Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Suelen Goularte Carvalho

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciência da Computação

Orientador: Marco Aurélio Gerosa, Ph.D.

São Paulo, Julho de 2016

Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Esta é a versão original da dissertação elaborada pela candidata Suelen Goularte Carvalho, tal como submetida a Comissão Julgadora.

Comissão Julgadora:

- Marco Aurélio Gerosa, Ph.d. – IME-USP
- Alfredo Goldman vel Lejbman, Ph.d. – IME-USP
- Paulo Roberto Miranda Meirelles, Ph.d. – IME-USP

Dedico esta dissertação de mestrado a minha mãe.

"O motivo do tempo é que tudo não acontece de uma vez só."
– Albert Einstein

Agradecimentos

A fazer.

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 se tornado 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 anomalias ou maus cheiros, e podem degradar a qualidade do projeto, tornando-o de difícil manutenção. Consequentemente, desenvolvedores de software precisam identificar trechos de código problemáticos 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 maus cheiros de código. Apesar de já existirem diversos maus cheiros 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 maus 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 maus cheiros 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 maus cheiros 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, maus cheiros, qualidade de código, engenharia de software, manutenção de software, métricas de código.

Abstract

A task that constantly software developers need to do is identify problematic code snippets so they can refactor, with the ultimate goal to have constantly a base code easy to be maintained and evolved. For this, developers often make use of code smells detection strategies. Although there are many code smells cataloged, such as *God Class*, *Long Method*, among others, they do not take into account the nature of the project. However, Android projects have relevant features and untested to date, for example the `res` directory that stores all resources used in the application or an ACTIVITY by nature, accumulates various responsibilities. Research in this direction, specific on Android projects, are still in their infancy. In this dissertation we intend to identify, validate and document code smells of Android regarding the presentation layer, where major distinctions when compared to traditional designs. In other works on Android code smells, were identified code smells related to security, intelligent consumption of device's resources or somehow influenced the experience or user expectation. Unlike them, our proposal is to catalog Android code smells which influence the quality of the code. It developers will have another ally tool for quality production code.

Keywords: android, code smells, code quality, software engineering, software maintanence, code metrics.

Sumário

Lista de Abreviaturas **vi**

Lista de Figuras **vii**

1 Introdução **1**

1.1 Questões de Pesquisa 4

1.2 Contribuições 5

1.3 Organização da Dissertação 5

2 Fundamentação Conceitual **7**

2.1 Refatoração 7

2.2 Maus Cheiros de Código 8

2.3 Android 9

2.3.1 Fundamentos do Desenvolvimento Android 9

2.3.2 Recursos do Aplicativo 12

2.3.3 Interfaces de Usuários 13

2.3.4 Camada de Apresentação Android 15

3 Trabalhos Relacionados **17**

3.1 Pesquisas sobre Android 17

3.2 Maus Cheiros Específicos 18

3.3 Maus Cheiros Android 18

4 Proposta de Dissertação **20**

4.1 Atividades 20

4.2 Cronograma 21

A Questionário sobre Boas e Más Práticas **23**

Referências Bibliográficas **26**

Lista de Abreviaturas

Software Development Kit Kit para Desenvolvimento de Software

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

Lista de Figuras

1.1 Unidades de disposivitos vendidos por plataforma móvel [7]. 1

1.2 Quantidade de aplicativos disponíveis na Google Play Store [8]. 2

2.1 Arquitetura do sistema operacional Android [38]. 10

2.2 Árvore hierárquica de `Views` e `ViewGroups` do Android [42]. 13

Capítulo 1

Introdução

Android é a plataforma móvel de maior crescimento e em 2017 completará uma década desde seu primeiro lançamento. A Figura 1.1 apresenta a quantidade de dispositivos vendidos para cada plataforma móvel, no período de 2009 à 2016 [7]. 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.

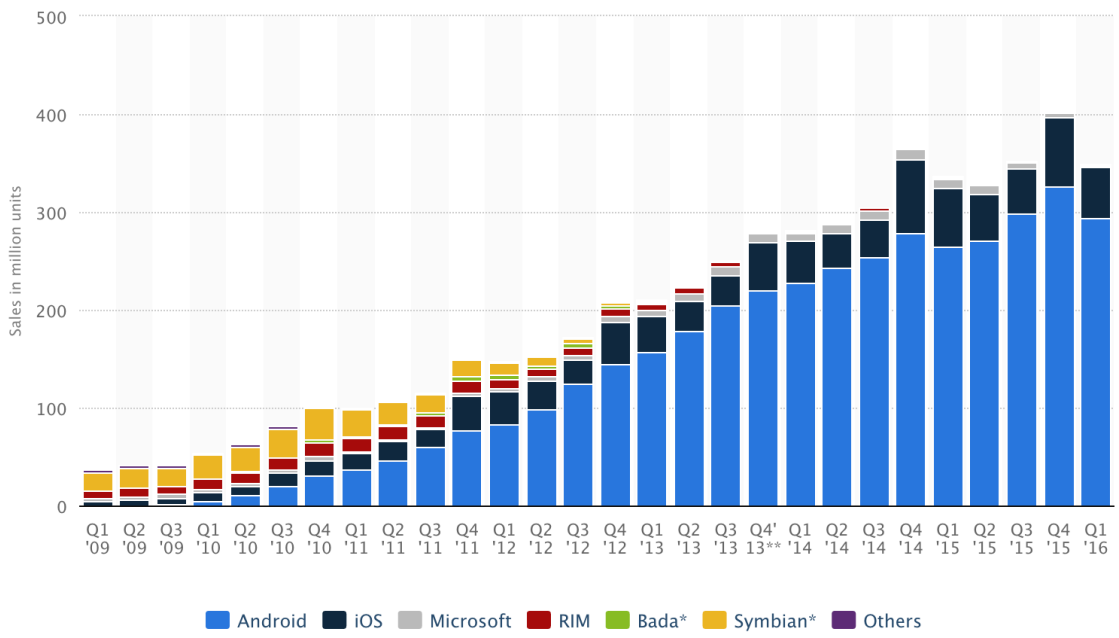

Figura 1.1: *Unidades de disposivitos vendidos por plataforma móvel [7].*

Com o crescimento da plataforma, a demanda por aplicativos também aumenta. Na Figura 1.2 é possível acompanhar o crescimento da quantidade de aplicativos disponíveis na

Google Play Store, loja oficial de aplicativos Android, ao longo de 2009 até 2016, sendo que nesse último ano, superou 200 milhões de aplicativos disponíveis [8].

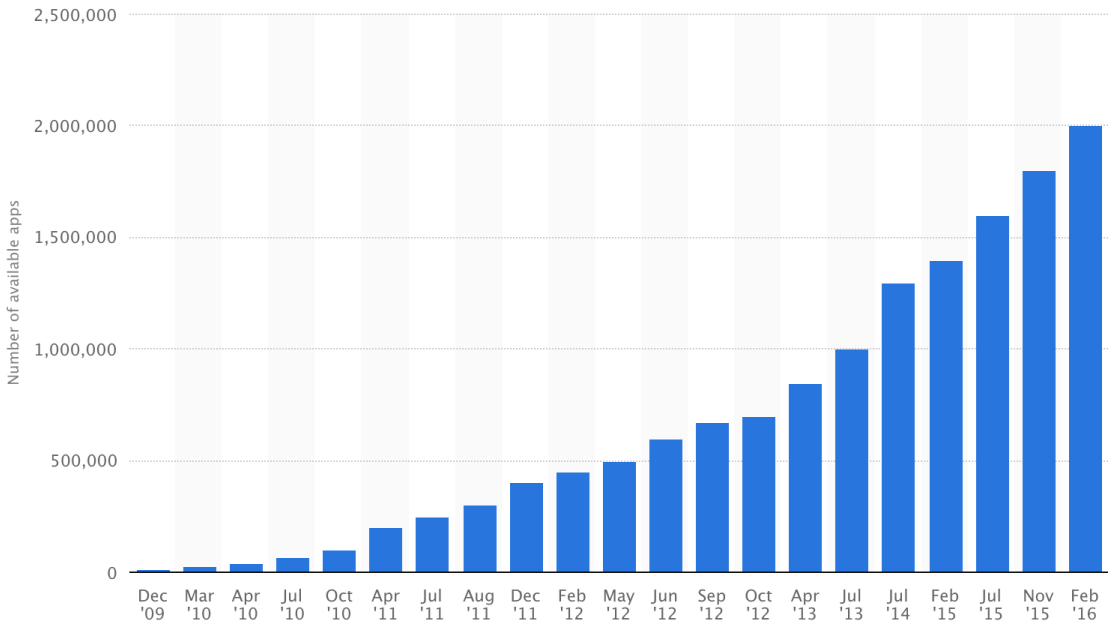

Figura 1.2: *Quantidade de aplicativos disponíveis na Google Play Store [8].*

Ainda, segundo Gartner Group [6] e IDC [12], mais de 83,5% dos dispositivos móveis no mundo usam o sistema operacional Android, e esse percentual vem crescendo ano após ano. Atualmente é possível encontrar Android em diversos tipos de dispositivos como: *smartphones*, *smart TVs*, *smartwatchs*, carros, dentre outros [3, 5].

Todo esse crescimento da plataforma impacta no processo de desenvolvimento de software desses aplicativos. Segundo Geoffrey [22] *"aplicativos móveis têm se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. No entanto, atender a esses requisitos pode resultar em más escolhas de design, conhecidas como antipatterns, que podem degradar a qualidade e desempenho do software"* (tradução livre). Verloop [47] também aborda esse cenário ao dizer que aplicativos móveis tem o ciclo de vida curto e precisam ser constantemente atualizados para se menterem relevantes e atender as expectativas de seus usuários. Verloop [47] complementa dizendo que a combinação do rápido crescimento e atualizações implicam no desenvolvimento de software, de modo que, os desenvolvedores devem aumentar a importância em manter seus projetos com fácil manutenção.

Apesar de ser possível analisar projetos Android através de maus cheiros "tradicionais"

(por exemplo, *God Classes* e *Long Methods*), pesquisas têm demonstrado que diferentes plataformas, linguagens e frameworks podem apresentar métricas de qualidade de código específicas [14, 52]. Projetos Android possuem características específicas principalmente com relação à camada de apresentação [47].

Conforme relatado por Hecht [22] com relação a projetos Android, "*antipatterns* específicos à plataforma Android são mais comuns e ocorrem mais frequentemente do que *antipaterns* OO (Orientados a Objetos)" (tradução livre). Vale lembrar que além de código Java, grande parte de um projeto Android é constituído por arquivos `XML`. Estes são os *recursos da aplicação* (do inglês *Application Resources*) e ficam localizados no diretório `res` do projeto. São responsáveis por apresentar algo ao usuário como, uma tela, uma imagem, uma tradução e assim por diante. No início do projeto, os recursos costumam ser poucos e pequenos. Conforme o projeto evolui, a quantidade e complexidade dos recursos tende a aumentar, trazendo problemas para encontrá-los, reaproveitá-los e entendê-los. Enquanto esses problemas já estão bem resolvidos em projetos orientados a objetos, ainda não é trivial encontrar uma forma sistemática de identificá-los em recursos de projetos Android.

Outra característica é com relação à Activities, que são classes específicas da plataforma Android responsáveis pela apresentação e interações do usuário com a tela [1]. Activities também possuem muitas responsabilidades [47], estão vinculadas a um Layout que representa uma interface com o usuário e normalmente precisam de acesso a classes do modelo da aplicação. Analogamente ao padrão MVC, Activities fazem os papéis de View e Controller simultaneamente. Isto posto, é razoável considerar que o mau cheiro *God Class* [34] é aplicável nesse caso, no entanto, conforme bem pontuado por Aniche et al. [14] "*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).

Na prática, desenvolvedores Android percebem estes problemas frequentemente. Muitos deles já se utilizam de práticas para solucioná-los, conforme relatado por Reimann et al. [33] "*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-los estão distribuídos pela internet*" (tradução livre). Ou seja, não é encontrado atualmente um catálogo único de boas e más práticas, tornando difícil a detecção e sugestão de refatorações apropriadas às particularidades da plataforma.

Pesquisas sobre Android ainda são poucas. Nas principais conferências de manutenção de software, dentre 2008 a 2015, 5 artigos foram sobre maus cheiros Android, dentro de um total de 52 artigos sobre o assunto [27]. 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 forma 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 maus cheiros específicos à camada de apresentação de projetos Android. Desta forma, trabalhamos a seguinte questão de pesquisa:

Existem Maus Cheiros específicos à Camada de Apresentação Android?

Para isso, exploramos as seguintes questões:

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

Nesta questão, nós investigamos a existência de maus cheiros em elementos da camada de apresentação Android como Activities e Adapters. Para responder a esta pergunta aplicamos questionário e realizamos entrevistas com desenvolvedores especialistas em Android. Também coletamos postagens em fóruns e blogs técnicos sobre Android.

Q2: Qual a relação entre os maus cheiros propostos e a tendência a mudanças e defeitos no código?

Estudos prévios mostram que maus cheiros tradicionais (e.g., *Blob Classes*) podem impactar na tendência a mudanças em classes do projeto [14]. Desta forma, esta questão pretende, por meio de um experimento com desenvolvedores Android, analisar o impacto dos maus cheiros propostos na tendência a mudanças e defeitos em projetos Android.

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

Com esta questão complementamos com dados qualitativos as análises quantitativas realizadas no contexto de Q2. Desta forma, investigamos se códigos afetados pelos maus cheiros definidos para a camada de apresentação Android são percebidos como problemáticos por desenvolvedores.

Fizemos uso de diferentes métodos de pesquisa durante esta dissertação. Desta forma, cada método usado é abordado no capítulo respectivo à questão. Todos os capítulos exigem do leitor conhecimento prévio sobre Android, Maus Cheiros de Código e Métricas de Código. Apresentamos uma breve introdução a esses três assuntos no capítulo 2.

1.2 Contribuições

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

1. A definição do termo **Camada de Apresentação Android**. Com embasamento teórico sobre a origem de interfaces gráficas e na documentação oficial do Android provemos uma definição sobre quais elementos compõem a camada de apresentação Android.
2. Um catálogo validado de maus cheiros da camada de apresentação Android. Os maus cheiros foram definidos com a participação de mais de 50 desenvolvedores em questionários e entrevistas.
3. Um estudo quantitativo sobre a tendência a mudanças e defeitos dos maus cheiros propostos. Realizaremos um experimento com desenvolvedores Android de modo a coletar quantitativamente se classes afetadas pelos maus cheiros possuem uma maior tendência a mudanças e introdução de defeitos.
4. Um estudo sobre a percepção de desenvolvedores sobre os maus cheiros propostos. Realizaremos um experimento com desenvolvedores Android de modo a identificar se classes afetadas pelos maus cheiros 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 aqui são: Qualidade de Código, Maus Cheiros e 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** Camada de Apresentação Android

  Esta pesquisa limita-se em mapear boas e más práticas apenas na camada de apresentação de aplicativos Android. Neste capítulo pretende-se explanar para o leitor o que é considerado como camada de apresentação Android.

- **Capítulo 5** Proposta de Dissertação

  Neste capítulo apresentamos a proposta da dissertação e o cronograma de atividades.

- **Capítulo 6** 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 7** 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 8** Percepção dos Desenvolvedores

  Neste capítulo respondemos a Q3. É apresentado a motivação da questão, explicamos o experimento conduzido e os resultados obtidos.

- **Capítulo 9** 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 claro a definição de 3 itens, são eles: Refatoração, Maus Cheiros de Código e Android.

2.1 Refatoração

Refatoração é definido por Fowler como "uma técnica para reestruturação de um código existente, alterando sua estrutura interna sem alterar seu comportamento externo" [18]. 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 [47]. 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 [46].

Para auxiliar no processo de refatoração, é comum desenvolvedores se utilizarem de ferramentas de detecção de cheiros de código [14], sintomas de design ruim e más práticas de programação [11, 14, 18, 47]. De modo a ser possível automatizar a detecção de cheiros de código, é importante tê-los mapeados e catalogados. Muitos cheiros de código já foram catalogados [18, 44, 48] como veremos em mais detalhes na Seção 2.2.

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 [27]. 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 forma 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.

2.2 Maus Cheiros de Código

Maus cheiros de código são sintomas de design ruim e más práticas de programação [11, 14, 18, 47]. Diferentemente de erros sistêmicos, eles não resultam em comportamentos errôneos [11, 47]. Maus cheiros apontam fraquezas no *design* que podem desacelerar o desenvolvimento e aumentar o risco à defeitos, falhas e mudanças [11, 14, 24, 26]. Do ponto de vista do desenvolvedor, maus cheiros são heurísticas para refatorações que indicam quando refatorar e qual técnica de refatoração usar [11].

O livro de Webster (1995) [48] deve ter sido o primeiro local onde o termo cheiro de código foi usado referindo-se a más práticas. Mais adiante, **mau cheiro de código** foi usado por Martin Fowler e Kent Beck no livro Refactoring (1999) [18] referindo-se a um trecho de código que pode se beneficiar de refatoração. Em uma postagem em 2006 em seu site [19], 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) [18] é mau cheiro de código (*bad smell in code*) e na postagem em seu site [19] é cheiro de código (*code smell*). Essa diferença pode ser justificada pela afirmativa de Fowler, na postagem em seu site, onde 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) [44] 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 a degradação da qualidade em projetos de software [33].

Outro termo comum de se ver juntamente com (mau) cheiro de código é o termo *anti-pattern*. Esse, por sua vez, descreve uma solução comum para um problema que gera consequências negativas [15], também conhecido como má prática. 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 [15, 20], perfeitamente bom no contexto errado [15]. Geralmente, (maus)

cheiros de código são sintomas da presença de *antipattern*. Mario et al. [26] concluem que *antipatterns* impactam de forma negativa aplicações móveis, em particular métricas de qualidade relacionadas ao aumento do risco de falhas.

Muitos autores já trabalharam na definição de diferentes catálogos de cheiros de código. Riel [35] definiu mais de 60 características de um bom código orientado a objetos. Martin Fowler [18] definiu mais de 20 maus cheiros como *God Classes* [18], que são classes que fazem ou sabem muitas coisas e *Feature Envy* [18], 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 [9] e Sonar [10].

Nesta dissertação utilizamos o termo **cheiro de código** para se referir a ambos os termos (mau) cheiro de código e *antipattern* como em trabalhos prévios [26, 29] e nosso foco está em definir e detectar maus cheiros relacionados especificamente à camada de apresentação de projetos Android.

2.3 Android

Android é um sistema operacional de código aberto, baseado no kernel do Linux criado para um amplo conjunto de dispositivos. Na Figura 2.1 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.

2.3.1 Fundamentos do Desenvolvimento Android

Aplicativos Android são escritos na linguagem de programação Java. O Kit para Desenvolvimento de Software (*Software Development Kit*) 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 [37].

Os elementos base para a construção de aplicativos Android são os componentes. Cada 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 [37]. 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

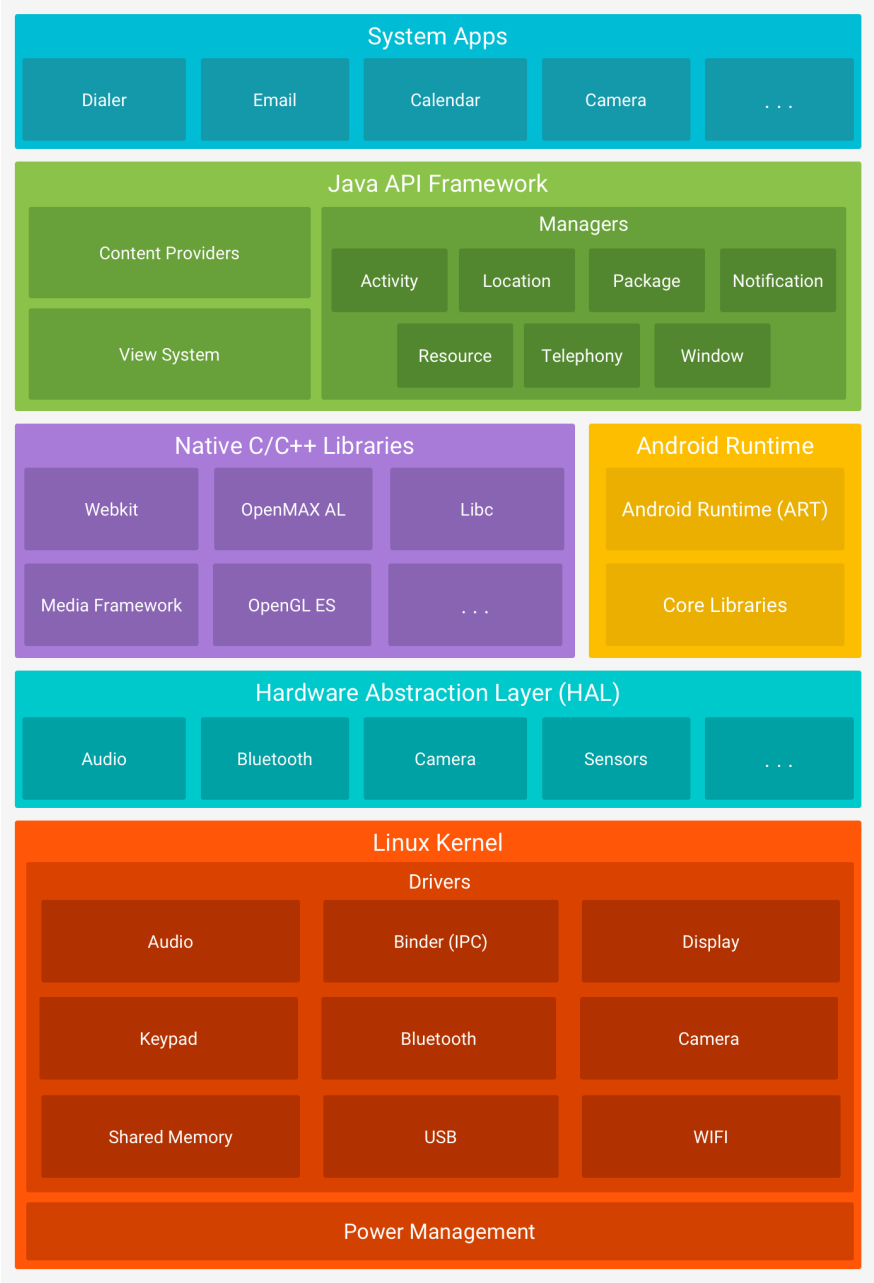

Figura 2.1: *Arquitetura do sistema operacional Android [38].*

destruído [37]. 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` [37].

- **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` [37].

- **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` [37].

- **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` [37].

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 [37]. 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*.

2.3.2 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 [41]. Recursos costumam ser arquivos XML que usam o vocabulário definido pelo Android [37].

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 [37]. Segundo levantamento, em 2015 foram encontrados mais de 24 mil dispositivos diferentes com Android [4].

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 [37].

- **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.

2.3.3 Interfaces de Usuários

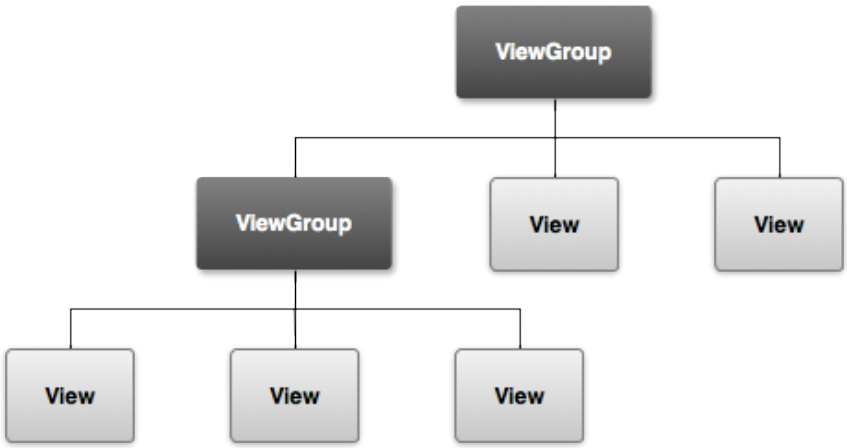

Figura 2.2: *Árvore hierárquica de `Views` e `ViewGroups` do Android [42].*

Arquivos de *layout* são recursos localizados no subdiretório `res/layout` que possuem a extensão `.xml` [41]. Todos os elementos de UI (Interface de Usuário, do inglês UI, *User Inter-*

face) de um aplicativo Android são construídos usando objetos do tipo `View` e `ViewGroup` como mostrado na figura 2.2 [42].

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 container invisível que organiza `Views` filhas. O encadeamento desses objetos formam uma árvore hierárquica que pode ser tão simples ou complexa quanto se precisar [41].

É possível criar um *layout* programaticamente instanciando `Views` e `ViewGroups` no código e construir a árvore hierárquica manualmente, no entanto, a forma mais indicada é por meio de um XML de *layout* [42].

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()` [42].

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:id="@+id/text"
8                 android:layout_width="wrap_content"
9                 android:layout_height="wrap_content"
10                android:text="I am a TextView" />
11
12    <Button android:id="@+id/button"
13              android:layout_width="wrap_content"
14              android:layout_height="wrap_content"
15              android:text="I am a Button" />
16
17 </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 nestes elementos, 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 [39].

2.3.4 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 de forma a 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 [40] 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" [37]. 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 [31]. 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 [32, 45].

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" [49].

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 áreas: (i) estudos recentes realizados sobre Android, (ii) estudos que analisam maus cheiros específicos à alguma tecnologia, plataforma ou framework e (iii) estudos sobre maus cheiros específicos ao Android.

3.1 Pesquisas sobre Android

Em 2017 o Android completará uma década desde seu primeiro lançamento em 2007. Muitas pesquisas, em diversas áreas como segurança, consumo inteligente, comunicação entre aplicativos, dentre outros, têm surgido nos últimos anos.

Adrienne et al. [17] realizou 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. O estudo concluiu que 17% dos usuários prestam atenção as permissões apresentadas durante a instalação. O estudo conclui com recomendações para melhorar o entendimento e atenção dos usuários com relação as permissões solicitadas.

Outro estudo de Erika et al. [16], tem relação com segurança porém com relação a comunicação entre aplicativos Android. Os autores afirmam que, apesar de o sistema de comunicação entre aplicativos do Android ser rico, foram identificados riscos relacionados a segurança. Esses riscos foram mapeados na ferramenta ComDroid [2], que pode ser usada para detecção automática de vulnerabilidades de comunicação. Com esta ferramenta, foram analisados 20 aplicativos dos quais, 12 continham pelo menos 1 vulnerabilidade.

Lance et al. [13] teve seu artigo também sobre comunicação entre aplicativos Android, publicado na MOBILESoft 2016. Foi analisado as diferentes estratégias para se implementar intercomunicação de aplicativos Android desenvolvidos com a IDE Android Studio e a

plataforma web App Inventor. A pesquisa apresenta o método de envio e recebimento de mensagens em ambas as plataformas e suas limitações.

3.2 Maus Cheiros Específicos

Atualmente já existem diversos catálogos de cheiros de código relacionados a projetos orientado a objetos. Por exemplo as práticas catalogadas por Webster [48] ou os cheiros de código catalogados por Fowler [18]. Qualquer que seja o projeto, desde que siga o paradigma orientado a objeto, poderá se beneficiar desses catálogos. No entanto, pesquisas têm demonstrado que domínios diferentes podem apresentar cheiros de código específicos [26, 14].

Aniche et al. [14] em seu trabalho de doutorado, encontrou cheiros de código específicos ao framework Java Spring MVC, foram eles *Promiscuous Controller*, *Smart Controller*, *Meddling Service*, *Smart Repositories*, *Laborious Repository Method* e *Fat Repository*.

Mario et al. [26] salienta a importância do domínio ao se tratar de cheiros de código ao concluir em sua pesquisa que *antipatterns* em aplicações com determinados domínios tem sua qualidade negativamente impactada mais do que outros.

Gharachorlu [21] avaliou a existência de más práticas em arquivos CSS de projetos de código aberto. Essas más práticas foram extraídas de sites e blogs técnicos. De acordo com o autor, as más práticas mais frequentes eram valores fixos, estilos que desfaziam o que estilos anteriores faziam e o uso de seletores id.

Silva et al. [36] executou uma análise em sistemas de código aberto e mostrou que códigos Javascript tendem a usar conceitos de orientação a objetos em 36% dos sistemas analisados.

3.3 Maus Cheiros Android

Diversas pesquisas em torno de maus cheiros de código vem sendo realizadas ao longo dos últimos anos. Já existem inclusive diversos maus cheiros mapeados, porém poucos deles são específicos da plataforma Android [27]. Segundo Hecht [22] estudos sobre maus cheiros de código sobre aplicações Android ainda estão em sua infância. Outro ponto que reafirma esta questão são os trabalhos de Linares-Vásquez [26] e Hecht [22] onde concluem que, em projetos Android, é mais comum maus cheiros específicos do que maus cheiros Orientado a Objetos.

O trabalho de Verloop [47] avalia a presença de maus cheiros definidos por Fowler [18] e Minelli e Lanza [28] em projetos Android. Apesar das relevantes contribuições feitas, a

conclusão sobre a incidência de tais maus cheiros não é plenamente conclusiva, visto que dos 6 maus cheiros analisados (*Large Class*, *Long Method*, *Long Parameter List*, *Type Checking*, *Feature Envy* e *Dead Code*) apenas dois deles, *Long Method* e *Type Checking*, se apresentam com maior destaque (duas vezes mais provável) em projetos Android. Os demais apresentam uma diferença mínima em classes Android quando se comparados a classes não específicas do Android. Por fim, acaba por não ser conclusivo quanto a maior relevância deles em Android ou não.

Desta forma, Verloop [47] conclui com algumas recomendações de refatoração de forma a mitigar a presença do mau cheiro *Long Method*. Estas recomendações são o uso do atualmente já reconhecido padrão *ViewHolder* em classes do tipo `Adapters`. Ele também sugere um *ActivityViewHolder* de forma a extrair código do método `onCreate` e deixá-lo menor. Sugere também o uso do atributo `onClick` em XMLs de LAYOUT e MENU.

Diferentemente de validar a presença de maus cheiros previamente catalogados conforme feito por Verloop [47], esta dissertação objetiva identificar, validar e catalogar, com base na experiência de desenvolvedores, boas e más práticas específicas à **camada de apresentação de projetos Android**.

Outro trabalho muito relevante realizado neste tema é o de Reimann et al. [33] que, baseado na documentação do Android, documenta 30 *quality smells* específicos para Android. No texto *quality smells* são definidos como *"uma determinada estrutura em um modelo, indicando que influencia negativamente a requisitos específicos de qualidade, que podem ser resolvidos por refatorações particulares ao modelo"* (tradução livre). Estes requisitos de qualidade são centrados no usuários (estabilidade, tempo de início, conformidade com usuário, experiência do usuário e acessibilidade), consumo inteligente de recursos (eficiência geral e no uso de energia e memória) e segurança.

Esta dissertação se difere do trabalho Reimann et al. [33] pois pretende-se encontrar maus cheiros em termos de qualidade de código, ou seja, que influenciam na legibilidade e manutenibilidade do código do projeto.

[]

Capítulo 4

Proposta de Dissertação

Conforme apresentado no Capítulo 1, podem existir maus cheiros específicos a um domínio, tecnologia ou plataforma (por exemplo, Android) [14, 26, 47]. Geoffrey [22] 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 [27]. Desta forma, neste capítulo é apresentada a proposta da dissertação e o cronograma de atividades planejadas.

4.1 Atividades

Para obter as ideias iniciais para a derivação dos maus cheiros 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 feito uma análise para derivar os maus cheiros, essa análise será feita com base em estratégias já utilizadas em trabalhos anteriores como o de Aniche et al. [14]. Para reduzir viés sobre os maus cheiros definidos, os mesmos serão validados com mais de um especialista em Android. A derivação dos maus cheiros está relacionada a Q1 definida na seção 1.1 e as atividades planejadas são:

- Bibliografia e Trabalhos Relacionados.
- Survey Boas e Más práticas Android.
- Derivação dos Maus Cheiros.

- Validação Maus Cheiros c/ Especialista.

Evidências na literatura sugerem que maus cheiros de código podem esconder manutenibilidade de código [43, 50, 51] e aumentar a tendência a mudanças e introdução de defeitos [23, 25]. Mario et al. [26] mostra que *antipatterns* impactam negativamente métricas relacionadas a qualidade em projetos móveis, em particular métricas relacionadas a propensão de falhas. Desta forma, pretende-se avaliar o impacto dos maus cheiros propostos na tendência a mudanças e introdução de defeitos no código. Para isso será realizado um experimento presencial com desenvolvedores Android. Este experimento está relacionado ao Q2 definida na Seção 1.1 e a atividade planejada é:

- Experimento Impacto em Mudanças/Defeitos.

Evidências na literatura também sugerem que maus cheiros de código são percebidos por desenvolvedores [30], desta forma pretende-se avaliar se desenvolvedores Android percebem códigos afetados pelos maus cheiros propostos como indicativos de trechos de códigos ruins. Para isso será conduzido outro experimento também com desenvolvedores Android. Esse experimento está relacionado a Q3 definida na Seção 1.1 e a seguinte atividade está planejada:

- Experimento Percepção Desenvolvedores.

4.2 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 Maus Cheiros			•			
Validação Maus Cheiros c/ 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] Comdroid tool. Last accessed at 25/11/2016. 17

[3] 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. 2

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

[5] 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. 2

[6] 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. 2

[7] 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. vii, 1

[8] 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. vii, 2

[9] Pmd (2016). https://pmd.github.io/, 2016. Last accessed at 29/08/2016. 9

[10] Sonarqube (2016). http://www.sonarqube.org/, 2016. Last accessed at 29/08/2016. 9

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

[12] 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. 2

[13] Lance A. Allison and Mohammad Muztaba Fuad. Inter-app communication between android apps developed in app-inventor and android studio. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 17–18, New York, NY, USA, 2016. ACM. 17

[14] MaurÃŋcio Aniche, Marco Gerosa, Bullet Paulo, SÃčo and-INPE, Bullet Sant, and Anna ufba. Architectural roles in code metric assessment and code smell detection. 2016. Regras Arquiteturais na AvaliaÃğÃčo de MÃľtricas de CÃşdigo e DetecÃğÃčo de Maus Cheiros Regras Arquiteturais na AvaliaÃğÃčo de MÃľtricas de CÃşdigo e DetecÃğÃčo de Maus Cheiros. 3, 4, 7, 8, 18, 20

[15] 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

[16] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM. 17

[17] 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. 17

[18] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. 7, 8, 9, 18

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

[20] 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

[21] Golnaz Gharachorlu. *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model*. PhD thesis, The University of British Columbia, 2014. 18

[22] Geoffrey Hecht. An approach to detect android antipatterns. page 766âĂŞ768, 2015. 2, 3, 18, 20

[23] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 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, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society. 21

[24] Foutse Khomh, Massimiliano Penta, Yann-Gael GuÃľhÃľneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243âĂŞ275, 2012. 8

[25] 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):243–275, June 2012. 21

[26] Mario Linares-VÃąsquez, Sam Klock, Collin Mcmillan, Aminata SabanÃľ, Denys Poshyvanyk, and Yann-Gael GuÃľhÃľneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 8, 9, 18, 20, 21

[27] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding code smells in android applications. 4, 7, 18, 20

[28] 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. 18

[29] Naouel Moha, Yann-GaÃńl GuÃľhÃľneuc, Anne-FranÃğoise Meur, and Laurence Duchien. Fundamental approaches to software engineering. 2008. 9

[30] 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. 21

[31] Danny Preussler. Writing better adapters. https://medium.com/\spacefactor\ @mdpreussler/writing-better-adapters-1b09758407d2#.tbvvw3krr, 2016. Last accessed at 27/10/2016. 15

[32] Eric Steven Raymond. *The Art of Unix Usability*. 2004. Last accessed at 26/10/2016. 16

[33] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2013. 3, 8, 19

[34] A.J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley Publishing Company, 1996. 3

[35] Arthur Riel J. *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 1996. 9

[36] Leonardo Silva, Miguel Ramos, Marco Valente, Alexandre Bergel, and Nicolas Anquetil. Does javascript software embrace classes? pages 73–82, 2015. 18

[37] Android Developer Site. Andriod fundamentals. https://developer.android.com/guide/components/fundamentals.html. Last accessed at 04/09/2016. 9, 10, 11, 12, 15

[38] Android Developer Site. Plataform architecture. https://developer.android.com/guide/platform/index.html. Last accessed at 04/09/2016. vii, 10

[39] Android Developer Site. Ui events. https://developer.android.com/guide/topics/ui/ui-events.html. Last accessed at 25/11/2016. 15

[40] Android Developer Site. DocumentaÃğÃčo site android developer. https://developer.android.com, 2016. Last accessed at 27/10/2016. 15

[41] Android Developer Site. Resource type. https://developer.android.com/guide/topics/resources/available-resources.html, 2016. Last accessed at 12/09/2016. 12, 13, 14

[42] Android Developer Site. Ui overview. https://developer.android.com/guide/topics/ui/overview.html, 2016. Last accessed at 23/10/2016. vii, 13, 14

[43] Dag Sjoberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *Ieee T Software Eng*, 39(8):1144–1156, 2013. 21

[44] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014. 7, 8

[45] TecMundo. A histÃşria da interface grÃąfica. http://www.tecmundo.com.br/historia/9528-a-historia-da-interface-grafica.htm. Last accessed at 26/10/2016. 16

[46] 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

[47] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 2, 3, 7, 8, 18, 19, 20

[48] Bruce Webster F. *Pitfalls of Object-Oriented Development*. M & T Books, 1995. 7, 8, 18

[49] Wikipedia. Interface do utilizador. https://pt.wikipedia.org/wiki/Interface_do_utilizador. Last accessed at 26/10/2016. 16

[50] 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. 21

[51] 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. 21

[52] Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. How does context affect the distribution of software maintainability metrics? *In IEEE International Conference on Software Maintenance, paÌąginas 350âĂŞ359. IEEE*, 2013. Encontrado na dissertaÃğÃčo do Aniche. 3