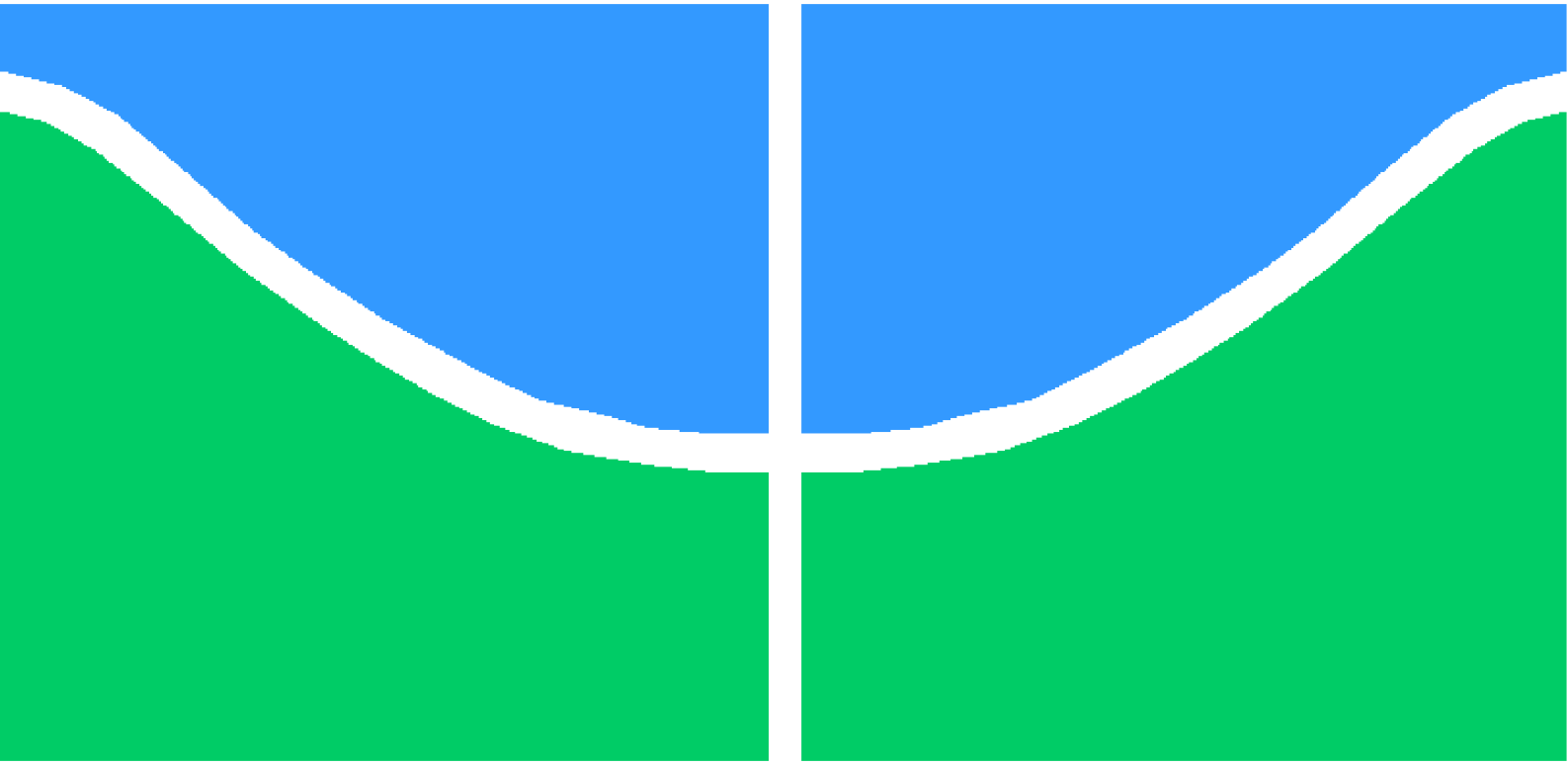

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Estudo da correlação entre métricas de código fonte do sistema Android e seus aplicativos

Autores: Marcos Ronaldo Pereira Júnior

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2014

Marcos Ronaldo Pereira Júnior

Estudo da correlação entre métricas de código fonte do 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

2014

Marcos Ronaldo Pereira Júnior

Estudo da correlação entre métricas de código fonte do sistema Android e seus aplicativos/ Marcos Ronaldo Pereira Júnior. – Brasília, DF, 2014-

46 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 , 2014.

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 da correlação entre métricas de código fonte do sistema Android e seus aplicativos

CDU

Marcos Ronaldo Pereira Júnior

Estudo da correlação entre métricas de código fonte do 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, Dezembro de 2014:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Prof. Marcelino Monteiro de Andrade
Convidado 1

Prof. Luiz Augusto Fontes Laranjeira
Convidado 2

Brasília, DF
2014

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 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 de qualidade para o desenvolvimento de aplicativos desenvolvidos para o Android. Traçar então um perfil que caracteriza o próprio sistema a partir de suas métricas de código fonte, assim como traçar esse perfil para os aplicativos do sistema e de terceiros e verificar o grau de aproximação que eles encontram em seu design.

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

Lista de ilustrações

1 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.

1.1 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.

1.2 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.

1.3 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.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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 3, 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 2.

1.4.4 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 4.

1.4.5 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.

2 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 em 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.

Neste trabalho, serão trabalhadas métricas estáticas de código fonte para avaliar a qualidade de um produto de software através de suas decisões arquiteturais, que são refletidas essencialmente nas métricas OO e métricas de complexidade. Métricas de tamanho serão utilizadas com o objetivo de relacionar as demais métricas de forma relativa em vez de uma comparação direta. Complexidade tem forte relação com o tamanho do código, e a necessidade de manter as duas é justificada para realizar análises mais adequadas e chegar a resultados mais consistentes.

2.1 Métricas utilizadas

Várias métricas podem ser utilizadas para avaliar quantidade de código fonte, como por exemplo *Lines of Code (LOC)* ou mesmo métricas de tamanho e volume de halstead. Manter o monitoramento de métricas de volume de código em conjunto com outras métricas é de extrema importância para que comparações sejam feitas para sistemas de tamanho semelhante. Comparar um pequeno applicativo de 2 classes com o código de um sistema operacional não apresenta resultados válidos. 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. LOC pode ser bastante representativa para qualidade de software e probabilidade de erros em código (GYIMOTHY; FERENC; SIKET,

2005), e é uma métrica extremamente simples e rápida de ser coletada. Essa é a melhor 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. Não será levada em consideração a comparação entre diversas métricas neste trabalho, baseando a escolha então totalmente na simplicidade e na ferramenta escolhida para coleta, devido a limitações de tempo e escopo.

R.Basili, Briand e Melo (1995) apresenta um estudo com métricas de Chidamber e Kemerer, 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. 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 facilidade em que o software pode ser testado e mantido. Métricas de complexidade ciclomática, introduzidas por McCabe (1976), foram criadas para avaliar a quantidade de caminhos que a execução do software pode seguir em sua execução, dando uma boa visão dos possíveis casos de teste para o software. Demonstrar que um software possui muitos caminhos paralelos de execução para serem testados pode demonstrar um grande esforço com testes, bem como um sistema bem difícil de manter com o passar do tempo. Da mesma forma, um software com baixa capacidade ciclomática indica um software que pode ser testado mais facilmente, podendo indicar um menor esforço futuro com possíveis modificações. Há outras métricas bem difundidas para avaliar a complexidade de software, como as métricas de Halstead que associam tamanho, volume, dificuldade e esforço. As métricas de halstead podem ser consideradas como métricas de tamanho, embora possam ser derivados vários valores com significados distintos das métrica básicas apresentadas. Essas métricas não são diretamente compatíveis pois medem coisas diferentes, mesmo usando “complexidade” como terminologia, e seus resultados levam a conclusões completamente distintas.

É importante ressaltar que as métricas discutidas neste capítulo, utilizadas durante todo o estudo, são coletadas de forma unificada pela ferramenta Analizo, o que foi um dos motivos de sua escolha.

2.1.1 Descrição das métricas

(listar e explicar cada uma)

3 Metodologia

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, traçar um perfil que caracteriza o próprio sistema a partir de suas métricas de código fonte, assim como traçar esse perfil para os aplicativos do sistema e verificar o grau de aproximação que ambos encontram em seu design.

Assumindo que o código dos aplicativos do sistema possui uma arquitetura ideal de aplicativos por ser mantido pela Google, bem como pela comunidade de desenvolvedores por ter código aberto, criar um fator de aproximação para aplicar a aplicativos em desenvolvimento para avaliar sua qualidade de acordo com a aproximação ao sistema e à aplicativos nativos, em termos de métricas estáticas de código. Esse fator de aproximação pode ser calculado utilizando métodos estatísticos ou aprendizado de máquina, de forma que possa ser avaliada a aproximação de uma nova amostra aos perfis existentes.

Caso o grau de aproximação de um aplicativo qualquer seja bastante elevado, pode-se inferir uma boa qualidade de código, pelo fato de se aproximar àquelas desenvolvidas pelos próprios arquitetos do sistema.

Utilizar para a análise métricas de código orientado a objetos, bem como outras métricas que refletem decisões arquiteturais e de design. É importante incluir métricas que refletem o volume de código para dar resultados relativos ao tamanho do aplicativo sendo avaliado, uma vez que várias métricas tem seu valor de referência variável de acordo com o tamanho do código. Utilizar valores do sistema, como milhões de linhas de código, como referencia para analisar um aplicativo com apenas algumas centenas de linhas resulta em uma comparação inválida para os valores de algumas métricas, como discutido no Capítulo 2.

Algumas conclusões talvez já possam ser tiradas de acordo com a aproximação do código do sistema aos aplicativos nativos, caso os mesmos sejam a referencia. Por exemplo, a respeito dos aplicativos nativos, se os resultados mostrarem que nenhum se aproxima a mais de 70% do sistema, então pode-se inferir que a aproximação ideal ao sistema seja esta, e talvez uma maior que 70 não seja desejada em aplicativos de terceiros. Da mesma forma, uma aproximação muito alta pode indicar a possibilidade de utilizar os valores da mesma como referência, utilizando os aplicativos para gerar mais amostras relativas de

tamanho.

3.1 Pesquisa

Baseando-se nas ideias apresentadas, é levantada a seguinte questão de pesquisa:

- É possível monitorar métricas estáticas de código fonte de aplicativos Android de acordo com a análise e predição da evolução 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 as questões de pesquisa que foram definidas, algumas hipóteses devem ser estudadas e avaliadas:

- É possível identificar padrões e tendências na evolução da arquitetura do sistema Android e nos aplicativos desenvolvidos para ele.
- O desenvolvimento de aplicativos Android pode ser guiado pelo resultado de uma análise evolutiva do código do próprio sistema.

Como uma hipótese adicional neste trabalho, será avaliado, através da comparação com o sistema, o resultado parcial inicial deste trabalho, que corresponde a um aplicativo Android 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. Surge então uma hipótese adicional, para avaliar se as decisões tomadas com base em experiência de desenvolvedor e padrões de projeto são refletidas e semelhantes as decisões que podem ser tomadas com base em resultados de análise estática de código fonte, como as análises realizadas neste trabalho.

- As decisões arquiteturais teóricas aplicadas no estudo de caso e-lastic estão relacionadas com decisões arquiteturais baseadas em métricas.

O projeto do aplicativo desenvolvido durante o início deste trabalho para esse estudo de caso específico também deve ser submetido a análise de aproximação ao código do sistema e de aplicativos nativos, de forma a validar as decisões tomadas durante o desenvolvimento do mesmo.

3.2 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 do sistema, como dependência de código de terceiros dentro do projeto de aplicativos Android. O objetivo não é 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 idéia 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 linear até redes neurais. Embora os resultados apresentados no artigo não tenham sido muito promissores, algumas observações podem ser tiradas como exemplo para 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) se mostrou não muito determinística, 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 como o resultado de uma comparação do código com a arquitetura do sistema e de aplicativos nativos, 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 de Chidamber e Kemerer, métricas de complexidade e de volume de código fonte. Outro ponto importante é este estudo será validado por um modelo preditivo desenvolvido para esse fim.

3.3 Coleta de Dados

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

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

Para a análise da API do sistema, foram escolhidas 16 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 KitKat, as versões 4.4 e 4.4.4. 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 gran-

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

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

des 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
13. *Android KitKat* 4.4 r1
14. *Android KitKat* 4.4.4 r2
15. *Android Lollipop* 5.0.0 r1.0.1
16. *Android Lollipop* 5.1.0 r1

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

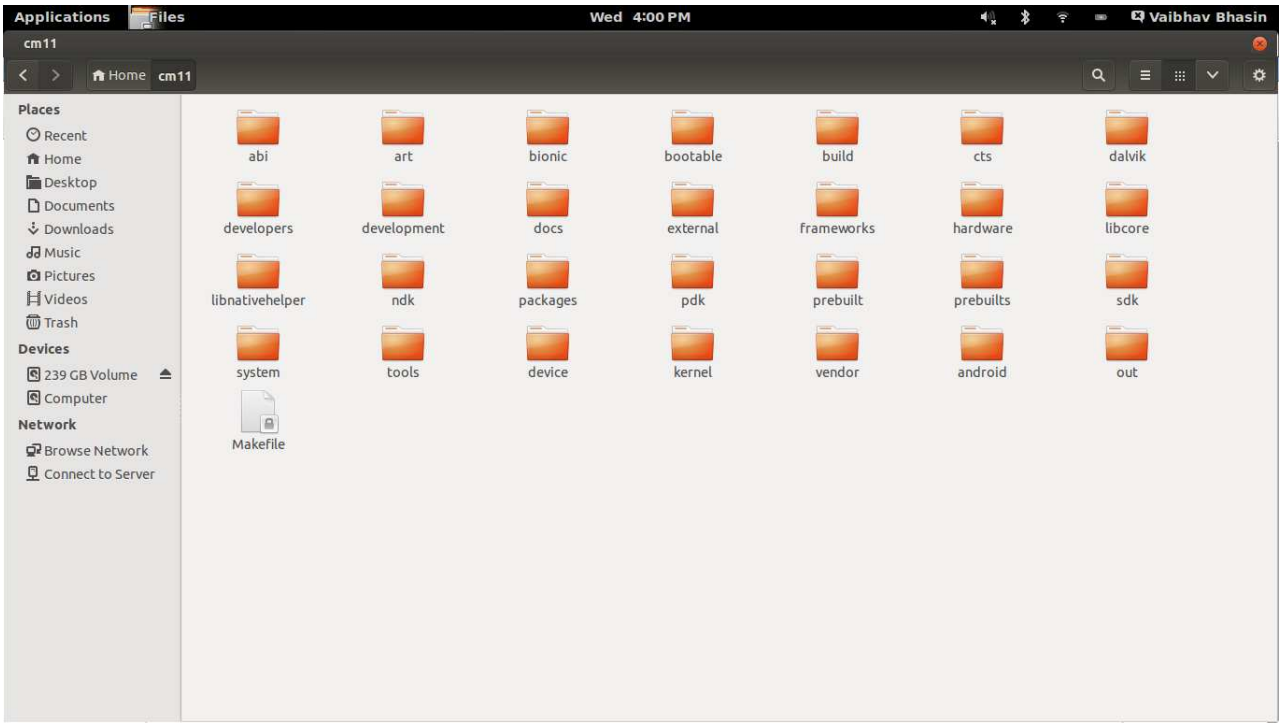

Figura 1 – Exemplo de estrutura de arquivos de diretório raiz do AOSP

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

O restante dos diretórios do AOSP contém desde adaptações de bibliotecas para o Android como o bionic, até código fonte para o *Android Run Time* (ART), que substituiu a dalvik nas 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*

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

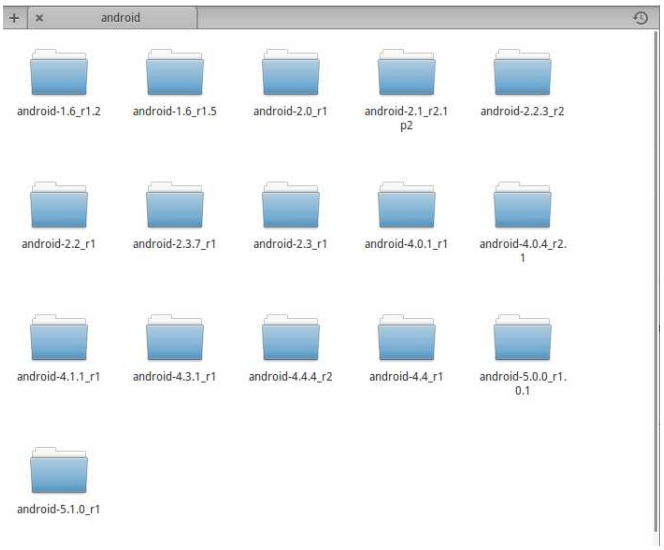

Figura 2 – Diretório onde foram armazenadas cada versão a ser analisada

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 a performance 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 na análise.

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 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

3.4 Análise de dados

Após o download de todas as versões escolhidas, foi utilizada a ferramenta Analizo[4] para análise estática de código e coleta de métricas. Um dos motivos da escolha do sistema Debian foi a facilidade de instalação da ferramenta. O Analizo 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. Foram coletadas essencialmente métricas de código fonte orientado a objetos. São métricas desse tipo que melhor refletem as decisões arquiteturais ou de design aplicadas durante o desenvolvimento do sistema, como discutido no Capítulo 2.

Neste trabalho, foi utilizada a funcionalidade de *batch* do Analizo, de forma a coletar métricas de todas as *tags* de uma só vez. A saída da ferramenta é um arquivo CSV para cada projeto ou versão a ser analisada, assim como um arquivo CSV que centraliza os valores de cada métrica a 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.

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

As métricas finais escolhidas foram capturadas pela ferramenta Analizo e estão listadas a seguir:

- ACCM - *Average Cyclomatic Complexity per Method*
- AMLOC - *Average Method Lines Of Code*
- CBO - *Coupling Between Objects*
- COF - *Coupling Factor*
- DIT - *Depth in Inheritance Tree*
- LCOM4 - *Lack of Cohesion in Methods*
- LOC - *Lines of Code*
- NOC - *Number of Children*

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

- RFC - *Response For a Class*

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.

Para a análise de código fonte em C, que é uma linguagem estruturada, com essas métricas orientadas a objetos, algumas observações devem ser ressaltadas: na ferramenta analizo, em vez de classes, são considerados módulos, e as funções são utilizadas como métodos(MEIRELLES, 2013). 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. No Capítulo 4 serão discutidas, no contexto deste trabalho, as variações de valores para algumas métricas quando analisados nesses diferentes paradigmas. 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, foram calculados, utilizando 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 máximo valor. 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. 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-5.1.0_r1	20129	0	0	0	0	1	1	2	3.5	5	11	158.6

Tabela 1 – Percentis para os valores de complexidade ciclomática nas versões do Android 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. A figura XXXXX apresenta um gráfico de área contendo os valores da métrica XXXXX para cada versão analisada. Fica bem claro o aumento/diminuição(XXXXX) dos valores da métrica para as versões mais recentes do Android. Detalhes da interpretação dos resultados serão discutidos no Capítulo 4.

Alguns outros gráficos também podem ser gerados a partir dos dados completos, com os valores de cada métrica para cada classe. Gráficos como o histograma, demonstrando os valores mais frequentes, e o boxplot, que ajuda a verificar valores mais discrepantes do restante da amostra, são bastante úteis para a análise dos dados coletados. A figura XXXXX é um boxplot para a métrica XXXXX plotado com todos os dados do sistema. Pode-se perceber claramente que os valores XXX são muito frequentes enquanto os demais são menos frequentes, sendo que alguns são valores atípicos nessas amostras. Outliers, que são esses valores discrepantes, devem ser identificados e removidos para não gerar ruído na interpretação dos dados. Por esse motivo, as amostras mais utilizadas nas análises serão em geral até o 95º percentil. Assim como Meirelles (2013), os resultados das análises discutidos no Capítulo 4 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.

4 Resultados

4.1 Análise preliminar

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.

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.

4.1.1 Average Method Lines Of Code

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. Neste trabalho não serão feitas comparações diretas dos valores desta métrica, então ela será utilizada principalmente para relativização da comparação de outras métricas, quando aplicável, uma vez que os valores de algumas delas podem ser relacionados a esta.

A métrica LOC por si só não será discutida aqui, pois seu valor é mais relativo e deve ser comparado com outras métricas para ter significado mais completo. Uma classe com valor alto de LOC pode ter um baixo valor de AMLOC e valor maior para NOM, ainda mantendo um valor aceitável de LCOM. Em suma, a análise de outras métricas abrange as explicações relacionadas a métrica LOC e também a NOM, então essas métricas de tamanho não serão explanadas em separado, mas juntamente com a explicação de outras métricas.

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 LCOM e RFC. Uma classe com muitos métodos privados pequenos tende a ter um valor maior de RFC, o que não implica que esteja mal projetada, desde que os métodos ali presentes estejam bem posicionados segundo o padrão de projeto OO especialista da informação, mantendo por consequencia um baixo valor de LCOM. Como referência geral de resultados para AMLOC, quanto menor o valor, melhor o resultado.

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-5.1.0_r1	20129	0	0	0	0	2	5.5	12	24	37.8	105	708

Tabela 2 – Percentis para a métrica *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. 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 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,

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
SoundRecorder	6	1	1.32	2.6	4.2	9	10	11.8	19.15	21.6	23.56	24.05
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 – Percentis para a métrica *Average Method Lines of Code* nos aplicativos nativos

quando olhamos os valores aplicativos do sistema, demonstrados na Tabela 3, podemos perceber uma grande semelhança nos resultados. 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.

Em linhas gerais, 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, porem 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;

4.2 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.

4.3 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.

4.4 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.

4.5 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.

Referências

ANDROIDDEVELOPERS. *Google guide to android development*. 2014. Disponível em: <http://developer.android.com/develop/index.html>. Citado na página 9.

BRAHLER, S. *Analysis of the Android Architecture*. 2010. Citado na página 39.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. 1994. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=295895&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel1%2F32%2F7320%2F00295895>. Citado na página 22.

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 16 e 39.

DEVOS, M. *Bionic vs Glibc Report*. 2014. Disponível em: <http://irati.eu/wp-content/uploads/2012/07/bionic_report.pdf>. Citado na página 9.

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 20.

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 25.

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 2 vezes nas páginas 21 e 26.

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 10 e 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 13 e 14.

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 13, 16, 17, 18 e 42.

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 26.

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 25.

MCCABE, T. J. A complexity measure. 1976. Disponível em: <http://www.computer.org/csdl/trans/ts/1976/04/01702388-abs.html>. Citado na página 22.

MEIRELLES, P. R. M. Monitoramento de métricas de código-fonte em projetos de software livre. In: . São Paulo: [s.n.], 2013. Citado 3 vezes nas páginas 32, 33 e 36.

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 25.

OLIVEIRA, C. M. F. Kalibro: Interpretação de métricas de código fonte. In: . São Paulo: [s.n.], 2013. Citado 2 vezes nas páginas 36 e 37.

R.BASILI, V.; BRIAND, L.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. 1995. Disponível em: <http://drum.lib.umd.edu/bitstream/1903/715/2/CS-TR-3443.pdf>. Citado 2 vezes nas páginas 22 e 26.

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 13.

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 18.

SYER, M. et al. Exploring the development of micro-apps: A case study on the blackberry and android platforms. 2011. Disponível em: <http://sailhome.cs.queensu.ca/~mdsyer/wp-content/uploads/2011/07/Exploring-the-Development-of-Micro-Apps-A-Case-Study-on-the-BlackBerry-and-Android-Platforms.pdf>. Citado na página 25.

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 14, 16 e 39.

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 na página 26.