Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Suelen Goularte Carvalho

Texto submetido
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para o
Exame de Qualificação para o Grau de Mestre em
Ciência da Computação

Orientador: Prof. Dr. Marco Aurélio Gerosa

São Paulo, 12 de Novembro de 2016

Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Membros do Comitê:

- Prof. Dr. Marco Aurélio Gerosa (orientador) – IME-USP
- Prof. Dr. Alfredo Goldman – IME-USP
- Prof. Dr. Paulo Roberto Miranda Meirelles – UnB

Resumo

Android é o sistema operacional para dispositivos móveis mais usado atualmente, com 83% do mercado mundial e mais de 2 milhões de aplicativos disponíveis na loja oficial. Aplicativos Android têm evoluído para complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. Esse contexto pode levar a decisões ruins de *design* de código, conhecidas como cheiros de código, podendo se tornar anomalias que degradam a qualidade do projeto, tornando-o de difícil manutenção. Consequentemente, desenvolvedores de software precisam identificar trechos de código problemáticos que podem se beneficiar de refatoração, com o objetivo de ter constantemente uma base de código que favoreça a manutenção e evolução. Para isso, desenvolvedores costumam fazer uso de estratégias de detecção de cheiros de código. Apesar de já existirem diversos cheiros de código catalogados, como por exemplo *God Class* e *Long Method*, eles não levam em consideração a natureza do projeto. Pesquisas têm demonstrado que diferentes plataformas, linguagens e frameworks apresentam métricas de qualidade de código específicas. Projetos Android possuem características específicas, como um diretório que armazena todos os recursos usados e uma classe que tende a acumular diversas responsabilidades. Nota-se também que pesquisas específicas sobre Android ainda são poucas. Nesta dissertação objetivamos identificar, validar e documentar cheiros de código Android com relação à camada de apresentação, onde se encontra as maiores diferenças quando se comparado a projetos tradicionais. Em outros trabalhos sobre Android, foram identificados cheiros de código relacionados à segurança, consumo inteligente de recursos ou que de alguma forma impactam a experiência ou expectativa do usuário. Diferentemente deles, nossa proposta é catalogar cheiros de código Android que influenciem na qualidade do código. Com isso, os desenvolvedores terão mais um recurso para a produção de código de qualidade.

Palavras-chave: android, cheiros de código, qualidade de código, anomalias de código, métricas de código.

Abstract

Android is the most widely used mobile operating system with 83% of the world market and more than 2 million applications available in the official store. Android applications have been made complex software projects that need to be quickly developed and regularly evolved to meet the requirements of users. This context can lead to bad code design choices, known as code smells, that can become anomalies that degrade project quality, making it difficult to maintain. Consequently, software developers need to identify problematic code snippets in order to have a code base that favors maintenance and evolution. For this, developers often make use of techniques to detect code smells. Although there are already several code smells cataloged, such as God Class and Long Method, they do not take into account the nature of the project. Researches demonstrated that different platforms, languages and frameworks present specific code quality metrics. Android projects have specific characteristics, such as a directory that stores all the resources used and a class that tends to accumulate various responsibilities. Research on Android are still few. In this dissertation we aim to identify, validate and document code smells related with presentation layer of Android, where the greatest differences are found when compared to traditional projects. In other works on Android, were identified code smells related to safety, intelligent features or somehow impact the experience or user expectation consumption. Unlike them, our proposal is to catalog code smells Android that influence the quality of the code. With this, developers will have another resource for producing quality code.

Keywords: android, code smells, code quality, code anomalies, code metrics.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Atualmente o Android é a plataforma móvel com maior crescimento. Em 2017 completará uma década do seu lançamento [2]. No primeiro quadrimestre de 2016, foram vendidos aproximadamente 300 milhões de dispositivos com Android contra aproximadamente 50 milhões com iOS, seu concorrente mais próximo [9]. Com o crescimento da plataforma, a demanda por aplicativos também aumentou. A Google Play Store [1] em 2016 superou 200 milhões de aplicativos disponíveis [10]. Ainda, segundo Gartner Group [8] e IDC [14], mais de 83,5% dos dispositivos móveis usam o sistema operacional Android, e esse percentual vem crescendo ano após ano. Atualmente é possível encontrar a plataforma Android em diferentes tipos de dispositivos como *smartphones*, *smart TVs*, *smartwatchs*, carros, dentre outros [5, 7].

O crescimento acentuado da plataforma Android impacta no processo de desenvolvimento dos aplicativos. Segundo Geoffrey [39], aplicativos móveis têm se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários, entretanto, esse contexto pode resultar em más escolhas de *design* que podem degradar a qualidade e desempenho do software. Verloop [74] também trata desse contexto de atualizações rápidas ao dizer que aplicativos móveis possuem o ciclo de vida curto e precisam ser constantemente atualizados para se manterem relevantes e atender as expectativas de seus usuários. Verloop [74] complementa dizendo que a combinação do rápido crescimento e atualizações implicam no desenvolvimento de software de modo que desenvolvedores devem aumentar seus esforços em manter seus projetos com um alto grau de manutenibilidade.

Atualmente existe um extenso conjunto de ferramentas que apoiam desenvolvedores na tarefa de manter o código "limpo". Podemos citar estilos arquiteturais, padrões de *design*

[1] Google Play Store (originalmente Android Market), loja oficial de aplicativos Android

e boas práticas consolidadas [16, 18, 36, 50] que desenvolvedores utilizam como ponto de partida. Por exemplo, o padrão de projeto Model-View-Controller (MVC) [46] sugere separar o código em três camadas distintas: MODELS, VIEWS e CONTROLLERS. Cada camada possui uma responsabilidade. Classes do tipo CONTROLLERS são responsáveis por coordenar o processo entre a VIEW e o MODEL, enquanto classes do tipo MODEL representam conceitos de negócio e assim por diante. Essa separação guia o desenvolvedor de modo a implementar classes mais específicas e coesas, facilitando a manutenção.

Para detectar trechos de código problemáticos que possam se beneficiar de *refatoração*, é comum o uso de ferramentas de detecção de maus cheiros [18] como PMD [11] e Sonarqube [12]. Essas ferramentas entregam relatórios sobre o projeto indicando classes "boas" e "ruins". Por exemplo, para detectar o mau cheiro *God Class* [41], que aparece em classes que "fazem" ou "sabem" de mais [33], essas ferramentas fazem uso de um conjunto de métricas de qualidade de código [18]. Se a classe está acima (ou abaixo) das métricas, ela é considerada como "malcheirosa". Algumas métricas possibilitam configurações para se adaptar a diferentes contextos.

Nas últimas décadas, muitos cheiros de código para projetos orientado a objetos foram catalogados [33, 41]. Ocorre que pesquisas têm demonstrado que domínios diferentes podem apresentar também cheiros de código específicos [18, 37, 39, 47]. Aniche et al. [18] mapearam 6 cheiros de código relacionados ao framework Java Spring MVC e diz que "*enquanto [God Class] se encaixa bem em qualquer sistema orientado a objetos, ele não leva em consideração as particularidades arquiteturais da aplicação ou o papel desempenhado por uma determinada classe.*" (tradução livre). Gharachorlu [37] avaliou a existência de más práticas específicas de arquivos CSS. Linares et al. [47] concluem que cheiros de código impactam métricas de qualidade de formas distintas a depender do domínio. Os autores indicam como trabalho futuro entender os motivos dessa conclusão. Geoffrey [39] afirma que *antipatterns* específicos à plataforma Android são mais comuns e ocorrem mais constantemente do que *antipaterns* orientados a objetos, no entanto, salienta que pesquisas sobre *antipatterns* móveis ainda estão em sua infância.

Em parte, projetos Android seguem o paradigma de orientação a objetos, no entanto, possuem características diferentes de projetos tradicionais [39, 49, 60]. Além de código Java, grande parte de um projeto Android é constituído por arquivos `XMLs`, chamados de recursos do aplicativo [68]. São interfaces visuais, imagens, textos, dentre outros elementos visualizados pelo usuário.

Outra característica do Android é com relação à ACTIVITIES, que são classes específicas da plataforma Android responsáveis pela apresentação e interações do usuário com a tela [1]. ACTIVITIES possuem muitas responsabilidades [74], estão vinculadas a um LAYOUT e normalmente precisam de acesso a classes do modelo da aplicação. Analogamente ao padrão MVC, ACTIVITIES fazem os papéis de VIEW e CONTROLLER simultaneamente. Geoffrey [39] e Jan et al. [60] apontam ainda especificidades relacionadas à limitação de recursos do dispositivo, como bateria e memória e Umme et al. [49] apontam especificidades relacionadas à estrutura e fluxo de trabalho.

Enquanto há mais soluções documentadas para projetos orientados a objetos, ainda não é trivial encontrar uma forma sistemática de identificar códigos problemáticos com relação às características específicas de projetos Android [49, 60]. Na prática, desenvolvedores Android percebem esses problemas. Muitos deles, inclusive, se utilizam de práticas para solucioná-los. Reimann et al. [60] reforçam essa ideia ao dizer que "*o problema no desenvolvimento móvel é que desenvolvedores estão cientes sobre maus cheiros apenas indiretamente porque estas definições [dos maus cheiros] são informais (boas práticas, relatórios de problemas, fóruns de discussões, etc.) e recursos onde encontrá-las estão distribuídos pela internet*" (tradução livre).

De acordo com Umme et al. [49], dentre as principais conferências de manutenção de software (ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MRS e ESEM), no período de 2008 a 2015, a maior parte dos artigos (47) investigaram cheiros de código em aplicações *desktop* e apenas 5 (9,6%) consideraram projetos Android. Nenhuma outra plataforma móvel foi considerada.

A ausência de um catálogo de maus cheiros Android resulta em (i) uma carência de conhecimento sobre boas e más práticas a ser compartilhado entre profissionais da plataforma, (ii) indisponibilidade de uma ferramenta de detecção de maus cheiros de modo a alertar automaticamente os desenvolvedores da existência dos mesmos e (iii) ausência de estudo empírico sobre o impacto dessas más práticas na manutenibilidade do código de projetos Android.

1.1 Questões de Pesquisa

Esta dissertação tem por objetivo investigar cheiros de código específicos à camada de apresentação de projetos Android. Desta forma, trabalhamos a seguinte questão de pesquisa primária:

Quais cheiros de código são específicos à Camada de Apresentação Android?

Para isso, exploramos as seguintes questões secundárias:

Q1: O que desenvolvedores consideram boas e más práticas com relação à Camada de Apresentação em projetos Android?

Nesta questão investigamos a existência de cheiros de código em elementos da camada de apresentação Android como ACTIVITIES e ADAPTERS. Para isso realizamos um estudo empírico com a aplicação de questionário online e entrevistas com desenvolvedores Android. Muito conteúdo sobre boas e más práticas Android estão distribuídos na internet, desta forma pretende-se também coletar dados em postagens de fóruns e blogs técnicos sobre Android.

O questionário foi aplicado na comunidade de desenvolvimento Android do Brasil e exterior e foram coletadas 44 respostas. A análise desses dados, bem como outras atividades planejadas, estão previstas no cronograma de atividades apresentado no Capítulo 4.

Como resposta a esta questão espera-se derivar um conjunto de cheiros de código relacionados à camada de apresentação Android.

Q2: Qual a relação entre os cheiros de código propostos e a tendência a mudanças e defeitos?

Estudos prévios mostram que cheiros de código tradicionais (e.g., *Blob Classes*) podem impactar na tendência a mudanças em classes do projeto [18]. Desta forma, esta questão pretende, por meio de uma análise de código de projetos Android, analisar o impacto dos cheiros de código propostos na tendência a mudanças e defeitos em projetos Android.

Q3: Desenvolvedores Android percebem os códigos afetados pelos cheiros de código propostos como problemáticos?

Para responder esta questão complementamos com dados qualitativos as análises quantitativas realizadas no contexto de Q2. Desta forma, investigamos se códigos afetados pelos cheiros de código propostos para a camada de apresentação Android são percebidos como problemáticos por desenvolvedores. Os dados a serem usados para basear a resposta a esta questão serão extraídos do mesmo experimento realizado na Q2.

Faremos uso de diferentes métodos de pesquisa durante esta dissertação. Cada método usado é abordado no capítulo respectivo à questão. Todos os capítulos exigem do leitor conhecimento prévio sobre Android, Maus Cheiros de Código. Apresentamos uma breve introdução a esses dois assuntos no Capítulo 2.

1.2 Contribuições

As principais contribuições desta dissertação, na ordem em que aparecem, são:

1. Um catálogo validado de cheiros de código da camada de apresentação Android. Os cheiros de código serão definidos com base em questionário online, entrevistas e coleta de dados relevantes em sites e blogs técnicos sobre Android.

2. Um estudo quantitativo sobre a tendência a mudanças e defeitos dos cheiros de código propostos. Realizaremos um estudo de projetos com desenvolvedores Android de modo a coletar quantitativamente se classes afetadas pelos cheiros de código propostos possuem uma maior tendência a mudanças e introdução de defeitos.

3. Um estudo qualitativo sobre a percepção de desenvolvedores com relação aos cheiros de código propostos. Baseado no experimento realizado na Q2 pretende-se coletar dados de modo a identificar se classes afetadas pelos cheiros de código propostos são percebidas como problemáticas por desenvolvedores Android.

1.3 Organização da Dissertação

O restante desta dissertação está organizada da seguinte forma:

- **Capítulo 2** Fundamentação Conceitual: Neste capítulo é passado ao leitor informações básicas relevantes para o entedimento do trabalho. Os assuntos aprofundados são: Cheiros de Código e Android. Esta dissertação limita-se em mapear boas e más práticas apenas relacionadas à camada de apresentação de aplicativos Android. Desta forma, neste capítulo também é apresentado ao leitor o que é considerado, para efeitos deste trabalho, como camada de apresentação Android.

- **Capítulo 3** Trabalhos Relacionados: Neste capítulo, pretende-se apresentar estudos relevantes já feitos em torno do tema de maus cheiros Android e em que esta dissertação se diferencia deles.

- **Capítulo 4** Proposta de Dissertação: Neste capítulo, apresentamos a proposta da dissertação e o cronograma de atividades.

- **Capítulo 5** Boas e Más Práticas na Camada de Apresentação: Neste capítulo, respondemos a Q1. É apresentada a motivação da questão, os métodos de pesquisa utilizados e o catálogo resultante de maus cheiros.

- **Capítulo 6** Impacto na Tendência a Mudanças e Defeitos: Neste capítulo, respondemos a Q2. É apresentada a motivação da questão, explicamos o experimento conduzido e os resultados obtidos.

- **Capítulo 7** Percepção dos Desenvolvedores: Neste capítulo, respondemos a Q3. É apresentado a motivação da questão, explicamos o experimento conduzido e os resultados obtidos.

- **Capítulo 8** Conclusão: Neste capítulo, são apresentadas as conclusões do trabalho, bem como as suas limitações e sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Conceitual

Para a compreensão deste trabalho é importante ter compreensão de dois temas: Maus Cheiros de Código e Android.

2.1 Maus Cheiros de Código

Maus cheiros de código são sintomas de design ruim e más práticas de programação [13, 18, 33, 74]. Também conhecido por anomalias de código [54]. Diferentemente de erros sistêmicos, eles não resultam em comportamentos errôneos [13, 74]. Maus cheiros apontam fraquezas no *design* que podem desacelerar o desenvolvimento e aumentar o risco a defeitos, falhas e mudanças [13, 18, 45, 47]. Do ponto de vista do desenvolvedor, maus cheiros são heurísticas para refatorações que indicam quando refatorar e qual técnica de refatoração usar [13].

Refatoração é definido por Fowler como "uma técnica para reestruturação de um código existente, alterando sua estrutura interna sem alterar seu comportamento externo" [33]. Escolher não resolver um mau cheiro pela refatoração não resultará na aplicação falhar mas irá aumentar a dificuldade de mantê-la. A refatoração ajuda a melhorar a manutenibilidade de uma aplicação [74]. Uma vez que os custos com manutenção são a maior parte dos custos envolvidos no ciclo de desenvolvimento de software, aumentar a manutenibilidade através de refatoração irá reduzir os custos de um software no longo prazo [73].

O livro de Webster (1995) [78] deve ter sido um dos primeiros onde o termo cheiro de código, referindo-se a más práticas, foi usado. Mais adiante, **mau cheiro de código** foi usado por Martin Fowler e Kent Beck no livro Refactoring (1999) [33] referindo-se a um trecho de código que pode se beneficiar de refatoração. Em uma postagem em 2006 em seu site [34], Martin Fowler define um **cheiro de código** como sendo "uma indicação

superficial no código que usualmente corresponde a um problema mais profundo no sistema" e complementa dizendo que "primeiramente um mau cheiro é por definição algo rápido de ser detectado - farejável - e segundo, que um mau cheiro nem sempre indica um problema' (tradução livre).

Note que o termo usado no livro Refactoring (1999) [33] é mau cheiro de código (*bad smell in code*) e na postagem em seu site [34] é cheiro de código (*code smell*). Essa diferença pode ser justificada pela afirmativa de Fowler, na postagem em seu site, em que diz que "cheiros de código não são inerentemente ruins por conta própria - eles são muitas vezes um indicador de um problema e não o problema em si" (tradução livre).

No livro Refactoring for Software Design Smells (2014) [70] encontramos uma definição sobre cheiro de código relacionada a princípios e qualidade, onde diz que "[cheiros de código] são certas estruturas que indicam violação de princípios de *design* fundamentais e impactam negativamente na qualidade do *design*". De fato, pesquisas têm apontado que maus cheiros de código estão relacionados à degradação da qualidade em projetos de software [60].

Outro termo comum de se ver juntamente com (mau) cheiro de código é o termo *antipattern*, também conhecido como má prática. Esse, por sua vez, descreve uma solução comum para um problema que gera consequências negativas [21]. Pode ser resultado de um desenvolvedor não ter conhecimento suficiente ou experiência em resolver um determinado tipo de problema, ou ter aplicado um *design pattern*, solução reutilizável para um problema de *design* recorrente [21, 35], perfeitamente bom no contexto errado [21]. Geralmente, (maus) cheiros de código são sintomas da presença de *antipattern*. Linares et al. [47] concluem que *antipatterns* impactam de forma negativa aplicações móveis, em particular métricas de qualidade relacionadas ao aumento do risco de falhas.

Para auxiliar no processo de refatoração, é comum desenvolvedores se utilizarem de ferramentas de detecção de cheiros de código [18], sintomas de design ruim e más práticas de programação [13, 18, 33, 74]. Para automatizar a detecção de cheiros de código, é importante tê-los mapeados e catalogados. Muitos cheiros de código já foram catalogados [33, 70, 78].

Diversos autores já trabalharam na definição de diferentes catálogos de cheiros de código. Riel [41] definiu mais de 60 características de um bom código orientado a objetos. Fowler [33] definiu mais de 20 maus cheiros como *God Classes* [33], que são classes que fazem ou sabem muitas coisas e *Feature Envy* [33], que são métodos que estão mais interessados em outras classes do que na própria classe. Existem diversas ferramentas atualmente que conseguem realizar a detecção automática de diversos desses maus cheiros como PMD [11] e Sonar [12].

No entanto, nas principais conferências de manutenção de software, dentre 2008 a 2015, 5 de um total de 52 artigos, foram sobre maus cheiros Android [49]. A ausência de um catálogo de maus cheiros Android resulta em (i) uma carência de conhecimento sobre boas e más práticas a ser compartilhado entre praticantes da plataforma, (ii) indisponibilidade de uma ferramenta de detecção de maus cheiros de modo a alertar automaticamente os desenvolvedores de sua existência e (iii) ausência de estudo empírico sobre o impacto dessas más práticas na manutenibilidade do código de projetos Android.

Nesta dissertação utilizamos o termo **cheiro de código** para nos referir a ambos os termos (mau) cheiro de código, *antipattern* e anomalias de código, como em trabalhos prévios [47, 53, 54], e nosso foco está em definir e detectar maus cheiros relacionados especificamente à camada de apresentação de projetos Android.

2.2 Android

Atualmente o Android é a plataforma móvel com maior crescimento. Em 2017 completará uma década do seu lançamento [2]. A Figura 2.1 apresenta a quantidade de dispositivos vendidos para cada plataforma móvel, no período de 2009 à 2016 [9]. É possível observar que no Q1 de 2016, foram vendidos aproximadamente 300 milhões de dispositivos com Android contra aproximadamente 50 milhões com iOS, seu concorrente mais próximo.

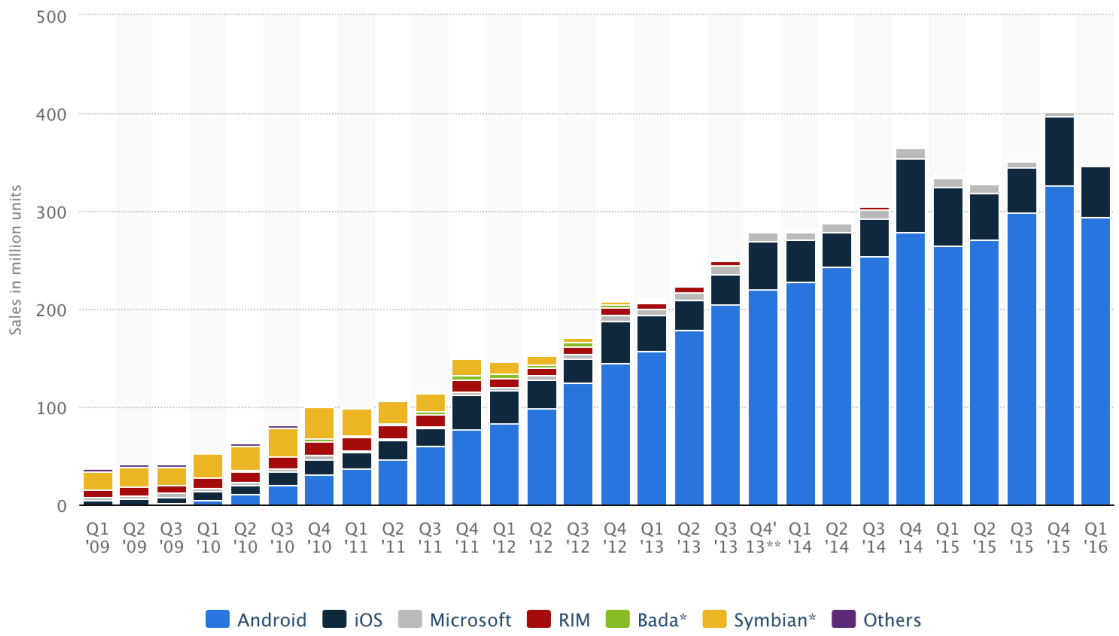

Figura 2.1: *Unidades de dispositivos vendidos por plataforma móvel [9].*

Com o crescimento da plataforma, a demanda por aplicativos também aumentou. Na Figura 2.2 é possível acompanhar o crescimento da quantidade de aplicativos disponíveis na Google Play Store [1] ao longo de 2009 até 2016, sendo que nesse último ano, superou 200 milhões de aplicativos disponíveis [10].

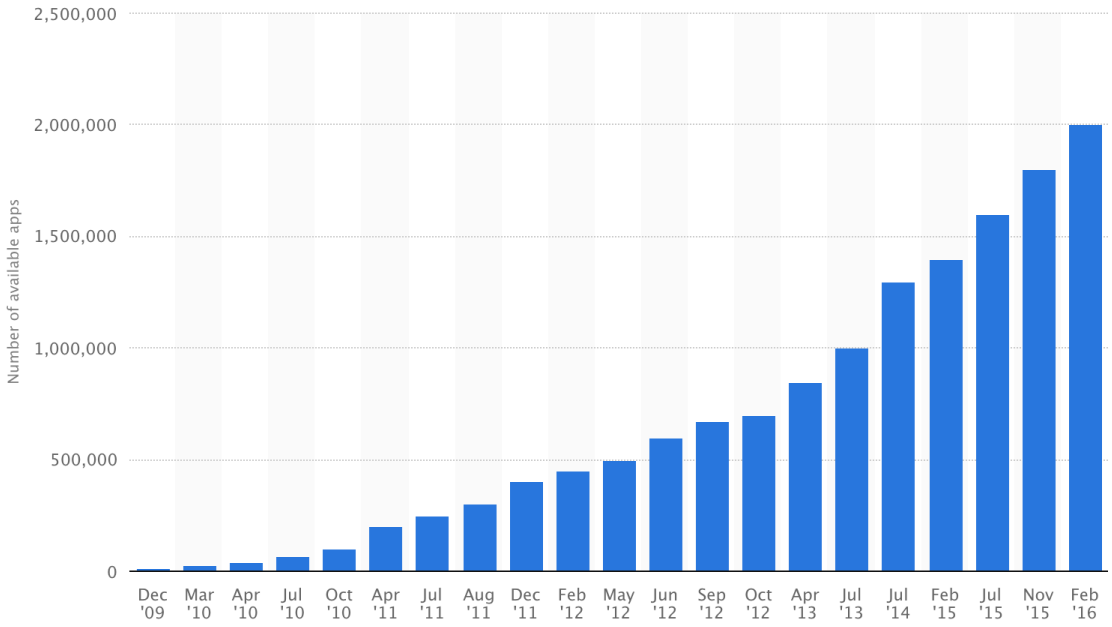

Figura 2.2: *Quantidade de aplicativos disponíveis na Google Play Store [10].*

Android é um sistema operacional de código aberto, baseado no kernel do Linux criado para um amplo conjunto de dispositivos. Na Figura 2.3 é apresentada a arquitetura geral da plataforma Android. Todas as funcionalidades da plataforma Android estão disponíveis para os aplicativos por meio de APIs Java (*Application Programming Interface*). Essas APIs compõem os elementos básicos para a construção de aplicativos Android.

2.2.1 Fundamentos do Desenvolvimento Android

Aplicativos Android são escritos na linguagem de programação Java. O **SDK!** (**SDK!**) Android compila o código, junto com qualquer arquivo de recurso ou dados, em um arquivo APK (*Android Package*). Arquivos APKs, arquivo com extensão `.apk`, são usados por dispositivos para a instalação de aplicativos [62].

Os elementos base para a construção de aplicativos Android são os componentes. Cada

[1] Google Play Store (originalmente Android Market) é a loja oficial de aplicativos Android. Pode ser acessada pelo URL https://play.google.com/store [3].

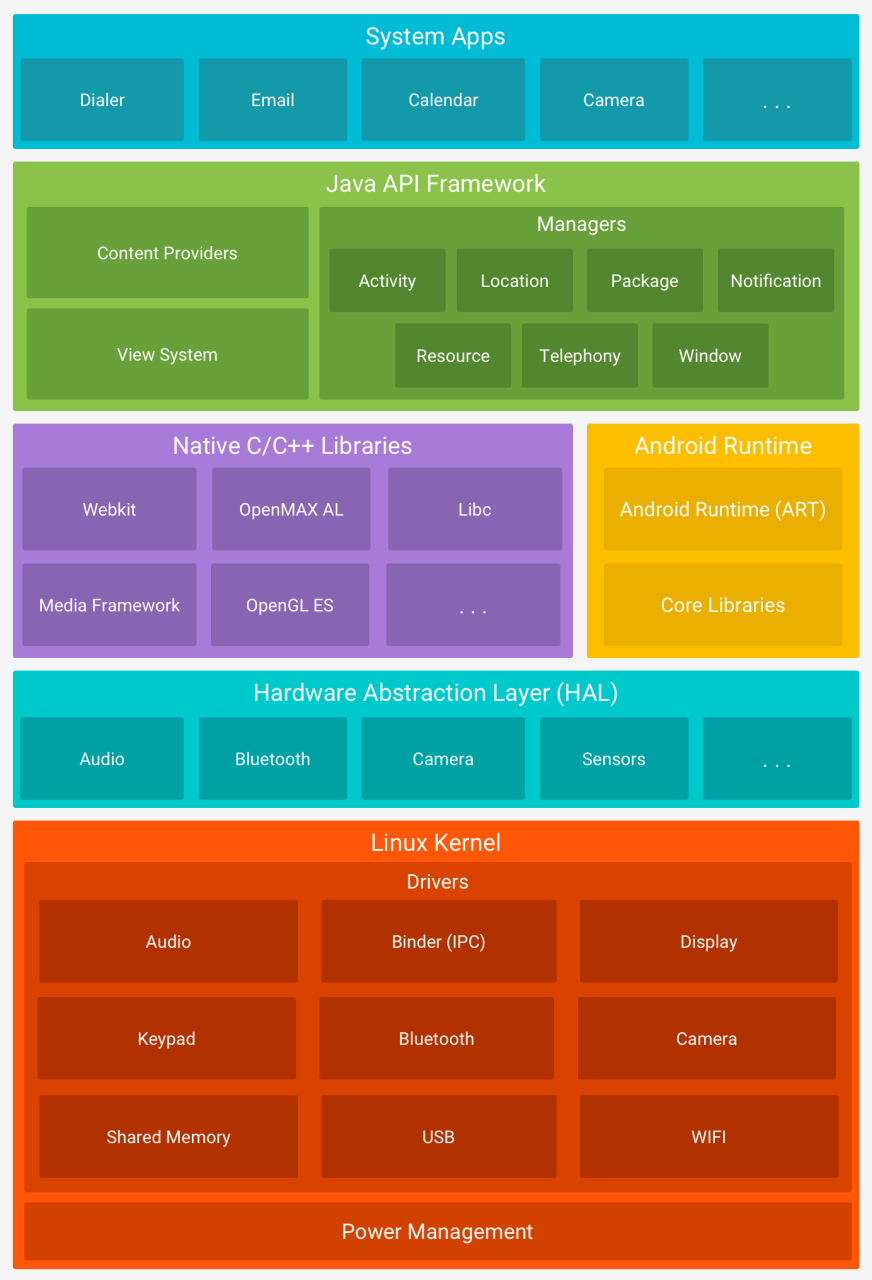

Figura 2.3: *Arquitetura do sistema operacional Android [63].*

componente é um diferente ponto de entrada por meio do qual o sistema aciona o aplicativo. Nem todos os componente são pontos de entrada para o usuário e alguns são dependentes entre si [62]. Há quatro tipos diferentes de componentes Android, cada qual serve um propósito distinto e possui diferentes ciclos de vida, ou seja, como o componente é criado e destruído [62]. São eles:

- **Activities**

  Uma *activity* representa uma tela com uma interface de usuário. Por exemplo, um aplicativo de email pode ter uma *activity* para mostrar a lista de emails, outra para redigir um email, outra para ler emails e assim por diante. Embora *activities* trabalhem juntas de modo a criar uma experiência de usuário (UX do inglês *User Experience*) coesa no aplicativo de emails, cada uma é independente da outra. Desta forma, um aplicativo diferente poderia iniciar qualquer uma dessas *activities* (se o aplicativo de emails permitir). Por exemplo, a *activity* de redigir email no aplicativo de emails, poderia solicitar o aplicativo câmera, de modo a permitir o compartilhamento de alguma foto. Uma *activity* é implementada como uma subclasse de `Activity` [62].

- **Services**

  Um *service* é um componente que é executado em plano de fundo para processar operações de longa duração ou processar operações remotas. Um *service* não provê uma interface com o usuário. Por exemplo, um *service* pode tocar uma música em plano de fundo enquanto o usuário está usando um aplicativo diferente, ou ele pode buscar dados em um servidor remoto através da internet sem bloquear as interações do usuário com a *activity*. Um *service* é implementado como uma subclasse de `Service` [62].

- **Content Providers**

  Um *content provider* gerencia um conjunto compartilhado de dados do aplicativo. Estes dados podem estar armazenados em arquivos de sistema, banco de dados SQLite, servidor remoto ou qualquer outro local de armazenamento que o aplicativo possa acessar. Por meio de *content providers*, outros aplicativos podem consultar ou modificar (se o *content provider* permitir) os dados. Por exemplo, a plataforma Android disponibiliza um *content provider* que gerencia as informações dos contatos dos usuários, possibilitando que qualquer aplicativo, com as devidas permissões, possa consultar, ler ou escrever informações sobre um contato. Um *content provider* é implementado como uma subclasse de `ContentProvider` [62].

- **Broadcast Receivers**

  Um *broadcast receiver* é um componente que responde a mensagens enviadas pelo sistema. Muitas destas mensagens são originadas da plataforma Android, por exemplo, o desligamento da tela, baixo nível de bateria e assim por diante. *Broadcast receivers*

não possuem interface de usuário. Para informar o usuário que algo ocorreu, *broadcast receivers* podem criar notificações. Um *broadcast receiver* é implementado como uma subclasse de `BroadcastReceiver` [62].

Para a plataforma iniciar quaisquer dos componentes mencionados, ela busca pela existência deles por meio da leitura do arquivo `AndroidManifest.xml` do aplicativo (arquivo de manifesto). O arquivo de manifesto é um arquivo XML, localizado na raiz do projeto, que contém informações sobre o aplicativo tais como: permissões de usuário, configurações de dependências do projeto, versão do Android, declarações dos componentes do aplicativo, dentre outras [62]. Por exemplo, uma *activity* pode ser declarada conforme o Código-fonte 2.1.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest ... >
3     <application android:icon="@drawable/app_icon.png" ... >
4         <activity android:name="com.example.project.ExampleActivity"
5                   android:label="@string/example_label" ... >
6         </activity>
7         ...
8     </application>
9 </manifest>
```

Código-fonte 2.1: *Arquivo `AndroidManifest.xml`*

No elemento `<application>`, o atributo `android:icon` aponta para o ícone, que é um recurso do tipo imagem, que identifica o aplicativo. No elemento `<activity>`, o atributo `android:name` especifica o nome completamente qualificado da *Activity*, e por fim, o atributo `android:label` especifica um texto para ser usado como título da *Activity*.

2.2.2 Recursos do Aplicativo

Um aplicativo Android é composto por outros arquivos além de código Java, ele requer **recursos** como imagens, arquivos de áudio, animações, menus, estilos e qualquer recurso relativo a apresentação visual do aplicativo [66]. Recursos costumam ser arquivos XML que usam o vocabulário definido pelo Android [62].

Um dos aspectos mais importantes de prover recursos separados do código-fonte é a habilidade de prover recursos alternativos para diferentes configurações de dispositivos como por

exemplo idioma ou tamanho de tela [62]. Segundo levantamento, em 2015 foram encontrados mais de 24 mil dispositivos diferentes com Android [6].

Deve-se organizar os recursos dentro do diretório `res` do projeto, usando subdiretórios que agrupam os recursos por tipo e configuração. Para qualquer tipo de recurso, pode-se especificar uma opção padrão e outras alternativas [62].

- **Recursos padrões** são aqueles que devem ser usados independente de qualquer configuração ou quando não há um recurso alternativo que atenda a configuração atual. Por exemplo, arquivos de *layout* padrão ficam em `res/layout`.
- **Recursos alternativos** são todos aqueles que foram desenhados para atender a uma configuração específica. Para especificar que um grupo de recursos é para ser usado em determinada configuração, basta adicionar um qualificador ao nome do diretório. Por exemplo, arquivos de *layout* para quando o dispositivo está em posição de paisagem ficam em `res/layout-land`.

O Android irá aplicar automaticamente o recurso apropriado através da identificação da configuração corrente do dispositivo. Por exemplo, o recurso do tipo *strings* pode conter textos usados nas interfaces do aplicativo. É possível traduzir estes textos em diferentes idiomas e salvá-los em arquivos separados. Desta forma, baseado no qualificador de idioma usado no nome do diretório deste tipo de recurso (por exemplo `res/values-fr` para o idioma francês) e a configuração de idioma do dispositivo, o Android aplica o conjunto de *strings* mais apropriado.

2.2.3 Interfaces de Usuários

Arquivos de *layout* são recursos localizados no subdiretório `res/layout` que possuem a extensão `.xml` [66]. Todos os elementos de UI (Interface de Usuário, do inglês UI, *User Interface*) de um aplicativo Android são construídos usando objetos do tipo `View` e `ViewGroup` como mostrado na Figura 2.4 [67].

Uma `View` é um objeto que desenha algo na tela do qual o usuário pode interagir como caixas de texto e botões. Um `ViewGroup` é um contêiner invisível que organiza `Views` filhas. O encadeamento desses objetos formam uma árvore hierárquica que pode ser tão simples ou complexa quanto se precisar [66].

É possível criar um *layout* programaticamente instanciando `Views` e `ViewGroups` no

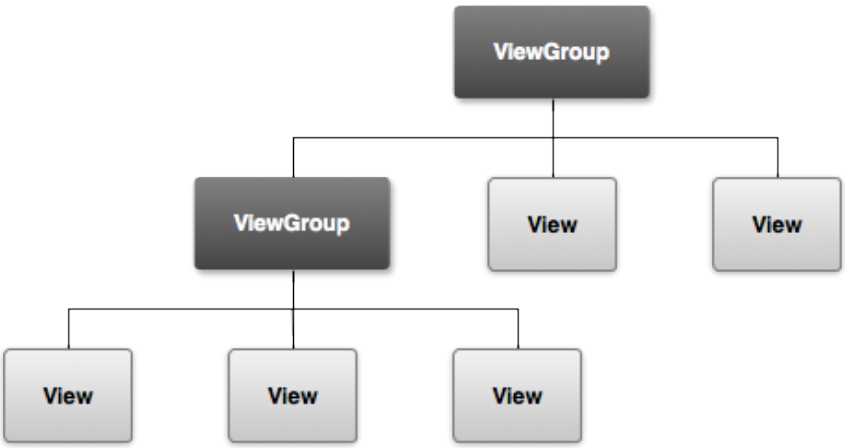

Figura 2.4: *Árvore hierárquica de `Views` e `ViewGroups` do Android [67].*

código e construir a árvore hierárquica manualmente, no entanto, a forma mais indicada é por meio de um XML de *layout* [67].

O vocabulário XML para declarar elementos de UI segue ou é muito próxima a estrutura de nome de classes e métodos, onde os nomes dos elementos correspondem aos nomes das classes e os atributos correspondem aos nomes dos métodos, como por exemplo, o elemento `<EditText>` tem o atributo `text` que corresponde ao método `EditText.setText()` [67]. Um layout vertical simples com uma caixa de texto e um botão se parece com o Código-fonte 2.2.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout ...
3                 android:layout_width="fill_parent"
4                 android:layout_height="fill_parent"
5                 android:orientation="vertical">
6
7     <TextView android:layout_width="wrap_content"
8                 android:layout_height="wrap_content"
9                 android:text="I am a TextView" />
10
11    <Button   android:layout_width="wrap_content"
12                android:layout_height="wrap_content"
13                android:text="I am a Button" />
14
15 </LinearLayout>
```

Código-fonte 2.2: *Arquivo exemplo de layout.*

Quando o conteúdo é dinâmico ou não pré-determinado, como por exemplo uma lista de dados, pode-se usar um elemento que estende de `AdapterView` para popular o layout em tempo de execução. Subclasses de `AdapterView` usam uma implementação de `Adapter` para carregar dados em seu *layout*. `Adapters` agem como um intermediador entre o conteúdo a ser exibido e o *layout*, ele recupera o conteúdo e converte cada item, de uma lista por exemplo, dentro de uma ou mais `Views`.

Os elementos comumente usados para situações de conteúdo dinâmico ou não pré-determinado são: `ListView` e `GridView`. Para fazer o carregamento dos dados o Android provê alguns `Adapters` como por exemplo o `ArrayAdapter` que a partir de um `array` de dados popula os dados na `ListView` ou `GridView`.

Para responder a ações do usuário, cada `View` possui um conjunto de interfaces que podem responder a eventos, essas interfaces são chamadas de *event listeners*. Por exemplo, quando um botão é clicado, o evento `onClick` da interface `OnClickListener` é disparado. Ao associar a interface implementada ao botão, é possível implementar a resposta desejada para essa ação do usuário [64].

2.2.4 Camada de Apresentação Android

Um assunto essencial para o entendimento deste trabalho é explanar o que queremos dizer com "Camada de Apresentação Android". Nesta seção abordamos justamente este assunto com o objetivo de explanar como chegamos na definição aqui usada.

Em nossas pesquisas bibliográficas não foi encontrada uma definição formal sobre camada de apresentação Android. Encontramos porém, pontos na documentação oficial do Android [65] que afirmam que determinado elemento de alguma forma é parte desta camada. Por exemplo o trecho sobre *Activities* diz que "representa uma tela com interface do usuário". O trecho sobre recursos do aplicativo afirma que "um aplicativo Android é composto por outros arquivos além de código Java, ele requer recursos como imagens, arquivos de áudio e qualquer recurso relativo a apresentação visual do aplicativo" [62]. Encontramos também postagens em sites técnicos sobre Android que de alguma forma indicam que determinado elemento compõe a camada de apresentação Android, por exemplo Preussler relaciona *adapters* como parte da camada de apresentação [57]. Desta forma viu-se necessário definir quais são os elementos, para efeitos desta dissertação, que compõem a camada de apresentação em aplicativos Android.

Os primórdios de GUI (*Graphical User Interfaces* ou Interfaces de Usuário Gráficas)

foram em 1973 com o projeto Alto, desenvolvido pelos pesquisadores da Xerox Palo Alto Research Center (PARC), seguido do projeto Lisa da Apple em 1979. Estes dois projetos serviram de base e inspiração para o Machintosh, lançado pela Apple em 1985. As primeiras definições sobre GUI que surgiram nessa época abordavam sobre componentes de uso comum como ícones, janelas, barras de rolagem, menus suspensos, botões, caixas de entrada de texto; gerenciadores de janelas; arquivos de áudio, internacionalização e eventos. Antes deste período existiam apenas interfaces de linha de comando [58, 72].

Outra fonte define camada de apresentação como "informações gráficas, textuais e auditivas apresentadas ao utilizador, e as sequências de controle (como comandos de teclado, *mouse* ou toque) para interagir com o programa" [79].

Unindo as definições supracitadas, definimos que todos os elementos do Android que são apresentados ou interagem com o usuário de alguma forma auditiva, visual ou por comando de voz ou toque são elementos da **Camada de Apresentação**, são eles:

- **Activities e Fragments** Representam uma tela ou um fragmento de tela. A exemplo temos classes Java que herdam de `Activity`, `Fragment` ou classes similares.
- **Listeners** Meio pelo qual os comandos do usuário são capturados pelo aplicativo. A exemplo temos classes Java que implementam interfaces como `View.OnClickListener`.
- **Recursos do Aplicativo** Arquivos que apresentam textos, imagens, áudios, menus, interfaces gráficas (*layout*), dentre outros. Estão incluídos neste item todos os arquivos dentro do diretório `res` ainda que em seu formato Java. A exemplo podemos citar classes que herdam da classe `View` ou `ViewGroup`.
- **Adapters** Meio pelo qual são carregados conteúdos dinâmicos ou não pré-determinados na tela. A exemplo podemos citar classes que herdam da classe `BaseAdapter`.

Capítulo 3

Trabalhos Relacionados

Agrupamos os trabalhos relacionados em 3 seções: a Seção 3.1 trata de estudos recentes realizados sobre Android, a Seção 3.2 trata de estudos que analisam cheiros de código específicos à alguma tecnologia ou plataforma e a Seção 3.3, estudos sobre cheiros de código específicos à plataforma Android.

3.1 Pesquisas Focadas na Plataforma Android

Diversas pesquisas têm sido realizadas em torno da plataforma Android. É possível encontrar artigos sobre temas variados, como por exemplo segurança [25, 43, 87, 88, 27, 28, 81], análise estática de código [24, 48, 76, 43, 61] e autenticação de usuário [82, 29, 86, 85]. Neste seção apresentamos uma visão geral das pesquisas realizadas sobre Android.

Adrienne et al. [32] realizaram um estudo de usabilidade com usuários Android para entender a efetividade do sistema de permissões de usuários. Quando um usuário instala uma aplicação tem a oportunidade de rever as permissões solicitadas pelo aplicativo. Adrienne et al. [32] mostram que 17% dos usuários prestam atenção às permissões apresentadas durante a instalação. O autor conclui com recomendações para melhorar o entendimento e atenção dos usuários com relação as permissões solicitadas.

O sistema operacional Android além de possuir uma API aberta, tem um rico sistema de mensagens entre aplicações. Isso reduz o trabalho de desenvolvimento, facilitando a reutilização de componentes [25]. Entretanto o Android confere uma responsabilidade significativa aos desenvolvedores de aplicativos com relação ao risco de problemas de segurança como afirmam Kavitha et al. [43].

Infelizmente a comunicação entre aplicações pode ser detectada, capturada ou até mesmo substituída, comprometendo a privacidade e segurança do usuário conforme afirmam Chin

et al. [25].

Alguns autores têm explorado os riscos de problemas de segurança envolvendo Android [25, 43, 87, 88, 27, 28, 81]. Chin et al. [25] examinam iterações entre aplicações Android e identificam os riscos de segurança nos componentes. Kavitha et al. [43] investigam os problemas de segurança envolvendo o controle de permissão.

Uma das contribuições do trabalho de Chin et al. [25] é uma ferramenta para detecção de vulnerabilidades na comunicação entre aplicações chamada ComDroid. A ferramenta ComDroid baseia-se no DEX do aplicativo. Isto permite terceiros ou revisores utilizarem a ferramenta para avaliar aplicativos mesmo quando não têm disponível o código fonte. O funcionamento da ferramenta ComDroid é baseado na análise estática da saída do Dedexer [82]. A análise estática tem sido comumente utilizada para a detecção de bugs [24, 48, 76].

Kavitha et al. [43] também utilizam análise estatística em conjunto com análise dinâmica em um sistema de detecção de malware com base no controle de permissão do Android e os passos necessários para mitigar o acesso a permissões indesejadas.

Shabtai et al. [61] utilizam análise estática para classificar aplicações Android em dois tipos: ferramentas e jogos. Do mesmo modo que o trabalho de Chin et al. [25] os autores Shabtai et al. [61] também se baseiam no DEX, entretanto, mesmo utilizando aprendizado de máquina em conjunto com análise estática alguns aplicativos não conseguiram ser classificados corretamente.

Bläsing et al. [20] propõem um sandbox para análises de aplicações Android denominado AASandbox. O AASandbox é capaz de executar análises estáticas e dinâmicas em programas Android para detectar automaticamente aplicativos suspeitos. A análise estática do software tem foco em padrões maliciosos e utiliza um emulador Android na nuvem em ambiente isolado, fornecendo detecção rápida e distribuída de software suspeito. Além disso, o AASandbox pode ser utilizado para melhorar a eficiência dos antivírus disponíveis para o sistema operacional Android.

Os usuários confiam cada vez mais em aplicações móveis para necessidades computacionais. Android é uma plataforma móvel muito popular, desta maneira a confiabilidade de aplicativos Android está se tornando cada vez mais importante [40, 71]. De acordo com Wasserman [77], o desafio de engenharia de software com desenvolvimento de aplicações móveis é o de encontrar soluções eficazes para alcançar qualidades e definir técnicas e ferramentas adequadas para suportar seus testes. Testar aplicativos do Android é uma atividade desafiadora, com vários problemas abertos, problemas específicos e perguntas em aberto principalmente

em relação a GUI [40, 71, 17].

Hu e Neamtiu [40] apresentam uma abordagem para automatizar o processo de teste para aplicativos Android com foco em bugs de interface, como erros de ACTIVITY e EVENT. Primeiramente os autores realizam um estudo de bugs para entender a natureza e a frequência de bugs que afetam aplicativos Android. Em seguida Hu e Neamtiu [40] apresentam técnicas de detecção de erros GUI utilizando geração automática de casos de teste, utilizando eventos aleatórios da aplicação e instrumentação da máquina virtual, produzindo arquivos de log que serão utilizados na análise pós-execução. As técnicas apresentadas mostraram-se eficazes para erros de ACTIVITY, EVENT e TYPE ERRORS.

Amalfitano et al. [17] apresentam o AndroidRipper uma técnica automatizada que testa aplicativos Android através da GUI. AndroidRipper é baseado em um explorador automático da GUI do aplicativo com o objetivo de exercitar o aplicativo de forma estruturada. Uma das contribuições de Amalfitano et al. [17] é a disponibilização do aplicativo AndroidRipper com código aberto. Os resultados apresentados demonstra a capacidade de detectar falhas graves, previamente desconhecidas no código, e que a exploração estruturada supera a abordagem aleatória.

Dispositivos móveis tornaram-se uma parte importante na vida das pessoas [82, 85]. A identificação de dispositivos é de grande importância para a autenticação segura de usuários em redes móveis e tem atraído a atenção de muitos pesquisadores [82, 85, 30].

Yildirim et al. [85] propõem o uso do leitor biométrico somado a identidade de equipamento móvel internacional (IMEI) para a geração de uma senha única e expirável para a autenticação em sistemas web. Alguns fabricantes de dispositivos móveis permitem que os desenvolvedores usem os recursos de segurança do dispositivo em seus aplicativos por meio do kit de desenvolvimento do dispositivo (SDK) do dispositivo.

Wu et al. [82] propõem que os dados obtidos a partir de um smartphone podem ser usados para identificar o usuário do smartphone, sem a necessidade de alguma ação pelo usuário. O autor afirma que Métodos tradicionais utilizam identificadores explícitos como o IMEI, a identidade de assinante móvel internacional (IMSI) ou o Android ID. No entanto, alguns identificadores explícitos não são confiáveis e podem ser facilmente adulterados ou forjados, além de que, para obtê-los, é necessário solicitar permissão do usuário o que pode causar em abuso de permissões sensíveis e vazamento da privacidade [29, 86].

Para resolver esses problemas, Wu et al. [82] propõem três algoritmos de identificação de dispositivos baseado em identificadores implícitos que podem ser obtidos sem solicitar

qualquer permissão [82]. Os identificadores implícitos são compostos por dados obtidos (i) da camada física (hardware) como espaço em disco, (ii) da camada de aplicação (sistema operacional) como versão do Android e (iii) da camada do usuário (configurações) como time-zone.

3.2 Cheiros de Código Específicos de Tecnologia ou Plataforma

Diversos pesquisadores propuseram cheiros de código e práticas recomendadas para tecnologias ou plataformas específicas, como frameworks Java [23, 18], linguagem web como Cascading Style Sheets (CSS) [37] e Javascript [31] e outros.

Chen et al. [23] afirmam que frameworks Object-Relational Mapping (ORM) são amplamente utilizado na indústria. No entanto, os desenvolvedores geralmente escrevem código ORM sem considerar o impacto desse código no desempenho de banco de dados, levando a transações com "timeout"ou travamentos em sistemas em larga escala. Chen et al. [23] solucionam esse problema com implementação de um framework automatizado e sistemático para detectar e priorizar anti-patterns de desempenho para aplicações desenvolvidas usando ORM. Estudos de caso mostraram que o framework pode detectar centenas ou milhares de instâncias de anti-patterns de desempenho ao mesmo tempo que prioriza efetivamente a correção dessas instâncias [23]. Foi descoberto que a correção dessas instâncias de anti-patterns de desempenho pode melhorar o tempo de resposta dos sistemas em até 98% (e, em média, 35%). Além do framework que é extensível podendo agregar outros anti-patterns, Chen et al. [23] contribuem com o mapeamento de 2 anti-patterns específicos a frameworks ORM.

Aniche et al. [18] também investigaram cheiros de código relacionado a um framework. Segundo Aniche et al. [18], para escrever código fácil de ser mantido e evoluído, e detectar pedaços de código problemáticos, desenvolvedores fazem uso de métricas de código e estratégias de detecção de maus cheiros de código. No entanto, métricas de código e estratégias de detecção de maus cheiros de código não levam em conta a arquitetura do software em análise o que significa que todas classes são avaliadas como se umas fossem iguais às outras. Aniche et al. [18] afirmam que cada papel arquitetural possui responsabilidades diferentes o que resulta em distribuições diferentes de valores de métrica de código. Mostram ainda que classes que cumprem um papel arquitetural específico, como por exemplo Controllers, também contêm maus cheiros de código específicos. Uma das contribuições de Aniche et al. [18] é um catálogo com 6 cheiros de códigos específicos ao framework Spring MVC mapeados e validados.

CSS é amplamente utilizado nas aplicações web de hoje para separar a semântica de apresentação do conteúdo HTML [37]. De acordo como Gharachorlu [37] apesar da simplicidade de sintaxe do CSS, as características específicas da linguagem tornam a criação e manutenção de CSS uma tarefa desafiadora. Foi realizando um estudo empírico de larga escala em 500 sites, 5060 arquivos no total, que consistem de mais de 10 milhões de linhas de código CSS. Segundo o autor, os resultados indicaram que o CSS de hoje sofre significativamente de padrões inadequados e está longe de ser um código bem escrito. Por fim Gharachorlu [37] propõe o primeiro modelo de qualidade de código CSS derivado de uma grande amostra de aprendizagem de modo a ajudar desenvolvedores a obter uma estimativa do número total de cheiros de código em seu código CSS. Sua principal contribuição foi oito novos cheiros de código CSS detectados com o uso da ferramenta CSSNose, também implementada e disponibilizada pelo autor.

Javascript é uma flexível linguagem de script para o desenvolvimento de aplicações Web 2.0 [31]. Fard e Ali [31] afirmam que devido à essa flexibilidade, o Javascript é uma linguagem particularmente desafiadora para escrever e manter código. Os desafios são múltiplos: Primeiro, é uma linguagem interpretada, o que significa que normalmente não há compilador no ciclo de desenvolvimento que ajudaria os desenvolvedores a detectar código incorreto ou não otimizado. Segundo, tem uma natureza dinâmica, fracamente tipificada, assíncrona. Terceiro, ele suporta recursos intrincados, como prototypes [56], funções de primeira classe e "closures"[26]. E finalmente, ele interage com o DOM através de um mecanismo complexo baseado em eventos [75]. Os autores propõem um conjunto de 13 cheiros de código Javascript, sendo 7 cheiros de códigos bem conhecidos adaptados para o Javascript e 6 tipos específicos de códigos de Javascript devidos do trabalho. Também é apresentada uma técnica automatizada, chamada JSNOSE, para detectar esses cheiros de código.

Abílio et al. [15] descrevem no artigo um estudo exploratório sobre maus cheiros de código em um ambiente que não utiliza engenharia de software convencional. A abordagem das Linhas de Produtos de Software (Software Product Lines, SPL) centra-se no uso de técnicas de engenharia que permitem criar um grupo de sistemas de software similares a partir de um conjunto de especificações de software comuns a todos esses sistemas. Diferenciando da engenharia de software convencional principalmente pela presença de variação em alguns ou até todos os requisitos de software [51, 42, 80]. Programação Orientada a Recursos (do inglês *Feature-Oriented Programming*, FOP) é um paradigma para a modularização de software em que as características são as principais abstrações [22]. Os recursos podem ser realizados em artefatos separados usando abordagens de composição com FOP e Programação Orientada

a Aspecto (do inglês, *Aspect-Oriented Programming*, AOP) [19].

No entanto, a Programação Orientada a Recursos é uma técnica específica para lidar com a modularização de recursos no SPL [15]. Uma das linguagens FOP mais populares é a AHEAD e ainda faltam estudos sistemáticos sobre a categorização e detecção de cheiros de código em SPL baseado em AHEAD [15]. Para preencher essa lacuna, Abíio et al. [15] estendem as definições de três maus cheiros de código tradicionais, *God Method*, *God Class*, e *Shotgun Surgery*, para levar em conta as abstrações de FOP, propondo novas métricas FOP para quantificar características específicas de abordagens com o AHEAD e definindo estratégias para identificar os maus cheiros de código investigados.

3.3 Cheiros de Código na Plataforma Android

Verloop [74] e Minelli e Lanza [52] focam em estudar código-fonte de aplicativos para entender se e como eles diferem dos sistemas de software tradicionais e quais são as possíveis implicações para a manutenção de aplicativos. Os autores concluem que aplicações móveis são substancialmente diferentes das aplicações de software tradicionais, por exemplo, a falta de uma fonte de alimentação permanente, vida útil curta, times de desenvolvimento com poucos desenvolvedores, projetos menores, poder de processamento limitado, muitas dependências externas. Devido a essas diferenças, pesquisas feitas em aplicações de software tradicionais podem não se aplicar a aplicações móveis.

Minelli e Lanza [52] realizam sua análise com uma ferramenta desenvolvida por ele chamada Samoa. Samoa é descrita como uma plataforma de análise de software baseada na web para analisar aplicações móveis de uma perspectiva estrutural e histórica. Os autores mostram uma série de métricas de software para as aplicações que analisa e apresenta as métricas usando diferentes tipos de gráficos. As métricas utilizadas incluem o número de pacotes, o número de classes, o número de métodos, o número de chamadas internas, o número de chamadas externas, o número de elementos principais, dentre outras. Minelli e Lanza [52] acreditam que no futuro, aplicações móveis poderão enfrentar os mesmos problemas que aplicações tradicionais.

Outro ponto importante da tese de Verloop [74] consistia em encontrar cheiros de código em projetos Android para determinar se os cheiros de código ocorrem com mais frequência no código relacionado ao Android. Para isso, foram usadas ferramentas de detecção automática de cheiros de código. Das 8 ferramentas mencionadas, apenas uma foi desenvolvida especificamente para Android e dava suporte à linguagem XML (Lint). Vale considerar que,

um projeto de aplicativo Android é composto por muitos arquivos XML [68]. Desta análise Verloop [74] derivou 4 métricas (CoreCS, CoreNLOC, NonCoreCS e NonCoreNLOC, onde NLOC significa número de linhas de código (do inglês, Number Lines of Code) que foram usadas para calcular o número de cheiros de código. Na Figura 3.1 é apresentada uma tabela com o resultado.

Code Smell	Core	Non-Core
Long Method	7.2	4.7
Large Class	2.3	2.0
Long Parameter List	<0.1	0.3
Feature Envy	1.3	1.2
Type Checking	0.9	0.4
Dead Code	1.8	2.7

Figura 3.1: *Número de cheiros de código encontrados por 1000 LOC agrupador por cheiro de código.*

Verloop [74] mostra que o cheiro de código *Long Method* é quase duas vezes mais provável de ocorrer nas classes núcleo, classes que herdam da estrutura do Android. Também mostra que o cheiro de código *Large Class* é quase tão provável de ocorrer em classes núcleo como em classes não-núcleo. Uma possível razão para isso pode ser que o número de classes de ACTIVITIES em comparação com o número total de classes é pequeno. O cheiro de código de *Long Parameter List* é quase inexistente em classes de núcleo. O cheiro de código *Feature Envy*, tal como o cheiro *Large Class*, é quase tão provável de ocorrer nas classes núcleo como nas classes não-núcleo. O cheiro de código *Type Checking* foi encontrado menos de uma vez a cada 1000 LOC, mas foi encontrado duas vezes mais frequentemente nas classes principais. Por último, o cheiro de código *Dead Code* é mais provável de ser encontrado em classes não-núcleo. O autor ainda contribui com cinco possíveis refatorações e três delas foram implementadas em um plugin do Eclipse.

Verloop [74] diz que apesar do crescimento e mudanças do mercado móvel não há muita pesquisa a ser encontrada nesta nova área de desenvolvimento de software. Esses novos desafios enfrentados por desenvolvedores móveis e a quantidade limitada de pesquisa sobre o assunto têm tornado cheiros de código para aumentar a manutenibilidade dessas aplicações em um tópico interessante afirma Verloop [74].

Linares et al. [47] usaram a ferramenta DECOR para realizar a detecção de 18 diferen-

tes *anti-patterns* orientado a objetos em aplicativos móveis desenvolvidos com Java Mobile Edition (J2ME). Este estudo em larga escala mostra que a presença de antipatterns afeta negativamente as métricas de qualidade do software, em particular as métricas relacionadas à falha.

Gottschalk e Jelschen [38] explicam como tentam melhorar o uso de energia de aplicações móveis, procurando por desperdícios de energia, padrões esses que eles denominam de cheiro de código sobre energia (textitquality code smell). Os autores produzem um catálogo com um total de 6 cheiros de códigos de energia e salientam que os mesmos são *cross-platform*, ou seja, independente de plataforma móvel. Entretanto, um código Android é dado no exemplo. Cada cheiro de código de energia catalogado tem uma descrição, instruções de como detectar e reestruturar. Reimann et al. [60] também tratam sobre cheiros de qualidade e relacionados 20 que impactam no consumo inteligente de recursos de hardware do dispositivo como eficiência no uso de energia, processamento e memória.

Reimann et al. [60] correlacionam os conceitos de mau cheiro, qualidade e refatoração a fim de introduzir o termo cheiro de qualidade (do inglês *quality smell*). Um cheiro de qualidade é uma estrutura que influencia negativamente requisitos de qualidade específicos, que podem ser resolvidos por refatorações [59]. Os autores compilaram um catálogo de 30 cheiros de qualidade para Android. O formato dos cheiros de qualidade incluem: nome, contexto, requisitos de qualidades afetados e descrição, este formato foi baseado nos catálogos de Brown et al. [21] e Fowler [4]. Todo o catálogo pode ser encontrado em `http://www.modelrefactoring.org/smell_catalog` e os mesmos também foram implementados no framework Refactory [59]. O requisitos de qualidade tratados por Reimann et al. [60] são: centrados no usuário (estabilidade, tempo de início, conformidade com usuários, experiência do usuário e acessibilidade), consumo inteligente de recursos de hardware do dispositivo (eficiência no uso de energia, processamento e memória) e segurança. Os autores ainda citam que o problema no desenvolvimento móvel é que os desenvolvedores estão cientes de cheiros de qualidade apenas indiretamente porque suas definições são informais (melhores práticas, problemas de rastreamento de bugs, discussões de fóruns etc.) e os recursos onde encontrá-los são distribuídos pela web e que é difícil coletar e analisar todas essas fontes sob um ponto de vista comum e fornecer suporte de ferramentas para desenvolvedores.

Esta dissertação pretende pela primeira vez catalogar cheiros de código especificamente relacionados a camada de apresentação de aplicativos Android. Pretende-se reaproveitar diversos dos métodos utilizados para a detecção e documentação de cheiros de código dos

diversos trabalhos citados.

Capítulo 4

Proposta de Dissertação

Esta dissertação objetiva validar a hipótese de que existem cheiros de código específicos à plataforma Android tal como diversas pesquisas [18, 47, 74] veem demonstrando que há cheiros de código específicos a tecnologias e plataformas. Geoffrey [39] afirma que a detecção e especificação de padrões móveis ainda é um problema em aberto e que *antipatterns* Android são mais frequentes em projetos móveis do que *antipatterns* orientado a objetos. Pesquisas em torno de projetos de aplicativos móveis ainda são poucas [49]. Desta forma, neste capítulo é apresentada a proposta da dissertação e o cronograma de atividades planejadas.

4.1 Métodos de Pesquisa

4.1.1 Boas e Más Práticas na Camada de Apresentação

God Class, *Large Class* e *Long Method* são exemplos de cheiros de código amplamente reconhecidos por desenvolvedores [18, 74, 47]. De fato, é possível obter métricas de qualidade de código de projetos Android utilizando estes cheiros de código já catalogados [74, 47]. No entanto, pesquisas têm demonstrado que existem cheiros de código específicos a tecnologias, frameworks e plataformas [18], desta forma sugerimos que há cheiros de código específicos à camada de apresentação de aplicativos Android. Para iniciar nossas investigações sobre esse tema, fomentamos a seguinte pergunta:

Q1: O que desenvolvedores consideram boas e más práticas com relação à Camada de Apresentação em projetos Android?

Cheiros de código são padrões de código que estão associados com um design ruim e más práticas de programação [74]. Por si só, um mau cheiro não é algo ruim, ocorre

que frequentemente ele indica um problema mas não necessariamente é o problema em si [34]. Sendo assim, esta questão visa identificar quais práticas desenvolvedores Android reconhecem como sendo más práticas, e possivelmente cheiros de código, e quais práticas os desenvolvedores reconhecem como boas práticas, e possivelmente formas de refatorar o que foi considerado um mau cheiro.

Objetivamos obter como resposta um catálogo de cheiros de código na camada de apresentação Android. Para derivar esse catálogo aplicamos um questionário respondido por 45 desenvolvedores Android e pretendemos entrevistar outros desenvolvedores sobre boas e más práticas seguidas durante o desenvolvimento de aplicativos Android. Também iremos coletar postagens relacionadas a boas e más práticas em sites sobre Android. Após, iremos realizar um procedimento de codificação aberto a partir das respostas e postagens obtidas.

Por fim, pretendemos validar os cheiros de código derivados com 2 desenvolvedores especialistas em Android.

Nós coletamos boas e más práticas na camada de apresentação seguidas por desenvolvedores durante o desenvolvimento de aplicativos Android. A coleta de dados inclui três diferentes passos, são eles:

Passo 1: Questionário Online. Elaboramos um questionário em inglês com perguntas dissertativas. O questionário foi divulgado em comunidades de desenvolvedores Android do Brasil e exterior e em redes sociais como LinkedIn, Google Plus, Facebook e Twitter. De modo a obtermos alguma análise demográfica, as primeiras questões são sobre idade e país de residência.

Em seguida, perguntamos sobre boas e más práticas. Para cada elemento da camada de apresentação fizemos o seguinte par de perguntas, onde "X" indica o elemento da camada de apresentação em questão:

- Do you have any good practices to deal with X? (Você conhece algumas boas práticas para lidar com X?)
- Do you have anything you consider a bad practice when dealing whit X? (Você considera algo uma má prática para lidar com X?)

Os elementos questionados foram definidos no Capítulo 2.2.4, são eles: Activities, Fragments, Adapter, Listeners, layouts, styles, string e drawables. Os 4 últimos representam os 4 principais recursos de aplicativos Android. Optamos por não questionar

todos os tipos de recursos individualmente para não tornar o questionário muito extenso.

Por último, com o objetivo de capturar alguma outra boa ou má prática em qualquer outro elemento da camada de apresentação, adicionamos as seguintes perguntas:

- Are there any other *GOOD* practices in Android Presentation Layer we did not asked you or you did not said yet? (Existe alguma outra *BOA* prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?)
- Are there any other *BAD* practices in Android Presentation Layer we did not asked you or you did not said yet? (Existe alguma outra *MÁ* prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?)

O questionário completo pode ser encontrado no Apêndice 1.

Passo 2: Entrevistas Semi-Extruturadas. Realizaremos entrevistas semi-estruturadas com desenvolvedores Android. O objetivo da entrevista é fazer com que os desenvolvedores discutam sobre boas e más práticas na camada de apresentação Android usadas no dia-a-dia de desenvolvimento. Essas discussões serão focadas nos oito elementos da camada de apresentação questionados no passo anterior.

Passo 3: Pesquisa em Sites Técnicos. Realizaremos uma busca na internet por postagens técnicas sobre Android que apontem alguma boa ou má prática em algum dos elementos da camada de apresentação pesquisados nos passos anteriores.

De modo a complementar os dados para análise demográfica, tanto no questionário quanto na entrevista fizemos perguntas sobre a experiência com desenvolvimento de software, experiência com desenvolvimento de aplicativos Android nativos e quais linguagens de programação se consideravam proficientes.

4.1.2 Impacto na Tendência a Mudanças e Defeitos

Evidências na literatura sugerem que cheiros de código podem esconder manutenibilidade de código e aumentar a tendência a mudanças e introdução de defeitos. Objetivamos avaliar o impacto dos cheiros de código propostos na tendência a mudanças e introdução de defeitos no código. Para isso conduziremos um experimento presencial com desenvolvedores Android.

Pretendemos conduzir um experimento apresentando aos desenvolvedores um projeto Android onde alguns deles receberão este projeto com os cheiros de código e outros o receberam com o código já refatorado.

Será solicitado a todos que implementem uma mesma funcionalidade, esta do qual, necessitará alterar arquivos da camada de apresentação. Com este experimento pretendemos validar o impacto que os cheiros de código propostos têm na tendência a alteração de código e a introdução de defeitos.

4.1.3 Percepção dos Desenvolvedores

Após a definição de um catálogo de cheiros de código, pretende-se validar se desenvolvedores os percebem de fato como indicativos de trechos de códigos ruins. Para isso pretende-se conduzir um experimento apresentando aos desenvolvedores códigos com e sem os cheiros de código propostos, para cada código, será solicitado que ele indique o código como "com cheiro" ou "sem cheiro".

4.2 Atividades

Para obter as ideias iniciais para a derivação dos cheiros de código na camada de apresentação Android, foi aplicado um questionário sobre boas e más práticas Android na comunidade de desenvolvedores do Brasil e exterior. O questionário pode ser encontrado no Apêndice A e até o momento da escrita desta proposta de qualificação foram coletadas 44 respostas. Ainda de modo a complementar os dados coletados com o questionário, pretende-se realizar entrevista com desenvolvedores Android sobre o mesmo tema. Também será desenvolvida uma análise para derivar os cheiros de código. Essa análise será feita com base em estratégias já utilizadas em trabalhos anteriores como o de Aniche et al. [18]. Para reduzir viés sobre os cheiros de código definidos, os mesmos serão validados com mais de um especialista em Android. A derivação dos cheiros de código está relacionada à Q1 definida na Seção 1.1 e as atividades planejadas são:

- Bibliografia.
- Survey sobre Boas e Más práticas Android.
- Derivação dos cheiros de código.
- Validação de cheiros de código com especialista Android.

Evidências na literatura sugerem que cheiros de código podem dificultar manutenibilidade de código [69, 83, 84] e aumentar a tendência a mudanças e introdução de defeitos [44, 45]. Linares et al. [47] demonstram que *antipatterns* impactam negativamente métricas relacionadas com a qualidade em projetos móveis, em particular métricas relacionadas com a propensão de falhas. Desta forma, pretende-se avaliar o impacto dos cheiros de código propostos na tendência a mudanças e introdução de defeitos no código. Para isso, será realizado um estudo de projetos com desenvolvedores Android. Esse experimento está relacionado à Q2 definida na Seção 1.1 e a atividade planejada é:

- Experimento sobre o Impacto em Mudanças/Defeitos.

Evidências na literatura também sugerem que cheiros de código são percebidos por desenvolvedores [55], desta forma pretende-se avaliar se desenvolvedores Android percebem códigos afetados pelos cheiros de código propostos como indicativos de trechos de códigos ruins. Para isso será conduzido outro experimento também com desenvolvedores Android. Esse experimento está relacionado à Q3 definida na Seção 1.1 e a seguinte atividade está planejada:

- Experimento sobre a Percepção Desenvolvedores.

4.3 Cronograma

Na Tabela 4.1 são apresentadas as atividades previstas para a conclusão da dissertação bem como em qual período pretende-se realizá-la.

Atividades	2016		2017			
	3^o Tri	4^o Tri	1^o Tri	2^o Tri	3^o Tri	4^o Tri
Bibliografia e Trabalhos Relacionados	•	•				
Survey Boas e Más práticas Android		•				
Entrevista Boas e Más práticas Android			•			
Derivação dos cheiros de código			•			
Validação cheiros de código com Especialista			•			
Experimento Impacto em Mudanças/Defeitos				•		
Experimento Percepção Desenvolvedores				•		
Escrita da Dissertação	•	•	•	•	•	•
Defesa						•

Tabela 4.1: *Cronograma de atividades propostas.*

Apêndice A

Questionário sobre Boas e Más Práticas

Android Good & Bad Practices

Help Android Developers around the world in just 15 minutes letting us know what you think about good & bad practices in Android native apps. Please, answer in portuguese or english.

Hi, my name is Suelen, I am a Master student researching code quality on native Android App's Presentation Layer. Your answers will be kept strictly confidential and will only be used in aggregate. I hope the results can help you in the future. If you have any questions, just contact us at suelengcarvalho@gmail.com.

Questions about Demographic & Background. Tell us a little bit about you and your experience with software development. All questions throught this session were mandatory.

1. What is your age? (One choice beteen 18 or younger, 19 to 24, 25 to 34, 35 to 44, 45 to 54, 55 or older and I prefer not to answer)
2. Where do you currently live?
3. Years of experience with software development? (One choice between 1 year or less, one option for each year between 2 and 9 and 10 years or more)
4. What programming languages/platform you consider yourself proficient? (Multiples choices between Swift, Javascript, C#, Android, PHP, C++, Ruby, C, Python, Java, Scala, Objective C, Other)
5. Years of experience with developing native Android applications? (One choice between 1 year or less, one option for each year between 2 and 9 and 10 years or more)

6. What is your last degree? (One choice between Bacharel Student, Bacharel, Master, PhD and Other)

Questions about Good & Bad Practices in Android Presentation Layer. We want you to tell us about your experience. For each element in Android Presentation Layer, we want you to describe good & bad practices (all of them!) you have faced and why you think they are good or bad. With good & bad practices we mean anything you do or avoid that makes your code better than before. If you perceive the same practice in more than one element, please copy and paste or refer to it. All questions throught this session were not mandatory.

1. **Activities** An activity represents a single screen.
    - Do you have any good practices to deal with Activities?
    - Do you have anything you consider a bad practice when dealing with Activities?
2. **Fragments** A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities.
    - Do you have any good practices to deal with Fragments?
    - Do you have anything you consider a bad practice when dealing with Fragments?
3. **Adapters** An adapter adapts a content that usually comes from model to a view like put a bunch of students that come from database into a list view.
    - Do you have any good practices to deal with Adapters?
    - Do you have anything you consider a bad practice when dealing with Adapters?
4. **Listeners** An event listener is an interface in the View class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.
    - Do you have any good practices to deal with Listeners?
    - Do you have anything you consider a bad practice when dealing with Listeners?
5. **Layouts Resources** A layout defines the visual structure for a user interface.

- Do you have any good practices to deal with Layout Resources?
- Do you have anything you consider a bad practice when dealing with Layout Resources?

6. **Styles Resources** A style resource defines the format and look for a UI.
   - Do you have any good practices to deal with Styles Resources?
   - Do you have anything you consider a bad practice when dealing with Styles Resources?
7. **String Resources** A string resource provides text strings for your application with optional text styling and formatting. It is very common used for internationalizations.
   - Do you have any good practices to deal with String Resources?
   - Do you have anything you consider a bad practice when dealing with String Resources?
8. **Drawable Resources** A drawable resource is a general concept for a graphic that can be drawn to the screen.
   - Do you have any good practices to deal with Drawable Resources?
   - Do you have anything you consider a bad practice when dealing with Drawable Resources?

Last thoughts Only 3 more final questions.

1. Are there any other *GOOD* practices in Android Presentation Layer we did not asked you or you did not said yet?
2. Are there any other *BAD* practices in Android Presentation Layer we did not asked you or you did not said yet?
3. Leave your e-mail if you wish to receive more information about the research or participate in others steps.

Referências Bibliográficas

[1] Activities. https://developer.android.com/guide/components/activities.html. Last accessed at 29/08/2016. 3

[2] Android version history. https://en.wikipedia.org/wiki/Android_version_history. Last accessed at 27/11/2016. 1, 9

[3] Wikipedia google play store. Last accessed at 27/11/2016. 10

[4] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 25

[5] Google android software spreading to cars, watches, tv. http://phys.org/news/2014-06-google-android-software-cars-tv.html, June 2014. Last accessed at 26/07/2016. 1

[6] Android fragmentation visualized. http://opensignal.com/reports/2015/08/android-fragmentation, August 2015. Last accessed at 12/09/2016. 13

[7] Ford terá apple carplay e android auto em todos os modelos nos eua. http://g1.globo.com/carros/noticia/2016/07/ford-tera-apple-carplay-e-android-auto-em-todos-os-modelos-nos-eua.html, 2016. Last accessed at 26/07/2016. 1

[8] Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016. http://www.gartner.com/newsroom/id/3323017, May 2016. Last accessed at 23/07/2016. 1

[9] Global smartphone sales to end users from 1st quarter 2009 to 1st quarter 2016, by operating system (in million units). http://www.statista.com/statistics/266219/global-smartphone-sales-since-1st-quarter-2009-by-operating-system, 2016. Last accessed at 24/07/2016. v, 1, 9

[10] Number of available applications in the google play store from december 2009 to february 2016. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store, 2016. Last accessed at 24/07/2016. v, 1, 10

[11] Pmd (2016). https://pmd.github.io, 2016. Last accessed at 29/08/2016. 2, 8

[12] Sonarqube (2016). http://www.sonarqube.org, 2016. Last accessed at 29/08/2016. 2, 8

[13] Wikipedia code smell. https://en.wikipedia.org/wiki/Code_smell, 2016. Last accessed at 14/11/2016. 7, 8

[14] Worldwide smartphone growth forecast to slow to 3.1% in 2016 as focus shifts to device lifecycles, according to idc. http://www.idc.com/getdoc.jsp?containerId=prUS41425416, June 2016. Last accessed at 23/07/2016. 1

[15] Ramon Abilio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting code smells in software product lines – an exploratory study. 2015. 22, 23

[16] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2 edition, 2003. 2

[17] Domenico Amalfitano, Anna Fasolino, Porfirio Tramontana, Salvatore Carmine, and Atif Memon. Using gui ripping for automated testing of android applications. 2012. 20

[18] MaurÃŋcio Aniche and Marco Gerosa. Architectural roles in code metric assessment and code smell detection. 2016. 2, 4, 7, 8, 21, 26, 29

[19] Kastner C. Apel S. An overview of feature-oriented software development. 2009. 23

[20] Blasing and Thomas. An android application sandbox system for suspicious software detection. 2010. 19

[21] William J. Brown, Raphael C. Malveau, and Thomas J. Mowbray Hays W. "Skip" McCormick. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998. 8, 25

[22] Prehofer C. Feature-oriented programming: A fresh look at objects. 1997. 22

[23] Chen, TseHsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed Hassan, and Parminder Flora Mohamed Nasser. Detecting performance anti-patterns for applications developed using object-relational mapping. 2014. 21

[24] B. Chess and G. McGraw. Static analysis for security. 2004. 18, 19

[25] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. 2011. 18, 19

[26] D. Crockford. Javascript: the good parts. 2008. 22

[27] Enck, William, and Patrick Drew McDaniel Machigar Ongtang. Understanding android security. 2009. 18, 19

[28] Enck, William, and Patrick McDaniel Machigar Ongtang. Mitigating android software misuse before it happens. 2008. 18, 19

[29] Z. Fang and Y. Li W. Han. Permission based android security: Issues and countermeasures. 2014. 18, 20

[30] Zheran Fang and Yingjiu Li Weili Han. Permission based android security: Issues and countermeasures. 2014. 20

[31] A. Milani Fard and A. Mesbah. Jsnose: Detecting javascript code smells. 2013. 21, 22

[32] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM. 18

[33] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999. 2, 7, 8

[34] Martin Fowler. Code smell. http://martinfowler.com/bliki/CodeSmell.html, February 2006. 7, 8, 26

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison Wesley, Boston, 1995. 8

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 2

[37] Golnaz Gharachorlu. *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model*. PhD thesis, The University of British Columbia, 2014. 2, 21, 22

[38] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. Maus cheiros relacionados ao consumo de energia. 25

[39] Geoffrey Hecht. An approach to detect android antipatterns. 2015. 1, 2, 3, 26

[40] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. 2011. 19, 20

[41] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, 1996. 2, 8

[42] Pohl K. Software product line engineering: Foundations, principles and techniques. 2005. 22

[43] K Kavitha, P Salini, and V Ilamathy. Exploring the malicious android applications and reducing risk using static analysis. 2016. 18, 19

[44] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, Washington, DC, USA, 2009. IEEE Computer Society. 30

[45] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Softw. Engg.*, 17(3), June 2012. 7, 30

[46] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988. 2

[47] Mario Linares-Vásquez, Sam Klock, Collin Mcmillan, Aminata Sabanè, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. Domain matters: Bringing further evidence of the

relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 2, 7, 8, 9, 24, 26, 30

[48] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. 2005. 18, 19

[49] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding code smells in android applications. 2, 3, 9, 26

[50] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. 2

[51] John D. McGregor. Software product lines. 2004. 22

[52] Roberto Minelli and Michele Lanza. Software analytics for mobile applications, insights & lessons learned. *In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013. 23

[53] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Meur, and Laurence Duchien. Fundamental approaches to software engineering. 2008. 9

[54] Juliana Padilha. Detecção de anomalias de código usando métricas de software. 2013. 7, 9

[55] Fabio Palomba, Gabriele Bavota, Massimiliano Penta, Rocco Oliveto, and Andrea Lucia. Do they really smell bad? a study on developers' perception of bad code smells. pages 101–110, 2014. 30

[56] S. Porto. A plain english guide to javascript prototypes. 2013. 22

[57] Danny Preussler. Writing better adapters. https://medium.com/\spacefactor\@m{}dpreussler/writing-better-adapters-1b09758407d2#.tbvvw3krr, 2016. Last accessed at 27/10/2016. 16

[58] Eric Steven Raymond. *The Art of Unix Usability*. 2004. Last accessed at 26/10/2016. 17

[59] Jan Reimann and Uwe Assmann. Quality-aware refactoring for early detection and resolution of energy deficiencies. 2013. 25

[60] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2013. 2, 3, 8, 25

[61] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. *ieee*, 2010. 18, 19

[62] Android Developer Site. Andriod fundamentals. https://developer.android.com/guide/components/fundamentals.html. Last accessed at 04/09/2016. 10, 11, 12, 13, 14, 16

[63] Android Developer Site. Plataform architecture. https://developer.android.com/guide/platform/index.html. Last accessed at 04/09/2016. v, 11

[64] Android Developer Site. Ui events. https://developer.android.com/guide/topics/ui/ui-events.html. Last accessed at 25/11/2016. 16

[65] Android Developer Site. Documentação site android developer. https://developer.android.com, 2016. Last accessed at 27/10/2016. 16

[66] Android Developer Site. Resource type. https://developer.android.com/guide/topics/resources/available-resources.html, 2016. Last accessed at 12/09/2016. 13, 15

[67] Android Developer Site. Ui overview. https://developer.android.com/guide/topics/ui/overview.html, 2016. Last accessed at 23/10/2016. v, 14, 15

[68] Developer Android Site. Resources overview. https://developer.android.com/guide/topics/resources/overview.html, 2016. Last accessed at 08/09/2016. 2, 24

[69] Dag Sjoberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *Ieee T Software Eng*, 2013. 30

[70] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014. 8

[71] Takala, Tommi, and Julian Harty Mika Katara. Experiences of system-level model-based gui testing of an android application. 2011. 19, 20

[72] TecMundo. A história da interface gráfica. http://www.tecmundo.com.br/historia/9528-a-historia-da-interface-grafica.htm. Last accessed at 26/10/2016. 17

[73] Nikolaos Tsantalis. *Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings*. PhD thesis, University of Macedonia, August 2010. 7

[74] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 1, 3, 7, 8, 23, 24, 26

[75] W3C. Document object model (dom) level 2 events specication. 2000. 22

[76] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. 2000. 18, 19

[77] Anthony I Wasserman. Software engineering issues for mobile application development. 2010. 19

[78] Bruce Webster F. *Pitfalls of Object-Oriented Development*. M & T Books, 1995. 7, 8

[79] Wikipedia. Interface do utilizador. https://pt.wikipedia.org/wiki/Interface_do_utilizador. Last accessed at 26/10/2016. 17

[80] Wikipedia. Linhas de produtos de software. Last accessed at 28/10/2016. 22

[81] Wu and Lei. The impact of vendor customizations on android security. 2013. 18, 19

[82] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. Eficient fingerprinting-based android device identication with zero-permission identiers. 2016. 18, 19, 20, 21

[83] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Sept 2012. 30

[84] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 682–691, Piscataway, NJ, USA, 2013. IEEE Press. 30

[85] Nilan Yildirim and Asaf Varol. Android based mobile application development for web login authentication using fingerint recognition feature. 18, 20

[86] S. Yu. Big privacy: Challenges and opportunities of privacy study in the age of big data. 2016. 18, 20

[87] Yuan Zhang, Min Yang, Zhemin Yang, Guofei GU, and Binyu Zang Peng Ning. Exploring permission induced risk in androidâĂŞapplications for malicious detection. 2004. 18, 19

[88] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. Permission use analysis for vetting undesirable behaviors in android apps. 2014. 18, 19