O valor de ACC permanece relativamente constante enquanto o número de classes aumenta significativamente a cada versão, acarretando no aumento de COF. Essencialmente, o sistema como um todo está cada vez mais desacoplado e consequentemente manutenível.

A Tabela 17 apresenta os valores de COF para os aplicativos. COF não é sempre inversamente proporcional a quantidade de classes como já comentado, embora ele realmente tenha a tendencia a diminuir com o aumento do número de classes. Para os aplicativos, observamos um range de 33% para o menor dos projetos, com apenas 4 classes, e 0.4% para um dos maiores aplicativos.

Ferreira et al. (2009) apresenta como intervalos os valores de 0 a 0.02 (2%) como bons , 0.02 a 0.14 (14%) como regulares, e 0,14 em diante como valores ruins. Os valo-

| app | classes | total_cof |
|---|---|---|
| HTMLViewer | 4 | 0.3333333333 |
| BasicSmsReceiver | 5 | 0.0833333333 |
| Calculator | 10 | 0.1666666667 |
| Terminal | 17 | 0.1416666667 |
| PackageInstaller | 19 | 0.0653594771 |
| Music | 75 | 0.0111069974 |
| Gallery | 89 | 0.0432340648 |
| InCallUI | 117 | 0.028035982 |
| DeskClock | 121 | 0.0343837535 |
| Exchange | 135 | 0.0253619122 |
| Launcher2 | 161 | 0.0415487421 |
| Nfc | 178 | 0.0273497689 |
| QuickSearchBox | 196 | 0.0111551679 |
| LegacyCamera | 214 | 0.0230312694 |
| Dialer | 215 | 0.0119125971 |
| Calendar | 216 | 0.014844599 |
| Bluetooth | 239 | 0.0259014998 |
| Camera | 253 | 0.0195566939 |
| Browser | 259 | 0.0161825476 |
| Contacts | 265 | 0.0070284595 |
| ContactsCommon | 292 | 0.0110795118 |
| Launcher3 | 354 | 0.0148564254 |
| Email | 400 | 0.0083122379 |
| Camera2 | 462 | 0.0079222861 |
| Settings | 722 | 0.0040414548 |
| UnifiedEmail | 872 | 0.0044023912 |
| Gallery2 | 895 | 0.0055753048 |

Tabela 17 – COF nos aplicativos nativos

res para o sistema Android estão bem abaixo desses valores, sendo considerados então excelentes valores para o fator de acoplamento. Já os aplicativos se mantém nos valores bons quando passam a ter por volta de 200 classes, mas quando muito pequenos são classificados em sua maioria como regulares nessa escala.

Os intervalos aqui não são considerados em função de seus percentis, mas foram feitas as seguintes observações:

- Espera-se que o valor de COF para o sistema diminua à medida que novas versões surgem;

- Aplicativos desenvolvidos para o Android, quando muito pequenos, até poucas dezenas de classes, tendem a ficar abaixo de 10%, e não devem ultrapassar os 14% no valor de COF. Valores bem menores ainda podem ser obtidos e são preferíveis;

- A medida que ganham tamanho de várias dezenas à poucas centenas de classes, os valores de COF para aplicativos não devem ultrapassar os 4%, entretanto é recomendável que fique abaixo de 2%. Quando chegam a várias centenas de classes, os valores devem ser menores que 1%.

| Métrica | Intervalos | Rótulo |
|---|---|---|
| AMLOC | [0, 14] | Muito Frequente |
| | [14, 31] | Frequente |
| | [31, 55] | Pouco Frequente |
| | [55, ...] | Não Frequente |
| ACCM | [0, 2] | Muito Frequente |
| | [2, 4] | Frequente |
| | [4, 6] | Pouco Frequente |
| | [6, ...] | Não Frequente |
| RFC | [0, 31] | Muito Frequente |
| | [31, 85] | Frequente |
| | [85, 140] | Pouco Frequente |
| | [140, ...] | Não Frequente |
| DIT | [0, 1] | Muito Frequente |
| | [1, 2] | Frequente |
| | [2, 4] | Pouco Frequente |
| | [4, ...] | Não Frequente |
| NOC | 0 | Muito Frequente |
| | 1 | Frequente |
| | 2 | Pouco Frequente |
| | [3, ...] | Não Frequente |
| LCOM4 | [0, 3] | Muito Frequente |
| | [3, 7] | Frequente |
| | [7, 12] | Pouco Frequente |
| | [12, ...] | Não Frequente |
| ACC | [0, 3] | Muito Frequente |
| | [3, 12] | Frequente |
| | [12, 26] | Pouco Frequente |
| | [26, ...] | Não Frequente |
| COF | até 14% | Para poucas dezenas de classes |
| | até 4% | Para poucas centenas de classes |
| | até 2% | Para várias centenas de classes |

Tabela 18 – Intervalos definidos para sistema Android

### 3.1.9 Observações acerca das métricas

A Tabela 18 apresenta um resumo dos intervalos definidos para a API do sistema Android ao longo dessa seção. Embora algumas métricas tenham um indicativo de um problema pontual, nenhuma das métricas discutidas neste trabalho deve ser analisada isoladamente.

Valores baixos de AMLOC são sempre preferíveis, pois métodos mais enxutos tem menor responsabilidade, portanto estão mais sujeitos a reuso, e também são mais fáceis de se ler e se modificar. Entretanto essa métrica é preciso ser analisada em conjunto com outras métricas, como LCOM4 e RFC. Uma classe com muitos métodos privados pequenos tende a ter um valor maior de RFC, o que não implica que esteja mal projetada, desde que os métodos ali presentes estejam bem posicionados segundo o padrão de projeto OO especialista da informação, mantendo por consequência um baixo valor de LCOM4. Como referência geral de resultados para AMLOC, quanto menor o valor, melhor o resultado.

O resultados para a métrica ACCM, que é uma métrica bastante difundida e tem sua aplicabilidade bem clara, relacionados aos AMLOC, dão subsídio para reafirmar que

os valores da métrica AMLOC devem ser os menores possíveis para uma boa arquitetura orientada a objetos.

Entender a relação entre ACCM e AMLOC é importante também para pensar em possibilidade de refatoração de um código fonte alvo. Estão claras as consequências de se ter uma alta complexidade ciclomática, mas essa discussão nos leva rapidamente a ter a ideia de verificar os métodos da classe e avaliar se seu comportamento está tão enxuto como deveria ser, se algum método não está fazendo mais do que propõe. Dividir o comportamento em tarefas menores pode ser uma solução viável. Dividir o comportamento de um método em 2 provavelmente vai acarretar no aumento da métrica RFC. É importante ficar de olho também em métricas como a LCOM4 quando fazendo esse tipo de refatoração, pois às vezes uma tarefa menor que foi extraída de um método não está coesa na classe onde está e muitas vezes até já esteja implementada em uma outra classe que tem responsabilidade mais congruente com essa tarefa. Remover esse tipo de código duplicado ajuda a reduzir a falta de coesão dentro de uma classe, e todas essas melhorias derivaram do simples fato de perceber uma alta complexidade ciclomática em uma classe.

O ideal em uma classe é obter métodos pequenos com tarefas atômicas e bem definidas, que correspondam às responsabilidades dessa classe. A métrica RFC está diretamente relacionada a *Number Of Methods* (NOM), uma vez que um aumento neste último implica necessariamente em um aumento em RFC. Uma classe que tenha alto RFC e muitos métodos (valor alto de NOM) pode indicar que está fazendo mais tarefas do que é sua responsabilidade fazer, necessitando talvez rever a sua implementação para aumentar sua coesão. Da mesma forma, um valor alto de RFC e valor baixo de NOM indica que uma classe está fazendo muito o uso de métodos de terceiros, podendo-se inferir que alguns métodos possam ser extraídos para essas classes que estão sendo tanto chamadas, com o objetivo de diminuir o acoplamento entre essas classes.

Alto valor de profundidade de árvore de herança em uma classe auxilia no aumento da métrica RFC, uma vez que todos os comportamentos são herdados. Entretanto não é uma correlação direta significativa entre as duas, visto que a profundidade de herança raramente é alta para o sistema Android. Embora valor alto de DIT possa significar maior valor de RFC, valores muito altos de RFC tendem a indicar mais a falta de coesão e alto acoplamento, aumentando assim a complexidade estrutural da classe, e não uma profundidade de herança preocupante.

LCOM4 juntamente com RFC são bons indicadores da organização interna de uma classe, pois ambas as métricas dão subsídios para avaliar coesão. Essencialmente, métricas de coesão e acoplamento estão sempre relacionadas e são as métricas mais indicadas para avaliar a complexidade estrutural de uma classe. A ferramenta Analizo, por exemplo, calcula a métrica *Structural Complexity* (SC) como produto entre LCOM4 e CBO.

Dentro do escopo de um projeto, pela própria definição da métrica, o valor limite

para ACC é o próprio número de classes. Embora o valor seja limitado pelo número de classes, não é possível perceber uma relação direta entre o crescimento do número de classes e o valor de ACC, sendo que essa métrica é mais relacionada à forma como o sistema foi pensado e desenhado do que com o seu tamanho. Entretanto, para análise longitudinal em um mesmo projeto, a relação entre número de classes e COF se mostra válida.

## 3.2 Análise dos intervalos de referência

Após a obtenção e análise de distribuição dos dados de métricas OO nas diferentes versões do Android e em aplicativos nativos, foram feitos alguns estudos para validar os intervalos de referência definidos na primeira seção deste capítulo. Também tinham o propósito de utilizar os dados obtidos para auxiliar desenvolvedores a não necessitarem de uma análise manual, assim como na proposta de comparação de aplicativos com a API.

Análise de regressão é um processo estatístico para ver o impacto de variáveis independentes em alguma variável dependente, tentando descrever a relação entre essas variáveis por equações matemáticas. Pensamos na criação um modelo de regressão que conseguisse predizer o valor ideal de uma métrica OO a partir de uma variável independente de tamanho que representasse a escala do projeto, de forma a obter um valor proporcional daquela métrica. O desenvolvedor então poderia comparar esse valor com o valor real da métrica que o código apresenta, e com os intervalos que foram definidos neste trabalho, e então verificar se mudanças estruturais devem ser feitas.

Um modelo de regressão iria aprender os valores ideais para a API com os próprios dados crus das métricas, e então será possível comparar esses resultados com os valores definidos manualmente através dos percentis, discutidos na primeira seção deste capítulo.

### 3.2.1 Análise de regressão

A seguir estão listadas as métricas base que foram inicialmente idealizadas para relativização das métricas por regressão:

1. Número de classes, realizando uma regressão em escopo de software.
2. Número de classes por pacote, resultando em uma regressão em escopo de “módulo”.
3. NOM, realizando uma regressão em escopo de classe.
4. AMLOC, realizando uma regressão em escopo de classe.

Não é possível idealizar esse resultado para métricas OO, mensuradas para cada classe, para entradas mais granularizadas como valores de métodos. A grande maioria das

métricas tem seus valores para a classe como um todo, não podendo fazer distinção sobre a participação de cada método individualmente dentro daquele valor.

Nenhuma das formas propostas foi realmente implementada, mas foi realizada uma discussão e verificação das possibilidades com as métricas que estavam disponíveis. Métricas de tamanho foram o alvo da regressão como variáveis independentes porque seriam uma forma de abstrair os resultados para projetos de diferentes escalas. Essencialmente os intervalos de valores anteriormente definidos seriam validados se a métrica estivesse de acordo com os mesmos para os parâmetros específicos do projeto sendo analisado, fazendo uma comparação entre o modelo de regressão e a análise manual subjetiva deste trabalho.

#### 3.2.1.1 Regressão em escopo de software

A primeira observação que se faz sobre a realização de regressão em escopo de software utilizando os valores da evolução da API do sistema Android é capacidade de generalização de um modelo. De forma geral, o modelo seria criado com pontos com valores de número de classes variando de 5000 a 20000 classes. Como consequência, argumentamos que o modelo de regressão não teria uma boa capacidade de generalização para projetos muito pequenos, como acontece com aplicativos, que possuem muitas vezes apenas algumas dezenas de classes, e são o alvo principal para utilização dessa regressão. Da mesma forma, a capacidade de predição para projetos que contenham bem mais que 20000 classes também seria prejudicada.

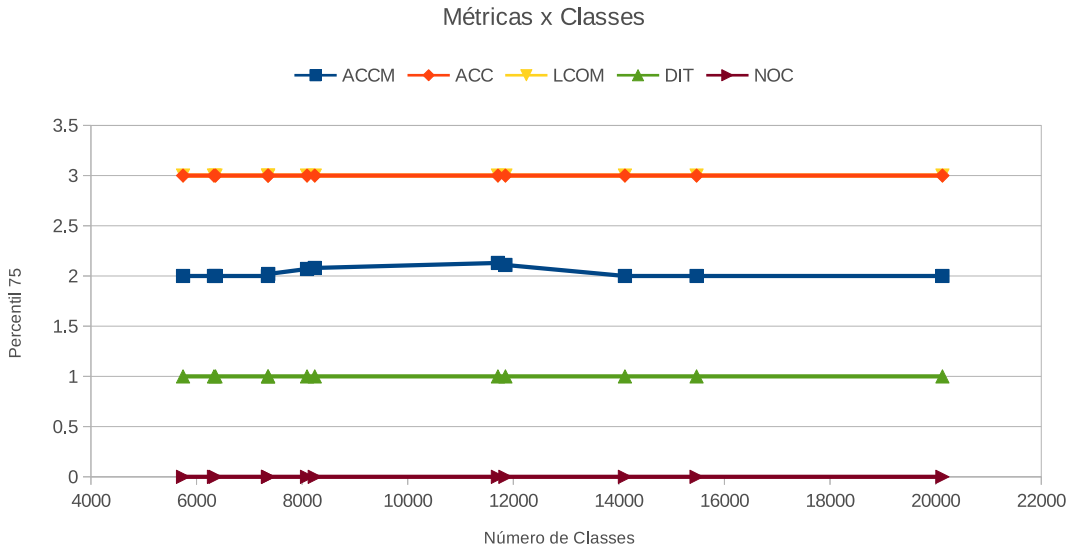


Figura 13 – Percentil 75 das métricas DIT, NOC, LCOM, ACC e ACCM

Para que isso pudesse ser realmente feito, os dados das métricas OO devem ser relacionados com as métricas de tamanho, demonstrando seu valor como dependente do valor dessas métricas. Se os dados obedecem um certo padrão, um funcional pode ser traçado para representar o comportamento dos mesmos.

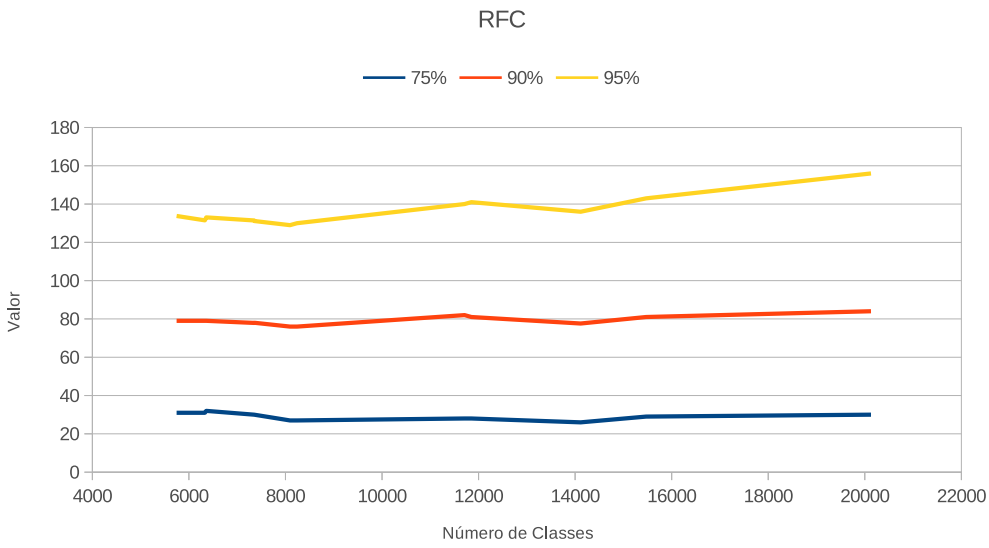


Figura 14 – Percentil 75, 90 e 95 de RFC em função do número de classes

Na Figura 13, onde é traçado o percentil 75 em função do número de classes, vemos que as métricas DIT, NOC, ACC, LCOM4 e ACCM podem ser representadas por uma linha reta horizontal, enquanto RFC, plotada na Figura 14, tem dados mais dispersos mas ainda podem ser representados por uma reta horizontal sem perda significativa nos percentis 75 e 90, que correspondem à grande maioria dos dados. Nas tabelas e gráficos apresentados na seção anterior confirmamos que dentro de cada percentil de cada métrica a variação é pequena. Não foram plotados os valores de outros percentis aqui por diferenciação de escala que exigiria uma grande quantidade de gráficos, além de não terem tanta representatividade quanto os valores muito frequentes apresentados no percentil 75.

Para algumas métricas essa regressão é plausível, isto é, um funcional realmente pode ser traçado que representa a variação dos valores das métricas com o número de classes do projeto. Entretanto, para a ínfima variação nos percentis mais significativos para métricas importantes como ACC, LCOM4, DIT, NOC, RFC, ACCM, que são as principais métricas aqui analisadas, argumentamos que um valor referência se mostra tão útil quanto essa regressão, tornando-a desnecessária.

Além de poder utilizar um valor referência sem perda de valor semântico sobre o valor ideal da métrica, uma regressão em escopo de software da forma como foi proposta, pelo número de classes, não contém, com os dados aqui obtidos, um conjunto suficiente de dados para uma boa regressão.

Os dados das métricas para os aplicativos se mostraram levemente diferentes para os dados da API do sistema, mesmo estando semelhantes, então argumentamos que utilizar apenas a API do sistema como insumo para o modelo de regressão resultaria em um modelo que não funciona tão bem para predizer valores de métricas de aplicativos.

Os dados indicam que um funcional pode ser traçado apenas quando analisando diferentes versões do mesmo software, então, para os aplicativos, que são projetos distintos, os resultados são mais dispersos.

Para realização de um modelo de regressão polinomial a nível de software, não foi encontrada ao longo deste trabalho, dadas as restrições de tempo e escopo, outra métrica que representasse bem o tamanho do projeto para relativização do resultado das métricas OO.

#### 3.2.1.2 Regressão em escopo de classe

Como uma segunda tentativa, foi verificada a possibilidade de utilizar valores das métricas para cada classe, e não para o projeto. Tentamos então avaliar o valor ideal de uma métrica para uma classe, dado alguma variável independente que represente seu tamanho, como AMLOC e NOM.

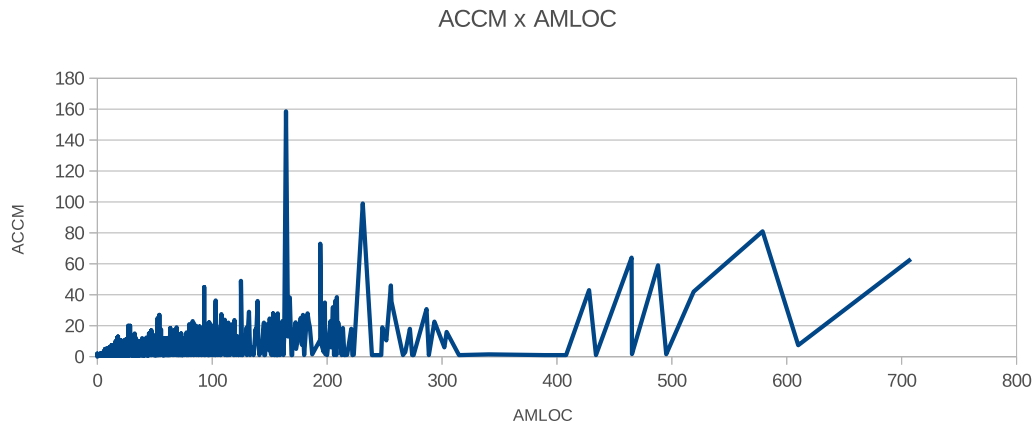


Figura 15 – ACCM em função de AMLOC na API versão 5.1.0

Cada classe dentro de cada versão seria uma amostra para esse método, que teria então um total de mais de 100000 classes. Dessa forma o problema de escala do escopo de software seria resolvido.

O problema nessa abordagem é que as métricas de tamanho que podem ser utilizadas também se mostraram independentes dos valores das métricas OO. Como mostram as Figuras 15 e 16, os valores de ACC e ACCM oscilam bastante. É importante notar que a parte de maior valor do gráfico representa uma quantidade muito pequena de amostras como os próprios percentis já descrevem. A Figura 17 demonstra isso com o fato de que pouco menos de 5% dos dados de maior valor foram retirados (1000 amostras), deixando aproximadamente os dados até o percentil 95, e a escala do gráfico caiu drasticamente. De forma geral, esses pontos de maior valor tem representação estatística muito pequena. Como os percentis representam a grande maioria de valores, iremos utilizá-los para fazer essa comparação, em vez do valor de cada classe.

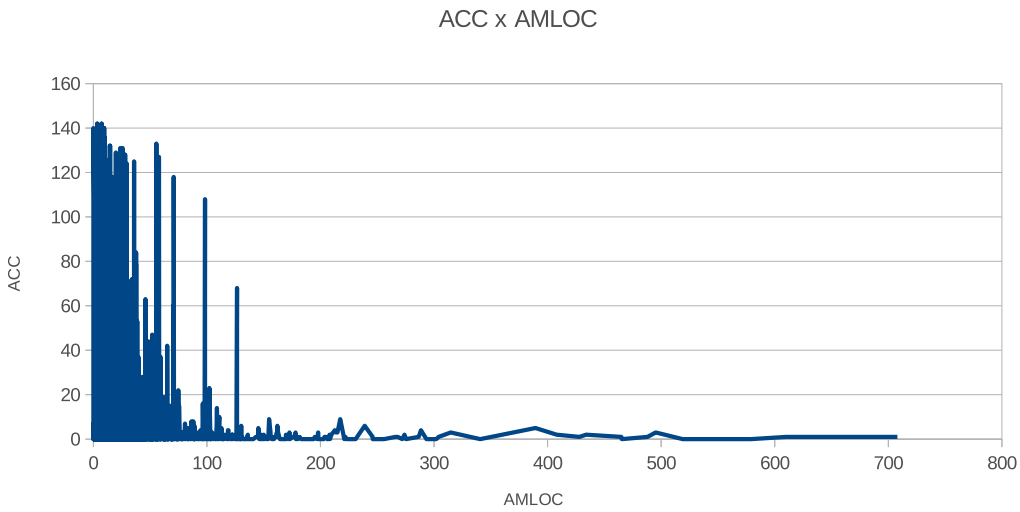


Figura 16 – ACC em função de AMLOC na API versão 5.1.0

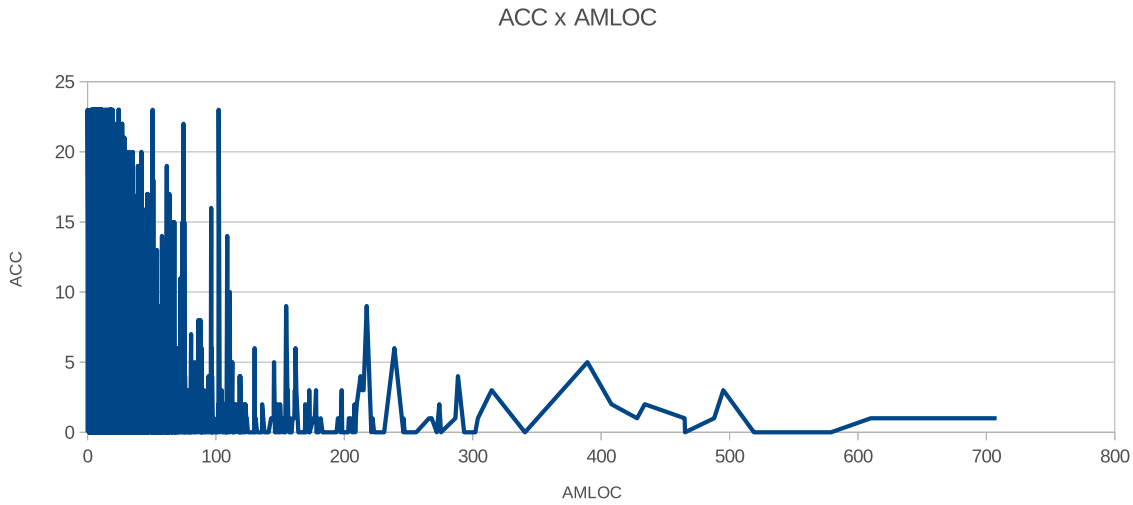


Figura 17 – ACC em função de AMLOC na API versão 5.1.0 com 95% dos dados

Como mostram as Figuras 18 e 19, a variação das métricas OO em seus percentis mais representativos em função de AMLOC ou NOM é mínima e os dados se apresentam na forma de uma linha horizontal. Apenas ACCM tem uma suave relação crescente com AMLOC como já discutido em seção anterior. Na verdade, por essas métricas não variarem muito ao longo das versões da API, esse resultado era previsível.

#### 3.2.1.3 Regressão em escopo de pacote

Para fazer esse teste, foi separado cada pacote presente em cada versão do sistema, e utilizado como métrica de tamanho o número de classes nesse pacote. Os percentis para cada métrica eram calculados individualmente por pacote.

O problema encontrado nessa abordagem foi que as métricas eram dependentes do

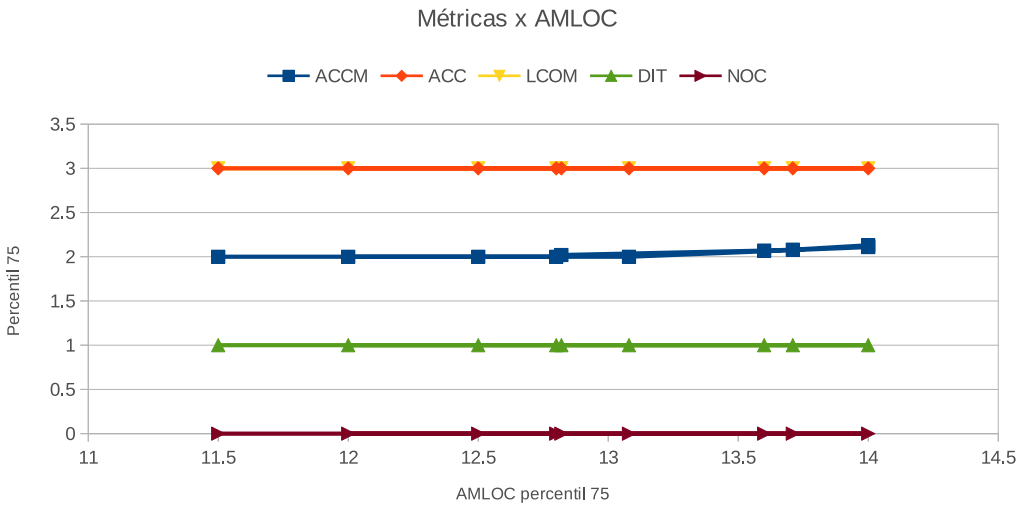


Figura 18 – DIT, NOC, LCOM, ACC e ACCM em função de AMLOC

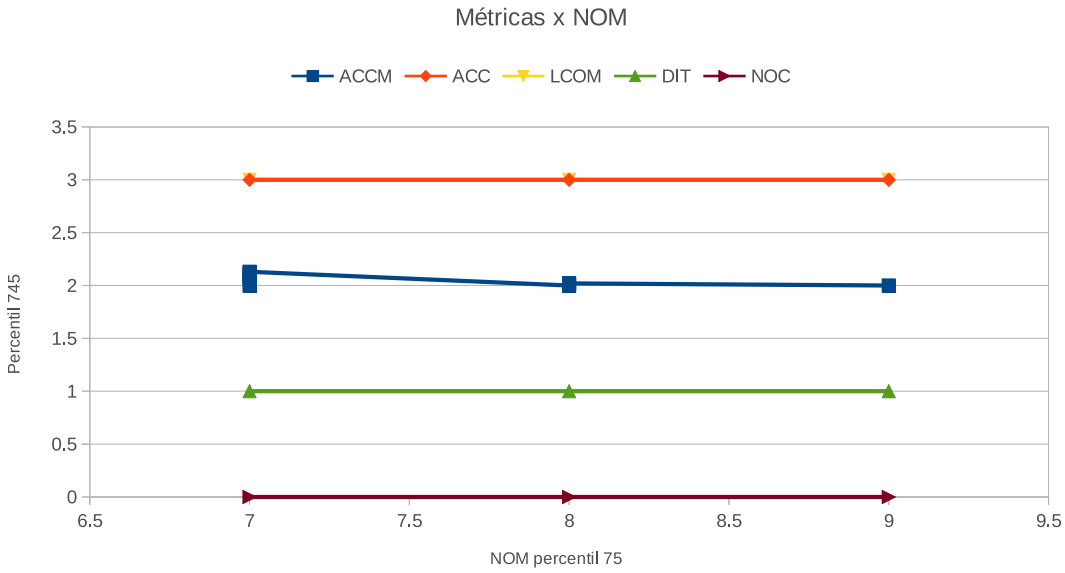


Figura 19 – DIT, NOC, LCOM, ACC e ACCM em função de NOM

próprio pacote analisado e seu propósito, mas independentes de seu tamanho. Isso quer dizer que vários pacotes com mesmo número de classes apresentavam valores com nenhuma relação entre si. Basicamente, foi encontrado o mesmo problema de independência dos dados em relação a variável de tamanho.

O gráfico da Figura 20 demonstra as métricas LCOM, ACC e ACCM para cada pacote em função dos seu número de classes. Os valores tem uma grande variação que aparentemente independe do número de classes do pacote. Pacotes maiores podem parecer variar um pouco menos, mas isso se da pela quantidade menor de amostras em relação a pacotes pequenos.

Emfim, regressão neste contexto de métricas OO e métricas de tamanho não se

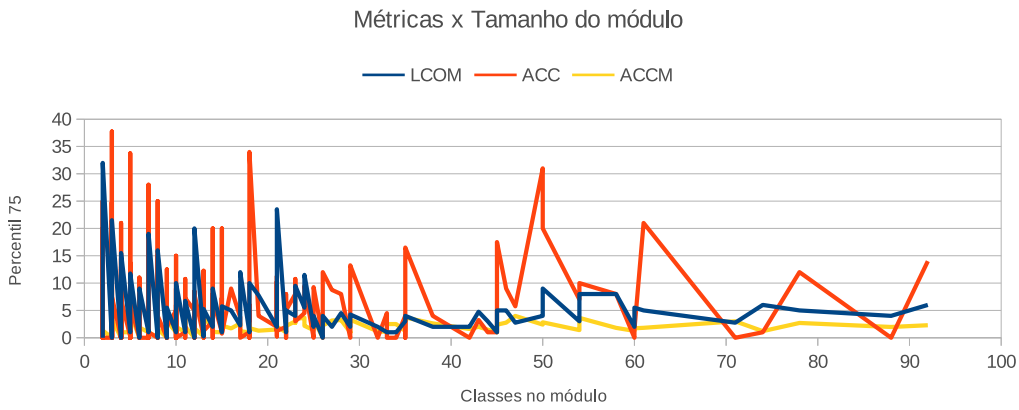


Figura 20 – LCOM, ACC e ACCM em função do tamanho de módulo

mostrou um esforço válido para auxiliar desenvolvimento futuro de aplicativos. Mesmo no caso do escopo de software onde os dados podiam ser representados por um funcional, como a maioria dos valores se mostrou quase invariável ao longo das versões, ou com intervalos fixos, utilizar intervalos de referência tem tanta utilidade quanto uma regressão apresentaria.

## 3.3 Comparação de similaridade com a API

Como foi comentado na seção anterior, as métricas OO não são dependentes de métricas de tamanho. Não foi encontrada uma forma de fazer a predição de um valor ideal de métrica para um determinado projeto sendo analisado.

Entretanto, podemos abstrair um pouco os significados das métricas e avaliar o projeto como um todo fazendo uma comparação com o sistema. Valores para as métricas semelhantes aos do sistema podem ser considerados bons. Então, verificar o valor das métricas de forma relativa ao sistema ajuda a perceber se o valor está bom ou ruim sem necessariamente conhecer quais os valores bons ou ruins nesse contexto de desenvolvimento de aplicativos Android.

Utilizando os valores identificados no início desse capítulo, neste trabalho considerados bons para o contexto Android, será proposto um cálculo de aproximação de aplicativos à API em termos de métricas estáticas de código. Essa comparação não se da de forma direta com o valor de uma classe de um projeto, mas sim em escopo de software. Para isso, assim como na primeira seção, utilizamos os percentis 75, 90 e 95 dos valores de cada métrica dentro do projeto, com o objetivo de levar em conta a distribuição de valores da métrica dentro do sistema, visto que a média e a mediana não são sempre representativas para as métricas utilizadas (MEIRELLES, 2013). A proposta de cálculo de similaridade da API então verifica as diferenças entre cada percentil do app e cada percentil da API, normalizados, e então unifica as diferenças entre as métricas em um só

valor.

### 3.3.1 Normalização de valores

Obter um valor apenas de similaridade para a API não é efetivo se as métricas têm diferentes grandezas. Deve-se fazer um trabalho de normalização dos valores das métricas para deixá-las em mesmo *range*, com uma certa equivalência, para calcular a similaridade de um aplicativo numa ideia semelhante ao que é feito com uma distância euclidiana, onde todas as variáveis tem sua distância unificada em um valor.

Normalizar métricas com valores sem um range específico pode ser bastante problemático. A métrica AMLOC, por exemplo, nada mais é do que a média de valores absolutos de tamanhos de métodos, que podem ser arbitrariamente grandes. Dessa forma, não há um valor limite superior para os valores, embora o valor mínimo seja 0 pela própria definição da métrica. Isso é válido para quase todas as métricas, com exceção de COF, que varia entre 0 e 1. Entretanto COF não tem uma abordagem de distribuição de valores por ser um valor único para o projeto, além de ser derivada de ACC, então não será utilizada no cálculo de similaridade.

Para resolver este problema, extraímos os limites superiores das numerosas amostras obtidas na coleta de dados, com dados de mais de 100 mil classes. Inicialmente considerou-se, para todas as métricas, o maior valor que a métrica apresentou, sem importar a versão onde esse valor apareceu. Como mostra a tabela 14, por exemplo, foi considerado para a métrica ACC um limite superior igual a 4180. Esse limite superior é considerado então como valor 1, e os valores menores como uma porcentagem desse valor, obtido a partir de uma divisão do valor por esse limite superior. Essa mesma forma de normalização pode ser aplicada em todas as métricas.

Entretanto essa abordagem é problemática em algumas métricas. Para DIT, por exemplo, o valor máximo foi 9 para todas as mais de 100 mil amostras. Essa métrica tem uma variação muito pequena dentro dos percentis utilizados (de 1 a 4), fazendo com que a variação de 1 para 4 resulte em uma distância de 0,33. Para a métrica ACCM, que geralmente tem grandezas semelhantes nesses percentis, como apresentam os valores da Tabela 4, variar de 2 à 6 resulta em uma distância de 0,025. Dessa forma a normalização não iria deixar valores equivalentes para a distância de cada métrica, tornando enviesado o cálculo da similaridade total do projeto.

É importante observar que os valores das métricas raramente chegam próximos ao valor máximo, e estes então aparecem em uma quantidade ínfima de classes. Como esses valores então são significativamente maiores que a maioria dos valores dos percentis analisados, as distâncias calculadas para cada métrica, com nada mais que uma subtração, são quase sempre valores muito pequenos. Entretanto, não é importante neste trabalho

obter uma saída normalizada, pois os números finais não têm um valor semântico relativo a sua grandeza, não têm uma medida. Queremos apenas uma comparação entre distâncias de diferentes aplicativos à API, bem como uma diferenciação entre positivo, neutro e negativo, para saber se os valores estão piores, semelhantes ou melhores, respectivamente. Dessa forma, a saída desse cálculo de semelhança será representado na forma de um *score*.

Na verdade 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. Valores de aplicativos que superem esse limite terão sua representação "normalizada" sendo maiores que 1, mas isso não tem impacto no resultado, pois, como já comentamos, a grandeza não tem significado. Um problema dessa utilização é que a métrica NOC contém um valor 0 no percentil 75, e então em sua normalização ocorreria divisões por 0. Entretanto, como já estamos utilizando uma métrica que reflete complexidade da árvore de herança, a métrica DIT, removemos NOC desse cálculo sem muitas perdas.

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 a métrica ACCM, referência para percentil 95, seria "normalizado" para 3 com esse cálculo, sendo interpretado então como 3 vezes o valor muito frequente da API. Da mesma forma, o valor 4 de DIT, que é a referência do Android para o percentil 95, seria 4. As grandezas estão mais equivalentes quando calculadas dessa forma.

### 3.3.2 Cálculo de similaridade

Inicialmente pensou-se em calcular distâncias de forma modular, isto é, uma mesmo valor para uma variação positiva ou negativa. Optou-se por não o fazer porque aplicativos com valores teóricos melhores que os do sistema eram "mascarados" por estarem distantes da API e erroneamente considerados como aplicativos de qualidade ruim. Calculando similaridade como sendo positiva ou negativa, 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 *app* 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, tentou-se utilizar pesos em relação a importância das métricas, semelhante ao realizado em Oliveira (2013) para a configuração do Kalibro.

Assim, métricas mais importantes teriam mais impacto para informar que um valor está melhor ou pior em relação a API. Assim como no trabalho citado, manteve-se equivalência de pesos entre métricas de complexidade e tamanho, e métrica de acoplamento e coesão são evidenciadas com peso maior por terem mais impacto na qualidade do software. Essencialmente, quase todas as métricas trabalham com complexidade, acoplamento ou coesão, então apenas métricas que trabalham com arvore de herança ficaram com peso 1, e o restante tem peso 2. AMLOC ganhou peso 2 para criar equivalência com ACCM.

Os valores para comparação com o sistema são os valores indicados na Tabela 18. Para ACCM, por exemplo, os valores de comparação são 2, 4 e 6, para os percentis 75, 90 e 95, respectivamente.

É importante notar que os percentis têm representatividade diferente. O intervalo que representa os valores muito frequentes contém 75% das amostras, enquanto os intervalos frequentes, representado pelo percentil 90, tem 15% das amostras, e 5% para o intervalo de valores pouco frequentes, no percentil 95. Assim, para cada diferenciação em cada percentil, foi multiplicado um peso relacionado a representatividade 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:

$$\frac{\sum_i^n \frac{(Mapi_i - Mapp_i)}{Mapi_{75}} \cdot W_i}{95} \tag{3.1}$$

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:

$$\frac{\sum_i^n d_i \cdot W_i}{\sum_i^n W_i} \tag{3.2}$$

Onde:

- $i$ varia entre as métricas;
- $d_i$ é a distância da métrica i calculada na Equação 3.1;
- $W_i$ é o peso da métrica i;
- $\sum_i^n W_i$ é a soma dos pesos das métricas.

### 3.3.3 Resultados

| | |
|---|---|
| HTMLViewer | -85 |
| BasicSmsReceiver | -71 |
| QuickSearchBox | -55 |
| Calculator | -51 |
| ContactsCommon | -40 |
| PackageInstaller | -37 |
| Contacts | -36 |
| Dialer | -34 |
| Email | -31 |
| Camera2 | -30 |
| UnifiedEmail | -28 |
| InCallUI | -23 |
| Terminal | -21 |
| Music | -19 |
| Settings | -17 |
| Browser | -14 |
| Gallery | -11 |
| Gallery2 | -10 |
| Exchange | -9 |
| Launcher3 | -3 |
| Camera | 0 |
| LegacyCamera | 11 |
| Calendar | 12 |
| Launcher2 | 12 |
| DeskClock | 19 |
| Nfc | 24 |
| Bluetooth | 74 |

Tabela 19 – *Scores* de similaridade

O resultado então foi multiplicado por 100 para levar os valores para uma grandeza maior e então truncamos as casas decimais, com o intuito de obter melhor visualização. A Tabela 19 apresenta os scores calculados com a Equação 3.2 para os aplicativos do sistema. O *app Bluetooth* ficou com o maior valor dentre os aplicativos, e portanto representa o

mais discrepante do sistema e o que contém, no geral, os piores valores para as métricas analisadas. Como observado nas tabelas da primeira seção deste capítulo, ele realmente tinha valores em intervalos ruins para quase todas as métricas. Já o *app Settings*, que possui bons valores em quase todas as métricas, com exceção de DIT, ficou negativo, e mais próximo do sistema que o *app Bluetooth*, como os valores anteriormente apresentados já indicavam. O *app Camera* ficou com score 0, bem próximo ao sistema, e como podemos ver nas tabelas da primeira seção, ele se manteve bem próximo aos intervalos na maioria das métricas, com pequenas variações, positivas ou negativas. O *app Calculator*, que contém um dos menores valores, apresenta valores teóricos em geral melhores que os da API, então sua posição com um número negativo maior que os demais é justificada, assim como o *HTMLViewer*, que é um projeto mais simples e também contém valores teóricos melhores que os do sistema.

No geral a grande maioria dos aplicativos ficou melhor que o sistema Android, sendo que a média dos valores de similaridade para os aplicativos do sistema ficou em -17,5, com desvio padrão de 31,5. Os valores dos aplicativos do sistema estão então no intervalo -48 a 14, indicando que no geral os aplicativos são próximos do sistema, como já verificado na primeira seção deste capítulo, porém esses valores podem implicar que aplicativos tendem a ter métricas com valores levemente melhores.

Apesar de algumas observações serem levantadas sobre os resultados apresentados, como números com grandezas não significativas e valores bons mascararem alguns ruins e vice versa, pelos resultados apresentados na Tabela 19, verificamos que o objetivo proposto para essa verificação de similaridade foi alcançado. Assim como se planejava, *scores* negativos se mostram melhores que o sistema, *scores* próximos a 0, com valor perto de -20 a 20 são bem próximos aos valores da API, e *scores* positivos são preocupantes. De certa forma, o cálculo de similaridade proposto pode ser utilizado como um indicador de problemas arquiteturais caso o valor seja positivo, para ser utilizado por desenvolvedores inexperientes e sem conhecimento direto de métricas de código fonte. Esse indicador de qualidade de código em comparação com a API parte da premissa de que a API tem uma boa qualidade de código, como verificado no início desse capítulo com intervalos de valores de métricas, e portanto terá sua validade enquanto essa premissa for verdadeira.

# 4 Exemplo de Uso

## 4.1 E-Lastic

O E-lastic é um sistema eletrônico que monitora e controla a execução de exercícios físicos realizados com equipamento que impõe sobrecarga à movimentação de segmentos corporais por meio de resistência elástica. Esse produto é resultado do trabalho de mestrado em processamento de sinais da aluna Fernanda Sampaio Teles, que resultou num pedido de patente com número de registro BR 5120130007631.

O produto em desenvolvimento apresenta um aparelho portátil, voltado para o controle de atividades físicas em ambientes fechados ou abertos. Trata-se de um sistema eletrônico embarcado que realiza o processamento digital do sinal originado num sensor de força e associa essas informações com variáveis de espaço e tempo, de forma a gerar informações suficientes para o controle e prescrição de exercícios resistidos. Esse sistema eletrônico permite a acoplagem do implemento elástico para a realização do exercício a ser monitorado, e interfaceia com o usuário por meio de um aplicativo. De forma simplificada, durante o exercício físico, a força aplicada pelo usuário ao elemento elástico é calculada no microcontrolador e enviada juntamente com as demais informações via *Bluetooth* para um dispositivo móvel com o e-lastic *app*, que contém opções de controle para a realização do exercício físico.

O aplicativo foi desenvolvido sem o monitoramento de nenhuma métrica, e seu código está disponível em um repositório aberto[1] para qualquer desenvolvedor através da plataforma gitlab. Não serão explorados aqui detalhes das funcionalidades do aplicativo e-lastic, apenas como foram pensados alguns de seus componentes internos.

## 4.2 Estado da Arquitetura

O desenvolvimento foi focado na base da arquitetura, juntamente com a funcionalidade básica do aplicativo. Nesta seção, serão chamados de componentes instanciações dos componentes Android, e de módulos conjunto mais complexo de classes que envolvem um ou mais componentes. Na implementação atual do aplicativo, existem os seguintes componentes e módulos principais:

- *BluetoothService* - Responsável pela conexão *bluetooth*, este componente é um *Android Service*[2] que se inicia juntamente com a aplicação e fica à espera de uma so-

[1] <http://gitlab.com/biodyn/biodynapp>

[2] *Services* são tarefas que são executadas em background no Android, sem interação com o usuário

licitação de conexão. Quando o usuário solicita a conexão com o hardware e-lastic, o aplicativo identifica o dispositivo a ser conectado e informa ao *BluetoothService*, que a partir daí é responsável por conectar e manter a conexão, recebendo os dados do microcontrolador presente no hardware e-lastic.

- *ExerciseService* - Este módulo reúne a maior parte da lógica de negócio. Ele gerencia qual é o exercício ativo e trata da execução do mesmo, controlando informações de entrada vindas do microcontrolador, bem como mudanças nos exercícios solicitadas pelo usuário.

- *BioFeedbackService* - Este módulo é responsável por gerar *biofeedback* para o usuário. Quando o valor de força aplicado é superior ao limite máximo, por exemplo, o exercício informa a esse componente que algum tipo de *feedback* deve ser acionado para informar ao usuário que o limite foi ultrapassado.

- *ExerciseActivity*[3] - Representa a tela principal e a própria interação do usuário com os componentes gráficos. Neste trabalho, apenas alguns componentes gráficos temporários estão disponíveis para demonstrar a funcionalidade básica do aplicativo em execução.

A comunicação de todos os componentes e módulos é feita de forma indireta por meio de *intents*. O *intent* tem a função de comunicar componentes independente da aplicação a qual pertencem. Entretanto, neste aplicativo em específico, essa comunicação deve ser feita apenas entre os componentes internos da aplicação, não sendo necessário que esses *intents* entre os componentes internos sejam enviados pelo sistema Android para outros componentes de outras aplicações. Para esse fim, a implementação foi feita utilizado um recurso da API Android chamado *LocalBroadcastManager*, que é responsável por realizar o envio de *broadcast intents* apenas para os componentes internos da aplicação, evitando que os mesmos sejam recebidos em outras aplicações. De forma análoga, os receptores desses *intents* são registrados não diretamente no sistema, mas dentro dessa instância de *LocalBroadcastManager* que é individual da aplicação, registrando-se apenas para receber *intents* enviados dentro da própria aplicação. O uso desse recurso evita a necessidade de criar permissões específicas para cada tipo de *intent* que será utilizado entro da aplicação, e não há perigo de que a aplicação receba *intents* forjados por alguma outra aplicação maliciosa ou mesmo que alguma aplicação maliciosa receba dados que são privados deste aplicativo.

Toda essa comunicação indireta foi reunida em uma classe responsável por gerenciar essas comunicações, a classe *Communicator*. Essa classe reúne todas as *actions*

[3] *Activity* é um componente Android que representa uma interface gráfica(*Graphic User Interface*) para a realização de uma tarefa específica

de todos os *intents* que as classes estão preparadas para receber. Basicamente, para saber que informações o *ExerciseService* espera, por exemplo, basta checar as constantes na classe *Communicator.ExerciseServiceActions*. Embora essa classe pareça ter um alto acoplamento devido a sua responsabilidade de comunicar todos os componentes, esse acoplamento é reduzido devido a comunicação indireta que é feita com o uso de *intents*. Basicamente, a classe *Communicator* tem a responsabilidade de enviar mensagens para todos os componentes sem realmente os conhecer. Se um componente for alterado, não há impacto algum dentro dessa classe.

Essa comunicação indireta que é implementada por meio dos *intents* tem a vantagem de deixar os componentes com uma conexão bastante fraca. Quando um componente envia uma mensagem, ele não sabe ao certo quem vai receber, portanto não precisa conhecer o receptor. As mensagens são enviadas via *broadcast*, para todos os componentes, e cada um recebe o que desejar receber. Da mesma forma, os receptores dessas mensagens não sabem quem as mandou, e portanto não há uma associação direta em nenhum dos lados da comunicação. Embora pareça deixar a comunicação confusa e embaralhada, isso permite que troquemos um componente inteiro do projeto com pequeno ou nenhum impacto nos demais componentes. A interface gráfica, por exemplo, recebe mensagens do componente de exercício informando sobre atualizações nos dados. Ela não conhece o exercício em execução e o exercício em execução não conhece quem está mostrando seus dados. Se a interface gráfica fosse inteiramente excluída da aplicação, com pouquíssimos ajustes, relacionados principalmente as mudanças no próprio *AndroidManifest.xml*, seria possível deixar várias outras funcionalidades ainda em funcionamento. Com esse acoplamento reduzido entre os componentes da aplicação, se torna muito mais fácil a manutenção e refatoração desses componentes isoladamente, ou mesmo a substituição completa dos mesmos. Trocar o módulo *bluetooth* por um módulo *wifi*, por exemplo, que recebe esses dados por *socket* de rede em vez de *bluetooth*, não traria quase nenhum impacto a aplicação já em produção, uma vez que esse novo módulo poderia simplesmente enviar mensagens da mesma forma que o *bluetooth* enviava utilizando a classe *Communicator*. Como já explicitado, para os demais módulos não importa quem envie os dados, desde que eles cheguem corretamente.

A única exceção para essa comunicação indireta é a própria inicialização dos componentes. Todos os serviços precisam ser iniciados em algum lugar na aplicação, e o lugar óbvio para o fazer é no início da aplicação. Não tem porque iniciar os componentes enquanto o usuário não abrir a parte gráfica da aplicação, e por esse motivo os serviços são inicializados assim que a *Activity* principal é criada, já que ela é o ponto de entrada da aplicação e-lastic. Muitos componentes podem ser o ponto de entrada em um aplicativo Android, mas neste caso específico a melhor escolha é a própria interface gráfica, representada pela *Activity*. Desse modo a *Activity* conhece os serviços a serem inicializados, porém não conhece seu comportamento e nem mesmo sua responsabilidade.

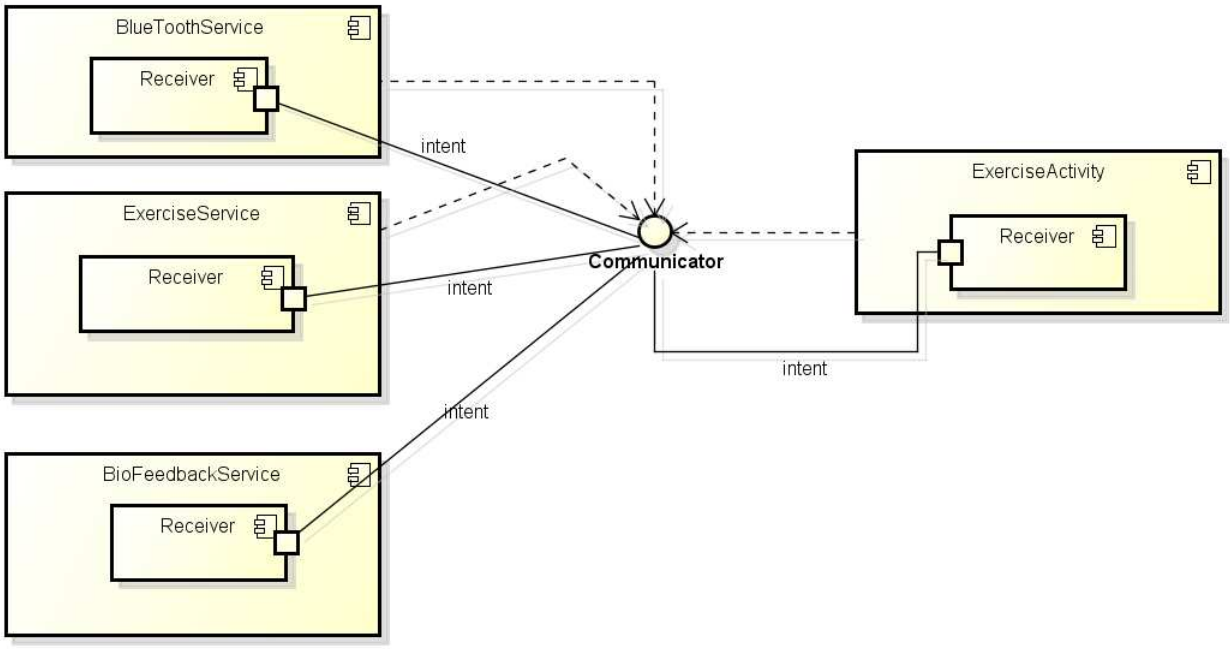


Figura 21 – Componentes e sua comunicação via classe *Communicator*

Essa comunicação por meio do *LocalBroadcastManager* funciona de maneira semelhante ao que vemos no padrão observador de orientação a objetos. Todos os "eventos" ou "mensagens" gerados em um componente/módulo são informados ao objeto exclusivo ao contexto do aplicativo, que por sua vez os entrega a quem se inscreveu para obtê-los. Uma diferença básica em relação a essa comparação é que no padrão observador, a interface gráfica geralmente conhece a classe de modelo na qual se inscreve para receber atualizações, e portanto existe uma relação unilateral fraca entre os objetos. Da forma implementada neste trabalho, cada objeto apenas conhece as mensagens que quer receber e como elas são formatadas mas não o componente que as envia, deixando as classes mais desacopladas. Entretanto, o resultado de uma análise de fator de acoplamento pode indicar um valor não tão pequeno devido a utilização da classe *Communicator* pelos demais componentes. Nessa arquitetura, é uma classe com alto acoplamento unilateral, pois, embora ela não conheça os componentes que receberão as mensagens, a maioria dos componentes a conhecem e a utilizam como "mensageiro".

A Figura21 é um diagrama de componentes que representa de forma gráfica a ideia básica da comunicação entre os componentes da aplicação e-lastic.

Embora representado no diagrama de componentes como uma caixa apenas (*ExerciseService*), o módulo de exercício contém a lógica da comunicação do componente via *Communicator* e o gerenciamento do exercício em execução. É utilizada a generalização para que o *service* tenha o mesmo comportamento independente do exercício em execução, e portanto as manipulações do exercício são feitas em cima de um objeto do tipo *Exercise*, que é uma classe abstrata. A Figura 22 contém um diagrama de classes para demonstrar

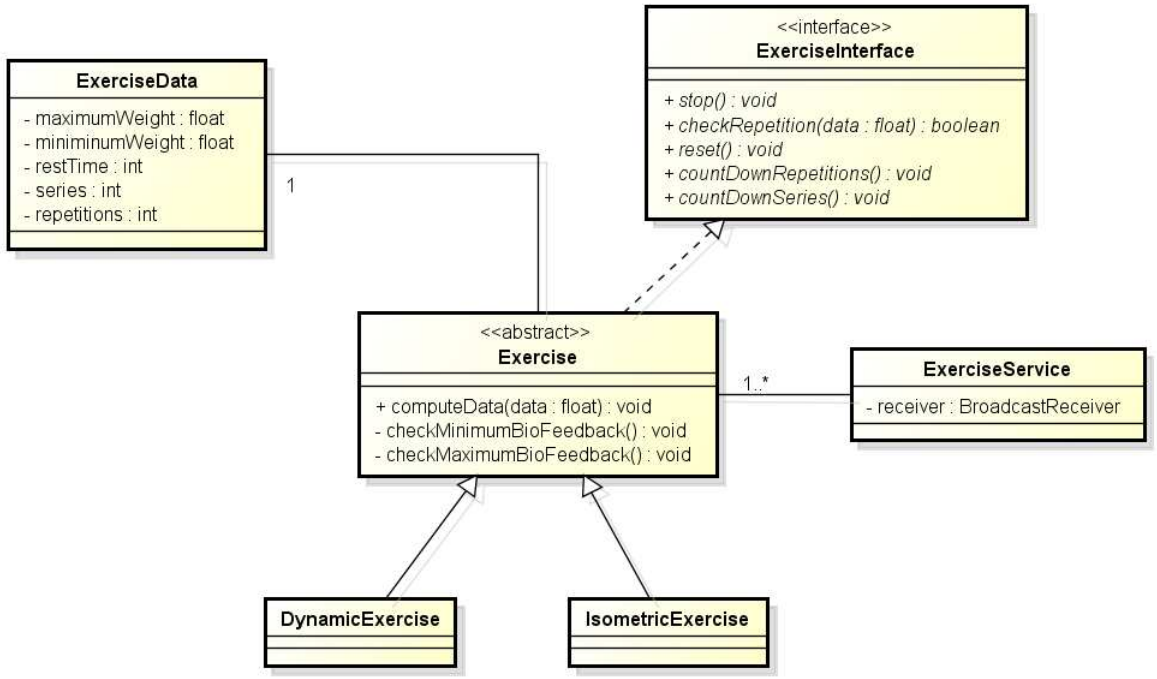


Figura 22 – Principais classes dentro do módulo de exercício e suas relações

a estrutura básica desse módulo.

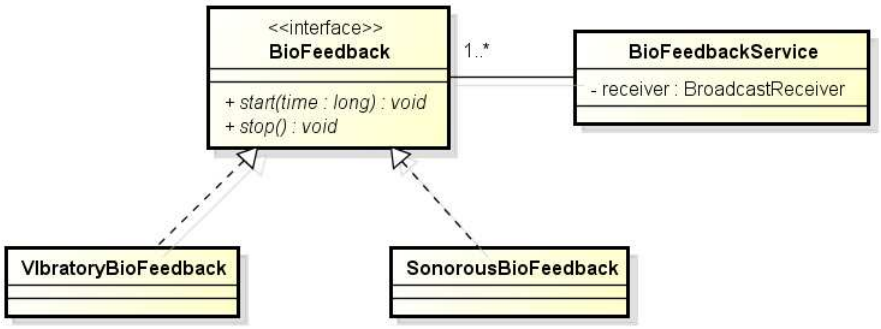


Figura 23 – Principais classes dentro do módulo de *biofeedback* e suas relações

As classes internas do módulo de *biofeedback* podem ser vistas na Figura 23. Também é utilizada uma generalização dos tipos de *biofeedback* com a interface *BioFeedback*, que deve ser implementada para criar um novo tipo de *biofeedback*. Da mesma forma que o módulo de exercício, aqui também existe um *service* que fica em segundo plano a espera de uma requisição de execução de *biofeedback*. Em suma, ele fica em *standby* até que algum pedaço da aplicação solicite que ele ative algum dos seus tipos de *feedback*. Reunir essa função em um módulo específico da aplicação faz com que se possa ter controle de quais tipos de *biofeedback* podem ser utilizados, e eles tem um acesso padrão independente do

tipo, com a utilização do *Communicator* e *flags* que identificam que tipos de *biofeedback* ativar. Tocar um som e vibrar o celular são feitos de formas completamente diferentes, e só a própria classe precisa saber como o fazer, sendo que o *service* só conhece a interface de início e término e para ela não importa sua implementação. Com a implementação dessa forma, basta uma adição de uma linha de código no mapa de *biofeedbacks* e a adição de uma *flag*, além da implementação do novo *biofeedback*, para que um novo tipo de *biofeedback* esteja pronto para ser utilizado.

## 4.3 Cálculo de distância aplicado ao E-lastic

A arquitetura básica do e-lastic foi submetida a uma coleta de métricas assim como os demais aplicativos do sistema, como apresentado no Capítulo 3. O Score proposto neste trabalho foi aplicado na base arquitetural do aplicativo e-lastic com base nessas métricas coletadas.

| métrica | 75% | 90% | 95% |
|---|---|---|---|
| accm | 1.8 | 2.33 | 3.63 |
| amloc | 9.97 | 20.4 | 23.27 |
| nom | 3 | 5.9 | 10 |
| noc | 0 | 0 | 0.95 |
| dit | 1 | 1 | 1 |
| lcom4 | 2 | 3 | 3.95 |
| acc | 0 | 2.9 | 4 |
| rfc | 10.75 | 18.8 | 50.9 |

Tabela 20 – Percentis 75, 90 e 95 para as métricas analisadas no aplicativo e-lastic

Tomando como referência os intervalos aqui definidos e reunidos na Tabela 18, o aplicativo e-lastic apresenta valores teóricos melhores que os do sistema em todas as métricas, como mostra a Tabela 20. O cálculo resultou em um score de -82, menor que a grande maioria dos aplicativos do sistema, refletindo os ótimos resultados que a Tabela 20 apresenta. Em suma, o resultado da análise de métricas classifica o aplicativo e-lastic como tendo uma boa qualidade de código.

Os resultados encontrados afirmam que a hipótese 4 (H4) definida no Capítulo 2 não pode ser negada, isto é, as decisões tomadas ao longo do desenvolvimento do aplicativo com base em padrões de projeto estão de fato relacionadas às decisões tomadas com base em métricas, uma vez que decisões tomadas com este propósito tentariam levar os valores para os menores possíveis, tentando alcançar valores em intervalos ótimos, como os que foram encontrados no aplicativo parcial desenvolvido.

# 5 Conclusão

Este trabalho teve como objetivo principal o monitoramento de métricas estáticas de código fonte, essencialmente métricas OO, e fazer um estudo da evolução de seus valores nas diferentes versões da API do sistema operacional Android, estudar as semelhanças com aplicativos do sistema e então verificar a possibilidade de utilizar os dados obtidos para auxiliar no desenvolvimento de aplicativos para o Android. Esses objetivos foram alvejados na análise exploratória de dados de métricas realizada neste trabalho na API Android.

Foram conceituadas e aplicadas várias métricas na API de desenvolvimento de aplicativos Android, a fim de avaliar a qualidade do código fonte da API em relação aos valores das métricas e sua distribuição dentro do sistema. Foi realizado um estudo da própria plataforma Android e sua arquitetura para auxiliar na interpretação de valores de métricas aplicados neste contexto.

Foi verificado neste trabalho que a API e aplicativos desenvolvidos para o Android tem muitas semelhanças no que diz respeito a métricas OO, o que reforça a afirmação de alguns trabalhos aqui citados no sentido de existir um alto acoplamento entre eles. Os valores de métricas encontrados para os aplicativos estão muito semelhantes com os do sistema, refletindo um estilo arquitetural intrínseco da plataforma.

O cálculo de similaridade proposto tinha o objetivo principal de verificar se um aplicativo se aproximava da API Android, com o intuito de avaliar sua qualidade com esse referencial. O resultado mostrou que *scores* negativos se mostram valores melhores que os da API, e *scores* positivos são preocupantes. Assim como o objetivo propôs, o *score* calculado pode ser utilizado como um indicador de problemas arquiteturais, caso supere o valor 0, que são os valores da API. Esse indicador de qualidade de código em comparação com a API parte da premissa de que a API tem uma boa arquitetura, que foi confirmada neste trabalho, e portanto terá sua validade enquanto essa premissa for verdadeira.

Em relação à questão de pesquisa e às hipóteses levantadas no Capítulo 2, temos as seguintes observações:

- *H1 - É possível identificar padrões e tendências na evolução da arquitetura do sistema Android e nos aplicativos desenvolvidos para ele.*

Temos a resposta de H1 na análise da evolução das métricas do código fonte contida no Capítulo 3, onde foi verificado que é possível identificar um padrão de comportamento nos valores das métricas, que indica uma permanência dos valores de métricas OO em

certos intervalos, independentemente da versão ou de seu tamanho. Da mesma forma, os aplicativos parecem obedecer um mesmo padrão que foi identificado para a API, acompanhando os valores na grande maioria dos casos.

- *H2 - O desenvolvimento de aplicativos Android pode ser guiado pelo resultado de uma análise evolutiva do código do próprio sistema.*

H2 é uma hipótese que também é avaliada a partir do acoplamento entre a API e seus aplicativos. A validade dos mesmos intervalos de valores para ambos os casos implica na utilização dos mesmos como referência no desenvolvimento de aplicativos. A proposta de *score* de similaridade é uma tentativa de utilização dos dados obtidos com esse propósito de auxiliar desenvolvedores na avaliação da qualidade do software em desenvolvimento, e como já comentado, esse cálculo de similaridade cumpriu seu objetivo.

- *H3 - Uma grande aproximação ao sistema implica em uma boa qualidade de código.*

H3 se torna verdadeira a medida que encontramos excelentes valores de métricas para a API Android na análise no Capítulo 3. Se aproximar, em termos de métricas de código fonte, de uma arquitetura consolidada e que contém ótimos valores de métricas, sem dúvidas pode ser utilizado como um indicador de qualidade de produto de software para ser utilizado em aplicativos.

- *H4 - As decisões arquiteturais aplicadas no estudo de caso e-lastic têm resultados equivalentes à decisões arquiteturais baseadas em métricas.*

Foi observado que as decisões tomadas ao longo do desenvolvimento com base em padrões de projeto estão relacionadas às decisões tomadas com base em métricas, uma vez que decisões tomadas baseadas em métricas tentariam levar os valores para os menores possíveis, tentando alcançar valores em intervalos ótimos, como os que foram encontrados no aplicativo parcial "e-lastic" desenvolvido neste trabalho. Foram tomadas decisões arquiteturais com base em padrões de projeto e princípios de design levando em conta experiência de programação com a plataforma, e o *score* encontrado utilizando o indicador de similaridade proposto reforça a ideia de que isso resultou em uma boa arquitetura.

- *QP - É possível monitorar métricas estáticas de código fonte de aplicativos Android de acordo com a análise de intervalos e aproximação às métricas do código do sistema Android?*

Por fim, este trabalho mostrou que a semelhança entre os dois faz com que os valores de referência sejam muito parecidos e podem então ser unificados em um só intervalo de referência. Assim, os intervalos definidos que caracterizam o código fonte da arquitetura do sistema são aplicáveis também aos aplicativos. A própria evolução do sistema demonstrou um certo padrão de permanência dos valores independentemente do tamanho do projeto, uma vez que a API aumentou seu tamanho em quatro vezes ao longo das versões analisadas, mas não teve variação significativa no tamanho das métricas. Esses resultados mais uma vez implicam na validade dos intervalos para o escopo de aplicativos.

## 5.1 Limitações

Uma limitação encontrada ao longo do trabalho foi a quantidade de dados utilizados para análise. Poucas versões do sistema Android puderam ser analisadas devido ao tempo que cada análise leva para ser concluída. Dados de poucas versões dificultam a generalização dos resultados para toda a plataforma, além de dificultar detecção de padrões nos valores de métricas na evolução do sistema.

Outra possível limitação foram as métricas selecionadas. Embora tenham sido utilizadas métricas consolidadas para arquiteturas OO, um possível argumento contra este trabalho é que as métricas aqui trabalhadas podem não ser suficientes para avaliar a qualidade da arquitetura, visto que diversos estudos utilizam um conjunto diferente de métricas. Várias métricas não foram usadas devido ao escopo reduzido deste trabalho, além de que seria necessária a utilização de mais de uma ferramenta de coleta.

Um fator que pode ameaçar a validade de alguns resultados deste trabalho foi a forma de normalização dos dados para o cálculo de similaridade proposto no Capítulo 3. Foi feita uma relativização dos valores com relação a uma referência frequente dentro da própria métrica, porém podem ser encontradas outras formas melhores de equivalência entre os valores das métricas.

A respeito do cálculo de similaridade, alguns valores aqui utilizados, como os pesos aplicados no cálculo, embora tenham algum fundamento matemático, são subjetivos e podem ser substituídos por um estudo mais detalhado e focado nesse propósito.

## 5.2 Trabalhos Futuros

Propomos inicialmente a coleta e análise de métricas para outras versões do Android. Com mais amostras, a análise pode ser feita de forma mais detalhada, além do que pode ser melhor estudada uma proposta de regressão de valores para algumas métricas.

A utilização de *machine learning* (ML) poderia ser uma boa contribuição para detecção de padrões no sistema e auxílio na análise dos valores das métricas. Um método

de ML que obtivesse os dados ideais da arquitetura seria de grande valia para comparação com os intervalos definidos.

Incluir mais métricas também é uma contribuição para este trabalho, avaliando a distribuição e evolução das mesmas ao longo das versões do sistema. Além disso, métricas adicionais podem melhorar o fator de similaridade que foi proposto.

Sobre o cálculo de similaridade, é interessante o testar com outras formas de normalização dos dados, e avaliar os resultados com valores distintos de normalização, e pesos distintos tanto para os percentis quanto para as métricas. Também pode ser criada uma escala de valores para melhor classificar o score de saída.

Além disso, mais informações poderiam ser apresentadas com o *score* de similaridade, como por exemplo dicas de refatoração obtidas a partir das diferenças mais discrepantes que contribuíram para o valor de saída. Isso auxiliaria bastante desenvolvedores iniciantes que não tem conhecimento profundo sobre interpretação de valores de métricas de código fonte, pois eles poderiam receber diretamente um indicador de qualidade com dicas de melhoria, sem olhar diretamente os valores das métricas.

# Referências

ABREU, F. B.; CARAPUÇA, R. Object-oriented software engineering: Measuring and controlling the development process. In: *Proceedings of the 4th international conference on software quality*. [S.l.: s.n.], 1994. v. 186. Citado 2 vezes nas páginas 49 e 51.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado 2 vezes nas páginas 40 e 43.

FERREIRA, K. A. M. et al. Reference values for object-oriented software metrics. Belo Horizonte, Brazil, 2009. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbes/2009/007.pdf>. Citado 3 vezes nas páginas 30, 45 e 52.

GRAY, A.; MACDONELL, S. A comparison of techniques for developing predictive models of software metrics. 1997. Disponível em: <https://aut.researchgateway.ac.nz/bitstream/handle/10292/3823/Gray%20and%20MacDonell%20(1997)%20I%26ST.pdf?sequence=2&isAllowed=y>. Citado na página 24.

GYIMOTHY, T.; FERENC, R.; SIKET, I. Empirical validation of object-oriented metrics on open source software for fault prediction. 2005. Disponível em: <http://flosshub.org/system/files/Gyimothy.pdf>. Citado na página 24.

LANUBILE, F.; VISAGGIO, G. Evaluating predictive quality models derived from software measures: Lessons learned. 1997. Disponível em: <http://www.di.uniba.it/~lanubile/papers/jss97.pdf>. Citado na página 24.

LINARES-VASQUEZ, M. Supporting evolution and maintenance of android apps. 2014. Disponível em: <http://www.cs.wm.edu/~mlinarev/pubs/ICSE14DS-Android-CRC.pdf>. Citado na página 23.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. São Paulo, 2013. Citado 14 vezes nas páginas 19, 29, 30, 32, 33, 35, 39, 42, 43, 44, 46, 48, 50 e 62.

MINELLI, R.; LANZA, M. Software analytics for mobile applications - insights and lessons learned. 2013. Disponível em: <http://old.inf.usi.ch/faculty/lanza/Downloads/Mine2013a.pdf>. Citado na página 23.

MOORE, A. Mobile as 7th of the mass media: an evolving history. 2007. Disponível em: <http://smlxtralarge.com/wp-content/uploads/2008/03/smlxl-m7mm-copy.pdf>. Citado na página 19.

OLIVEIRA, C. M. F. Kalibro: Interpretação de métricas de código fonte. São Paulo, 2013. Disponível em: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-25092013-142158/en.php>. Citado 8 vezes nas páginas 29, 30, 35, 36, 39, 45, 50 e 64.

R.BASILI, V.; BRIAND, L.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. 1995. Disponível em: <http://drum.lib.umd.edu/bitstream/1903/715/2/CS-TR-3443.pdf>. Citado 2 vezes nas páginas 19 e 24.

SHEPPERD, M. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, IET, v. 3, n. 2, p. 30–36, 1988. Citado na página 37.

SYER, M. et al. Exploring the development of micro-apps: A case study on the blackberry and android platforms. 2011. Disponível em: <http://sailhome.cs.queensu.ca/~mdsyer/wp-content/uploads/2011/07/Exploring-the-Development-of-Micro-Apps-A-Case-Study-on-the-BlackBerry-and-Android-Platforms.pdf>. Citado na página 23.

TERCEIRO, A.; CHAVEZ, C. Structural complexity evolution in free software projects: A case study. 2009. Citado na página 30.

XING, F.; GUO, P.; R.LYU, M. A novel method for early software quality prediction based on support vector machine. 2005. Disponível em: <http://www.cs.cuhk.hk/~lyu/paper_pdf/issre05_pguo.pdf>. Citado 2 vezes nas páginas 19 e 24.