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

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 tem se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. Este 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. Desta forma, desenvolvedores de software constantemente 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 tem demonstrado que diferentes plataformas, linguagens e frameworks podem apresentar métricas de qualidade de código específicas. Projetos Android possuem características não experimentadas até o momento 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 pretendemos 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 OO 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

Software Development Kit Kit para Desenvolvimento de Software

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

Lista de Símbolos

Σ Sistema de transição de estados

Lista de Figuras

Capítulo 1

Introdução

Em 2017 o Android completará uma década desde seu primeiro lançamento em 2007. Atualmente há disponível mais de 2 milhões de aplicativos na Google Play Store, loja oficial de aplicativos Android [6]. Mais de 83,5% dos dispositivos móveis no mundo usam o sistema operacional Android, e esse percentual vem crescendo ano após ano [5, 7]. Atualmente é possível encontrá-lo também em outros dispositivos como *smart TVs*, *smartwatchs*, carros, dentre outros [2, 4].

Aplicativos Android tem se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. Este 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 e evolução [11]. Apesar de ser possível analisar projetos Android através de maus cheiros "tradicionais" (por exemplo *God Classes* e *Long Methods*), pesquisas tem demonstrado que diferentes plataformas, linguagens e frameworks podem apresentar métricas de qualidade de código específicas [8, 29]. Projetos Android possuem características ainda não experimentadas em projetos orientados a objetos, principalmente com relação à camada de apresentação, onde apresenta suas maiores diferenças.

Conforme relatado por Hecht [11] com relação a projetos Android, "*antipatterns* específicos a 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 estes 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 [27], 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 simultâneamente. Isto posto, é razoável considerar que o mau cheiro *God Class* [18] é aplicável neste caso, no entanto, conforme bem pontuado por Aniche et al. [8] "*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 pelo Reimann et al. [17] "*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, apenas 5 artigos foram sobre maus cheiros Android, dentro de um total de 52 artigos sobre o assunto [13]. 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 ter 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 ímpacto destas más práticas na manutenabilidade do código de projetos Android.

1.1 Questões de Pesquisa

Esta dissertação tem por objetivo investigar maus cheiros específicos a camada de apresentação em 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 passamos por aplicação de questionário e 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 [8]. Desta forma, esta questão pretende, através 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 a questão. Todos os capítulos exigem do leitor conhecimento prévio sobre Android, Maus Cheiros de Código e Métricas de Código.

Apresentamos uma breve introdução à estes três assuntos no capítulo 2.

1.2 Contribuições

As principais contribuições dessa 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 compoõem 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 13 desenvolvedores Android de forma 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 forma a identificar se classes afetadas pelos maus cheiros são percebidas como problemáticas por desenvolvedores Android.

1.3 Organização da Tese

Esta dissertação está organizada da seguinte forma:

- **Capítulo 1** Introdução

  Neste capítulo é introduzido o contexto atual do desenvolvimento de aplicativos Android. Apresenta-se quais são as motivações e o problema a ser resolvido. É dado também uma breve introdução sobre como pretende-se resolvê-lo.

- **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 o 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 por camada de apresentação Android.

- **Capítulo 5** Boas e Más Práticas na Camada de Apresentação

  Neste capítulo respondemos a Q1. É apresentado 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 apresentado 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 7** 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: Qualidade de Código, Maus Cheiros de Código e Android.

2.1 Qualidade de Código

2.2 Maus Cheiros

Um mau cheiro de código (*code smell*) é uma indicação superficial que usualmente corresponde a um problema mais profundo em um software. Por si só, um *code smell* não é algo ruim, ocorre que frequentemente ele indica um problema mas não necessáriamente é o problema em si [10]. O termo *code smell* foi cunhado pela primeira vez por Kent Beck enquanto ajudava Martin Fowler com o seu livro Refactoring [9, 10].

Maus cheiros são padrões de código que estão associados com um design ruim e más práticas de programação. Diferentemente de erros de código eles não resultam em comportamentos erroneos. Maus cheiros apontam para áreas na aplicação que podem se beneficiar de refatorações. [27]. Refatoração é definido por "uma técnica para reestruturação de um código existente, alterando sua estrutura interna sem alterar seu comportamento externo" [9].

Escolher não resolver um mau cheiro pela refatoração não resultará na aplicação falhar mas irá aumentar a dificuldade de mantê-la. Logo, a refatoração ajuda a melhorar a manutenabilidade de uma aplicação [27]. Uma vez que os custos com manutenção são a maior parte dos custos envolvidos no ciclo de desenvolvimento de software [26], aumentar a manutenabilidade através de refatoração irá reduzir os custos de um software no longo prazo.

2.3 Android

Arquitetura da Plataforma

Android é um sistema operacional de código aberto, baseado no kernel do Linux criado para um amplo conjunto de dispositivos. Para prover acesso aos recursos específicos dos dispositivos como câmera ou *bluetooth*, o Android possui uma camada de abstração de *hardware* (HAL do inglês *Hardware Abstraction Layer*) exposto aos desenvolvedores através de um arcabouço de interfaces de programação de aplicativos (APIs do inglês *Applications Programming Interface*) Java. Estes e outros elementos explicados a seguir podem ser visualizados na figura 2.1 [20].

Cada aplicativo é executado em um novo processo de sistema que contém sua própria instância do ambiente de execução Android. A partir da versão 5 (API nível 21), o ambiente de execução padrão é o Android Runtime (ART), antes desta versão era a Dalvik. ART foi escrita para executar multiplas instâncias de máquina virtual em dispositivos com pouca memória. Suas funcionalidades incluem: duas forma de compilação, a frente do tempo (AOT do inglês *Ahead-of-time*) e apenas no momento (JIT do inglês *Just-in-time*), o coletor de lixo, ferramentas de depuração e um relatório de diagnósticos de erros e exceções.

Muitos dos componentes e serviços básicos do Android, como ART e HAL, foram criados a partir de código nativo que depende de bibliotecas nativas escritas em C e C++. A plataforma Android provê arcabouços de APIs Java para expôr as funcionalidade de algumas destas bibliotecas nativas para os aplicativos. Por exemplo, OpenGL ES pode ser acessado através do arcabouço Android Java OpenGL API, de forma a adicionar suporte ao desenho e manipulação de gráficos 2D e 3D no aplicativo.

Todo as funcionalidades da plataforma Android estão disponíveis para os aplicativos através de APIs Java. Estas APIs compõem os elementos básicos para a construção de aplicativos Android. Dentre eles, os mais relevantes para esta dissertação são:

- Um rico e extensível **Sistema de Visualização** para a contrução de interfaces com o usuário, também chamadas de arquivos de *layout*, do aplicativo. Incluindo listas, grades ou tabelas, caixas de textos, botões, dentre outros.
- Um **Gerenciador de Recursos**, provendo acesso aos recursos "não-java" como textos, elementos gráficos, arquivos de *layout*.
- Um **Gerenciador de Activity** que gerencia o ciclo de vida dos aplicativos e provê

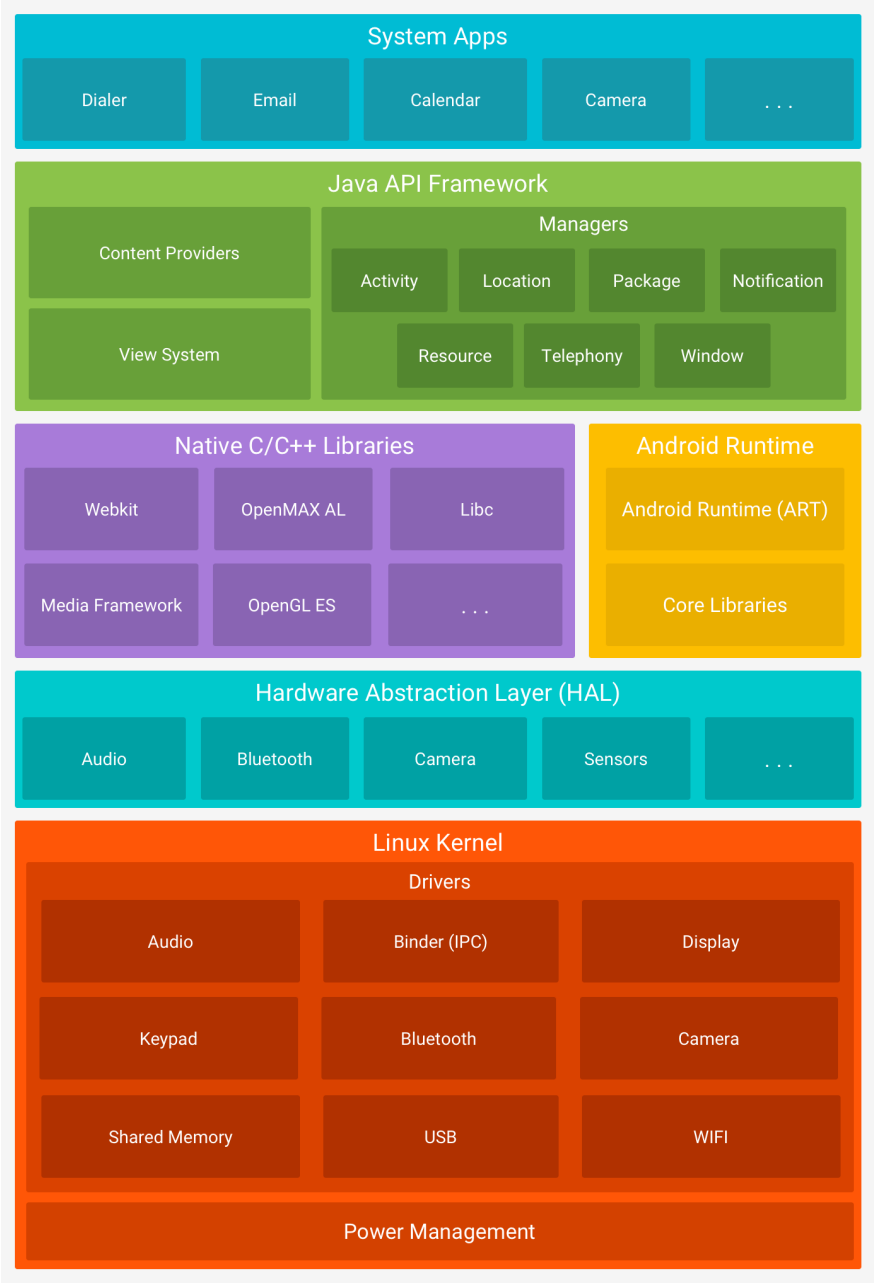

Figura 2.1: *Arquitetura do sistema operacional Android.*

uma navegação comum.

O Android já vem com um conjunto de aplicativos básicos como por exemplo, para envio e recebimento de SMS, calendário, navegador, contatos e outros. Estes aplicativos vindos com a plataforma não possuem nenhum diferencial com relação aos aplicativos de terceiros. Todo aplicativo tem acesso ao mesmo arcabouço de APIs do Android, seja ele aplicativo da

plataforma ou de terceiro. Desta forma, um aplicativo de terceiro pode se tornar o aplicativo padrão para navegar na internet, receber e enviar SMS e assim por diante.

Aplicativos da plataforma provem capacidades básicas que aplicativos de terceiros podem reutilizar. Por exemplo, se um aplicativo de terceiro quer possibilitar o envio de SMS, o mesmo pode redirecionar esta funcionalidade de forma a abrir o aplicativo de SMS já existente, ao invés de implementar por si só.

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 Android Package (APK). Um APK, arquivo com extensão `.apk`, é usado por dispositivos para a instalação de um aplicativo [19].

Componentes Android são os elementos base para a construção de aplicativos Android. Cada componente é um diferente ponto através do qual o sistema pode acionar o aplicativo. Nem todos os componente são pontos de entrada para o usuário e alguns são dependentes entre si, mas cada qual existe de forma autônoma e desempenha um papel específico.

Existem quatro tipos diferentes de componentes Android. Cada tipo serve um propósito distinto e tem diferentes ciclos de vida, que definem como o componente é criado e destruído. Os quatro componentes são:

- **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 forma 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 destas *activities* (se o aplicativo de emails permitir). Por exemplo, a *activity* de redigir email no aplicativo de emails, poderia solicitar o aplicativo câmera, de forma a permitir o compartilhamento de alguma foto. Uma *activity* é implementada como uma subclasse de `Activity`.

- **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*. Outros componente, como uma *activity*, podem iniciar um *service* e deixá-lo executar em plano de fundo. É possível interagir com um *service* durante sua execução. Um *service* é implementado como uma subclasse de `Service`.

- **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. Através 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. Desta forma, qualquer aplicativo, com as devidas permissões, pode consultar parte do *content provider* (como `ContactsContract.Data`) para ler e escrever informações sobre um contato específico. Um *content provider* é implementado como uma subclasse de `ContentProvider`.

- **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. Aplicativos de terceiros também podem enviar mensagens, por exemplo, informando que alguma operação foi concluída. No entanto, *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`.

Antes de a plataforma Android poder iniciar qualquer um dos componente supramencionados, a plataforma precisa saber que eles existem. Isso é feito através da leitura do arquivo `AndroidManifest.xml` do aplicativo (arquivo de manifesto). Este arquivo deve estar no diretório raiz do projeto do aplicativo e deve conter a declaração de todos os seus componentes.

O arquivo de manifesto é um arquivo XML e pode conter muitas outras informações além das declarações dos componentes do aplicativo, por exemplo:

- Identificar qualquer permissão de usuário requerida pelo aplicativo, como acesso a internet, acesso a informações de contatos do usuário e assim por diante.
- Declarar o nível mínimo do Android requerido para o aplicativo, baseado em quais APIs são usadas pelo aplicativo.
- Declarar quais funcionalidades de sistema ou *hardware* são usadas ou requeridas pelo aplicativo, por exemplo câmera, *bluetooth* e assim por diante.
- Declarar outras APIs que são necessárias para uso do aplicativo (além do arcabouço de APIs do Android), como a biblioteca do Google Maps.

Os elementos usados no arquivo de manifesto são definidos pelo vocabulário XML do Android. Por exemplo, uma *activity* pode ser declarada conforme o *listing* 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>
```

Listing 2.1: *Arquivo* `AndroidManifest.xml`

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

Para declarar cada um dos quatro tipos de componentes, deve-se usar os elementos a seguir:

- `<activity>` elemento para *activities*.

- `<service>` elemento para *services*.
- `<receiver>` elemento para *broadcast receivers*.
- `<provider>` elemento para *content providers*.

Recursos do Aplicativo

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 [19]. Também é possível definir animações, menus, estilos, cores e arquivos de *layout* das *activities*. Recursos costumam ser arquivos XML que usam o vocabulário definido pelo Android.

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. Este aspecto se torna mais importante conforme mais dispositivos são lançados com configurações diferentes. Segundo levantamento, em 2015 foram encontrados mais de 24 mil dispositivos diferentes com Android [3].

De forma a prover compatibilidade com diferentes configurações, deve-se organizar os recursos dentro do diretório `res` do projeto, usando sub-diretórios que agrupam os recursos por tipo e configuração. Para qualquer tipo de recurso, pode-se especificar uma opção padrão e outras alternativas.

- **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 com os recursos disponíveis no aplicativo. 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 frânces) e a configuração de idioma do dispositivo, o Android aplica o conjunto de *strings* mais apropriado.

A seguir são listados os tipos de recursos que podem ser utilizados no Android [23]. Para cada tipo de recurso existe um conjunto de qualificadores que podem ser usados para prover recursos alternativos:

- **Recursos de animações** Definem animações pré-determinadas. Ficam nos diretórios `res/anim` ou `res/animator`.
- **Recursos de lista de cores de estado** Definem recursos de cores que alteram baseado no estado da *View*. Ficam no diretório `res/color`.
- **Recursos de desenhos** Definem recursos gráficos como *bitmap* ou XML. Ficam no diretório `res/drawable`.
- **Recursos de *layouts*** Definem a parte visual da interface com o usuário. Ficam no diretório `res/layout`.
- **Recursos de menus** Definem os conteúdos dos menus da aplicação. Ficam no diretório `res/menu`.
- **Recursos de textos** Definem textos, conjunto de textos e plurais. Ficam no diretório `res/values`.
- **Recursos de estilos** Definem os estilos e e formatos para os elementos da interface com usuário. Ficam no diretório `res/values`.
- **Outros recursos** Ainda existem outros recursos como inteiros, *booleanos*, dimensões, dentre outros. Ficam no diretório `res/values`.

Interfaces de Usuários

Arquivos de layout são recursos localizados na pasta `res/layout` que possuem a extensão `.xml`.

Todos os elementos de UI (Interface de Usuário, do inglês UI, *User Interface*) de um aplicativo Android são construídos usando objetos do tipo `View` ou `ViewGroup`. Uma `View`

é um objeto que desenha algo na tela do qual o usuário pode interagir. Um `ViewGroup` é um objeto que agrupa outras `Views` e `ViewGroups` de forma a desenhar o layout da interface [24].

A UI para cada componente do aplicativo é definida usando uma hierarquia de objetos `View` e `ViewGroup`, como mostrado na figura 2.2. Cada `ViewGroup` é um container invisível que organiza `Views` filhas, enquanto as `Views` filhas são caixas de texto, botões e outros componentes visuais que compoem a UI. Esta árvore hierárquica pode ser tão simples ou complexa quanto se precisar, mas quanto mais simples melhor o desempenho.

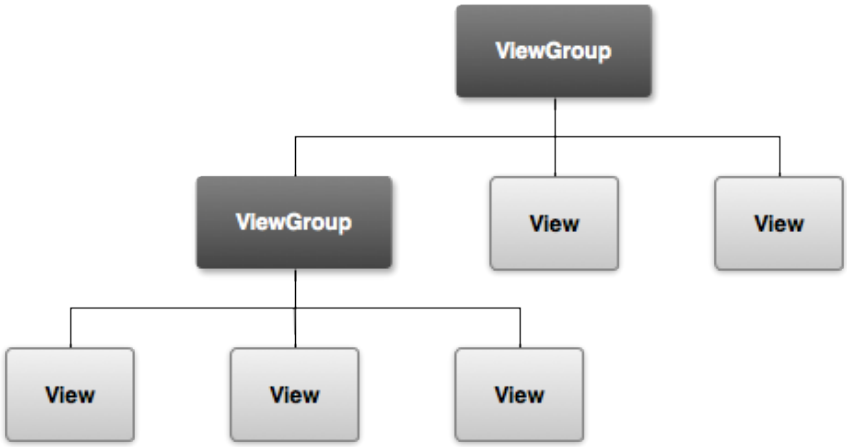

Figura 2.2: *Árvore hierárquica de `Views` e `ViewGroups` do Android.*

É possível criar um layout programaticamente, instanciando `Views` e `ViewGroups` no código e construir a árvore hierárquica manualmente, no entanto, a forma mais simples e efetiva de definir um layout é través de um XML de layout. O XML de layout oferece uma estrutura legível aos olhos humanos, similar ao HTML, podendo ser utilizados elementos aninhados.

O vocabulário XML para declarar elementos de UI segue 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. De fato, a correspondência frequentemente é tão direta que é possível adivinhar qual atributo XML correspodne a qual método de classe, ou adivinhar qual a classe correspondente para determinado elemento. No entanto, algumas classes possuem pequenas diferenças como por exemplo, o elemento `<EditText>` tem o atributo `text` que correponde ao método `EditText.setText()`.

Um layout vertical simples com uma caixa de texto e um botão se parece com o código no listing 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>
```

Listing 2.2: *Arquivo exemplo de layout.*

Quando um recurso de layout é carregado pelo aplicativo, o Android inicializa um objeto para cada elemento do layout, desta forma é possível recuperá-lo programaticamente para definir comportamentos, modificar o layout ou mesmo recuperar o estado.

O Android provê uma série de elementos de UI comuns pré-prontos como: caixa de texto, botão, lista suspensa, dentre muitos outros. Desta forma, o desenvolvedor não precisa implementar do zero estes elementos básicos através de `Views` e `ViewGroups` para escrever uma interface de usuário.

Cada subclasse de `ViewGroup` provê uma forma única de exibir o conteúdo dentro dele. Por exemplo, o `LinearLayout` organiza seu conteúdo de forma linear horizontalmente, um ao lado do outro, ou verticalmente, um abaixo do outro. O `RelativeLayout` permite especificar a posição de uma `View` relativa ao posicionamento de alguma outra [22].

Quando o conteúdo é dinâmico ou não pré-determinado, como por exemplo uma lista de dados, pode-se usar um elemento que extende de `AdapterView` para popular o layout em momento 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`.

Eventos de Interface

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so.

Within the various View classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the onTouchEvent() method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every View object in order to handle such an event would not be practical. This is why the View class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called event listeners, are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the Button class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class event handlers.

Capítulo 3

Trabalhos Relacionados

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 [13]. Segundo Hecht [11] 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 [12] e Hecht [11] onde concluem que, em projetos Android, é mais comum maus cheiros específicos do que maus cheiros orientados a objetos.

O trabalho de Verloop [27] avalia a presença de maus cheiros definidos por Fowler [9] e Minelli e Lanza [14] 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 [27] 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 [27], 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. [17] 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. [17] pois pretende-se encontrar maus cheiros em termos de qualidade de código, ou seja, que influenciam na legibilidade e manutenabilidade do código do projeto.

Capítulo 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 econtrada uma definição formal sobre camada de apresentação Android. Encontramos porém, pontos na documentação oficial do Android [21] 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" [19]. 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 [15]. 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 supensos, 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 [16, 25].

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" [28].

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 5

Boas e Más Práticas na Camada de Apresentação

Neste capítulo é abordado detalhes dos métodos de pesquisa utilizados para a descoberta dos maus cheiros na camada de apresentação Android.

5.1 Introdução

God Class, *Large Class*, *Long Method* são exemplos de maus cheiros de código amplamente reconhecidos por desenvolvedores. De fato, é possível obter métricas de qualidade de código de projetos Android utilizando estes maus cheiros já catalogados. No entanto, pesquisas tem demonstrado que existem maus cheiros de código específicos a tecnologias, frameworks e plataformas [8], desta forma sugerimos que há maus cheiros específicos à camada de apresentação de aplicativos Android. Para iniciar nossas investigações sobre este tema, fizemos a seguinte pergunta:

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

Maus cheiros de código são padrões de código que estão associados com um design ruim e más práticas de programação [27]. Por si só, um mau cheiro não é algo ruim, ocorre que frequentemente ele indica um problema mas não necessáriamente é o problema em si [10]. Sendo assim, esta questão visa identificar quais práticas desenvolvedores Android reconhecem como sendo más práticas, e possívelmente maus cheiros, e quais práticas os desenvolvedores reconhecem como boas práticas, e possívelmente formas de refatorar o que foi considerado um mau cheiro.

Neste capítulo nós apresentamos um catálogo de maus cheiros na camada de apresentação de projetos Android. Para derivar este 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 código aberto a partir das respostas e postagens obtidas.

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

5.2 Método

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 forma a obtermos alguma análise demográfica, as primeiras questões questionavam 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 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. Estas discussões serão focadas nos 8 elementos da camada de apresentação questionados no passo anterior.

Passo 3: Pesquisa Bibliográfica 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 apresentacão pesquisados nos passos anteriores.

De forma 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 proeficientes.

5.3 Q1: Catálogo Resultante de Maus Cheiros da Camada de Apresentação Android

Capítulo 6

Impacto na Tendência a Mudanças e Defeitos

6.1 Introdução

Evidencias na literatura sugerem que maus cheiros de código podem esconder manutenabilidade de código e aumentar a tendência a mudanças e introdução de defeitos. Neste capítulo pretendemos avaliar o impacto dos maus cheiros propostos na tendência a mudanças e introdução de defeitos no código. Para isso conduziremos um experimento presencial com desenvolvedores Android.

6.2 Método

Pretende-se conduzir um experimento apresentando aos desenvolvedores um projeto Android onde alguns deles receberão este projeto com os maus cheiros 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 maus cheiros propostos tem na tendência a alteração de código e a introdução de defeitos.

6.3 Q2: Qual o impacto dos maus cheiros propostos na tendência a mudanças e defeitos?

Capítulo 7

Percepção dos Desenvolvedores

7.1 Introdução

Após a definição de um catálogo de maus cheiros, 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 maus cheios propostos, para cada código, será solicitado que ele indique o código como "com cheiro" ou "sem cheiro".

7.2 Método

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

Capítulo 8

Conclusão

A fazer.

8.1 Trabalhos futuros

A fazer.

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. 2

[2] 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

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

[4] 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

[5] 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

[6] 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. 1

[7] 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

[8] MaurÃŋcio Aniche, Marco Gerosa, SÃčo Paulo, Bullet 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. 1, 2, 3, 21

[9] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999. 6, 17

[10] Martin Fowler. Code smell. http://martinfowler.com/bliki/CodeSmell.html, February 2006. 6, 21

[11] Geoffrey Hecht. An approach to detect android antipatterns. page 766âĂŞ768, 2015. 1, 17

[12] Mario Linares-VÃąsquez, Sam Klock, Collin Mcmillan, Aminata SabanÃľ, Denys Poshyvanyk, and Yann-GaÃńl GuÃľhÃľneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 17

[13] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding code smells in android applications. 2, 17

[14] 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. 17

[15] Danny Preussler. Writing better adapters. https://medium.com/\spacefactor\@mdpreussler/writing-better-adapters-1b09758407d2#.tbvvw3krr, 2016. Last accessed at 27/10/2016. 19

[16] Eric Steven Raymond. *The Art of Unix Usability.* 2004. Last accessed at 26/10/2016. 19

[17] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2013. 2, 18

[18] A.J. Riel. *Object-oriented Design Heuristics.* Addison-Wesley Publishing Company, 1996. 2

[19] Android Developer Site. Andriod fundamentals. https://developer.android.com/guide/components/fundamentals.html. Last accessed at 04/09/2016. 9, 12, 19

[20] Android Developer Site. Plataform architecture. https://developer.android.com/guide/platform/index.html. Last accessed at 04/09/2016. 7

[21] Android Developer Site. DocumentaÃğÃčo site android developer. https://developer.android.com, 2016. Last accessed at 27/10/2016. 19

[22] Android Developer Site. Layouts. https://developer.android.com/guide/topics/ui/declaring-layout.html, 2016. Last accessed at 23/10/2016. 15

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

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

[25] 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. 19

[26] 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. 6

[27] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 2, 6, 17, 21

[28] Wikipedia. Interface do utilizador. https://pt.wikipedia.org/wiki/Interface_do_utilizador. Last accessed at 26/10/2016. 20

[29] 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. EncontradoÃğÃčo do Aniche. 1