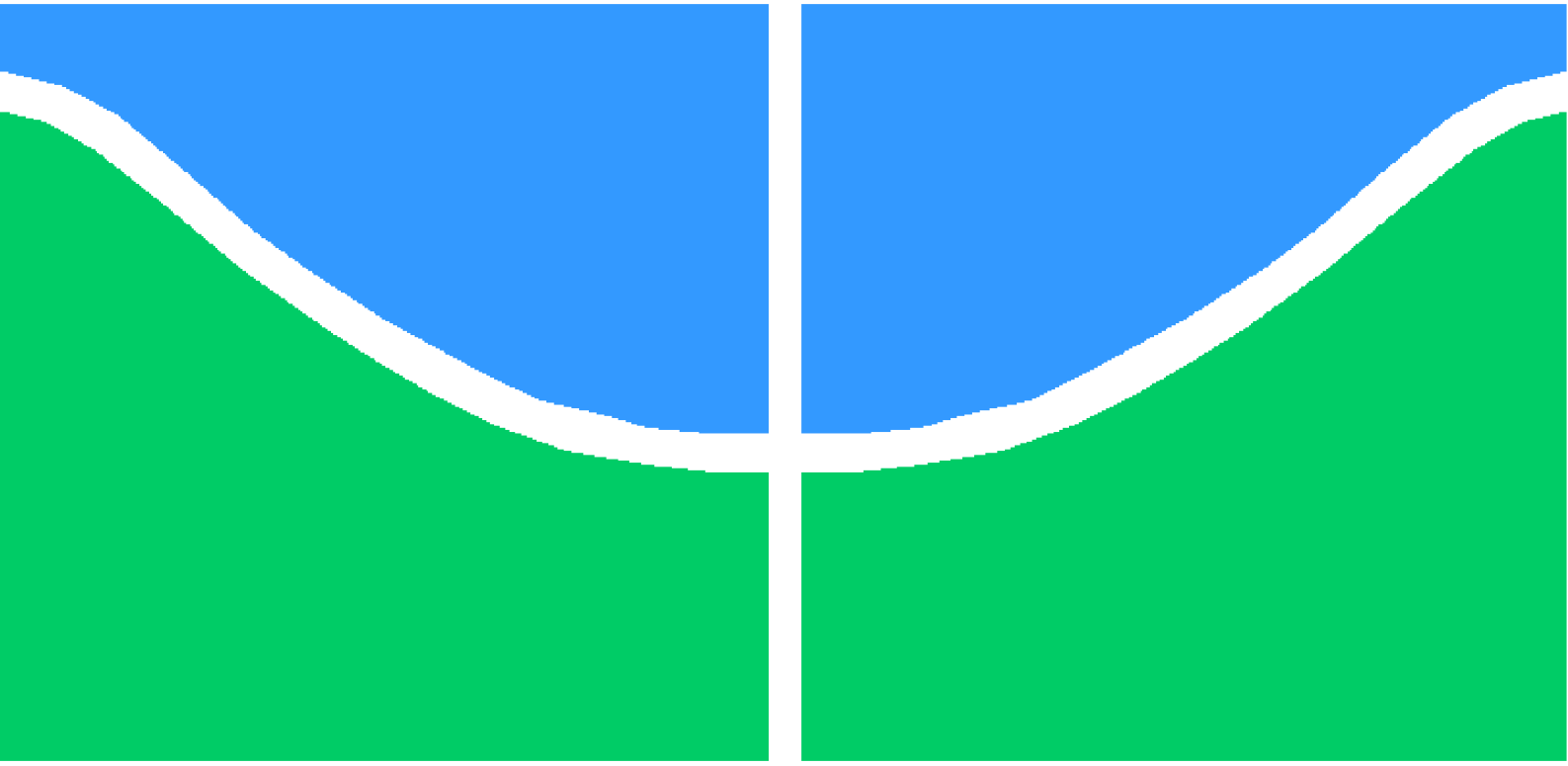

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

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-
113 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 OO, 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. It has emerged innovative projects that leverage the constant presence of these devices 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

UnB	Universidade de Brasília
OO	Orientado a Objetos
API	*Application Programming Interface*
JNI	*Java Native Interface*
UID	*User Identification*
DVM	*Dalvik Virtual Machine*
ART	*Android RunTime*
JVM	*Java Virtual Machine*
DEX	*Dalvik Executable*
GUI	*Graphic User Interface*
ICS	*Ice Cream Sandwich*
CSV	*Comma Separated Values*
AMLOC	*Average Method Lines Of Code*
ACCM	*Average Cyclomatic Complexity per Method*
RFC	*Response For a Class*
DIT	*Depth in Inheritance Tree*
NOC	*Number of Children*
LCOM4	*Lack of Cohesion in Methods*
ACC	*Afferent Connections per Class*
COF	*Coupling Factor*
LOC	*Lines of Code*
NOM	*Number of Methods*
CBO	*Coupling Between Objects*

AOSP	*Android Open Source Project*
MVC	*Model-View-Controller*
AIDL	*Android Interface Definition Language*
RPC	*Remote Procedure Call*
SDK	*Software Development Kit*
ADT	*Android Development Tools*
IDE	*Integrated Development Environment*
AVD	*Android Virtual Devices*
App	*Application*
MES	Manutenção e Evolução de Software

Sumário

1 Introdução

1.1 Contexto

A computação eletrônica tem avançado muito desde o século XX, e esse avanço continua bastante visível no século XXI (FONSECA, 2007). Aparelhos e maquinas que antes custavam fortunas e ocupavam salas inteiras puderam avançar ao ponto de caber em uma única caixa. Engrenagens e inúmeras válvulas reduzidas a um micro chip. Várias ferramentas utilizadas no dia a dia de muitos trabalhadores reunidas em um único computador, que adquiriu muitas outras funções além de fazer cálculos.

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 e alta capacidade de armazenamento de dados, além da conectividade que faz com que a utilização de cabos pareça insensata. 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) propõe 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.

1.2 Objetivos

O objetivo principal deste trabalho é o monitoramento de métricas estáticas de código fonte na API do sistema operacional Android, essencialmente métricas OO, 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.

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 validação de intervalos de referência para valores de métricas;
- Proposta de utilização dos intervalos definidos para auxílio no desenvolvimento de aplicativos;

O trabalho teve uma etapa inicial na qual foi levantada a maioria dos dados do Capítulo 2, e foi então desenvolvido um pedaço de um aplicativo Android com base em um estudo de caso específico, com o objetivo de alcançar uma boa arquitetura a partir de padrões de projeto. O resultado dessa etapa inicial pode ser visto no Apêndice B será utilizado para comparação das decisões de design aplicadas, e as possíveis decisões que podem ser tomadas a partir de análise de métricas de código fonte.

1.3 Estrutura do trabalho

Este documento está dividido em 5 capítulos. No Capítulo 2, é apresentada uma descrição geral da plataforma Android, bem como critérios de seleção de plataformas móveis, além de conceitos da engenharia de software aplicados nesse contexto. O Capítulo 3 conceitua um conjunto métricas de código fonte e cita uma ferramenta para coleta das mesmas. No Capítulo 4 são melhor explicados os objetivos do trabalho e os métodos para o alcance dos objetivos propostos. Em seguida, o Capítulo 5 apresenta discussões sobre análise dos dados coletados, validação dos mesmos, e uma proposta de aplicação dos valores definidos. 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 Sistema Operacional Android[1]

Este capítulo descreve um pouco o sistema Android, e tem o intuito de ser uma documentação básica resumida, devido a falta de documentação em português sobre a plataforma, visto que a principal fonte das informações, a documentação oficial em uma página web, se encontra apenas em inglês. Dessa forma, algumas seções se apresentam relativamente extensas e boa parte das informações contidas em algumas delas pode ser encontrada no site oficial em inglês. Ainda neste capítulo também foram utilizados conceitos da engenharia de software para contextualizar etapas de desenvolvimento de software em plataformas móveis, e em especial, no Android.

2.1 Descrição Geral

O Android é um sistema operacional para dispositivos móveis com base em kernel linux modificado, com várias bibliotecas modificadas ou refeitas, de forma a deixar o sistema tão eficiente quanto possível para o hardware limitado que os dispositivos alvo apresentam. A exemplo disso está a biblioteca C *Bionic*, que foi desenvolvida para menor consumo de espaço físico, memória e poder de processamento que as bibliotecas padrão C como a GNU C (glibc) (DEVOS, 2014). Aplicações desenvolvidas para Android são feitas essencialmente em linguagem Java, com a possibilidade de utilizar outras linguagens como C e C++ através da *Java native interface* (JNI).

O sistema Android tira vantagem do kernel linux no que diz respeito a identificação e separação de processos rodando no sistema, atribuindo a cada aplicação um *UID* (*User Identification*), e executando cada uma em um processo diferente, isolando umas das outras. Independentemente de a aplicação ser desenvolvida em Java ou com código nativo, essa separação de processos do kernel, conhecida como *Application Sandbox*, garante que a aplicação está isolada das demais e portanto sujeita aos mesmos mecanismos de segurança inclusive que os aplicativos do sistema, como contatos, câmera, entre outros.

Nas versões anteriores ao Lollipop, cada uma dessas aplicações no sistema funcionava em uma instância diferente da *Dalvik Virtual Machine* (DVM), enquanto atualmente a DVM foi substituída pela Android Run Time (ART), introduzida opcionalmente desde a versão kitkat. Ambas são máquinas virtuais semelhantes a Java Virtual Machine (JVM). Códigos em Java são compilados e traduzidos para formato .dex (*dalvik executable*), que é executado pela DVM, semelhante ao formato .jar do Java. Enquanto a DVM utiliza *just-in-time compilation*, compilando trechos do código para execução nativa em tempo

[1] Este capítulo é baseado na documentação oficial disponível em AndroidDevelopers (2014)

de execução, a nova ART introduz o *Ahead-of-time compilation*, realizando compilações em tempo de instalação. Embora a instalação possa levar mais tempo dessa forma, essa mudança permite que os aplicativos tenham maior performance em sua execução. Esse isolamento de aplicativos, onde cada um é executado em sua própria instancia da máquina virtual, permite que uma falha em um processo de uma aplicação não tenha impacto algum em outra aplicação.

Para interagir com determinados serviços do sistema bem como outras aplicações, uma aplicação deve ter os privilégios correspondentes a ação que deseja executar. Por exemplo, o desenvolvedor pode solicitar ao sistema que sua aplicação tenha acesso a internet, privilégio não concedido por padrão pelo sistema. O usuário então no momento da instalação dessa aplicação é informado que a mesma deseja acesso a internet, e ele deve permitir acesso se quiser concluir a instalação. Todas as permissões requisitadas pelo desenvolvedor e necessárias para o aplicativo realizar suas funções são listadas no momento de instalação, e todas devem ser aceitas, caso contrário a instalação é cancelada. Não é permitido ao usuário selecionar quais permissões ele quer conceder e quais rejeitar à aplicação sendo instalada, tendo apenas as opções de aceitar todas elas, ou rejeitar a instalação.

O Android procura ser o mais seguro e facilmente utilizado sistema móvel, modificando a forma que várias tarefas são executadas para alcançar esse objetivo, como por exemplo fazer o isolamento de aplicações utilizando a separação de processos e usuários do kernel linux para gerenciar aplicativos instalados e proteger os dados dos mesmos. Aplicativos devem ser assinados e obrigatoriamente isolados uns dos outros (incluindo aplicativos do sistema) e devem possuir permissões explícitas para acessar recursos do sistema e outros aplicativos. As decisões arquiteturais relacionadas a segurança foram tomadas desde o início do ciclo de desenvolvimento do sistema, e continuam sendo prioridade.

Todo o código do sistema, incluindo a bionic, a Dalvik Virtual Machine e Android Run Time, é aberto para contribuição de qualquer desenvolvedor. Através do *Android Open Sorce Project* (AOSP), as fabricantes de dispositivos obtém o código do sistema, modificam conforme desejarem, adicionam aplicativos próprios e distribuem com seus produtos.

2.2 Estrutura de uma aplicação

Aplicações no Android são construídas a partir de quatro tipos de componentes principais: *Activities, Services, Broadcast Receivers*, e *Content Providers* (HEUSER et al., 2014).

1. Uma *Activity* é basicamente o código para uma tarefa bem específica a ser reali-

zada pelo usuário, e apresenta uma interface gráfica(*Graphic User Interface*) para a realização dessa tarefa.

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

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

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

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

Para executar a tarefa de ler um email, por exemplo, um usuário instala um aplicativo de gerenciamento de emails. Então ele deseja abrir um anexo de um email que está em formato PDF. O aplicativo de email não precisa necessariamente prover um leitor de PDF para que o usuário consiga ter acesso a esse anexo. Ele pode mandar ao sistema a intenção de abrir um arquivo PDF a partir de um *Intent*, e então o sistema encontra um outro componente que pode fazer isso, e o instancia. Caso mais de um seja encontrado, o sistema pergunta para o usuário qual é o componente que ele deseja utilizar. O sistema então invoca uma *Activity* de um outro aplicativo para abrir esse arquivo. Continuando no mesmo exemplo, o usuário clica em um link dentro do arquivo pdf. Esse aplicativo, por sua vez, pode enviar ao sistema a intenção de abrir um endereço web, que mais uma vez encontra um aplicativo capaz de o fazer.

É importante perceber que para uma atividade mais complexa de interação com o usuário, vários aplicativos são envolvidos sem que os mesmos tenham conhecimento

dos demais. Cada componente se "registra" no sistema para realizar determinada tarefa, e o sistema se encarrega de encontrar os componentes adequados para cada situação. Esse registro dos componentes é realizado através do AndroidManifest.xml, que é um arquivo incluso em toda aplicação sendo instalada. Ele reúne todos os componentes de uma aplicação, as permissões necessárias para acessar cada um deles, e as permissões que eles utilizam, bem como outras informações.

Uma vez que os componentes de uma aplicação podem ser utilizados por outras aplicações, é necessário um controle maior sobre quem pode ter acesso a cada um deles. Cada desenvolvedor pode criar permissões customizadas para seus componentes, e exigir que o aplicativo que requisite a tarefa tenha essa permissão para acessar o componente. Da mesma forma, o aplicativo que criou essa permissão determina os critérios para conceder a mesma para outros aplicativos. Um simples exemplo de uso desse mecanismo é o fato de uma empresa apenas criar vários aplicativos para tarefas distintas e querer integração entre os mesmos. O desenvolvedor pode definir uma permissão específica para acessar um dos seus *Content Providers*, por exemplo, e definir que apenas aplicativos com a mesma assinatura (assinados pelo mesmo desenvolvedor) possam receber essa permissão. Dessa forma, todos os aplicativos desenvolvidos por essa empresa podem ter acesso aos dados gerenciados por esse *Content Provider*, enquanto as demais aplicações não tem esse acesso. Aplicativos pré instalados ou internos do sistema podem conter um tipo de permissão específica que só é dada a aplicativos do sistema, e não pode ser obtida por nenhum outro aplicativo instalado pelo usuário.

2.3 Diversidade e Compatibilidade

O Android foi projetado para executar em uma imensa variedade de dispositivos, de telefones a *tablets* e televisões. Isso é muito interessante no ponto de vista do desenvolvedor, que tem como mercado para seu software usuários de diversos dispositivos de diversas marcas diferentes. Entretanto, isso trás uma necessidade de fazer uma interface flexível, que permita que um aplicativo seja utilizável em vários tipos de dispositivos, com vários tamanhos de tela. Para facilitar esse problema, o Android oferece um *framework* em que se pode prover recursos gráficos distintos e específicos para cada configuração de tela, publicando então um aplicativo apenas que se apresenta de forma diferente dependendo do dispositivo onde ele está sendo executado.

A interface gráfica no Android é essencialmente construída em XML, e tem um padrão de navegação para as aplicações, embora fique a critério do desenvolvedor a aparência de sua aplicação. O desenvolvedor por criar, por exemplo, uma interface gráfica com arquivos XML para cada tamanho de tela, e também diferenciar entre modo paisagem e modo retrato. Entretanto, em se tratando de interface gráfica, vários componentes

vão sendo adicionados a API Android ao longo de sua evolução, e portanto vários recursos gráficos necessitam de uma versão mínima do sistema para serem utilizados. Utilizar um recurso presente apenas a partir da versão ICS 4.0.4 (*Ice Cream Sandwich*), por exemplo, implica que o aplicativo não tenha compatibilidade com versões anteriores do sistema.

Da mesma forma, devido a diversa variedade de modelos e fabricantes de hardware, é preciso ficar atento aos recursos de hardware disponíveis para cada dispositivo. Alguns sensores hoje mais comuns aos novos dispositivos sendo lançados no mercado não existiam em modelos mais antigos. O desenvolvedor pode especificar no *Android manifest* os recursos necessários para o funcionamento completo de sua aplicação, de forma que a mesma seja apenas instalada em dispositivos que os apresentarem. Também pode ser feita uma checagem em tempo de execução e apenas desativar uma funcionalidade do aplicativo caso algum recurso de hardware não esteja disponível no dispositivo, se isso for o desejo do desenvolvedor. De forma geral, é relativamente simples a forma com que o desenvolvedor especifica os dispositivos alvo para sua aplicação, tornando essa grande diversidade de dispositivos mais vantajosa do que dispendiosa.

Android é um sistema livre, que pode ser utilizado em modificado e utilizado segundo a licença Apache versão 2.2. Shanker e Lal (2011) apresenta alguns tópicos relacionados a portabilidade do sistema Android em um novo hardware. Embora não seja discutida nesse documento, a relativamente fácil portabilidade do sistema para vários tipos de hardware foi uma das razões que levaram seu rápido crescimento, fazendo com que várias fabricantes possam fazer uso do mesmo sistema e lançar vários tipos de dispositivos distintos no mercado, com diferentes *features* e preços, alcançando parcelas do mercado que possuem condições de aquisição muito variadas.

2.4 Engenharia de Software aplicada ao Android

Engenharia de software é definida pela (IEEE, 2014) como aplicação de uma abordagem sistemática, disciplinada e mensurável ao desenvolvimento, operação e manutenção de software. As subseções desta seção citam alguns dos processos da engenharia de software que são importantes em todo o ciclo de vida do produto de software, que inclui desde a concepção do produto até a manutenção do mesmo em ambiente de produção, no contexto de plataformas móveis, e em específico, no Android.

2.4.1 Seleção de plataformas móveis

Holzer e Ondrus (2009) Por exemplo, propõe 3 critérios para seleção de plataformas móveis no ponto de vista do desenvolvedor:

1. Remuneração - Relacionado ao número de potenciais clientes que podem adquirir o

produto. Plataforma com grande número de usuários e crescimento constante é de grande valia para desenvolvedores. Um ponto centralizado de venda de aplicativos também é um atrativo para a comercialização dos softwares desenvolvidos.

2. Carreira - As oportunidades que o desenvolvimento para a plataforma pode gerar. A possibilidade de trabalhar para grandes e renomadas empresas no mercado pode ser um fator decisivo para a escolha da plataforma. Para aumentar sua credibilidade na comunidade de desenvolvedores e ganhar visibilidade no mercado, Holzer e Ondrus (2009) sugere que o desenvolvedor participe de projetos de software livre, o que é facilitado quando a própria plataforma móvel é aberta.

3. Liberdade - Liberdade criativa para desenvolver. O programador deve sentir que na plataforma ele pode programar o que quiser. Uma plataforma consolidada com excelentes kits de desenvolvimento e dispositivos atrativos do ponto de vista de hardware e sistema operacional atraem muitos desenvolvedores. Plataformas fechadas e com muitas restrições tendem a afastar desenvolvedores que querem essa liberdade, enquanto plataformas abertas apresentam maior liberdade para desenvolvimento.

Com seu grande crescimento e consequente tomada da maior fatia do mercado de dispositivos móveis, a plataforma Android consegue ser bastante atrativa no ponto de vista primeiro tópico, em contraste com as demais plataformas do mercado, como WindowsPhone [2] e iOS [3]. Mais e mais empresas aparecem no contexto do Android tanto em desenvolvimento de hardware quanto software, e as oportunidades de trabalho crescem juntamente com o crescimento da própria plataforma, como sugerido no segundo tópico. Por ser aberto, a plataforma Android também permite que desenvolvedores enviem suas contribuições e correções de bugs ao próprio sistema operacional, aumentando sua visibilidade. Da mesma forma, o Android também apresenta um sistema totalmente aberto e com kits de desenvolvimento consolidados e extensiva documentação disponível online [4], no mínimo se equiparando ao seu principal concorrente iOS em liberdade de desenvolvimento na data de escrita desse documento.

Segundo Wasserman (2010), alguns aspectos devem ser pensados quando desenvolvendo software para dispositivos móveis:

1. Requisitos de interação com outras aplicações;

2. Manipulação de sensores;

3. Requisitos web que resultam em aplicações hibridas (*mobile* - *web*);

[2] <http://www.windowsphone.com/>
[3] <https://www.apple.com/br/ios/>
[4] <http://developer.android.com/>

4. Diferentes famílias de hardware;

5. Requisitos de segurança contra aplicações mal intencionadas que comprometem o uso do sistema;

6. Interface de Usuário projetadas para funcionar com diversas formas de interação e seguir padrões de design da plataforma;

7. Teste de aplicações móveis são em geral mais desafiadores pois são realizados de maneira diferente da maneira tradicional;

8. Consumo de energia;

Todos esses tópicos mencionados são muito importantes para o desenvolvimento em várias plataformas móveis, e modificam a forma com que várias atividades da engenharia de software devem ser abordadas. A plataforma Android tem sua arquitetura projetada para atender a vários desses quesitos:

- As aplicações são isoladas e se comunicam de forma unificada com outras aplicações ou com componentes e recursos do sistema;

- A linguagem Java de programação da uma imensa liberdade de utilizar diversas bibliotecas e *frameworks* desenvolvidos anteriormente para o Java;

- A camada de máquina virtual correspondente a DVM e ART permite uma abstração do uso dos componentes físicos e aumenta assim a compatibilidade com diversos tipos e famílias de hardware;

- A segurança foi prioridade desde os primeiros estágios de desenvolvimento, tentando prever inclusive ataques de engenharia social que tentam convencer o usuário a instalar aplicações maliciosas em seu dispositivo;

- A interface de usuário dos aplicativos é extremamente customizável, ainda podendo manter facilmente um padrão de navegação;

- Testes no sistema foram projetados para cada componente isoladamente, tentando facilitar o processo de teste de aplicativos;

- Recursos do sistema tem controle minucioso para melhor gerenciamento de uso de energia.

Tomando os critérios apresentados como base, a escolha da plataforma Android para desenvolvimento neste trabalho é feita de forma clara.

2.4.2 Requisitos

Área de conhecimento em requisitos de software é responsável pela elicitação, análise, especificação e validação de requisitos de software, bem como a manutenção gerenciamento desses requisitos durante todo o ciclo de vida do produto (IEEE, 2014).

Requisitos de software representam as necessidades de um produto, condições que ele deve cumprir para resolver um problema do mundo real que o software pretende atacar. Essas necessidades podem ser funcionalidades que o software deve apresentar para o usuário, chamados requisitos funcionais, ou outras condições que restringem as funcionalidades de alguma forma, seja por exemplo sobre tempo de execução, requisitos de segurança ou outras restrições, conhecidas como requisitos não funcionais.

Requisitos não funcionais são críticos para aplicações móveis, e estas podem precisar se adaptar dinamicamente para prover funcionalidade reduzida (DEHLINGER; DIXON, 2011). Embora o hardware de dispositivos móveis tenha avançado bastante nos últimos anos, dispositivos móveis ainda apresentam capacidade reduzida de processamento devido a limitações como o tamanho reduzido e capacidade limitada de refrigeração. Devido a essas e outras limitações e a grande variedade de dispositivos Android no mercado, com poder computacional bem variado, aplicativos devem ser projetados para funcionar em hardware limitado. Em suma, deve-se pensar sempre em requisitos de performance e baixo consumo de recursos: uso de rede (3g/4g/wifi/bluetooth...), energia, ciclos de processamento, memória, entre outros. Wasserman (2010) afirma que o sucesso de qualquer aplicação, *mobile* ou não, depende de uma grande lista de requisitos não funcionais.

Segundo Dehlinger e Dixon (2011), deve-se analisar bem requisitos de contexto de execução de aplicativos dispositivos móveis. Aplicações móveis apresentam contextos de execução que não eram obtidos em tecnologias anteriores, com dados adicionais como localização, proximidade a outros dispositivos, entre outros, que podem alterar a forma com que os aplicativos são utilizados. Aplicativos móveis tem que ser pensados para se adaptar com essas mudanças de contexto.

O sistema Android permite checar a disponibilidade de recursos de hardware em tempo de instalação ou execução para que o desenvolvedor possa ajustar as funcionalidades apresentadas ao usuário e prevenir que o usuário encontre problemas na utilização de determinadas funcionalidades. Por exemplo, um jogo simples como um *tic tac toe* que utilize *bluetooth* para *multiplayer* pode desativar essa funcionalidade para dispositivos antigos que o não tenham disponível e trabalhar apenas com jogo *single player* contra algum tipo de jogador virtual. Da mesma forma, a ausência de algum recurso pode impedir que algum aplicativo seja instalado no dispositivo. O whatsapp, por exemplo, não pode ser instalado em dispositivos que não possuem comunicação com rede móvel via cartão SIM, como tablets que possuem apenas comunicação WIFI. Dessa forma é possível prevenir a

apresentação para o usuário de funcionalidades que ele na verdade não pode executar.

Requisitos de software podem ser representados de diversas formas, sendo possível a utilização de vários modelos distintos. Na metodologia ágil scrum, por exemplo, os requisitos normalmente são registrados na forma de *User Stories*, onde são geralmente descritos na visão do usuário do sistema. Em outros contextos, podem ser descritos em casos de uso, com descrições textuais e diagramas, ou outras várias formas de representação.

Requisitos de software geralmente tem como fonte o próprio cliente que contrata o serviço do desenvolvimento de software, e são extraídos da descrição de como esse cliente vê o uso do sistema. Todo esse processo é muitas vezes chamado de "engenharia de requisitos".

Essa diferenciação que pode ser observada em requisitos para plataformas móveis em relação a software convencional também é refletida nas outras fases de desenvolvimento. Um produto idealizado de forma diferente acarreta em um produto trabalhado totalmente de forma diferente. As possíveis interações entre usuário e sistema, ou entre usuários, dão novos contextos de utilização para o produto de software. A forma como os usuários utilizam o próprio sistema alvo do software sendo desenvolvido muda a forma com que atividades de criação e inovação, planejamento, desenvolvimento, implantação e até mesmo distribuição e marketing são conduzidas.

2.4.3 Desenho

Desenho de software é o processo de definição da arquitetura, componentes, interfaces, e outras características de um sistema ou um componente (IEEE, 2014). É durante o desenho de software que os requisitos são traduzidos na estrutura que dará base ao software sendo desenvolvido. As necessidades então são traduzidas em modelos, que descrevem os componentes e as interfaces entre os componentes. A partir desses modelos é possível avaliar a validade da solução desenhada e as restrições associadas a mesma, sendo possível avaliar diferentes soluções antes da implementação do software. A partir do design da arquitetura, é possível prever se alguns requisitos elicitados podem ou não ser atingidos com determinada solução, e mudá-la conforme necessário com pouco ou mesmo nenhum custo adicional.

A área de desenho de software pode variar conforme a tecnologia sendo utilizada. A arquitetura do sistema pode variar conforme o sistema operacional alvo, ou mesmo conforme a linguagem de programação que se está utilizando no desenvolvimento. Existem vários princípios de design de software amplamente conhecidos que se aplicam a uma imensidade de situações, sempre com o intuito de encontrar a melhor solução para cada situação e deixar o software modularizado e manutenível.

Aplicativos para o sistema Android são construídos em módulos, utilizando os

componentes da API, embora possam ser criadas classes em Java puro sem a utilização de nenhum recurso da API do sistema, e utilizá-las nos componentes assim como em uma aplicação Java padrão desenvolvida para *desktop*. Várias classes de modelo em uma arquitetura MVC (*Model-View-Controller*), por exemplo, possivelmente serão criadas em Java puro. Por outro lado, o Android não impõe nenhuma arquitetura específica no desenvolvimento de aplicações, deixando livre para o desenvolvedor fazer suas escolhas.

Sokolova, Lemercier e Garcia (2013) apresentam alguns tipos de arquitetura derivados do bem difundido MVC, e demonstram uma possibilidade de adaptação do MVC ao Android. Embora a arquitetura MVC possa ser utilizada no Android, ela não é facilmente identificada, e não é intuitiva de ser implementada. *Activities* são os componentes mais difíceis de serem encaixados na arquitetura MVC padrão, embora sejam bem adaptadas às necessidades do desenvolvedor. Por padrão elas tem responsabilidades correspondentes ao *Controller* e ao *View*, e são interpretadas de forma diferente por vários desenvolvedores para o MVC.

O Android provê um *framework* para desenvolver aplicativos baseado nos componentes descritos no início deste capítulo. Os aplicativos são construídos com qualquer combinação desses componentes, que podem ser utilizados individualmente, sem a presença dos demais. Cada um dos componentes pode ser uma entrada para o aplicativo sendo desenvolvido. De forma geral, para desenhar uma arquitetura para sistema Android deve-se levar em conta todos esses componentes e conhecer bem sua aplicabilidade e a comunicação entre os mesmos. Mais detalhes sobre componentes Android podem ser encontrados no apêndice deste trabalho.

Neste trabalho a arquitetura do sistema e de aplicativos será avaliada e comparada através do resultado de métricas estáticas de código fonte, essencialmente métricas OO, que refletem várias das decisões arquiteturais. Como será discutido no Capítulo 4, a principal questão de pesquisa é desenvolvida em cima da relação entre a API do sistema android e os aplicativos desenvolvidos para a mesma, relação que pode ser refletida em métricas estáticas de código como será discutido no Capítulo 3.

2.4.4 Construção

Essa área de conhecimento é responsável pela codificação do software, por transformar a arquitetura desenhada e seus modelos em código fonte. A construção de software está extremamente ligada ao desenho e às atividades de teste, partindo do primeiro e gerando insumo para o segundo (IEEE, 2014).

Várias medidas podem ser coletadas do próprio código pra auxiliar a avaliação da qualidade do produto sendo construído e gerar insumo para o próprio desenvolvedor reavaliar sua implementação antes da fase de testes.

Este trabalho visa auxiliar a fase de construção de aplicativos Android através da análise de métricas estáticas de código que refletem o design nesse contexto de desenvolvimento Android. O objetivo dessa análise é auxiliar desenvolvedores de aplicativos nos primeiros estágios de desenvolvimento e continuamente durante a construção do produto, provendo uma avaliação do estado atual e uma base de comparação para o projeto durante todo o ciclo de vida, trabalhada a partir do código do sistema e de aplicativos nativos, como email, calendário, contatos, câmera, calculadora e o *web browser*.

Essa base de comparação deve ser idealizada como uma referência válida para este contexto de desenvolvimento de aplicativos, então as conclusões tiradas para o código do sistema devem se mostrar válidas também para aplicativos desenvolvidos para o mesmo. O acoplamento de aplicativos com a própria API do sistema indica que isso é uma possibilidade bastante plausível, o que será discutido no Capítulo 5.

2.4.5 Manutenção

A manutenção de software trata dos esforços de desenvolvimento com o software já em produção, isto é, em funcionamento no seu devido ambiente. Problemas que passaram despercebidos durante as fases de construção e testes são encontrados e corrigidos durante a manutenção do sistema. Da mesma forma, o usuário pode requisitar novas funcionalidades que ele não havia pensado antes do uso do sistema, e o desenvolvimento dessas novas funcionalidades é também tratado como manutenção uma vez que o software já se encontra em produção, um processo conhecido como evolução de software. Revisões de código e esforços com manutenibilidade também podem ser consideradas atividades de manutenção, embora possam acontecer antes do sistema entrar em produção.

A manutenção de software geralmente acontece por período mais longo que as demais fases do desenvolvimento do software citadas nos tópicos anteriores, ocupando a maior parte do ciclo de vida do produto.

A separação de componentes independentes apresentados neste capítulo é de grande valia para a manutenibilidade do sistema. Essa separação permite que componentes tenham sua funcionalidade específica melhor compreendida e possam ser substituídos sem grandes impactos nos demais.

Atividades de design de software devem ser reforçadas para garantir uma fase de manutenção com a menor quantidade de problemas possível. Uma arquitetura modularizada é mais fácil de ser entendida e modificada e consequentemente mantida. Também é importante que se utilize de padrões de codificação, identação e documentação para que a manutenção seja facilitada inclusive para desenvolvedores que não tem familiaridade com o código desenvolvido. Empresas tendem a colocar equivocadamente engenheiros junior para dar manutenção a sistemas e engenheiros mais experientes para desenvolver

novos projetos, e adotar práticas de desenvolvimento que agem em favor à manutenibilidade ajudam a amenizar os problemas causados por esse tipo de alocação de recursos humanos.

Os resultados de métricas que refletem decisões arquiteturais geralmente tem relação com a manutenibilidade do software. Alto acoplamento entre objetos, por exemplo, indica que uma mudança em um pequeno trecho de código pode trazer resultados catastróficos no restante do software.

Sendo um sistema de código aberto, o Android permite que o desenvolvedor possa o analisar e consequentemente o entender a ponto de corrigir *bugs* do sistema, criar funcionalidades novas, e também portá-lo para novos *hardwares* (GANDHEWAR; SHEIKH, 2010). Isso é um ponto importante para atividades de manutenção do sistema Android como um todo, e permitiu que a plataforma crescesse de forma extremamente acelerada em um pequeno espaço de tempo. Um dos motivos para a própria plataforma Linux estar em um estado tão estável nos dias atuais foi os anos que a mesma teve de contribuição da comunidade sendo um software livre e portanto tendo seu código aberto a qualquer desenvolvedor, assim como o sistema Android.

Inserido no contexto de manutenção e evolução do sistema Android, este trabalho avalia resultado de métricas em um sistema operacional consolidado e amplamente utilizado para auxiliar qualquer etapa que envolva manipulação de código fonte no ciclo de desenvolvimento de outros projetos relacionados para esta plataforma. Desenvolvedores poderão utilizar resultados deste estudo para tomar decisões relacionadas a remodelagem e refatoração de seus projetos que já estejam em produção, a fim de melhorar sua qualidade e facilitar sua evolução.

A atividade de testes, descrita no apêndice deste documento, corresponde a outra atividade essencial no contexto de manutenção de um produto de software. É importante ressaltar que, assim como podem refletir a qualidade de um produto de software, métricas de código fonte podem ajudar a prever esforços de teste já na fase de construção. A complexidade ciclomática em valores muito altos, por exemplo, pode indicar um software praticamente intestável, e portanto muito difícil de se manter.

Em linhas gerais, impactos positivos podem ser vistos em todas as etapas do ciclo de vida do software quando se monitora o desenvolvimento do produto desde sua concepção.

3 Métricas de Código

A ISO/IEC 9126, reunida agora na ISO/IEC 25000, apresenta características de qualidade e um guia para o uso dessas características, de forma a auxiliar e padronizar o processo de avaliação de qualidade de produtos de software. Ela separa qualidade de software essencialmente em qualidade interna e qualidade externa. Qualidade externa é a visão do produto de software de uma perspectiva externa, resumindo-se basicamente em qualidade do sistema em execução. Já a qualidade interna trata da visão interna do produto de software, podendo incluir documentos, modelos e o código fonte. A qualidade interna pode começar a ser medida em estágios mais iniciais do desenvolvimento por não haver ainda nesses estágios uma visão externa do sistema. Esses valores de qualidade interna podem ser utilizados para prever os valores de qualidade externa que o produto vai apresentar. É importante perceber essa relação entre qualidade interna e externa para perceber que qualidade interna impacta fortemente na qualidade externa de um produto, e então observar a importância de começar esforços de medição e acompanhamento da qualidade interna desde o início do projeto. 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 (MEIRELLES, 2013).

3.1 Métricas de código fonte selecionadas

Métricas de código fonte caracterizam bem o produto de software em qualquer estado do seu desenvolvimento. Existem vários tipos de métricas de código fonte. Entre as principais categorias, temos métricas de tamanho, complexidade, acoplamento e coesão.

Várias métricas de tamanho podem ser utilizadas para avaliar quantidade de código fonte, como por exemplo Linhas de código - *Lines of Code (LOC)*. Manter o monitoramento de métricas de volume de código em conjunto com outras métricas é importante para que comparações sejam feitas para sistemas de tamanho semelhante. A exemplo disso temos métricas de complexidade como relacionadas ao tamanho do software. Essas métricas de tamanho devem ser consideradas para que as comparações sejam feitas de forma adaptada para a "escala" de tamanho dos softwares sendo comparados. AMLOC (Média de linhas de código por método - *Average Methods Lines Of Code*) é semelhante a uma combinação de LOC e NOM (Número de métodos de uma classe - *Number of Methods*), duas métricas de tamanho, é uma boa escolha em termos de tamanho de código para o início deste trabalho devido a sua simplicidade, embora este estudo possa ser futuramente expandido com a utilização de métricas adicionais, para complementar ou substituir essa métrica.

R.Basili, Briand e Melo (1995) apresenta um estudo com métricas CK, introduzidas por Chidamber e Kemerer (1994), que são métricas para sistemas orientados a objetos, para avaliação da arquitetura do sistema e qualidade do código fonte. São essencialmente métricas que verificam coesão e acoplamento de classes, além de complexidade de estrutura hierárquica de objetos, característica de projetos OO. Essas métricas são bastante úteis para prever estados futuros já nas primeiras etapas do ciclo de vida. Tais métricas podem ser bastante úteis como indicadores de qualidade de sistemas orientados a objetos(R.BASILI; BRIAND; MELO, 1995).

Métricas de complexidade também dão uma visão do estado atual do software no que diz respeito a escolhas arquiteturais e facilidade de compreensão do mesmo e também tem impacto direto na manutenibilidade. A métrica de complexidade ciclomática, por exemplo, introduzida por McCabe (1976), foi criada para avaliar a quantidade de caminhos que a execução do software pode seguir em sua execução, indicando complexidade de leitura e entendimento do código, assim como esforço para testá-lo.

Todas as métricas que serão descritas neste capítulo tem a interpretação geral de ter o valor tão pequeno quanto possível, indicando simplicidade no projeto. Simplicidade no projeto é geralmente característica de uma boa arquitetura.

3.1.1 Ferramenta de captura de métricas

O Analizo[1] é uma ferramenta livre e extensível para análise de código com suporte a várias linguagens, incluindo Java, que será o foco da análise neste trabalho. Uma grande quantidade de métricas são coletadas pela ferramenta, embora apenas algumas sejam utilizadas para esta análise. Um dos motivos da escolha do sistema Debian para execução das análises foi a facilidade de instalação da ferramenta.

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 quanto 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

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

contabiliza essa herança, como será discutido na análise da métrica DIT no Capítulo 5.

É importante ressaltar que as métricas discutidas neste capítulo, utilizadas durante todo o estudo, são coletadas de forma unificada pela ferramenta Analizo, o que também foi um dos motivos de sua escolha. Mais detalhes sobre a escolha das métricas podem ser vistos no Capítulo 4. As métricas finais escolhidas estão listadas a seguir:

- 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)
- 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)

3.1.2 Descrição das métricas

Este capítulo tem o intuito de descrever os conceitos das métricas utilizadas, enquanto nos capítulos seguintes serão apresentados mais detalhes sobre a escolha das métricas, bem como interpretação das mesmas diretamente no contexto de dispositivos móveis em plataforma Android e linguagem Java. No Capítulo 5, serão discutidos valores dessas métricas para a API do sistema Android, bem como para aplicativos nativos, e então será proposta a verificação de similaridade de aplicativos em relação à API, com base nos valores dessas métricas aqui conceituadas.

3.1.2.1 Linhas de Código (LOC) / Média de linhas de código por método (AMLOC)

LOC representa o número de linhas de código fonte de uma classe, enquanto AMLOC a média do número de linhas dos métodos daquela classe. Número de métodos (*Number of Methods* - NOM) conta o número de métodos de uma classe e também é uma métrica de tamanho (SHARMA et al., 2012).

A primeira observação que deve ser feita quando analisando LOC nesse contexto é a diferenciação das linguagens. Embora um módulo em C seja mapeado para uma classe, arquivos fonte em C tendem a ser maiores que uma classe em Java, por exemplo, devido aos diferentes paradigmas que essas linguagem utilizam. Arquivos em C++ e Java

também podem ter valores bem distintos para a mesma funcionalidade devido ao número de bibliotecas padrões que a linguagem apresenta e a natureza da própria sintaxe da linguagem. Dessa forma, comparações dessa métrica devem ser feitas somente dentro da mesma linguagem.

A métrica LOC por si só não será discutida aqui, pois seu valor é totalmente independente e deve ser comparado com outras métricas para ter significado mais completo. AMLOC tem significado semelhante a uma combinação de LOC e NOM, e utilizá-la pode ser considerado uma utilização indireta dessas métricas. AMLOC apresenta uma interpretação mais concisa que as demais, uma vez que métodos grandes "abrem espaço" para problemas de complexidade excessiva.

Em suma, a análise de outras métricas podem abranger as explicações relacionadas a métrica LOC e também a NOM, então essas métricas de tamanho não serão explanadas em separado, mas ocasionalmente citadas na explicação de outras métricas.

3.1.2.2 Média de complexidade ciclomática por método (ACCM)

Complexidade ciclomática nada mais é do que 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. 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 interpretação do valor de complexidade ciclomática é relativamente simples: O valor 1 é o valor mínimo e ideal para se ter como resultado, pois significa que o software tem apenas uma forma de executar e será executado necessariamente daquela forma e naquela sequência. Como consequência disso, se tem um software que pode ser mais facilmente lido e modificado. A implicação dessa métrica é mais notada na atividade de testes do código fonte, pois além de dificultar a compreensão dos possíveis comportamentos de um pedaço de código, cada caminho adicional que pode ser seguido é um trecho diferenciado que deve ser testado. Isso quer dizer que o esforço de teste é diretamente proporcional ao resultado dessa métrica, pois para garantir o funcionamento correto do sistema, todos as possibilidades devem ser devidamente testadas. Em termos práticos, atingir uma cobertura de código de 100% é uma tarefa árdua quando há um valor muito grande de complexidade ciclomática.

Inserida então no contexto de manutenção e testes, ACCM é uma excelente candidata para ser constantemente monitorada ao longo da evolução do código fonte. Embora não tenha muita relação com outras métricas OO, ela tem uma relação óbvia do nú-

mero de linhas de código de um método, pois um método com poucas linhas de código não tem possibilidade de ter um valor muito alto de complexidade ciclomática. Entretanto, essa relação não faz com que elas possam ser utilizado para propósitos semelhantes (WATSON; MCCABE; WALLACE, 1996).

3.1.2.3 Resposta para uma classe (RFC)

Response for a Class 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 então pode indicar baixa coesão e alto acoplamento.

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

DIT é uma métrica que mede a profundidade que uma classe se contra na árvore de herança, e caso haja herança múltipla, DIT mede a distancia máxima até o nó raiz da árvore de herança (CHIDAMBER; KEMERER, 1994). Se ela não herda nada, tem DIT igual a 0. Se herda de uma classe, a profundidade é 1, e assim por diante.

NOC mede a quantidade de filhos que uma classe tem (CHIDAMBER; KEMERER, 1994). Caso ninguém herde dela, o valor é 0, e aumenta em 1 para cada classe que a estende diretamente, ou seja, filhos de filhos não são contabilizados.

DIT e NOC são métricas relativamente semelhantes por trabalhar com a árvore de herança, entretanto tem interpretações diferentes. São métricas que indicam complexidade no design, assim como a maioria das métricas OO.

Altos valores de DIT indicam que a classe herda de várias outras recursivamente, podendo tornar seu comportamento mais imprevisível, pois não se sabe todos os seus possíveis comportamentos sem analisar as demais. Classes com alto DIT tendem a ser mais complexas por adicionar o comportamento de todas suas classes precursoras. Entretanto, por se tratar de herança, altos valores de DIT também indicam maior reuso de código fonte.

NOC também indica maior potencial de reuso em altos valores, assim como na métrica DIT. Entretanto, ela também indica a importância de uma classe e seus comportamentos no design. Um NOC alto significa que uma mudança na classe pode ter consequências graves, pois seus métodos são utilizados em muitos filhos. Consequentemente é recomendado que classes com altos valores de NOC sejam muito bem testadas.

3.1.2.5 Falta de coesão em métodos (LCOM)

LCOM é uma métrica que mede coesão de uma classe. Existem algumas variações da métrica LCOM definida por Chidamber e Kemerer (1994) criadas por outros estudos que não serão abordadas neste estudo. A variação calculada pela ferramenta Analizo e utilizada neste trabalho é a LCOM4 (HITZ; MONTAZERI, 1995).

LCOM4 gira em torno da ideia de que os métodos da classe estão coesos se eles utilizam os mesmos atributos dentro dessa classe. Se algum método não utiliza nada da classe, ou utiliza apenas métodos/atributos de outra classe, ele provavelmente está no lugar errado.

A métrica então calcula quantos conjuntos de métodos relacionados existem dentro dessa classe, isto é, métodos que compartilham utilização de algum atributo ou que se referenciam. Caso existam 2 conjuntos de métodos distintos, ou seja, cada conjunto utiliza um conjunto diferente de atributos e um conjunto não utiliza nenhum método do outro, o valor de LCOM4 é 2, e significa que essa classe pode ser dividida em 2 para aumentar a coesão. O valor ideal 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.

É possível notar, pela própria definição da métrica, que LCOM4 é limitada pelo número de métodos da classe (NOM), embora não sejam diretamente relacionadas. Uma classe com 2 métodos não pode ter mais que 2 conjuntos distintos de métodos relacionados, então seu LCOM4 não passa de 2. Essa observação apenas quer dizer que classes pequenas tendem a ter menores valores de LCOM4.

3.1.2.6 Acoplamento entre Objetos (CBO) / Conexões aferentes de uma classe (ACC)

A métrica de acoplamento entre objetos (CBO), definida por Chidamber e Kemerer (1994), calcula as conexões de entrada e de saída de uma classe, isto é, para uma classe A, são contabilizadas classes que utilizam algum método ou variável de A, como também todas as classes que A referencia. Entretanto neste trabalho não a utilizaremos por problemas encontrados na coleta, como será discutido no Capítulo 4, ficando com ACC como métrica de acoplamento.

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 referencia a ela, por meio da utilização de algum método ou atributo. 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.

Uma classe com altos valores de ACC é utilizada em muitas outras. A interpretação dessa métrica é semelhante a métrica NOC no que diz respeito a impacto de mudanças. Ter muitas classes clientes indica que é necessário um maior cuidado ao realizar edições nessa classe, uma vez que impactos dessas modificações podem ocorrer em um número de classes tão grande quanto o valor de ACC.

De forma geral, deseja-se ter classes tão independentes quanto possível, levando o valor de ACC para baixo.

3.1.2.7 Fator de acoplamento (COF)

COF é uma métrica MOOD proposta por Abreu e Carapuça (1994), e é nada mais é que uma relativização do valor de ACC explicado na seção anterior para o tamanho do projeto, sendo então um valor apenas para todo o código fonte desse projeto. 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 pode variar de 0 a 1. Caso todas as X conexões possíveis aconteçam em um software, COF para ele será X/X, que é igual a 1. O ideal então como acoplamento para um projeto qualquer é que o valor de COF esteja tão próximo de zero quanto possível, indicando que as classes são mais independentes e desacopladas.

Naturalmente o incremento no número de classes de um projeto tende a fazer com que o valor da métrica caia, embora não seja sempre inversamente proporcional a ponto de esperar que projetos distintos com números maiores de classes sempre tenham menores valores de COF, pois depende bastante do design de sua arquitetura.

4 Metodologia

4.1 Objetivos

O objetivo geral deste trabalho é realizar uma análise da 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, 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. Verificar então a possibilidade da validação dos intervalos de valores através de regressão polinomial utilizando métricas OO em função de métricas de tamanho.

Assumindo que o código da API do sistema possui uma boa arquitetura, 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 a nível de software é 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. Usar esse score como forma de avaliar a qualidade em comparação com a API, julgando então como resultados teóricos melhores, próximos a API, ou piores, para valores negativos, próximos ao 0, e positivos, respectivamente.

Algumas conclusões talvez já possam ser tiradas de acordo com a aproximação do código do sistema[1] aos aplicativos nativos. Se a média de scores indicar um valor negativo, por exemplo, indica que na média os aplicativos do sistema tem valores teóricos melhores que os da plataforma, e talvez o objetivo de um novo *app* seja se manter dentro dos intervalos de valores do sistema, mas tentando ficar com valores ainda menores, não necessariamente iguais aos da API. Essa verificação de scores é reflexão dos intervalos de valores que serão definidos no Capítulo 5, e será discutida no mesmo capítulo.

[1] "Código do sistema" está sendo utilizado para referenciar a API de desenvolvimento de aplicativos

4.2 Questão de pesquisa e hipóteses

Baseando-se nas ideias apresentadas, é 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 do código do sistema?

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 que foi definida, 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 teóricas aplicadas no estudo de caso e-lastic estão relacionadas com decisões arquiteturais baseadas em métricas.

A hipótese H1 será testada com a identificação manual de um comportamento de métricas no código fonte de diversas versões do sistema. O objetivo principal dessa hipótese é identificar se o sistema contém algum padrão nos valores das métricas ao longo de sua evolução, ou se os valores são completamente diferentes das demais versões e portanto não relacionados. A identificação de um padrão de permanência/aumento/diminuição nos ajuda a definir intervalos bons de métricas válidos para o sistema, e ter uma ideia se eles continuarão válidos para versões futuras.

H2 é tratada com análise do código fonte da API e dos aplicativos, identificando semelhança entre ambos. Identificando semelhanças entre ambos, podemos confirmar algumas premissas como o alto acoplamento entre API e aplicativos, que já é esperado como outros trabalhos já sugerem. Essa confirmação nos leva a afirmar que os valores de métricas para o sistema, já mais consolidado, podem ser referência para comparação direta com aplicativos sem preocupação com escala do projeto, podendo ser utilizados durante desenvolvimento de aplicativos para Android.

H3 parte da ideia de que, caso a API tenha bons valores de métricas, o que será verificado no Capítulo 5, e portanto boa qualidade de código fonte, uma aproximação de aplicativos em relação à API em termos de métricas indica uma boa qualidade de código

fonte. Para avaliar essa hipótese, verificaremos os resultados da proposta de cálculo de semelhança de aplicativos que será discutida também no Capítulo 5.

Para a Hipótese H4, será avaliado, através da comparação com o sistema, o resultado parcial inicial deste trabalho, que corresponde a um aplicativo Android (descrito no Apêndice B) desenvolvido com o objetivo de criar uma arquitetura modularizada, flexível e manutenível. Foram tomadas diversas decisões teóricas com o objetivo de alcançar a melhor arquitetura para um aplicativo Android, desenvolvido para um estudo de caso específico, e então desejamos avaliar se essas decisões tomadas, com base em experiência de desenvolvedor e padrões de projeto, são semelhantes às decisões que podem ser tomadas com base em resultados de análise estática de código fonte, como as análises realizadas neste trabalho.

4.3 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 dependência maior em relação ao código do sistema torna o código de aplicativo desenvolvido para Android 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

ultimo 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, considerada ideal por ser desenvolvida pela criadora e mantenedora do sistema operacional. 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.

4.4 Coleta de Dados

O código fonte para análise foi retirado diretamente do *Android Open Source Project*[2] (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 ultima versão do sistema, com atualizações funcionais e de segurança, enquanto trabalham em ideias inovadoras para melhorar a experiência de usuário de seus dispositivos. A Motorola, por exemplo, tem o seu sistema levemente modificado para incluir algumas funcionalidades exclusivas em seus produtos, assim como várias outras grandes fabricantes como a Samsung, LG e outras. Essa forma de manter o sistema aberto e altamente customizável mantém uma competitividade entre as empresas, pois partindo do mesmo sistema base, todas as fabricantes entregam a seus clientes dispositivos com essencialmente as mesmas funcionalidades, com exceção das suas pequenas modificações, em um sistema operacional robusto e estável.

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

A ferramenta *repo*[3] 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 ultima 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 *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

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

13. *Android Lollipop* 5.1.0 r1

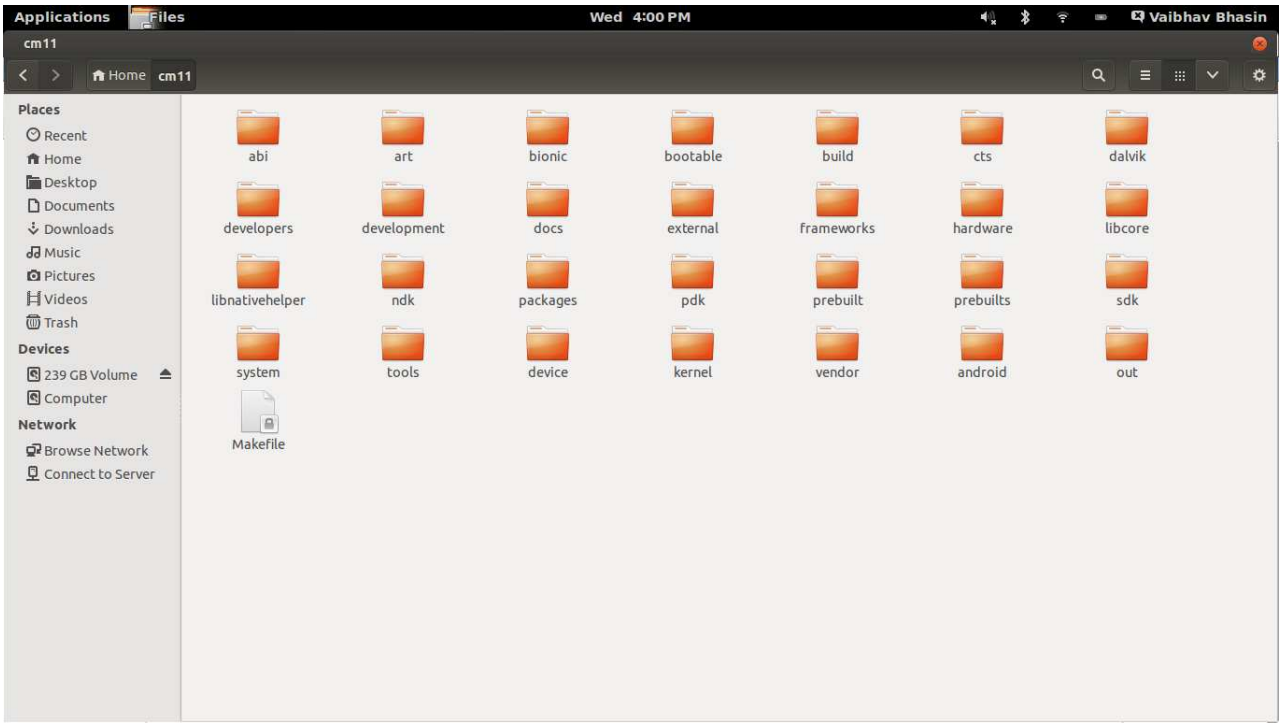

Figura 1 – Exemplo de estrutura de arquivos de diretório raiz do AOSP

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 página de ajuda a desenvolvedores[4] 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.

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

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

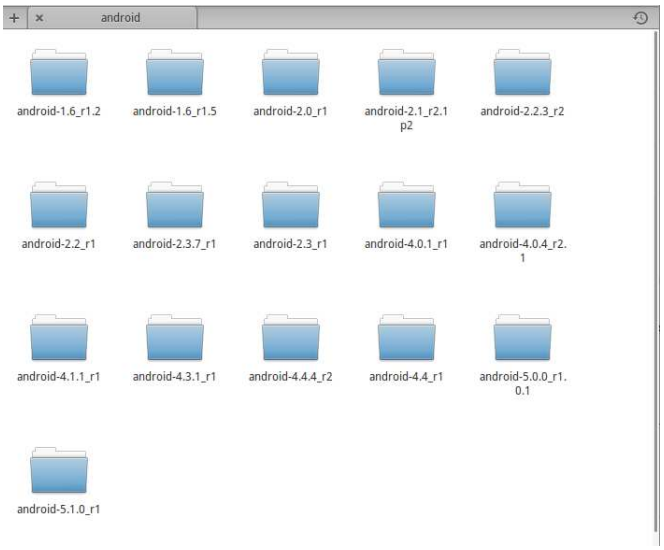

Figura 2 – Diretório onde foram armazenadas cada versão a ser analisada

a dalvik nas ultimas versões do sistema (especificamente desde as versões de *codename Lollipop*), e também códigos de baixo nível específicos para alguns dispositivos. Também 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 ultima 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 possam ser ressaltadas, embora, mais uma vez, não haja implicação substancial no resultados gerais apresentados.

4.5 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 resultado da análise de cada projeto contém os valores parciais das métricas para cada arquivo que foi analisado pela ferramenta. No total são 16 arquivos CSV contendo cada um as métricas para todas as classes/módulos de 1 versão do sistema. O arquivo CSV que contém métricas unificadas para todos os projetos não será utilizado neste trabalho.

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, como discutido no Capítulo 3. 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 5. 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, descritas no Capítulo 3, foram capturadas pela ferramenta Analizo e estão listadas a seguir:

- 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)
- 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 computação estatística e possui diversos módulos que facilitam análise estatística, com 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 5. Os dados coletados para aplicativos foram reunidos da mesma maneira, porém, como só uma versão foi analisada para os aplicativos, em vez de versões, o primeiro valor da tabela é o nome do aplicativo do sistema de onde as métricas foram coletadas.

version	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 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 5.

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 5 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 boa amostra estatística, 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 5. Ainda no capítulo de resultados, 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. É importante ressaltar que esse cálculo que será feito não resulta em um valor com significado semântico relacionado ao seu valor absoluto, pois o mesmo não tem uma medida, e portanto será utilizado apenas em comparação com os valores de similaridade calculados para outros aplicativos e verificação de distancia do valor de referência da API, representado pelo número 0. 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 Resultados

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 polinomial.

5.1 Análise de Distribuição

Com os dados coletados e devidamente preparados, várias conclusões podem ser tiradas dos valores das métricas e sua evolução ao longo do tempo. Esta seção é focada na análise subjetiva 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.

5.1.1 Distribuição dos dados

Os gráficos na Figura 3 apresentam as distribuições dos dados para as métricas LCOM4, ACC, ACCM e RFC. Podemos perceber que em todos os casos a probabilidade cumulativa, representada pelos percentis, apresenta um aumento de valor apenas nos últimos percentis, já demonstrando que que esses valores nos últimos percentis são bem discrepantes. Entretanto eles só começam a realmente aumentar significativamente a partir do percentil 95, então os 5% restantes podem ser descartados como um ruído estatístico.

Nas tabelas que serão discutidas nas seção seguintes, percebe-se que 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 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.

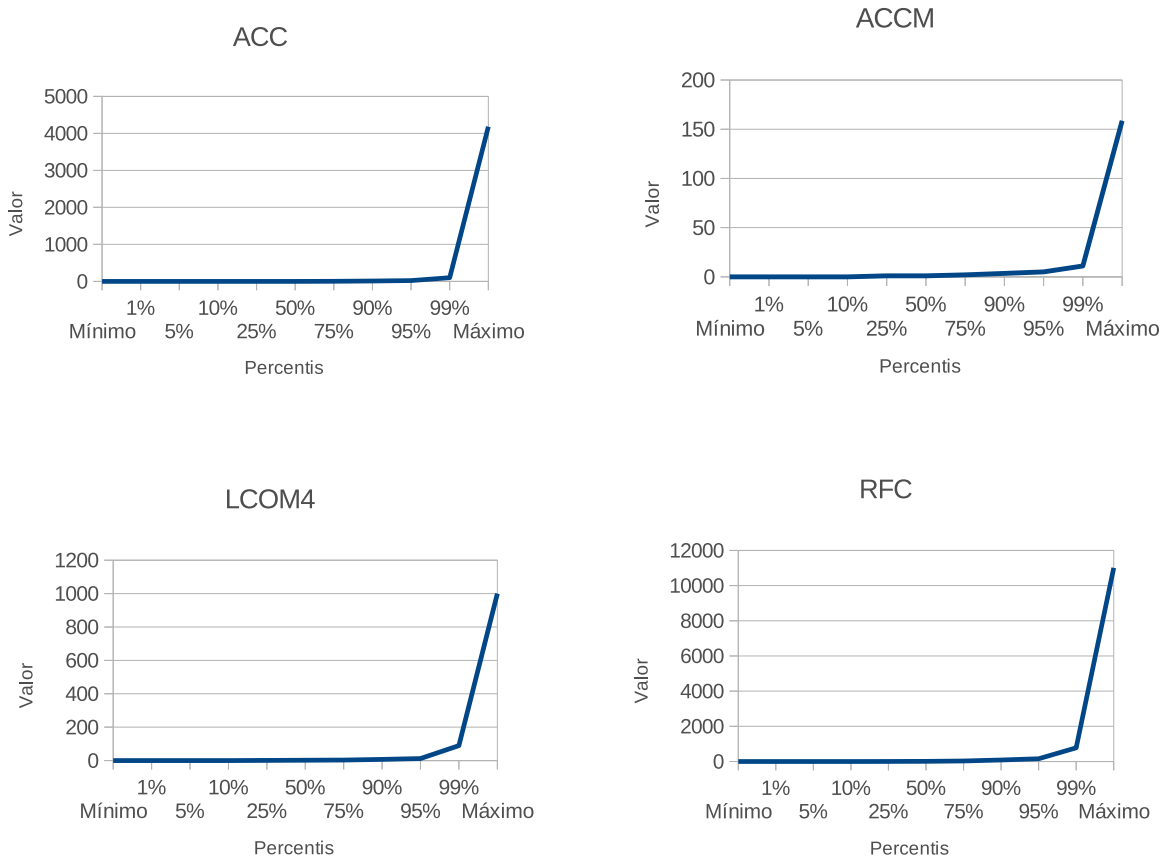

Figura 3 – Distribuição das métricas ACC, ACCM, LCOM4 e RFC na versão 5.1.0

5.1.2 Average Method Lines Of Code

version	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 – *Average Method Lines of Code* no Android

A Tabela 2 apresenta os valores para a métrica AMLOC nas versões do Android 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, onde é possível ver uma variação muito pequena de valores para os

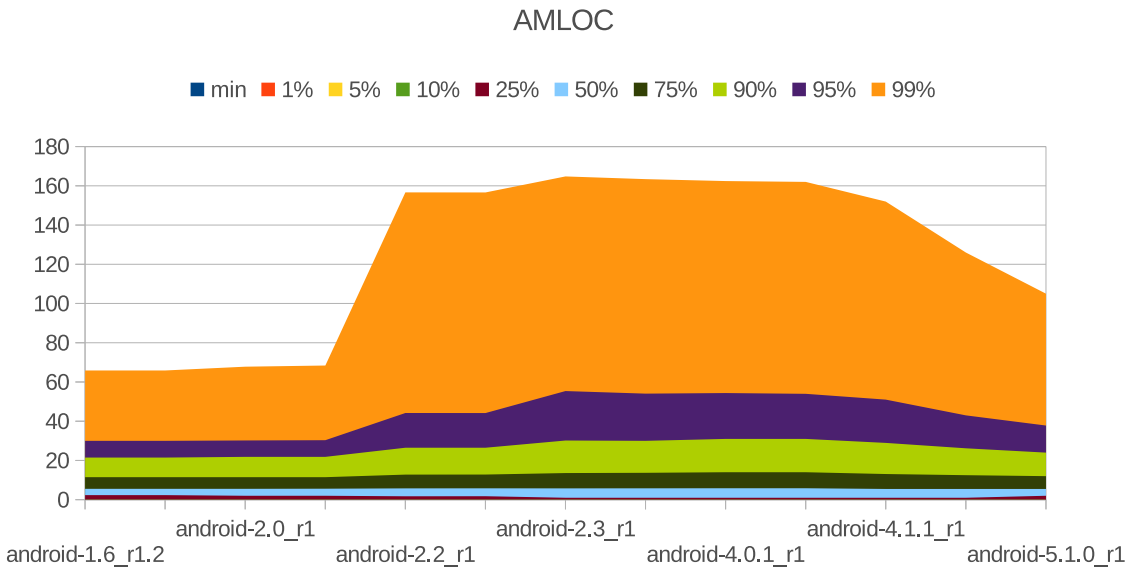

Figura 4 – Evolução da métrica AMLOC ao longo das versões da API

percentis 75 e 90. O percentil 99 demonstra uma variação maior, mas ele representa dados não frequêntes na análise.

Esses são valores que 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. É possível perceber que 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) analiza 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 linguagems C/C++/Java para esta métrica não é significativa, uma vez que a sintaxe entre as 3 é bastante semelhante. Dada essa afirmação, podemos comparar os intervalos definidos por ele, chegando a 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, quando olhamos os valores aplicativos do sistema, demonstrados na Tabela 3, podemos perceber uma grande semelhança nos resultados.

Métodos relacionados a 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

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 – *Average Method Lines of Code* nos aplicativos nativos

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, pode-se perceber que os intervalos se mantém válidos para a grande maioria dos aplicativos. Esses valores de aplicativos foram retirados dos aplicativos nativos da ultima 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 a 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 diferente propósito de cada aplicativo, que utiliza pedaços variados do sistema e tem sua codificação adaptada para seu propósito.

Intervalos encontrados:

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

5.1.3 Average Cyclomatic Complexity per Method

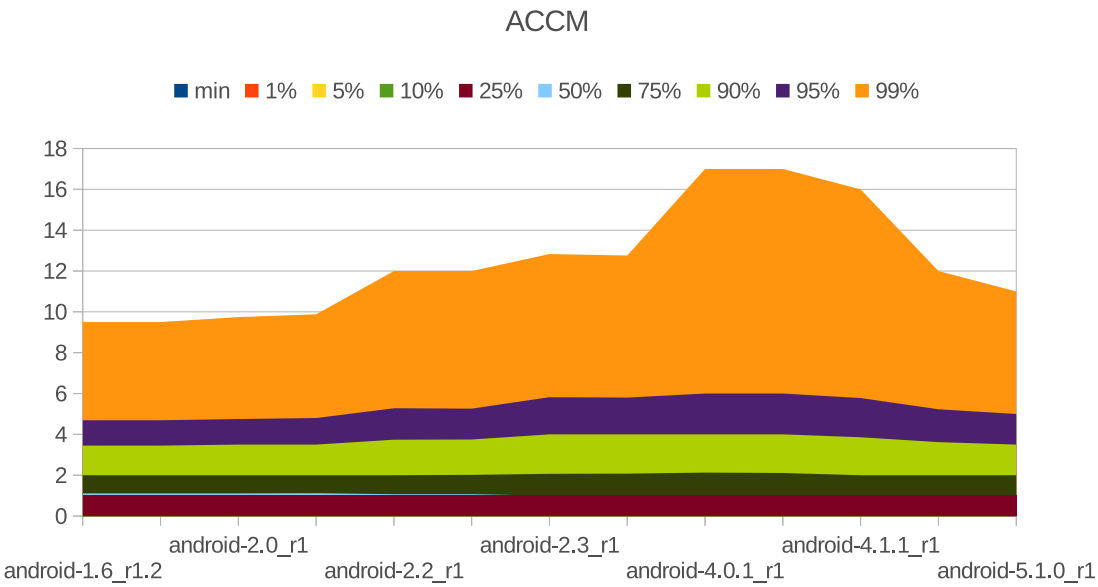

Figura 5 – Evolução da métrica ACCM ao longo das versões da API

A Figura 5 apresenta a evolução de ACCM métrica com a evolução da API. Podemos perceber que 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.

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 pode ser visto na Tabela 2. Essa relação direta também pode ser vista nos percentis 90 e 95, que representam valores frequentes e pouco requentes, respectivamente.

Essa relação pode ser vista para os aplicativos. Embora não seja totalmente determinístico, no geral aplicativos com maiores valores de AMLOC tendem a ter um maior

version	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 – *Average Cyclomatic Complexity per Method* no Android

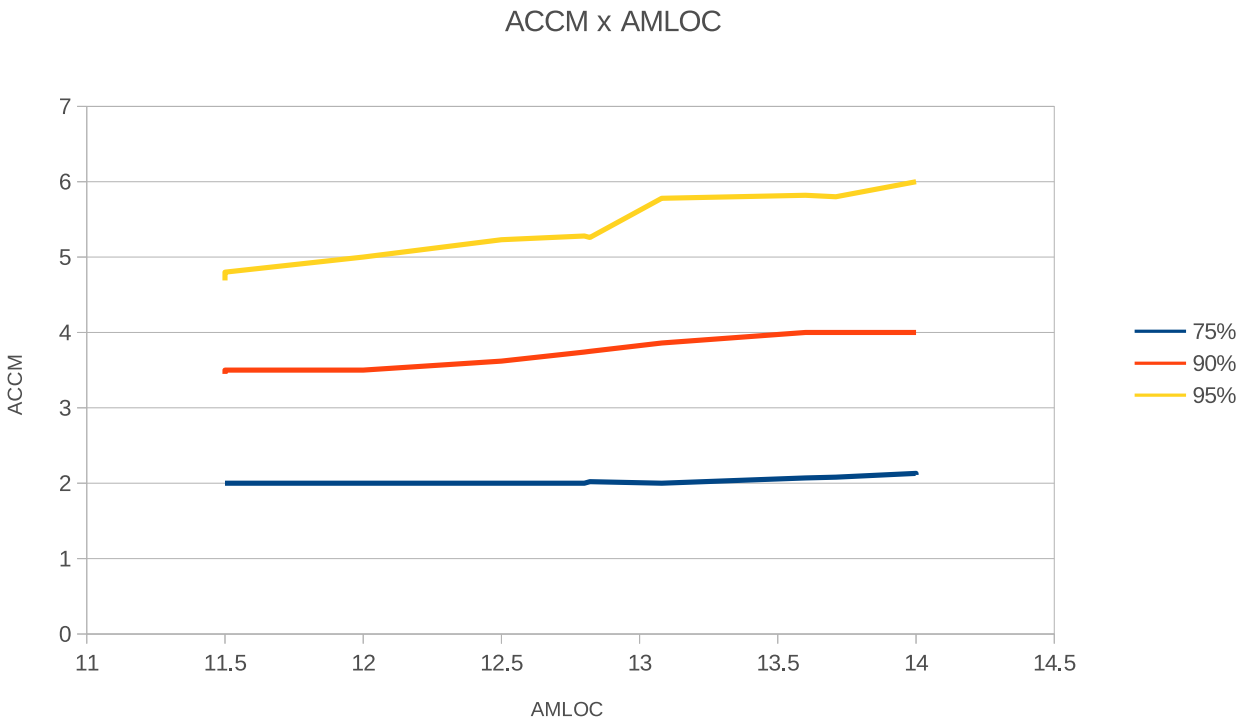

Figura 6 – ACCM por AMLOC nas Tabelas 2 e 4

valor de ACCM, assim como visto na API do sistema. 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. Observando as Tabelas 4 e 5 percebe-se 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 comple-

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 – *Average Cyclomatic Complexity per Method* nos aplicativos nativos

xidade 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 como um 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 se mostraram 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 menor ou igual a 4 pode ser vista 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. Os aplicativos do sistema quase não alcançaram esse valor;

- 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.

5.1.4 Response For a Class

version	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 – *Response For a Class* no Android

A API do sistema Android tende a ter um valor relativamente alto de RFC devido a forma como sua arquitetura foi desenhada, como pode ser visto na 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 *isDiscoverying()* e *cancelDiscovery()* para controlar a busca. Um acesso direto a uma variável booleana removeria a necessidade da chamada de método *isDiscoverying()*, entretanto perderia seu encapsulamento. De forma geral, encapsulamento de variáveis tende a aumentar o valor de RFC, que conta apenas métodos.

Contribuindo para o aumento, o resultado da busca de dispositivos bluetooth é realizado de forma assíncrona na forma de mensagens utilizando *intents* (vide Capítulo 2), 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

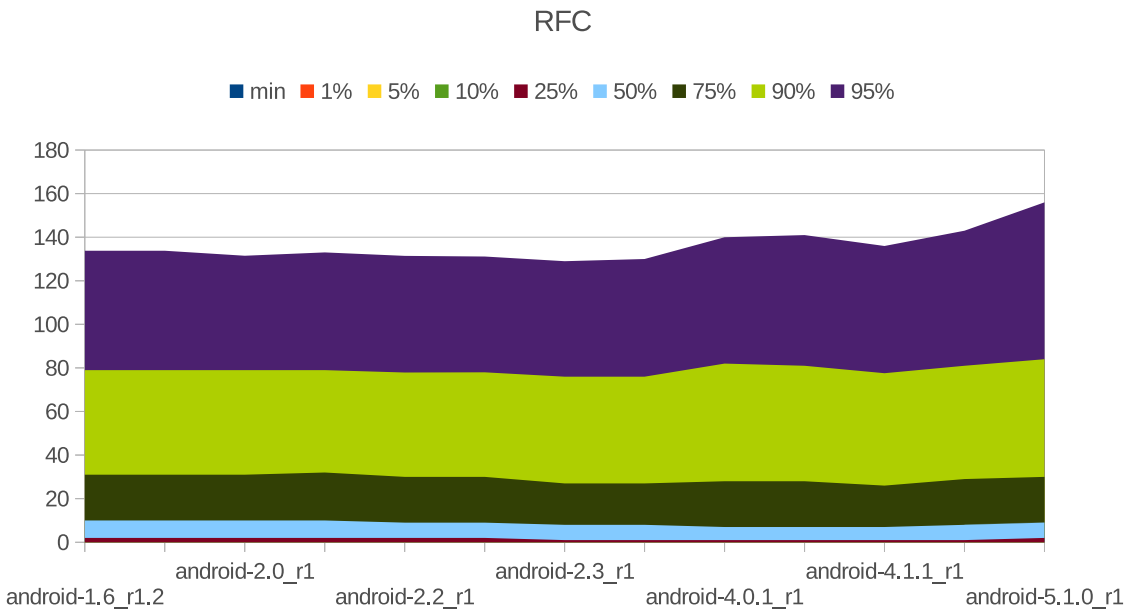

Figura 7 – Evolução da métrica RFC ao longo das versões da API

controle adicional que permite que o mesmo escalone melhor a utilização de recursos que necessitem de exclusão mútua. Por exemplo, um acesso direto a 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 utilização 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 demonstra que essa métrica não teve variação grande ao longo das versões do Android, tendendo inclusive a um leve aumento nas ultimas versões, demonstrando que os valores altos apresentados são mesmo uma característica da arquitetura do sistema.

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

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 – *Response For a Class* nos aplicativos nativos

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. Pode-ses perceber que 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, são considerados os seguintes intervalos:

- Valores abaixo 31 se mostraram muito 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, mas parecem ser comuns no Android, então são aceitáveis. 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, se mostram bastante semelhantes aos valores para o projeto Android como um todo, com predominância da linguagem C, encontrados por Meirelles (2013). Essa semelhança pode levar a interpretação de que estilos semelhantes de estruturação e design são utilizados em todo o AOSP, independente da linguagem utilizada em cada módulo.

5.1.5 Depth of Inheritance Tree / Number of Children

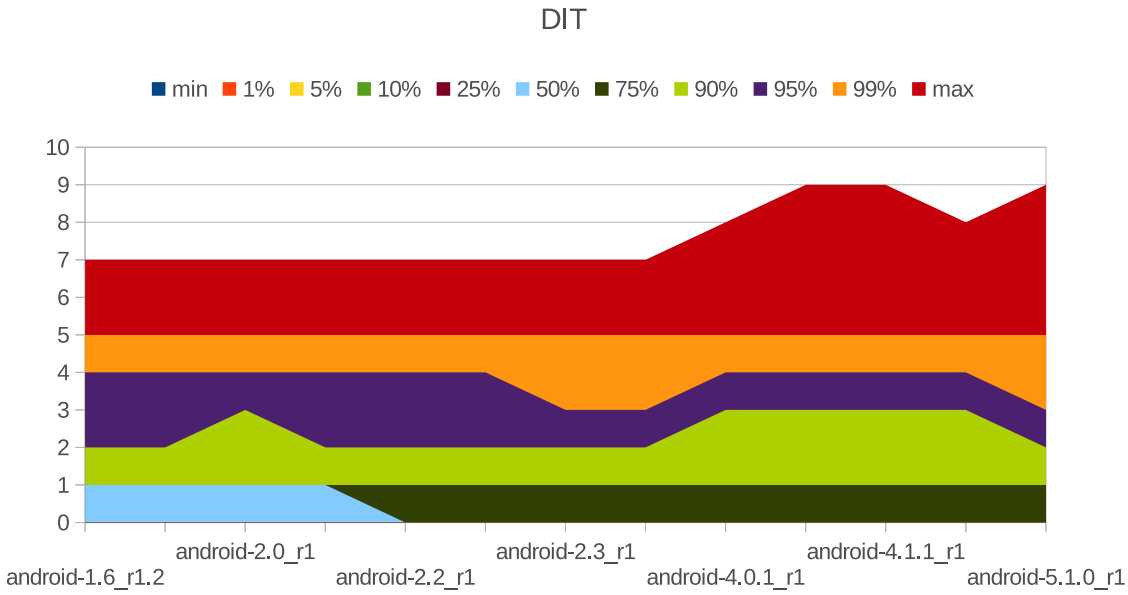

Figura 8 – Evolução da métrica DIT ao longo das versões da API

As Figuras 8 e 9 demonstram o comportamento dessas métricas com a evolução do sistema. É possível perceber que são valores relativamente baixos e com variação muito pequena para as duas métricas. Para DIT podemos perceber que 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.

version	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 – *Depth of Inheritance Tree* no Android

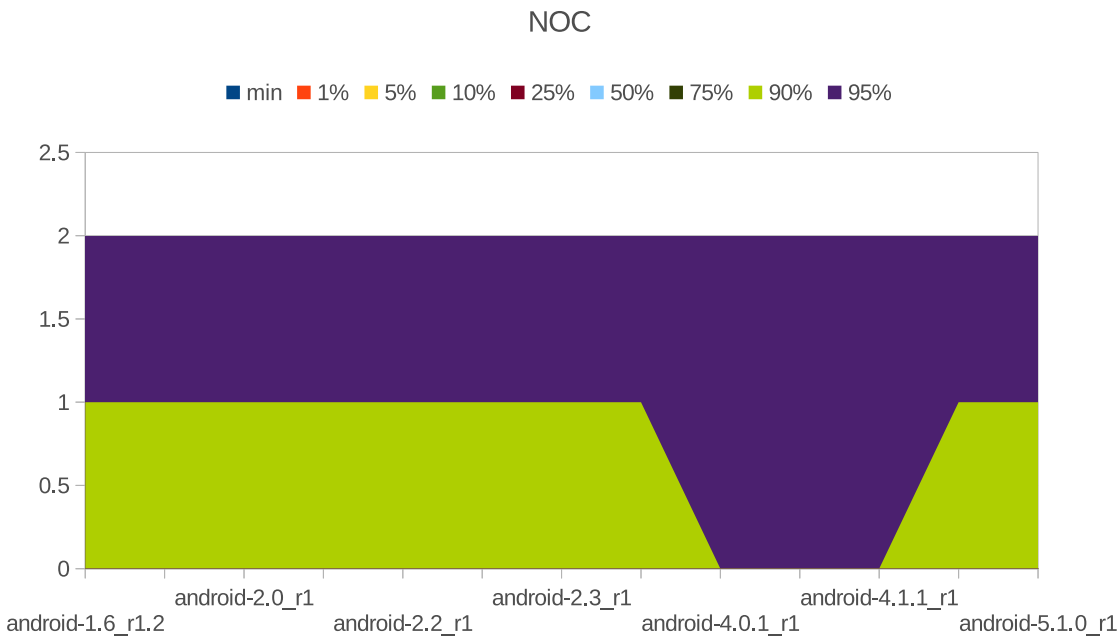

Figura 9 – Evolução da métrica NOC ao longo das versões da API

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 – *Depth of Inheritance Tree* nos aplicativos nativos

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 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. Tirando esse fato não temos muitas surpresas, 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 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.

version	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 – *Number of Children* no Android

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.

A Tabela 10 apresenta os valores da métrica NOC para o sistema Android. Percebe-se que 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.

A tabela 11 demonstra 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 a 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

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 – *Number of Children* nos aplicativos nativos

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.

5.1.6 Lack of Cohesion in Methods

A Figura 10 apresenta a continuidade dos valores para LCOM na API do android, que, mesmo quado 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.

A Tabela 12 demonstra 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.

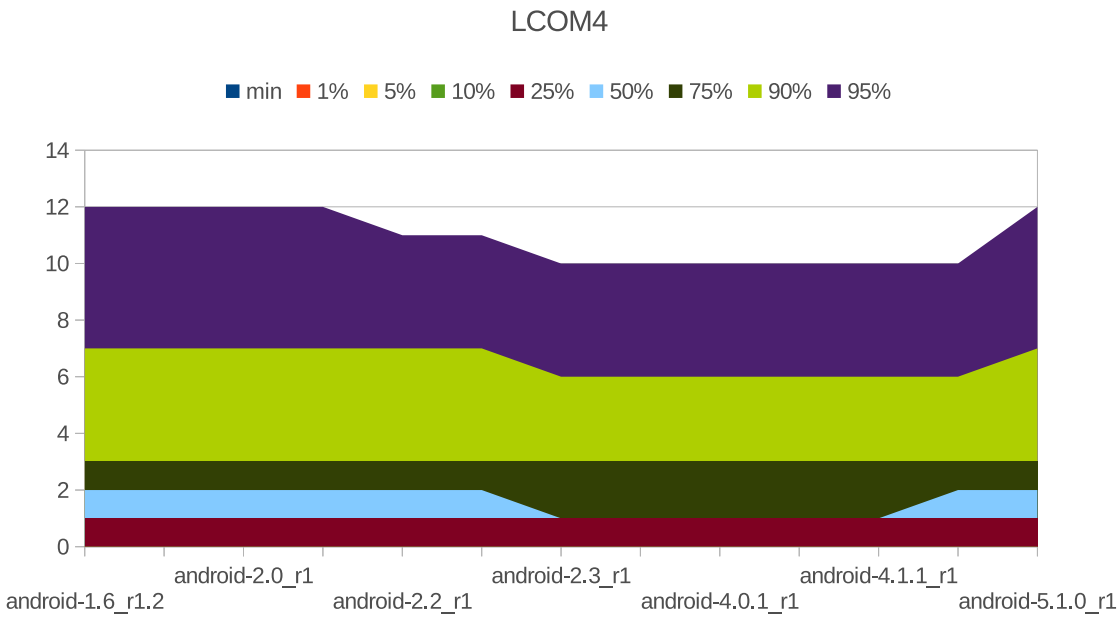

Figura 10 – Evolução da métrica LCOM4 ao longo das versões da API

version	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 – *Lack of Cohesion in Methods* no Android

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.

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 – *Lack of Cohesion in Methods* nos aplicativos nativos

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. Como pode ser visto na Tabela 13, os intervalos 0 a 4, 4 a 7, e 7 a 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.

5.1.7 Afferent Connections per Class

A Figura 11 e a Tabela 14 demonstram 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.

version	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 – *Afferent Connections per Class* no Android

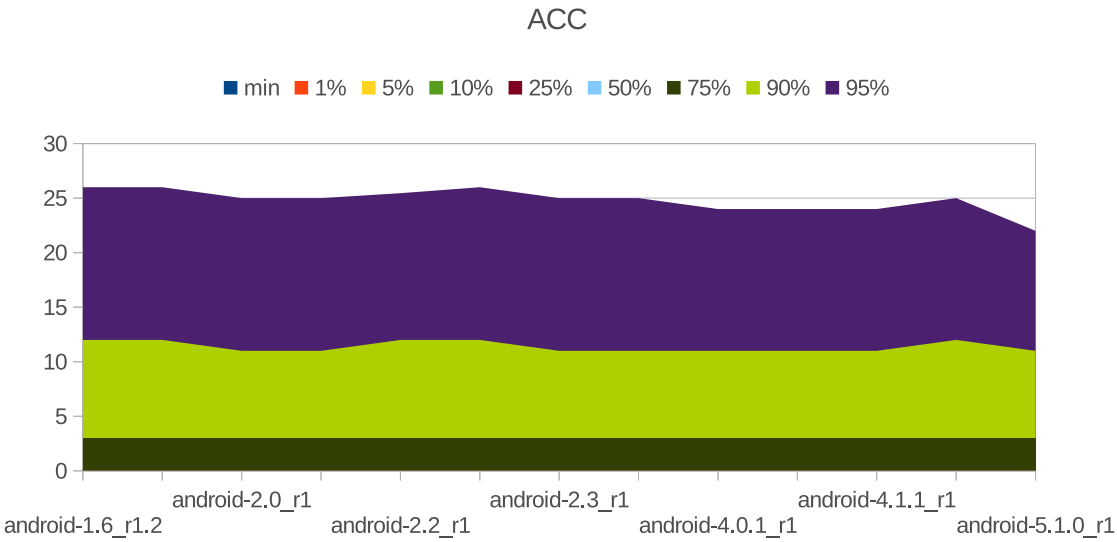

Figura 11 – Evolução da métrica ACC ao longo das versões da API

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 podemos ver na 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 podem ser vistos ao longo das versões. Como os intervalos encontrados são constantes ao longo das versões, parecem refletir o design do sistema e podem ser 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.

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 – *Afferent Connections per Class* nos aplicativos nativos

Classes de modelo tendem a ser utilizadas como variável de instância em diversos lugares em um aplicativo, e portanto podem ter valores de ACC mais altos. Verificou-se que 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.

5.1.8 Coupling Factor

O valor de COF para a API do Android ao longo de suas versões caiu de 0.13% para 0.057%, como pode ser visto na Figura 12 e na Tabela 16. Isso se dá pelo fato de que, como demonstrado na seção anterior, o valor de ACC permanece relativamente constante enquanto o número de classes aumenta significativamente a cada versão. Essencialmente, o sistema se mostra cada vez mais desacoplado e consequentemente manutenível.

version	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 – *Coupling Factor* no Android

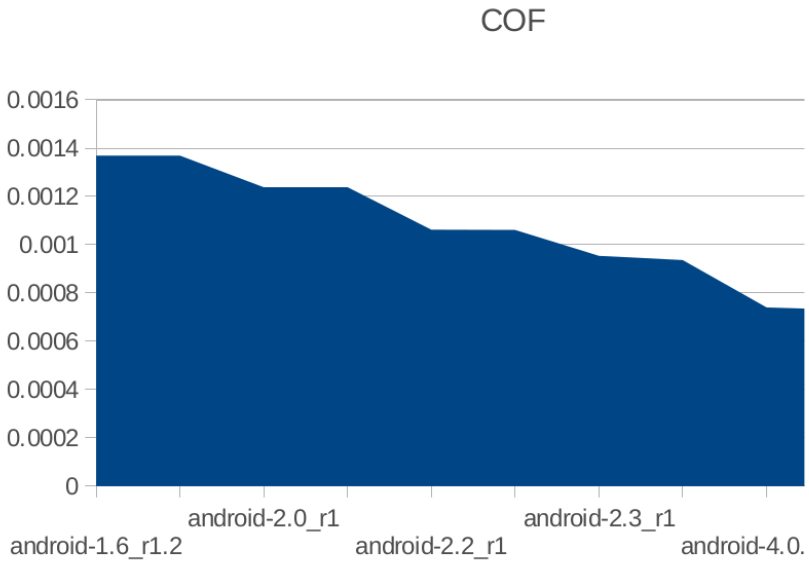

Figura 12 – Evolução da métrica COF ao longo das versões da API

A Tabela 17 apresenta os valores para os aplicativos e nela é possível perceber que o valor de 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, tivemos 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. Podemos perceber que os valores para o sistema Android estão bem abaixo desses valores, sendo considerados então excelentes valores para o fator de acoplamento. Já os aplicativos demonstraram que se mantém nos valores bons quando passam a ter por volta de 200 classes, mas quado muito pequenos se mostraram em sua maioria regulares nessa escala.

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

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 – *Coupling Factor* nos aplicativos nativos

- Espera-se que o valor de COF para o sistema se mostre cada vez menor 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%.

5.1.9 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, como explicado no Capítulo 3 e em outras seções deste capítulo, 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

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

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. As vezes dividir o comportamento em tarefas menores possa ser uma solução viável. Dividir o comportamento de um método em 2 provavelmente vai acarretar no aumento da métrica

RFC, mas mais uma vez ressalto que o valor de RFC não deve ser analisado por si só. É importante ficar de olho também em métricas como a LCOM quando fazendo esse tipo de refatoração, pois as 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 ultimo 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 pode auxiliar 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, como será discutido nas seções seguintes. 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.

5.2 Validação dos intervalos de referência

Após a obtenção e analise 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.

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 API com os próprios dados crus das métricas, e então sera possível comparar esses resultados com os valores definidos manualmente através dos percentis, discutidos na primeira seção deste capítulo.

5.2.1 Validação por regressão

A seguir métricas base que foram inicialmente idealizadas para relativização das métricas por regressão polinomial:

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, apenas foi realizado uma discussão 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.

5.2.1.1 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, uma vez que o modelo de regressão não veria nenhum nesse intervalo.

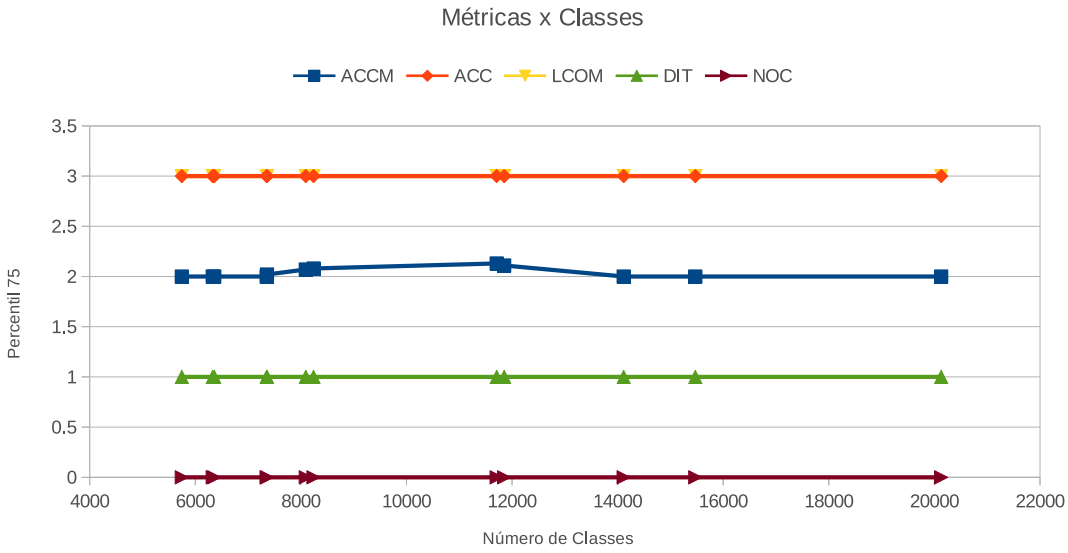

Figura 13 – Percentil 75 das métricas DIT, NOC, LCOM, ACC e ACCM

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.

Como podemos ver na Figura 13, onde é traçado o percentil 75 em função do número de classes, 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 a grande maioria dos dados. Nas tabelas e gráficos apresentados nas seção anterior pode ser confirmado 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 quantidade absurda de gráficos, além de

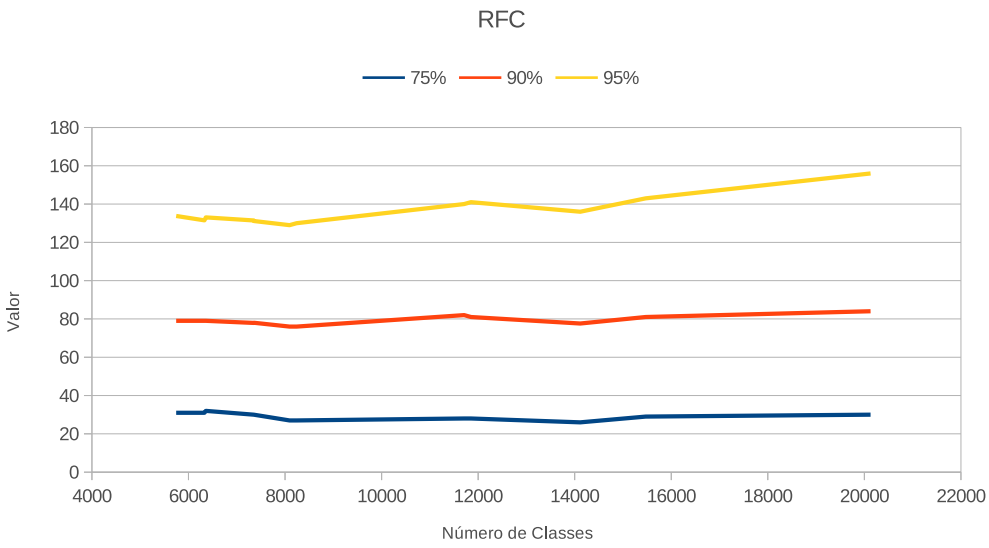

Figura 14 – Percentil 75, 90 e 95 de RFC em função do número de classes

não terem tanta importância quanto os valores muito frequêntes apresentados no percentil 75.

Para algumas métricas essa regressão parece ser possível, 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. Em suma, temos poucas amostras para realizar esse estudo.

Como um pequeno problema adicional, os dados das métricas para os aplicativos se mostraram levemente diferentes para os dados da API do sistema, mesmo estando bem 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. Para os aplicativos, que são projetos distintos, as métricas são extremamente dispersas em relação ao número de classes, tornando difícil representar bem os dados por um polinômio para realizar a regressão.

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

OO.

5.2.1.2 Regressão em escopo de classe

Como uma segunda tentativa na direção de uma regressão polinomial, 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 de alguma forma seu tamanho.

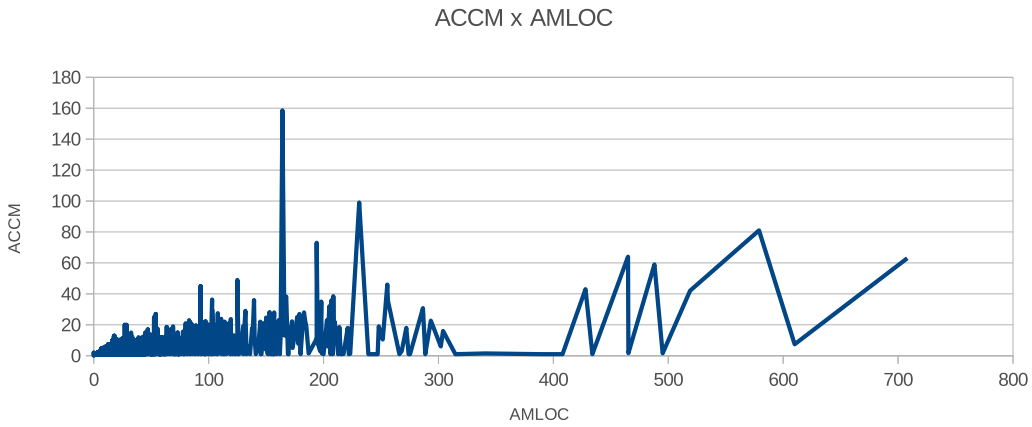

Figura 15 – ACCM em função de AMLOC na API versão 5.1.0

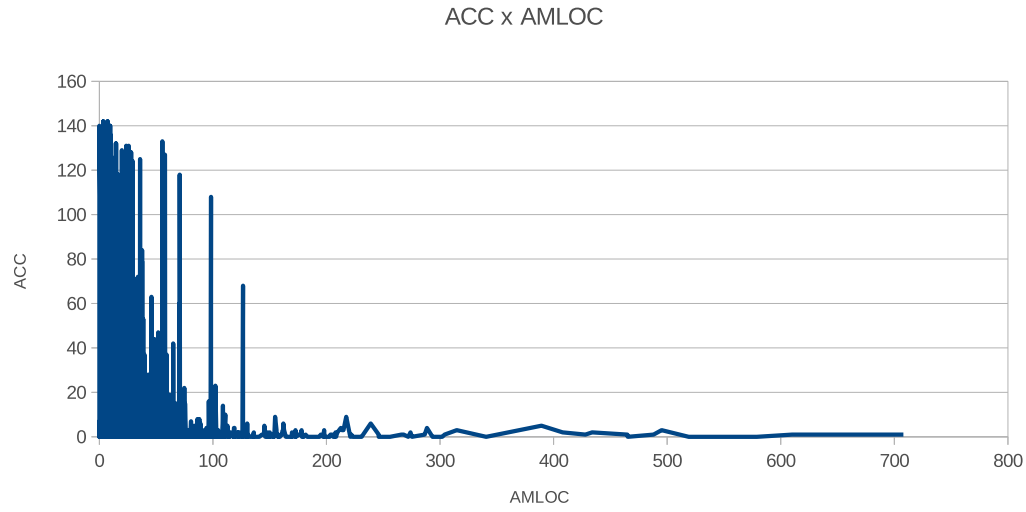

Figura 16 – ACC em função de AMLOC na API versão 5.1.0

Cada classe dentro de cada versão seria uma amostra para esse método, que teria então um total de mais de 100000 classes. Pensando 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 pode

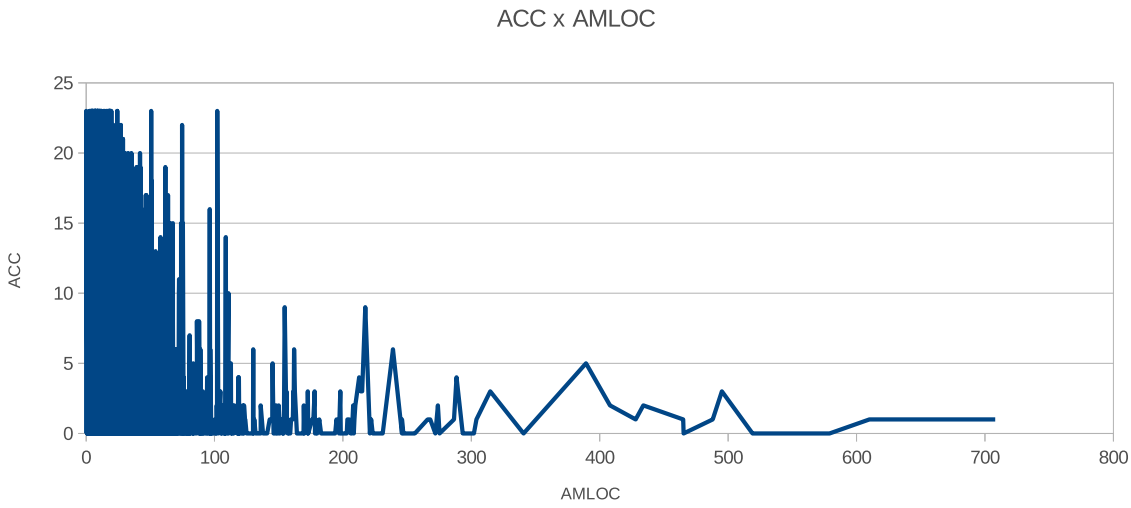

Figura 17 – ACC em função de AMLOC na API versão 5.1.0 com 95% dos dados

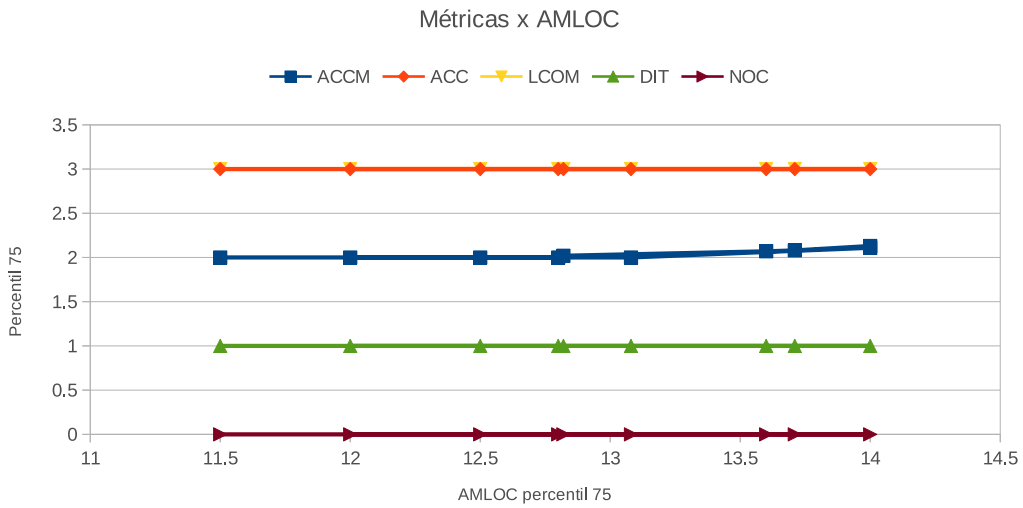

Figura 18 – DIT, NOC, LCOM, ACC e ACCM em função de AMLOC

ser visto nas Figuras 15 e 16, os valores de ACC e ACCM parecem oscilar 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.

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

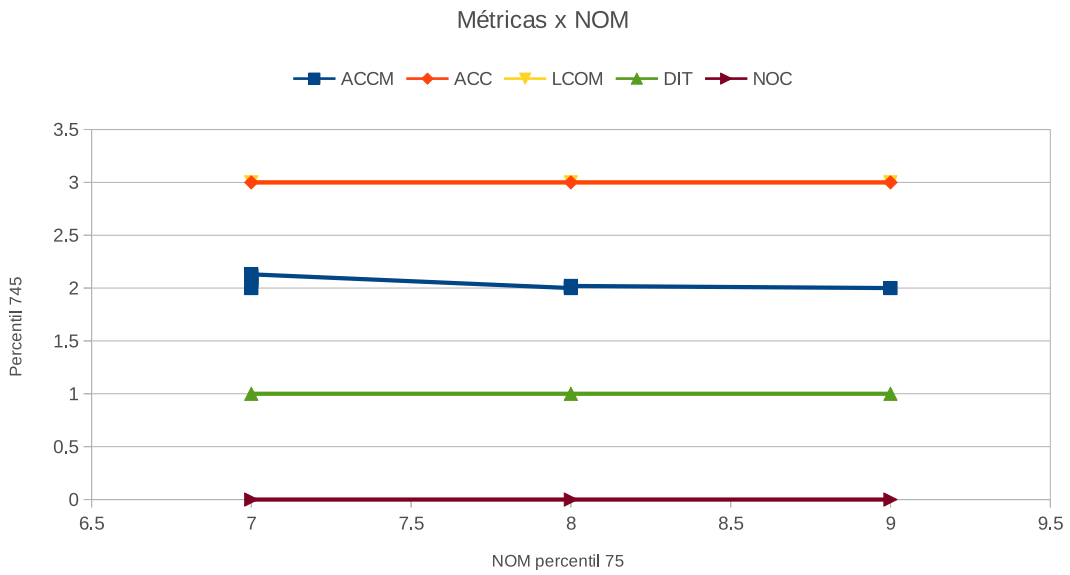

Figura 19 – DIT, NOC, LCOM, ACC e ACCM em função de NOM

AMLOC como já discutido na seção anterior. Na verdade, apenas pelo fato de essas métricas não variarem muito ao longo das versões, já se pode prever esse resultado.

5.2.1.3 Regressão em escopo de pacote

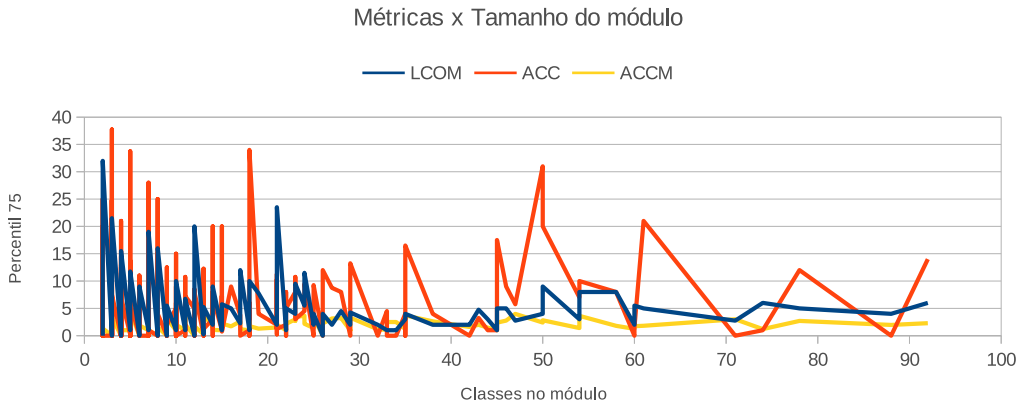

Figura 20 – LCOM, ACC e ACCM em função do tamanho de módulo

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 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, o mesmo problema de independência dos dados em relação a variável de tamanho foi encontrado aqui.

O gráfico da Figura 20 demonstra as métricas LCOM, ACC e ACCM para cada pacote em função dos seu número de classes. Podemos perceber que 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 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.

5.3 Comparação de similaridade com a API

Como foi comentado na seção anterior, as métricas OO não são dependentes de métricas de tamanho. Não foi encontrado 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, propomos então 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 para o projeto como um todo. 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.

5.3.1 Normalização de valores

Como as métricas geralmente se mostraram independentes, uma comparação direta pode ser bastante útil. Entretanto, obter um valor apenas de similaridade para a API não

é efetivo se as métricas tem diferentes grandezas de valores. Consequentemente temos que fazer um trabalho de normalização dos valores das métricas para deixar todas as métricas com um mesmo *range*, ou com grandezas equivalentes, e portanto ter uma certa equivalência, para calcular a distancia do app numa ideia semelhante ao que é feito com uma distancia euclidiana, onde todas as variáveis tem sua distancia 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 tão grandes quanto se queira. Dessa forma, não há um valor limite superior para os valores, embora o valor mínimo seja 0 pela própria forma de cálculo 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 do que é derivada de ACC, então não será utilizada no cálculo de distancia.

Para resolver este problema, foi considerado o fato de que termos um número grande de amostras de valores de métricas, com mais de 100 mil classes analisadas, podendo extrair os limites desses dados. Inicialmente pegamos então, para todas as métricas, o maior valor que a métrica apresentou, sem importar a versão onde esse valor apareceu. Como pode ser visto na 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, obtendo com uma divisão do valor por esse limite superior. Essa mesma forma de normalização pode ser feita para todas as métricas.

Entretanto essa abordagem pode ser um pouco problemática em algumas métricas. Por exemplo, para DIT, o valor máximo foi 9 para todas as mais de 100 mil amostras. A métrica tem uma variação muito pequena dentro dos percentis utilizados (de 1 a 4), então a variação de 1 para 4 é uma distancia de 0,33. Para a métrica ACCM que geralmente tem grandezas semelhantes nesses percentis, como apresentam os valores da Tabela 4, variando de 2 a 6, teria uma distancia de 0,025. Dessa forma a normalização não iria deixar valores equivalentes para a distância de cada métrica, tornando enviesado o cálculo da distância total do projeto.

É importante perceber 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 distancias 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 distancias 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 pode ser um valor em qualquer grandeza, e 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, podemos 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 comentado, a grandeza não tem significado semântico. 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, podemos remover NOC desse cálculo sem muitas perdas.

Dessa forma as distancias 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 assim.

5.3.2 Cálculo de similaridade

Inicialmente pensou-se em calcular distancias 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 bem menores 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, podendo fazer uma comparação entre ambos. É importante perceber 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 distancias, 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 tentar 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 perceber que os percentis tem diferentes representatividade. 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. 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:

$$\frac{\sum_{i}^{n} \frac{(Mapi_i - Mapp_i)}{Mapi_{75}} \cdot W_i}{95} \tag{5.1}$$

Onde:

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

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

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

Onde:

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

5.3.3 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 é multiplicado por 100 para levar os valores para uma grandeza maior e então truncamos as casas decimais, com o intuito de ter resultados melhor visualizados. A Tabela 19 apresenta os scores calculados com a Equação 5.2 para os aplicativos do sistema. Como podemos perceber, 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 podemos perceber 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.

metric	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

Quando esse calculo foi aplicado ao aplicativo parcial desenvolvido nas etapas iniciais desse projeto, com processo e arquitetura descritos no Apêndice B, os valores foram muito bons, como esperado. O aplicativo e-lastic obteve score -82, e contém valores teóricos melhores que os do sistema em todas as métricas, como mostra a Tabela 20. Isso nos dá subsídio para afirmar que a hipótese 4 (H4) definida no Capítulo 4 é verdadeira, isto é, as decisões tomadas ao longo do desenvolvimento do aplicativo com base em padrões de projeto estão 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.

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 então podem ser considerados como sendo de -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, podemos verificar 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 será sua validade enquanto essa premissa for verdadeira.

6 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.

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, em mais que algumas dezenas. Esse indicador de qualidade de código em comparação com a API parte da premissa de que a API tem uma boa arquitetura, e portanto terá sua validade enquanto essa premissa for verdadeira.

Em relação à questão de pesquisa e às hipóteses levantadas no Capítulo 4, 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 5, 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 5. 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 teóricas aplicadas no estudo de caso e-lastic estão relacionadas com 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 durante as etapas iniciais deste 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 do código do sistema?*

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.

6.1 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 do 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 5. 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 arbitrários e podem ser substituídos por um estudo mais detalhado e focado nesse propósito.

6.2 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.

Apêndices

APÊNDICE A – Android e a Engenharia de software

A.1 Design e Componentes Android

O Android provê um *framework* para desenvolver aplicativos baseado nos componentes descritos no início deste capítulo. Os aplicativos são construídos com qualquer combinação desses componentes, que podem ser utilizados individualmente, sem a presença dos demais. Cada um dos componentes pode ser uma entrada para o aplicativo sendo desenvolvido.

A comunicação direta entre os componentes de cada aplicativo é feita por meio do *Binder*[1], mecanismo de comunicação entre processos. O *Binder* comunica processos através da troca de mensagens (chamadas *parcels*), que podem referenciar dados primitivos e objetos da API assim como referencias para outros objetos *binder*. De forma geral, um *service* no Android pode ter sua interface definida em AIDL (*Android Interface Definition Language*), e uma aplicação que tiver referencia para o *binder* desse *service* pode executar chamadas de procedimento remoto (*RPC - remote procedure calls*) para qualquer método definido nessa interface AIDL de forma síncrona. Embora o *binder* apresente acesso direto para alguns componentes, essa comunicação pode ser feita de forma indireta utilizando *Intents*. Como descrito neste capítulo, Intents são geralmente recebidos por *receivers* que estão registrados para recebê-los. Esse registro é feito por meio de *Intent Filters*. Entretanto, *activities* e *services* também podem utilizar desse mecanismo para ser iniciados e finalizados. Quando um *Intent* é enviado ao sistema via *broadcast*, ele é recebido pelo sistema e resolvido pelo *Activity Manager Service* (AMS), que seleciona o melhor componente para tratá-lo, e então inicia o componente que o recebeu independente da aplicação a que ele pertença. Assim como já descrito, permissões podem ser criadas para restringir essa comunicação, mas a principio qualquer componente pode receber um *intent* de qualquer outra. Embora essas formas de comunicação entre processos sejam recomendadas, o sistema Android também suporta mecanismos de comunicação padrões do Linux como *sockets* e *pipes* (HEUSER et al., 2014).

Ainda que os componentes sejam geralmente registrados no sistema através do *AndroidManifest.xml*, *BroadcastReceivers* podem ser registrados dinamicamente dentro do ciclo de vida da aplicação. Isso quer dizer que é possível criar um *receiver* projetado para funcionar apenas enquanto a aplicação estiver em execução. Esse *BroadcastReceiver*

[1] <http://developer.android.com/guide/components/bound-services.html>

então é registrado por linha de comando da API e desativado quando requisitado, criando uma janela de tempo onde se deseja que esse componente funcione.

De forma geral, para desenhar uma arquitetura para sistema Android deve-se levar em conta todos esses componentes e conhecer bem sua aplicabilidade e a comunicação entre os mesmos. Se for necessário ter algum processo em background independente de *feedback* do usuário, por exemplo, deve-se utilizar do componente *service*. Caso contrário, quando o usuário fechar a interface gráfica do aplicativo, o aplicativo terá sua execução pausada e seu estado salvo para retorno posterior, deixando de executar alguma tarefa que não deveria ter sido interrompida. Como um exemplo mais palpável, o aplicativo do facebook precisa necessariamente utilizar de um *service* para que, mesmo quando não estiver em foco no sistema, notificações de mensagens e atualizações de status cheguem na tela do usuário.

Embora pareça restringir o design de aplicativos, o uso desses componentes unifica a forma com que os aplicativos são desenvolvidos. A API do Android já disponibiliza acesso a vários recursos de hardware do sistema na forma de componentes, e assim como é possível utilizar os componentes do sistema, é possível utilizar componentes de qualquer outra aplicação da mesma maneira - se a permissão for concedida- , e criar os seus próprios componentes para uso de terceiros de forma padronizada. Em suma, é tão fácil usar componentes criados por alguma aplicação qualquer quanto componentes internos do sistema graças ao *framework* de desenvolvimento Android.

A.2 Interface de usuário

O design de interface deve ser realizado com o intuito de alcançar usuários com capacidades motoras restritas, e outras limitações como problemas de visão ou audição. Dependendo do público alvo do aplicativo, esses aspectos devem ser considerados.

Com o desafio de fazer uso do pequeno espaço de tela, o design da interface gráfica apresenta mais importância do que jamais teve no desenvolvimento de aplicativos móveis (WASSERMAN, 2010). Usuários estão sempre tentando acessar várias tarefas diferentes, e geralmente tem baixa tolerância a aplicativos instáveis ou não responsivos, mesmo que gratuitos (DEHLINGER; DIXON, 2011).

A base da interface gráfica de usuário no Android é construída em cima da classe *View*, sendo que todos os elementos visíveis na tela, e alguns não visíveis, são derivados dessa classe (BRAHLER, 2010). O sistema Android disponibiliza vários componentes gráficos como botões, caixas de texto, imagens, caixas de seleção de dados, calendário, componentes de mídia (áudio e vídeo), entre outros, e a interface gráfica é construída em cima desses componentes gráficos, que podem ser customizados para um comportamento ou aparência um pouco diferente se assim for desejado, estendendo e customizando suas

respectivas classes em Java. Existe um padrão de design[2] que é recomendado que seja seguido.

A interface gráfica do usuário é construída geralmente em formato XML utilizando esses componentes gráficos já disponibilizados na API. Esses arquivos XML são então carregados pela API em Java quando necessário, e podem ser editados dinamicamente através da API, em linguagem Java.

Todo projeto de aplicativo Android possui um arquivo em Java chamado *R* (*Resources*) autogerado que contém identificação dos recursos do aplicativo. Cada componente gráfico disponibilizado na API e utilizado nos arquivos XML pode ser identificado de qualquer local do aplicativo pelo seu ID atribuído no arquivo XML. Dessa forma, é possível localizar facilmente dentro de sua *Activity* cada componente para ser carregado, modificado e utilizado.

A.3 Construção

Para construção de aplicativos para a plataforma Android, o Google disponibiliza um kit de desenvolvimento de aplicativos chamado Android SDK [3] (*Software Development Kit*). Ele provê as bibliotecas da API e as ferramentas necessárias para construir, testar e debugar aplicativos. Android SDK está disponível para os sistemas operacionais Linux, MAC e Windows.

A ferramenta oficial para desenvolvimento de aplicativos Android é o *Android Studio* [4]. O Android SDK atualmente é baixado instalado juntamente com o instalador do Android Studio. O IDE Eclipse ainda pode ser utilizado para desenvolvimento juntamente com *plugin* ADT (*Android Development Tools*), que inclui as ferramentas de visualização de interface em XML, ferramentas de *debug* de código, de análise de arquitetura, níveis de hierarquia de componentes gráficos, testes de desempenho, acesso a memória flash e outros recursos do dispositivo via IDE, entre outras funcionalidades. O SDK deve ser baixado separado e corretamente configurado se a ferramenta escolhida para desenvolvimento for o Eclipse. Todas essas funcionalidades estão inclusas no Android Studio, que foi feito especificamente para construção de aplicativos Android.

Após fazer o download do *Android Studio*, basta fazer o download das APIs desejadas durante a própria instalação do mesmo e utilizar do recurso da própria IDE para criar um projeto Android [5], que já vem com uma estrutura pronta para ser trabalhada, separando e categorizando código fonte e outros recursos utilizados no desenvolvimento. O Eclipse IDE foi utilizado como padrão por um bom tempo antes do lançamento oficial

[2] <https://developer.android.com/design/get-started/principles.html>
[3] <https://developer.android.com/sdk>
[4] <https://developer.android.com/sdk/installing/studio.html>
[5] <https://developer.android.com/tools/projects/index.html>

do Android Studio, e também monta uma boa estrutura de projeto, embora diferente da utilizada pelo Android Studio.

É necessário especificar a versão alvo do sistema para o qual se está desenvolvendo. Para manter a compatibilidade, geralmente desenvolvedores utilizam como versão mínima a 2.3 ou anterior, embora para utilizar alguns recursos seja necessário utilizar uma API mínima mais recente. Todas as APIs utilizadas precisam ser baixadas pela própria ferramenta de download do SDK ou automaticamente na instalação da IDE. Para desenvolver para Android kitkat 4.4.2 com compatibilidade com Android 2.3 por exemplo, deve-se obter ambas APIs. Cada vez mais as versões mais antigas estão deixando de ser utilizadas. Desenvolvedores de versões alternativas do Android como o Cyanogemod, vem dando suporte e oportunidade de update da plataforma para dispositivos cujo suporte para atualizações já foi abandonado pelo próprio fabricante, ajudando a fazer com que mesmo usuários mais antigos possam utilizar versões mais recentes do sistema Android. A grande maioria dos dispositivos em 2015 já utiliza API maior que a 15 segundo a própria ferramenta de desenvolvimento.

Juntamente com o SDK, o desenvolvedor tem acesso a uma ferramenta de criação de dispositivos virtuais AVD (*Android Virtual Devices*) para poder executar as aplicações sendo desenvolvidas sem a necessidade de um dispositivo físico disponível para esse fim. Entretanto vários recursos do sistema não podem ser utilizados por dispositivos virtuais, como por exemplo o GPS e o bluetooth. Nesses casos, é necessário um dispositivo físico para o desenvolvimento de aplicativos. Qualquer dispositivo pode ser utilizado, desde que a versão Android que ele apresente seja compatível com o aplicativo sendo desenvolvido e tenha ativado no sistema as opções de depuração USB.

Para o desenvolvimento para o AOSP, é necessário um dispositivo com *bootloader* desbloqueável, de forma que seja possível fazer o flash das imagens geradas pela compilação do sistema no dispositivo físico. Os dispositivos Nexus distribuídos pela própria Google são os mais recomendados para desenvolvedores e contribuidores para o código do sistema, uma vez que todos tem seu *bootloader* desbloqueado e podem ser modificados mais facilmente. Muitos dispositivos, embora não todos, podem ser utilizados para esse fim, desde que tenham *bootloader* desbloqueado. É importante ressaltar que fazer *flash* das imagens do sistema para um dispositivo físico requer que o mesmo seja restaurado para configuração de fábrica, removendo todos os dados presentes no mesmo, e resultará na perda de outras versões anteriormente instaladas do sistema. É preciso ter bastante cuidado para não danificar o dispositivo físico tentando fazer instalação de outras versões do sistema sem tomar as precauções necessárias.

Quando se desenvolvendo para o AOSP, é mais difícil ter uma visão total do software sendo modificado, pois o código fonte do sistema é muito extenso. Para carregar todo o código da api Java na memória da IDE Eclipse por exemplo, é necessário fazer

uma alteração nas configurações da ferramenta para que ela utilize mais memória (talvez alguns GB de RAM sejam necessários). Muitas vezes a melhor solução é fazer modificações isoladas em arquivos distintos em algum editor de texto qualquer sem ter o recurso de compilação automática das IDEs que ajudam a identificar erros em tempo de codificação. Nesses casos é necessária uma maior experiência do desenvolvedor para uma modificação consciente e cuidadosa nos arquivos de código fonte do sistema operacional.

A.4 Testes

Testes de software consistem em verificar se o produto de software se comporta da forma esperada em um determinado conjunto de casos específicos, selecionados com o intuito de representar o maior número de situações diferentes que podem ocorrer durante o uso do sistema, com o software em execução. Os testes têm que ser projetados para checar se o software está de acordo com as necessidades do usuário, procedimento conhecido como validação, e para verificar se as funcionalidades estão de acordo com a especificação, procedimento conhecido como verificação (IEEE, 2014). Testes podem ser realizados em vários níveis, desde o teste de pequenos trechos de código até a interação entre componentes e o teste da interface gráfica do usuário.

Existem vários tipos de testes aplicáveis a determinados tipos de sistema. De acordo com as necessidades e o ambiente onde o sistema irá funcionar, vários testes podem ser ou não necessários para garantir o funcionamento do sistema sob diversas condições. Sistemas web podem exigir testes de carga e stress para avaliar a quantidade de usuários simultâneos suportados, por exemplo. Sistemas críticos já também necessitam de testes de recuperação, para avaliar a capacidade do sistema de manter ou restaurar seu funcionamento após algum tipo de falta.

Ter uma boa suíte de testes é de grande valia para a manutenção de um produto de software, uma vez que sempre que uma modificação precisar ser feita no sistema, é possível verificar de forma automatizada se algum comportamento foi indevidamente alterado pela modificação realizada.

O sistema Android provê um *framework* para testes[6] , com várias ferramentas que ajudam a testar o aplicativo sendo desenvolvido em vários níveis, desde testes unitários a testes relacionados ao *framework* de desenvolvimento e a testes de interface de usuário. Todas as ferramentas necessárias para utilizar a API de testes disponível no *framework* de desenvolvimento Android são disponibilizadas juntamente com o Android SDK.

Podem existir classes em Java puro que não utilizam a API de desenvolvimento do Android. Conseguir fazer essa separação de classes que utilizam o *framework* e classes em Java puro pode significar uma maior facilidade em testar determinadas partes da

[6] <http://developer.android.com/tools/testing/testing_android.html>

aplicação, uma vez que podem ser diretamente utilizados os já bem difundidos *JUnit tests* para essas classes. Entretanto, muitas classes são construídas através dos componentes, e o funcionamento das mesmas também precisa ser testado.

As suítes de teste no Android são baseadas em *JUnit*, e então da mesma forma que é possível utilizar *JUnit* para desenvolver testes para classes que não utilizam a API Android, é possível utilizar as extensões do *JUnit* criadas especificamente para testar cada componente do aplicativo sendo desenvolvido. Existem extensões *JUnit* específicas para cada componente Android, e essas classes contém métodos auxiliares para criar *Mock Objects*. Estes são criados para simular outros objetos do contexto de execução real do aplicativo.

O kit de desenvolvimento para a plataforma Android inclui ferramentas automatizadas de teste de interface gráfica. O robotium, por exemplo, realiza testes de caixa preta em cima da interface gráfica do aplicativo. São criadas rotinas de teste em Java semelhantes ao *JUnit*, com *asserts* para validar os resultados. Com ele é possível criar rotinas robustas de testes para validar critérios de aceitação pré-definidos, simulando interações do usuário com uma ou várias *activities*. Existe a ferramenta Monkeyrunner, onde se cria *scripts* em Python para instalar e executar algum aplicativo, enviando comandos e cliques para o mesmo, e salvando *screenshots* do dispositivo no computador com resultados. Há também a ferramenta Monkey, que é utilizada para fazer testes de stress no aplicativo gerando inputs pseudo aleatórios.

APÊNDICE B – Estudo de Caso

B.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. Geralmente o controle de sobrecarga gerada por elementos elásticos é baseado na percepção subjetiva do esforço, com base na sensação de fadiga experimentada durante o exercício, e portanto o praticante não têm controle do esforço aplicado no exercício.

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. O sensor de força será fornecido calibrado juntamente com a central de processamento. 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 desenvolvido para um dispositivo móvel. 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.

Para a utilização do aparelho, o usuário deve selecionar o implemento elástico mais adequado ao exercício que será praticado, com resistência elástica compatível com o esforço que será aplicado. Um sensor de força compatível é acoplado ao implemento elástico e ligado ao módulo de processamento por uma entrada específica. O usuário então seleciona via aplicativo móvel um dos três modos de execução de exercício: dinâmico, isométrico e potência. Seleciona em seguida a configuração da sua rotina de treino, com os parâmetros de controle para o exercício específico. A partir daí o exercício pode ser iniciado, e informações precisas de controle são apresentadas na tela do dispositivo móvel em tempo real. Vibração do dispositivo, estímulos sonoros e outros tipos de sinais podem ser utilizados para dar *biofeedback* ao usuário sobre o exercício em curso. Após a realização do exercício, é apresentado um resumo da seção de treino executada. É possível que se apresente um histórico de treinamento dos indivíduos que realizam suas atividades utilizando o e-lastic.

Os parâmetros para controle podem variar de exercício para exercício, porém em

todos eles será selecionado um intervalo de intensidade, isto é, carga máxima e mínima em quilogramas, em que o exercício será trabalhado. As opções de exercícios são espécies de rotinas pré programadas, modos de utilização da resistência elástica para o exercício físico:

1. Modo dinâmico - Realização de movimentos cíclicos em velocidade lenta e cadência controlada. Os parâmetros de controle para esse exercício são número de séries de exercícios, número de repetições por série e a duração do descanso entre as séries.

2. Modo isométrico - Contrações estáticas em que o usuário mantém uma posição e sustenta a sobrecarga elástica nessa posição por um determinado tempo. Semelhante ao modo dinâmico, no modo isométrico o usuário seleciona os parâmetros de números de contrações realizadas (repetições do exercício), e duração do tempo de intervalo entre as séries, com adição da duração de cada contração, isto é, o tempo em que o esforço deve ser mantido para que uma repetição seja contabilizada.

3. Modo potência - Semelhante ao modo dinâmico, estes são exercícios cíclicos, porém com a maior velocidade possível. Os parâmetros para esse exercício são os mesmos do modo dinâmico, porém aqui a velocidade do movimento faz parte dos parâmetros de controle.

B.2 Desenvolvimento

O e-lastic começou como um produto desenvolvido para interfacear com usuário por modo de um dispositivo específico conectado ao sensor de força via cabo. A seleção dos exercícios e o próprio *feedback* para o usuário eram realizados por meio desse dispositivo, fazendo interação com o usuário por meio de um display LCD e um potenciômetro para seleção de parâmetros de exercício, e o *feedback* sendo feito por meio de LEDs e sinais sonoros, bem como o próprio display LCD. 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.

Por volta de 1 vez ao mês, foram realizadas reuniões com Fernanda e outros integrantes da equipe. Daqui em diante serão referenciados neste documento como *product owners*.

Os requisitos para o aplicativo foram a principio retirados do próprio comportamento dos primeiros protótipos. Esses comportamentos foram descritos e detalhados pelos *product owners*, que não só descreveram como os protótipos funcionavam, como também foi discutido como o aplicativo deveria se comportar de forma diferente dos mesmos, com adições de novas funcionalidades para o aplicativo sendo desenvolvido.

A ideia inicial do aplicativo, e foco principal do desenvolvimento da equipe ao longo desse semestre, foi fazer com que o aplicativo sendo desenvolvido para plataforma Android tenha no mínimo as mesmas funcionalidades dos primeiros protótipos.

O desenvolvimento do e-lastic app ao longo do semestre foi planejado e executado juntamente com a disciplina de Manutenção e Evolução de Software (MES). Desde o início do semestre, foi preparada uma equipe na disciplina para o desenvolvimento do aplicativo desde o início, de forma a preparar a base da arquitetura. Essa base será utilizada na segunda fase deste trabalho, quando o desenvolvimento será focado em alcançar todos os objetivos traçados nas reuniões com os *product owners*, que se resumem basicamente na construção e entrega do aplicativo completo e funcional.

Para este trabalho em conjunto com a disciplina de MES durante o semestre, o objetivo principal é conseguir uma base de arquitetura e alcançar dois tipos de exercícios com funcionalidade implementada: exercício dinâmico e isométrico. A interface gráfica foi deixada em segundo plano durante esse período de desenvolvimento, com o foco voltado então em uma base estável e manutenível com as funcionalidades básicas implementadas e com seus respectivos testes.

B.2.1 Ciclo de desenvolvimento

Durante o desenvolvimento, foram utilizadas práticas ágeis retiradas do Scrum. Embora o Scrum não esteja no escopo deste trabalho, informações sobre o mesmo podem ser obtidas em Schwaber e Beedle (2002).

Devido ao ritmo de trabalho mais lento proporcionado pelo tempo curto da disciplina de MES, as *sprints* foram planejadas para durar 15 dias (2 semanas). Inicialmente foram criadas várias histórias de usuário (*user stories*) e montado um *backlog* considerando a reunião inicial com os *product owners*. Novas histórias surgiam, bem como histórias iam sendo atualizadas conforme os requisitos ficavam mais claros com as reuniões subsequentes. A cada 2 semanas então eram consideradas as histórias de usuário do *backlog* e priorizadas para a iniciar a *sprint*. Nas medidas utilizadas pela equipe, com pontuação de 1 a 5 para cada história utilizando técnica do *planning poker*, foi implementada uma média de 7 *story points* por *sprint*. É importante ressaltar que histórias não completadas não entram nem parcialmente nesse cálculo e são jogadas para as *sprints* seguintes, e portanto para uma equipe inexperiente em constante aprendizado é possível inferir que mais esforço foi dedicado do que o próprio *velocity* pode indicar, representando a curva de aprendizado da equipe. Muito do esforço do principal autor deste trabalho foi direcionado ao aprendizado da equipe para alavancar o desenvolvimento do aplicativo.

Todo trabalho realizado foi feito utilizando técnica de programação em pares. Um quadro mapeando os integrantes que já trabalharam juntos esteve em constante atua-

lização para que a equipe circulasse da melhor forma possível. O principal autor deste trabalho participou do desenvolvimento dessa forma várias vezes ao longo do semestre, mesmo esse não sendo o objetivo principal, com o intuito de compartilhar de parte do conhecimento da plataforma obtido em projeto com parceria UnB-Positivo de desenvolvimento de aplicativos Android, bem como conhecimento adquirido em estágio no ICRI-SC (*Intel Collaborative Research Institute for Security Computing*), focado em segurança móvel.

O código sendo desenvolvido se encontra disponível em um repositório aberto[1] para qualquer desenvolvedor através da plataforma gitlab.com, e a equipe de alunos de MES não tem acesso direto de escrita para esse repositório. Para contribuir para o projeto, a equipe trabalha em uma cópia do repositório original (*fork*). Quando alcança algo significativo para o projeto, a equipe então cria um *merge request* pela própria plataforma gitlab, que é então avaliado e aceito pelos revisores, a exemplo do principal autor deste trabalho, ou rejeitado e mudanças são solicitadas. O mesmo procedimento pode ser realizado por qualquer desenvolvedor que deseja contribuir com o projeto. Como já dito, o software sendo desenvolvido é livre e portanto tem código aberto disponibilizado online. A única parte do produto e-lastic que será comercializada é o componente físico, com o módulo central de processamento, sensor de força e implemento elástico.

B.2.2 Equipe

Composta por 5 alunos da disciplina de MES, a equipe contém apenas graduandos em Engenharia de Software da Universidade de Brasília campus UnB Gama. Todos eles tem um tempo a dedicar ao desenvolvimento correspondente ao tempo de uma disciplina de 4 créditos, com 4 horas-aula e mais 4 horas semanais adicionais por integrante, totalizando 8 horas por semana de dedicação para cada um dos integrantes da equipe.

Todos os integrantes não tinham a priori nenhuma experiência no desenvolvimento de aplicativos para a plataforma Android, porém todos eles tinham algum conhecimento em linguagem de programação Java, necessária ao desenvolvimento, bem como experiência em atividades de verificação e validação, banco de dados, entre outras áreas. Todo esse conhecimento foi adquirido ao longo do curso de graduação em Engenharia de Software que está em andamento, provendo o background necessário para que eles pudessem trabalhar neste projeto durante o semestre. A equipe também apresentou conhecimento suficiente em gerência de configuração e controle de versionamento, o que ajudou significativamente em relação a manutenção do repositório do projeto.

Um dos integrantes da equipe faz o papel de *coach*, coordenando o restante da equipe para alcançar os objetivos de cada *sprint* e tomando frente da organização da equipe ao longo dos ciclos de desenvolvimento. Embora tenha essa responsabilidade adi-

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

cional, esse integrante participava das atividades de desenvolvimento tanto quanto os demais, e foi selecionado para esse papel pelo professor da disciplina, orientador deste trabalho, por ter mais experiência que os demais em metodologia ágil de desenvolvimento de software.

O principal papel exercido pelo principal autor deste trabalho ao longo do desenvolvimento foi em relação ao estabelecimento da arquitetura do aplicativo sendo desenvolvido, visto que a equipe não tinha experiência com componentes do Android e não conhecia a própria estrutura de um aplicativo para essa plataforma. Dessa forma, ele ficou responsável por criar a estrutura base para o desenvolvimento do aplicativo, bem como revisar todos os códigos submetidos para o repositório principal, e solicitar modificações caso não estivessem de acordo com a arquitetura base do repositório principal, ou não estivesse com qualidade aceitável. Como o autor participou ativamente do desenvolvimento, não houve muita surpresa nos *merge requests* e portanto poucos foram rejeitados.

Durante as aulas da disciplina de MES, toda a equipe esteve presente com o principal autor deste documento, onde era aproveitado o tempo em conjunto para a disseminação do conhecimento sobre a plataforma. Por várias vezes foram feitos DOJOs de treinamento para que o conhecimento circulasse dentro da equipe. Com o mesmo objetivo, a equipe sempre trabalhou em revezamento de pares e de tarefas. Uma dupla que trabalhou com um módulo específico durante uma semana passava a trabalhar em outro na semana seguinte, ou mesmo em atividades de teste, com outra dupla assumindo o trabalho da anterior. Os demais horários da equipe variam de acordo com a semana e com a dupla que trabalhará na semana. As duplas sempre alternavam, e os horários para cada dupla dependem da disponibilidade dos integrantes. Nesses horários de desenvolvimento, o autor deste trabalho não participou de forma presencial e não houve participação de toda a equipe reunida, cada dupla trabalhava em uma parte separada da aplicação.

Como os conhecimentos em desenvolvimento Android foram adquiridos ao longo do desenvolvimento, a primeira *sprint* teve menor *velocity* que as demais.

Embora seu principal papel fosse como arquiteto e revisor de código, o autor deste trabalho também participou da codificação do aplicativo, codificando e construindo boa parte dos principais componentes desenvolvidos ao longo do semestre. Um dos motivos para isso foi dar o impulso inicial que a equipe inexperiente na plataforma precisava para trabalhar de forma mais eficiente.

B.3 Estado da Arquitetura

As explicações neste tópico tem como objetivo principal tentar demonstrar o tipo de comunicação entre as partes da aplicação e a modularização da mesma, justificando as principais decisões acerca da arquitetura e demonstrando a preocupação da equipe

com manutenibilidade e princípios de design. Não serão explorados todos os detalhes da implementação parcial da aplicação desenvolvida para este trabalho.

O planejamento do desenvolvimento para a equipe da disciplina de MES foi focado na base da arquitetura, juntamente com a funcionalidade básica do aplicativo. Isso inclui o modo de exercício dinâmico e exercício isométrico com seus respectivos testes funcionais, e a comunicação com o microcontrolador via *Bluetooth* para o funcionamento dos exercícios. A interface gráfica de usuário foi deixada em segundo plano durante essa etapa do desenvolvimento, tendo apenas alguns componentes temporários apresentados na tela para ajudar na visualização da funcionalidade que foi desenvolvida. Ainda durante esse período de colaboração com a equipe de MES, também deve ser entregue a persistência dos parâmetros de execução dos exercícios, de forma que o usuário possa salvar suas preferências e carregá-las para reutilizar os mesmo parâmetros da próxima vez que praticar o exercício.

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, exitem os seguintes componentes e módulos principais:

- *BluetoothService* - Responsável pela conexão *bluetooth*, este componente é um *Android Service* que se inicia juntamente com a aplicação e fica a espera de uma solicitação de conexão vinda da tela de escolha de dispositivos. Quando o usuário solicita a conexão com o hardware e-lastic selecionando-o em uma lista, 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. Quando recebe um dado de força vindo do *bluetoothService*, o *exerciseService* informa ao exercício ativo os novos valores recebidos. Da mesma forma ele informa ao exercício sobre mudanças nos parâmetros vindas da interação com o usuário com o a tela do aplicativo.

- *BioFeedbackService* - Este módulo é responsável por gerar *biofeedback* para o usuário. Quando o valor de força 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. Por sua vez, este componente checa sua lista de *biofeedbacks* e aciona os que foram solicitados. Por exemplo, pode ser solicitado que o dispositivo móvel vibre quando os limites de força forem violados, como também pode ser requisitado que um som seja tocado. Essas configurações de

que tipo de *biofeedback* ativar estarão disponíveis para o usuário na versão final do aplicativo.

- *ExerciseActivity* - Representa a tela principal e a própria interação do usuário com os componentes gráficos. Este componente não será totalmente implementado neste período de desenvolvimento com parceria com a equipe de Manutenção e Evolução de software, e será trabalhado apenas na segunda etapa deste trabalho. Neste período, 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*. Como já citado em capítulos anteriores, o *intent* tem a função de comunicar componentes independente da aplicação a qual pertencem. Entretanto, neste aplicativo em desenvolvimento, 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* 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*. Simplificando, 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 co-

nhecer 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.

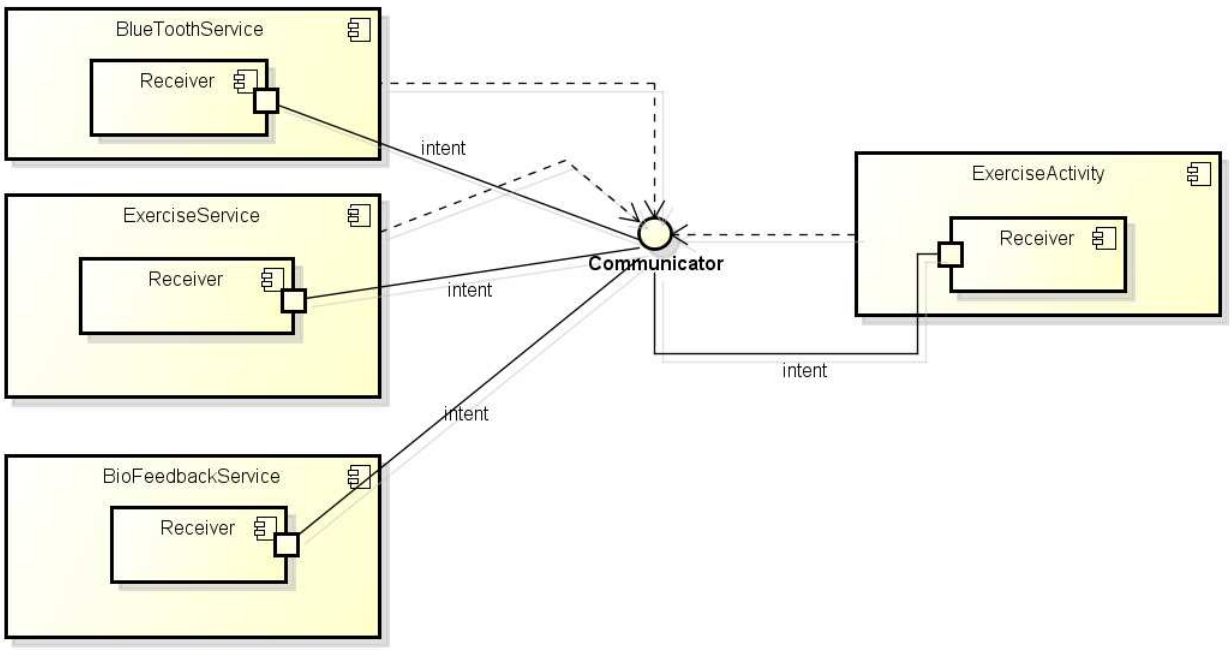

Figura 21 – Componentes e sua comunicação via classe *Communicator*

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.

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.

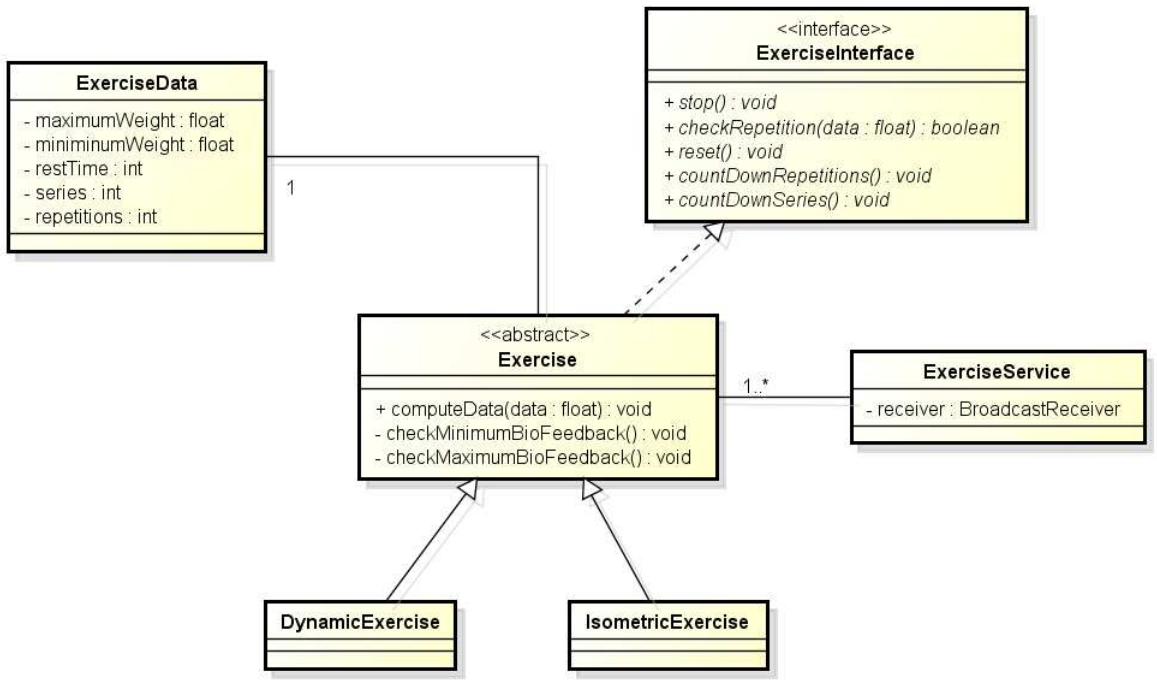

Figura 22 – Principais classes dentro do módulo de exercício e suas relações

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 a estrutura básica desse módulo.

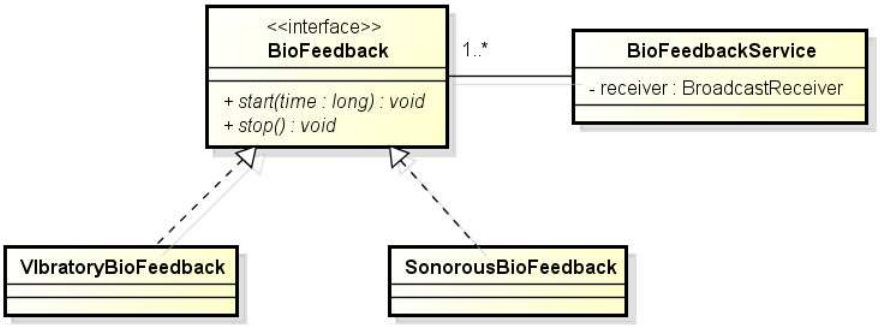

Figura 23 – Principais classes dentro do módulo de *biofeedback* e suas relações

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.

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 38 e 39.

ANDROIDDEVELOPERS. *Google guide to android development*. 2014. Disponível em: <http://developer.android.com/develop/index.html>. Citado na página 21.

BRAHLER, S. *Analysis of the Android Architecture*. 2010. Citado na página 96.

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 3 vezes nas páginas 34, 37 e 38.

DEHLINGER, J.; DIXON, J. Mobile application software engineering: Challenges and research directions. 2011. Disponível em: <http://www.mobileseworkshop.org/papers/7_Dehlinger_Dixon.pdf>. Citado 2 vezes nas páginas 28 e 96.

DEVOS, M. *Bionic vs Glibc Report*. 2014. Disponível em: <http://irati.eu/wp-content/uploads/2012/07/bionic_report.pdf>. Citado na página 21.

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 50, 65 e 71.

FONSECA, C. F. *História da Computação: O caminho do pensamento e da tecnologia*. [s.n.], 2007. Disponível em: <http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>. Citado na página 19.

GANDHEWAR, N.; SHEIKH, R. Google android: An emerging software platform for mobile devices. 2010. Disponível em: <http://www.enggjournals.com/ijcse/doc/003-IJCSESP24.pdf>. Citado na página 32.

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 44.

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 44.

HEUSER, S. et al. Asm: A programmable interface for extending android security. 2014. Disponível em: <http://www.icri-sc.org/fileadmin/user_upload/Group_TRUST/PubsPDF/heuser-sec14.pdf>. Citado 2 vezes nas páginas 22 e 95.

HITZ, M.; MONTAZERI, B. *Measuring coupling and cohesion in object-oriented systems*. [S.l.]: Citeseer, 1995. Citado na página 38.

HOLZER, A.; ONDRUS, J. Trends in mobile application development. 2009. Disponível em: <http://www.janondrus.com/wp-content/uploads/2009/07/2009-Holzer-BMMP.pdf>. Citado 2 vezes nas páginas 25 e 26.

IEEE. *Guide to the Software Engineering Body of Knowledge*. 2014. Disponível em: <http://www.computer.org/portal/web/swebok/swebokv3>. Citado 5 vezes nas páginas 25, 28, 29, 30 e 99.

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 44.

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 43.

MCCABE, T. J. A complexity measure. 1976. Disponível em: <http://www.computer.org/csdl/trans/ts/1976/04/01702388-abs.html>. Citado na página 34.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. São Paulo, 2013. Citado 13 vezes nas páginas 19, 33, 34, 50, 52, 53, 55, 58, 62, 63, 65, 68 e 81.

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 43.

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 34, 50, 55, 56, 58, 65, 68 e 83.

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 3 vezes nas páginas 19, 34 e 45.

SCHWABER, K.; BEEDLE, M. *Agile Sofware Development with Scrum*. [S.l.: s.n.], 2002. Citado na página 103.

SHANKER, A.; LAL, S. Android porting concepts. 2011. Disponível em: <http://p0-uni.googlecode.com/svn/trunk/p0-uni/Artikler/AndroidPortingConcepts.pdf>. Citado na página 25.

SHARMA, M. et al. A comparative study of static object oriented metrics. *International Journal of Advancements in Technology*, v. 3, n. 1, p. 25–34, 2012. Citado na página 35.

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 36.

SOKOLOVA, K.; LEMERCIER, M.; GARCIA, L. Android passive mvc: a novel architecture model for android application development. 2013. Disponível em: <http://www.thinkmind.org/download.php?articleid=patterns_2013_1_20_70039>. Citado na página 30.

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-Platforn pdf>. Citado na página 43.

TERCEIRO, A.; CHAVEZ, C. Structural complexity evolution in free software projects: A case study. 2009. Citado na página 50.

WASSERMAN, A. I. Software engineering issues for mobile application development. 2010. Disponível em: <http://repository.cmu.edu/cgi/viewcontent.cgi?article=1040&context=silicon_valley>. Citado 3 vezes nas páginas 26, 28 e 96.

WATSON, A. H.; MCCABE, T. J.; WALLACE, D. R. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, v. 500, n. 235, p. 1–114, 1996. Citado na página 37.

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 44.