

Maus cheiros de código em aplicativos Android: Um estudo sobre a percepção dos desenvolvedores

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: Ciência da Computação

Orientador: Prof. Dr. Marco Aurélio Gerosa

Coorientador: Prof. Dr. Maurício Aniche

São Paulo, 13 Dezembro de 2017

Maus cheiros de código em aplicativos Android: Um estudo sobre a percepção dos desenvolvedores

Esta é a versão original da dissertação elaborada pelo
candidato (Suelen Goularte Carvalho), tal como
submetida à Comissão Julgadora.

Agradecimientos

Under construction.

Resumo

SOBRENOME, A. B. C. **Título do trabalho em inglês.** 2010. 120 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2010.

Existem diversas práticas, ferramentas e padrões que auxiliam desenvolvedores a produzir código com qualidade. Dentre elas podemos citar catálogos de maus cheiros, que indicam possíveis problemas no código. Esses catálogos possibilitam a implementação de ferramentas de detecção automática de trechos de código problemáticos ou mesmo a inspeção manual. Apesar de já existirem diversos cheiros de código catalogados, pesquisas sugerem que tecnologias diferentes apresentaram cheiros de código específicos, e uma tecnologia que tem chamado a atenção de muitos pesquisadores é o Android. Neste artigo, nós investigamos a existência de cheiros de código em projetos Android. Como ponto de partida, por meio de um questionário online com 45 desenvolvedores, catalogamos 23 más práticas Android. Por meio de um experimento online, validamos a percepção de 20 desenvolvedores sobre quatro dessas más práticas, onde duas se confirmaram estatisticamente. Esperamos que nosso catálogo e metodologia de pesquisa, bem como sugestões de como mitigar as ameaças à validade possam colaborar com outros pesquisadores.

Keywords: keyword1, keyword2, keyword3.

Abstract

SOBRENOME, A. B. C. **Título do trabalho em inglês.** 2010. 120 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2010.

Existem diversas práticas, ferramentas e padrões que auxiliam desenvolvedores a produzir código com qualidade. Dentre elas podemos citar catálogos de maus cheiros, que indicam possíveis problemas no código. Esses catálogos possibilitam a implementação de ferramentas de detecção automática de trechos de código problemáticos ou mesmo a inspeção manual. Apesar de já existirem diversos cheiros de código catalogados, pesquisas sugerem que tecnologias diferentes apresentaram cheiros de código específicos, e uma tecnologia que tem chamado a atenção de muitos pesquisadores é o Android. Neste artigo, nós investigamos a existência de cheiros de código em projetos Android. Como ponto de partida, por meio de um questionário online com 45 desenvolvedores, catalogamos 23 más práticas Android. Por meio de um experimento online, validamos a percepção de 20 desenvolvedores sobre quatro dessas más práticas, onde duas se confirmaram estatisticamente. Esperamos que nosso catálogo e metodologia de pesquisa, bem como sugestões de como mitigar as ameaças à validade possam colaborar com outros pesquisadores.

Keywords: keyword1, keyword2, keyword3.

Sumário

Lista de Abreviaturas	ix
Lista de Figuras	xi
Lista de Tabelas	xii
1 Introdução	1
1.1 Considerações Preliminares	1
1.2 Objetivos	2
1.3 Contribuições	3
1.4 Organização do Trabalho	3
2 Fundamentação Conceitual	4
2.1 ANDROID	4
2.2 QUALIDADE DE SOFTWARE	13
2.2.1 Manutenibilidade	15
2.3 BOAS PRÁTICAS DE SOFTWARE	15
2.3.1 Padrões de Projeto	15
2.3.2 Anti-Padrões	16
2.4 MAUS CHEIROS DE CÓDIGO	16
2.4.1 Formato dos Maus Cheiros	17
3 Trabalhos Relacionados	19
4 Metodologia	21
4.1 QP1: Boas e Más Práticas Android	21

4.2	QP2: Percepção dos Desenvolvedores	25
5	Maus Cheiros de Interface Android	29
5.1	Maus Cheiros Em Código Java	29
5.1.1	CLASSE DE UI INTELIGENTE (SML-J1) ^{20*}	29
5.1.2	CLASSES DE UI ACOPLADAS (SML-J2) ^{6*}	30
5.1.3	COMPORTAMENTO SUSPEITO (SML-J3) ^{6*}	30
5.1.4	ENTENDA O CICLO DE VIDA (SML-J4) ^{5*}	31
5.1.5	ADAPTER CONSUMISTA (SML-J5) ^{5*}	31
5.1.6	USO EXCESSIVO DE FRAGMENT (SML-J6) ^{3*}	31
5.1.7	NÃO USO DE FRAGMENT (SML-J7) ^{3*}	32
5.1.8	CLASSES DE UI FAZENDO IO (SML-J8) ^{3*}	32
5.1.9	ACTIVITY INEXISTENTE (SML-J9) ^{2*}	32
5.1.10	ARQUITETURA NÃO IDENTIFICADA (SML-J10) ^{2*}	33
5.1.11	ADAPTER COMPLEXO (SML-J11) ^{2*}	33
5.1.12	FRAGMENT ANINHADO (SML-J12) ^{1*}	33
5.1.13	CONTROLE MANUAL DA PILHA DE ACTIVITIES (SML-J13) ^{1*}	34
5.1.14	ESTRUTURA DE PACOTES (SML-J14) ^{1*}	34
5.2	Maus Cheiros Em Recursos	34
5.2.1	RECURSO MÁGICO (SML-R1) ^{8*}	34
5.2.2	NOME DE RECURSO DESPADRONIZADO (SML-R2) ^{8*}	34
5.2.3	LAYOUT PROFUNDAMENTE ANINHADO (SML-R3) ^{7*}	35
5.2.4	IMAGEM DISPENSÁVEL (SML-R4) ^{6*}	35
5.2.5	LAYOUT LONGO OU REPETIDO (SML-R5) ^{5*}	35
5.2.6	IMAGEM FALTANTE (SML-R6) ^{4*}	36
5.2.7	LONGO RECURSO DE ESTILO (SML-R7) ^{3*}	36
5.2.8	RECURSO DE STRING BAGUNÇADO (SML-R8) ^{3*}	36
5.2.9	ATRIBUTOS DE ESTILO REPETIDOS (SML-R9) ^{3*}	37
5.2.10	REÚSO INADEQUADO DE STRING (SML-R10) ^{2*}	37
5.2.11	LISTENER ESCONDIDO (SML-R11) ^{2*}	37

6	Percepção dos Desenvolvedores	38
6.1	Percepção dos Desenvolvedores	38
6.1.1	Más práticas que afetam classes Java	38
6.1.2	Más práticas que afetam recursos	39
7	Discussão	41
7.1	Propostas de soluções	41
8	Ameaças à Validade	42
9	Conclusão	43
A	Questionário sobre Boas e Más Práticas	45
B	Exemplos de Respostas que Embasaram os Mau Cheiros	48
	Referências Bibliográficas	52

Acrônimos

Software Development Kit Kit para Desenvolvimento de Software

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

Lista de Abreviaturas

Lista de Figuras

- 2.1 Arquitetura do sistema operacional Android. 5
- 2.2 Árvore hierárquica de Views e ViewGroups do Android. 11
- 2.3 Qualidade de Software (ISO/EIC 9126-1 e ISO/IEC 25010 e [44]). *todo: corrigir caption* 13
- 4.1 Experiência com desenvolvimento de software e android dos desenvolvedores. 24
- 6.1 Gráficos violino das más práticas LCUI, LPA, RM e NRD. 38

Lista de Tabelas

4.1 Total de respostas obtidas por cada questão sobre boas e más práticas no
front-end Android. 24

Capítulo 1

Introdução

Qualidade de código já se provou com uma área de extrema importância no desenvolvimento de software e o rápido surgimento de novas tecnologias faz com que esta área continue muito relevante *todo: refs aqui*. Android é uma dessas novas tecnologias que surgiu na última década e tem apresentado um crescimento extremamente acentuado *todo: refs aqui*. Porém, estudos sobre qualidade de código Android ainda são poucos *todo: refs aqui*.

Uma das formas que desenvolvedores tem para aumentar a qualidade de seus códigos é por meio de refatoração. Refatoração é, segundo Kent Beck *todo: ref Refactoring*, uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável. Entretanto, uma parte importante para ser possível realizar refatorações é encontrar trechos códigos que precisem ser refatorados. Para isso é comum a identificação de maus cheiros no código.

O termo mau cheiro foi definido pela primeira vez no livro Refactoring de Martin Fowler com a participação de Kent Beck. Um mau cheiro de código descreve um contexto no código que pode apontar, ou não, para um problema mais profundo, mas geralmente não é o problema em si e não significa que existe alguma falha sistêmica. Trechos de código “maus cheirosos” são fortes candidatos a passarem por refatorações.

Nesta pesquisa investigamos a existência de maus cheiros de código específicos a plataforma Android.

1.1 Considerações Preliminares

Qualidade de código já se provou com uma área de extrema importância no desenvolvimento de software. Ao longo dos últimos anos, diversos estudos e livros foram publicados sobre esse assunto. Dentre as formas usadas para documentar o conhecimento gerado a cerca de qualidade de código estão a definição de padrões, anti-padrões e maus cheiros.

Muitos dos materiais documentados usam como base a linguagem de programação Java. Porém, ao longo dos últimos anos, novas tecnologias tem surgido cada vez mais rápido, tornando o tema qualidade de código ainda muito relevante. Perguntas como “o material já documentado também se aplica a determinada nova tecnologia?” e “determinada nova tecnologia teria formas novas para criar código com qualidade?” precisam ser respondidas.

Uma das tecnologias que surgiu na última década e vem se popularizando com velocidade é a plataforma Android para o desenvolvimento de aplicativos móveis. Mais de 83,5% dos dispositivos móveis no mundo usam o sistema operacional Android, e esse percentual vem crescendo ano após ano [3, 4]. Alguns estudos tem sido feito em torno dessa plataforma, mas ainda são muito poucos. Umme et al. [28] recentemente levantaram que no período de 2008 a 2015, nas principais conferências de manutenção de software (ICSE, FSE, OOPS-LA/SPLASH, ASE, ICSM/ICSME, MRS e ESEM) apenas 10% dos artigos consideraram projetos Android em suas pesquisas. Nenhuma outra plataforma móvel foi considerada.

Dentre os estudos já realizados sobre qualidade de código Android, podemos encontrar alguns buscando por maus cheiros tradicionais, como os definidos por Martin Fowler [17], em projetos Android e outros buscando por maus cheiros específicos a plataforma Android.

Estudos que buscaram relacionar de alguma forma maus cheiros tradicionais em projetos Android são, por exemplo, o estudo de Linares et al. [27] onde se utilizam do método DECOR para detectar 18 *anti-patterns* tradicionais em aplicativos móveis. Verloop [46] analisou se classes derivadas do SDK Android, conjunto de classes base para o desenvolvimento Android, são mais ou menos propensas a apresentar os maus cheiros tradicionais do que classes puramente Java, ou seja, que não precisam herdar de nenhuma classe do SDK Android. Estudos que buscaram por maus cheiros específicos ao contexto Android são, por exemplo, de Fulano **todo: arrumar** [21, ?] que identificaram cheiros de código específicos Android relacionados ao consumo inteligente de recursos do dispositivo, como bateria e memória, usabilidade, dentre outros.

Aniche et al. [8, 7] apontam a relevância de se estudar qualidade de código em tecnologias específicas ao concluir que a arquitetura do software é um fator importante e que deve ser levado em conta ao analisar a qualidade de um sistema. Notamos também que o desenvolvimento do *front-end* de aplicações costuma apresentar diferenças com relação ao *back-end*. Por exemplo, encontramos maus cheiros específicos a tecnologias de *front-end* web como CSS [19], Javascript [16] e o arcabouço Spring MVC [9].

1.2 Objetivos

Nossa premissa de pesquisa é que existem maus cheiros de código específicos ao *front-end* Android. Nossa pesquisa complementa as anteriores no sentido de que também buscamos por maus cheiros Android. E se difere delas pois buscamos maus cheiros relacionados à qualidade

de código no *front-end* Android. Por exemplo ACTIVITIES, FRAGMENTS e ADAPTERS são classes usadas na construção de telas e LISTENERS são responsáveis pelas interações com os usuários. Buscamos entender então quais são as *boas e más* práticas no desenvolvimento da interface visual de aplicativos Android.

Por meio de um questionário online com perguntas sobre boas e más práticas relacionadas ao *front-end* Android respondido por 45 desenvolvedores, derivamos 23 más práticas. Validamos a percepção dos desenvolvedores sobre essas más práticas através de um experimento com outros 20 desenvolvedores. Os resultados mostram que de fato existem boas e más práticas específicas ao *front-end* Android. Ainda que tenhamos validado com sucesso a percepção dos desenvolvedores sobre apenas duas, esperamos que este catálogo de más práticas possa contribuir com ideias iniciais para outras pesquisas sobre qualidade de código em projetos Android.

1.3 Contribuições

As contribuições deste trabalho são:

1. Catálogo com 23 más práticas e sugestões de soluções no desenvolvimento do *front-end* Android, derivadas a partir dos resultados obtidos com a aplicação de um questionário online respondido por 45 desenvolvedores.
2. A percepção de desenvolvedores sobre as quatro más práticas de alta recorrência através de um experimento online com 20 desenvolvedores. Obtivemos um resultado positivo e estatisticamente válido sobre duas delas.
3. Apêndice online [10] com roteiros dos questionários e outras informações da pesquisa para que outros pesquisadores possam replicar nosso estudo.

1.4 Organização do Trabalho

As seções seguintes deste artigo estão organizadas da seguinte forma: a Seção 4 aborda a metodologia de pesquisa. A Seção ?? apresenta o catálogo de más práticas a percepção de desenvolvedores sobre as quatro mais recorrentes. A Seção 8 aborda as ameaças à validade do estudo. A Seção 3 discute os trabalhos relacionados e o estado da arte sobre Android e cheiros de código. E por fim, a Seção 9 conclui.

Capítulo 2

Fundamentação Conceitual

Este trabalho trata de maus cheiros de código que encontra sua motivação quando pensamos em boas práticas para aumentar a qualidade do software.

Desta forma, esta seção define alguns termos considerados relevantes para o entendimento deste trabalho: qualidade de software, boas práticas e maus cheiros de código.

2.1 ANDROID

Android é um sistema operacional móvel, desenvolvido pelo Google, baseado no kernel do Linux e desenhado principalmente para o uso em dispositivos móveis com telas sensíveis ao toque [1]. Em 2017 é o sistema operacional móvel mais usado no mundo

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 [33].

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 múltiplas 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

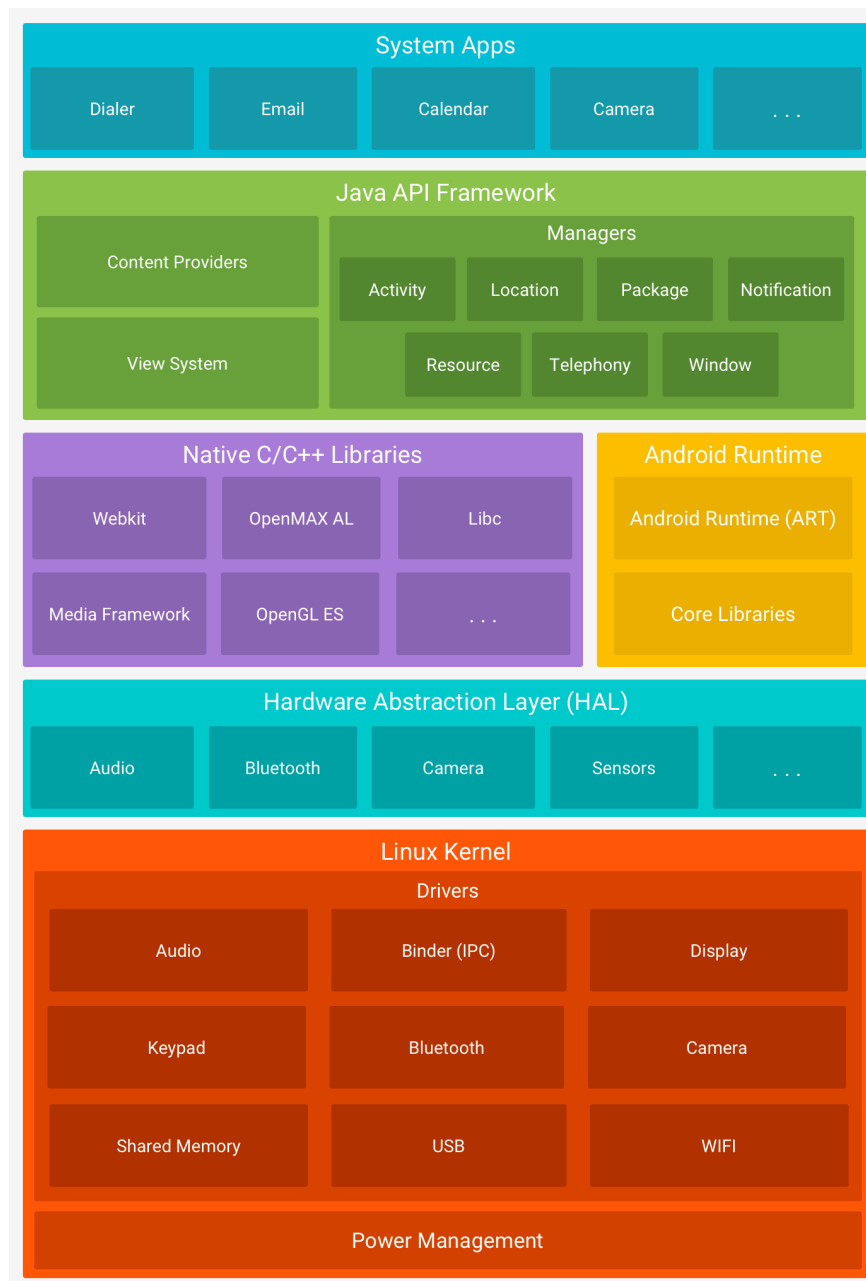


Figura 2.1: *Arquitetura do sistema operacional Android.*

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 construção 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 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 [40].

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 componentes 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>
```

Código-Fonte 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 estende 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 [40]. 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 [2].

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 francês) 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 [37]. 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 [39].

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.

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

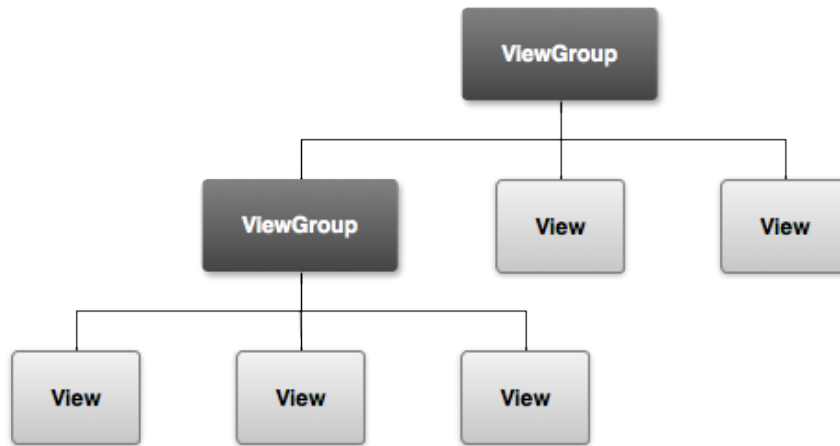


Figura 2.2: *Árvore hierárquica de Views e ViewGroups do Android.*

efetiva de definir um layout é através 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 corresponde 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 corresponde 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>
  
```

Código-Fonte 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 [36].

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 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.

2.2 QUALIDADE DE SOFTWARE

Qualidade de software é uma área do conhecimento da engenharia de software que trata das características desejadas em um projeto de software, sejam elas funcionais (o quanto o software atende as necessidades de negócio) ou estruturais (o quão bem o software funciona). Diversas características são desejáveis em um software, dentre elas confiabilidade, eficiência e manutenibilidade. A área de conhecimento sobre qualidade de software nos serve justamente para definir conceitos que suportam a criação de práticas e ferramentas que visam aumentar a qualidade de softwares. Muitos estudos já mostraram que problemas de qualidade em software demandam altos custos a empresas. Desta forma, essa área do conhecimento orienta nossos esforços para melhorar a qualidade de projetos de software visando, dentre outras coisas, reduzir os custos.

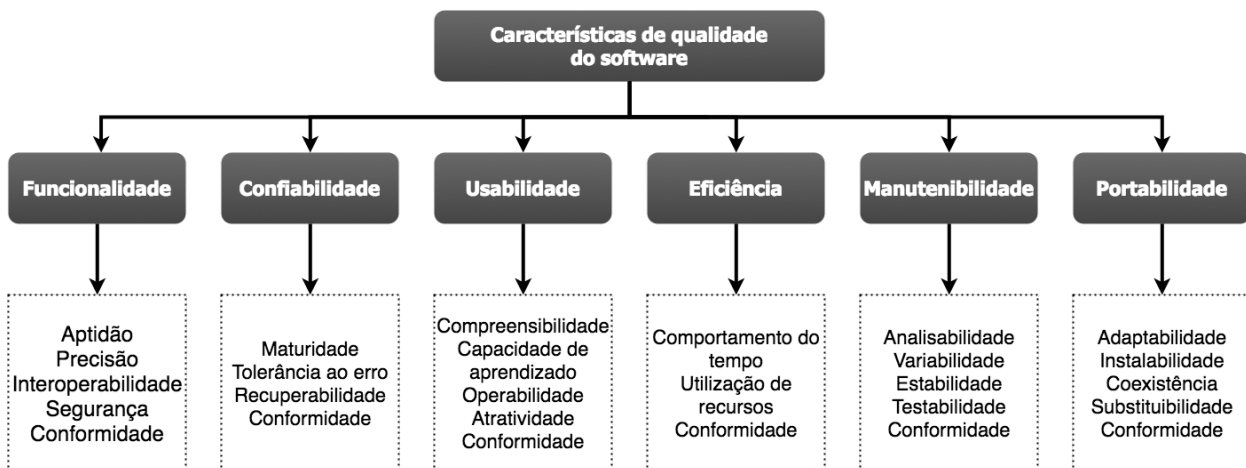


Figura 2.3: *Qualidade de Software (ISO/EIC 9126-1 e ISO/IEC 25010 e [44]).* **todo:** *corrigir caption*

todo: *para embasar as traduções na imagem https://pt.wikipedia.org/wiki/Qualidade_de_software*

As primeiras definições sobre *qualidade de software* são derivadas das normas ISO/EIC 9126-1 e ISO/IEC 25010 que diz que qualidade é a capacidade do produto de software satisfazer as necessidades implícitas e explícitas quando usado em condições específicas. Estas normas subdivide qualidade de software em 6 áreas, cada qual contendo um conjunto de

sub-áreas, conforme a Figura 2.3. Esta pesquisa está inserida no contexto da sub-área *manutenabilidade*, que discutiremos a seguir, onde engloba diversos conceitos, dentre os quais *analísabilidade*, quão fácil é para nós localizar os elementos que queremos melhorar ou corrigir e *alterabilidade*, quanto trabalho precisamos fazer para implementar uma modificação [44].

Existem diversas definições dadas ao termo qualidade de software. Segundo a norma ISO 9000 (versão 2000), qualidade é “o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes”. Já Juran, um dos principais autores sobre o assunto, tem duas definições para qualidade “a característica dos produtos que atendem as necessidades dos clientes, e, assim, proporcionar a satisfação do mesmo[2]” e “qualidade é a ausência de deficiências[2]”. Ao longo do tempo a teoria de qualidade de software foi evoluindo, resultando inclusive na criação dos padrões ISO/EIC 9126 e ISO/IEC 25000. Uma importante contribuição foi a divisão do conceito de qualidade em subconceitos, no caso de software, inicialmente proposto por McCall, Richards e Walters[3]. **todo: ajusta refs https://pt.wikipedia.org/wiki/Qualidade_de_software#cite_ref-6**

Muito embora tenha-se provado que investir em qualidade pode reduzir os custos de um projeto **todo: refs?**, aumentar a satisfação dos usuários e desenvolvedores **todo: refs??**, qualidade de software costuma ser esquecido ou deixado em segundo plano **todo: refs?**. Quem nunca ouviu a frase “depois eu testo”, ou “depois eu refatoro” no dia a dia? Manutenibilidade está relacionada as necessidades implícitas do software como legibilidade e alterabilidade, focado nesses aspectos, ao longo dos últimos anos foram criados diversos conceitos visando aumentar a qualidade relativa a eles. Por exemplo os padrões de projeto do GoF veem com o objetivo de documentar *melhores soluções para problemas comuns* originadas a partir do conhecimento empírico de desenvolvedores de software experientes. Anti-padrões são padrões antes recomendados que passaram a ser evitados pois percebeu-se que os problemas em usá-los superavam os benefícios **todo: referencia aqui do livro anti-patterns**, um dos maiores exemplos de anti-padrão é o *Singleton* **todo: breve desc aqui**. *Maus cheiros de código* que trataremos melhor adiante **todo: colocar seção aqui**, apontam sintomas que podem indicar um problema mais profundo no software. É interessante notar que, enquanto que padrões de projetos são conceitos que indicam “o que fazer”, anti-padrões e maus cheiros são conceitos que servem como alertas sobre “o que não fazer” ou sobre “o que evitar”. Esse conjunto de conceitos e ferramentas são comumente generalizados entre os desenvolvedores simplesmente pelo termo *boas práticas de software*.

Os erros de construção detectados no teste do sistema custam 10 vezes mais para consertar do que na fase de construção. Os erros de construção detectados postrelease custaram 10-25 vezes mais para consertar do que na fase de construção (Fagan 1976; Dunn 1984; Boehm & Turner 2004, Shull et al., 2002). O Laboratório de Engenharia de Software da NASA descobriu que a leitura de código detectou 3,3 defeitos por hora de esforço: o teste detectou

cerca de 1,8 erros por hora. A leitura do código encontrou 20 a 60% mais erros durante a vida do projeto do que os vários tipos de testes (Card 1987). 50% a 80% das declarações “if” simples deveriam ter tido um “else” cláusula (Elshoff, 1976). Poucas pessoas podem entender mais de 3 ou 4 níveis de ifs aninhados (Yourdon 1986; Ledgard & Tauer, 1987). A complexidade do fluxo de controle tem sido correlacionada com baixa confiabilidade e erros freqüentes (McCabe 1976, Shen et al., 1985, Ward, 1989).

2.2.1 Manutenibilidade

2.3 BOAS PRÁTICAS DE SOFTWARE

Em desenvolvimento de software, *boas práticas de código* são um conjunto de regras que a comunidade de desenvolvimento de software aprendeu ao longo do tempo, e que pode ajudar a melhorar a qualidade do software [30]. Dentre este conjunto de regras temos padrões de projetos, anti-padrões e *maus cheiros de código*.

2.3.1 Padrões de Projeto

***Padrão:** Algo que serve como modelo. Um exemplo para os outros seguirem.*

— Dicionário Oxford [2]

Não podemos falar sobre *padrões de projetos* sem antes falarmos sobre *padrões*. Segundo o dicionário Oxford um padrão é *algo que serve como modelo, um exemplo para os outros seguirem* [2]. Padrões não são invenção de algo novo, é uma forma de organizar o conhecimento de experiências [1].

Em engenharia de software, a principal definição sobre *padrões* foi cunhada no livro Uma Linguagem de Padrões do arquiteto Christopher Alexander (1977) [50] onde ele define um *padrão* como sendo uma regra de *três partes* que expressa a *relação* entre um certo **contexto**, um **problema** e uma **solução**. Martin Fowler apresenta uma definição mais simples que diz que *um padrão é uma ideia que foi útil em algum contexto prático e provavelmente será útil em outros* [9].

Inspirados por Alexander [50], Kent Beck e Ward Cunningham fizeram alguns experimentos do uso de padrões na área de desenvolvimento de software e apresentaram estes resultados na OOPSLA em 1987. E, apoiando-se na definição de padrões de Alexander, Design Pattern - GoF (1994) foi o primeiro livro sobre padrões de software a ser lançado. Documentando X padrões.

Logo, temos que padrões são modelos a serem seguidos visto que já...

2.3.2 Anti-Padrões

Um anti-padrão é uma resposta comum a um problema recorrente que geralmente é ineficaz e corre o risco de ser altamente contraproducente. O termo foi cunhado por Andrew Koenig em um artigo publicado em 1995, inspirado pelo livro GoF.

O termo se popularizou 3 anos após, em 1998 com o livro *Antipatterns*. Uma importante informação que este livro agrega é como diferenciar um anti-padrão de um mal hábito, má prática ou ideia ruim. Para isso o livro cita que um anti-padrão deve apresentar dois elementos: 1) um processo, estrutura ou padrão de ação comumente usado que, apesar de inicialmente parecer ser uma resposta adequada e efetiva a um problema, tem mais consequências ruins do que as boas, 2) existe outra solução que é documentada, repetitiva e provada ser eficaz.

2.4 MAUS CHEIROS DE CÓDIGO

“Se cheirar mau, troque-o.”

— Citação da avó Beck no Livro *Refactoring*
[48]

Podemos entender um mau cheiro de código como uma percepção empírica de que algo não está certo no software, percepção semelhante o dito popular “Algo não cheira bem!” quando uma pessoa está desconfiada de que há um problema em dada situação.

Um dos primeiros livros a usar o *conceito* de mau cheiro de código foi o livro de Webster (1995) [48] através do termo “*armadilha*”. No livro são apresentadas 82 *armadilhas* no Desenvolvimento Orientado a Objetos originadas da experiência do autor. Longos métodos e complexidade excessiva por exemplo são definidos na *armadilha Letting objects Become Bloated* [48, p. 180].

Apesar do conceito já permear em livros e publicações sobre desenvolvimento de software desde o começo da década de 90, o termo só se popularizou após o livro *Refatoração: Aperfeiçoando o Projeto de Código Existente* (Fowler, M 1999) [17]. Refatoração, segundo Fowler, é o ato de *alterarmos o código sem alterar seu comportamento* com o objetivo de torná-lo mais fácil de ser entendido e menos custoso de ser modificado. No livro são apresentadas diversas técnicas de refatoração. O termo mau cheiro de código é usado para explicar ao leitor o “quando” uma refatoração deve ser aplicada. Segundo o livro, “***mau cheiro de código são indicações de que existe um problema que pode ser resolvido por meio de uma refatoração***”. Essas indicações provêm do conhecimento empírico de desenvolvedores experientes, que ao longo dos anos, após trabalharem em diversos códigos bons e ruins, são capazes de listar características que podem indicar se um trecho de código é problemático.

Em uma postagem em seu site oficial em 2006 [18], Fowler afrouxou um pouco mais a definição do termo dizendo que “*mau cheiro de código são indicações superficiais que geralmente corresponde a um problema mais profundo no sistema*”.

No livro são apresentados 22 maus cheiros e mais de 70 técnicas de refatoração. *Código Duplicado* [17, p. 63] é um exemplo de mau cheiro que trata de problemas comuns que podem resultar na duplicação de código, por exemplo, ter a mesma expressão em dois métodos da mesma classe ou em duas subclasses irmãs. Para resolver este mau cheiro é indicada a refatoração *Extrair Método* [17, p. 89], ou seja, extrair a expressão duplicada para um novo método e substituí-lo nos lugares onde a expressão era usada.

Nos anos seguintes o termo mau cheiro se tornou frequente em livros [29, 45] e pesquisas acadêmicas. No livro *Clean Code* (Martin, R 2008) [29], que se tornou muito popular entre desenvolvedores de software, são definidos novos maus cheiros além de citar alguns já apresentados por Fowler [17]. Ele se apoia na definição de Fowler para explicar o conceito de mau cheiro.

2.4.1 Formato dos Maus Cheiros

Os primeiros maus cheiros definidos vieram em formato textual e diferentemente de padrões, não parece existir guias de como documentar um mau cheiro.

Fowler [17] por exemplo, se utiliza de *título* e um *texto explicativo*. No *texto explicativo*, é possível encontrarmos informações sobre contexto, exemplos de problemas comuns e possíveis refatorações, dentre as listadas no livro, que resolveriam o mau cheiro. A seguir, temos o mau cheiro *Código Duplicado* [17, p. 71]. O *título* do mau cheiro indica o contexto por ele tratado. No parágrafo (1) podemos observar uma breve resumo do que faz cheirar mal e uma possível refatoração. Nos parágrafos seguintes (2-4) são apresentadas em mais detalhes situações comuns que podem indicar a presença do mau cheiro.

Código Duplicado

O número um no *ranking* dos cheiros é o código duplicado. Se você vir o mesmo código em mais de um lugar, pode ter certeza de que seu programa será melhor se você encontrar uma maneira de unificá-los (1).

O problema mais simples de código duplicado é quando você tem a mesma expressão em dois métodos da mesma classe. Tudo o que você tem que fazer então é utilizar o *Extrair Método* e chamar o código de ambos os lugares (2).

Outro problema de duplicação comum é quando você tem a mesma expressão em duas subclasses irmãs. Você pode eliminar essa duplicação usando *Extrair Método* em ambas as classes e então *Subir Método na Hierarquia*. Se o código for similar mas não o mesmo, você

precisa usar *Extrair Método* para separar as partes semelhantes daquelas diferentes. Você pode então descobrir que pode usar *Criar um Método Padrão*. Se os métodos fazem a mesma coisa com um algoritmo diferente, você pode escolher o mais claro deles e usar *Substituir o Algoritmo* (3).

Se você tem código duplicado em duas classes não relacionadas, considere usar *Extrair Classe* em uma classe e então usar o novo componente na outra. Uma outra possibilidade é de que o método realmente pertença a apenas uma das classes e deva ser chamado pela outra ou que o método pertença a uma terceira classe que deva ser referida por ambas as classes originais. Você tem que decidir onde o método faz mais sentido e garantir que ele esteja lá e em mais nenhum lugar (4).

Martin [29] usa a mesma estrutura usada por Fowler e adiciona a ela uma *sigla*. O *texto explicativo* é apresentado em parágrafo único e é possível encontrar contexto, alguns exemplos de problemas comuns e exemplos de código, sendo que alguns maus cheiros apresentam todas essas informações ou apenas uma combinação delas. O *título* usado por Martin aponta de certa forma o problema (o uso de convenção durante o desenvolvimento) e a solução (sempre que possível, preferir o uso de estruturas acima da convenção).

A seguir temos o mau cheiro *G27: Estrutura acima de convenção* [29, p. 301]. No *texto explicativo* em parágrafo único, podemos observar em orações a mesma estrutura que vimos no mau cheiro anterior, porém em parágrafos. Na oração (1) temos algo relacionado ao que faz cheirar mal, na oração (2) uma possível refatoração, na (3) é dado um exemplo e na (4) é indicado o problema resultante de se basear em convenção.

G27: Estrutura acima de convenção

Insista para que as decisões do projeto baseiem-se em estrutura acima de convenção (1). Convenções de nomenclaturas são boas, mas são inferiores a estruturas, que forçam um certo cumprimento (2). Por exemplo, `switch/case` com enumerações bem nomeadas são inferiores a classes base com métodos abstratos (3). Ninguém é obrigado a implementar a estrutura `switch/case` da mesma forma o tempo todo; mas as classes bases obrigam a implementação de todos os métodos abstratos das classes concretas (4).

Em pesquisas que definem novos maus cheiros, observamos um formato similar aos mencionados porém, adicionado uma estratégia de detecção, com foco em automatizar a identificação do mau cheiro [8, 19]. Estas estratégias comumente se baseiam em métricas de software já existentes, como as mencionadas na sessão Métricas X.X.

Podemos dizer então que um padrão é a documentação formal de boas práticas.

Capítulo 3

Trabalhos Relacionados

Muitas pesquisas têm sido realizadas sobre a plataforma Android, muitas delas focam em vulnerabilidades [11, 13, 14, 25, 49, 52, 53], autenticação [15, 50, 51] e testes [24, 6]. Diferentemente dessas pesquisas, nossa pesquisa tem foco na percepção dos desenvolvedores sobre boas e más práticas de desenvolvimento na plataforma Android.

A percepção desempenha um importante papel na definição de code smells relacionados a uma tecnologia, visto que code smells possuem uma natureza subjetiva. Code smells desempenham um importante papel na busca por qualidade de código, visto que, após mapeados, podemos chegar a heurísticas para identificá-los e com essas heurísticas, implementar ferramentas que automatizem o processo de identificar códigos problemáticos.

Verloop [46] conduziu um estudo no qual avaliou por meio de 4 ferramentas de detecção automatizada de cheiros de código (JDeodorant, Checkstyle, PMD e UCDetector) a presença de 5 cheiros de código (Long Method, Large Class, Long Parameter List, Feature Envy e Dead Code) em 4 projetos Android. Nossa pesquisa se relaciona com a de Verloop no sentido de que também estamos buscando por cheiros de código, entretanto, em vez de buscarmos por cheiros de código já definidos, realizamos uma abordagem inversa na qual, primeiro buscamos entender a percepção de desenvolvedores sobre boas e más práticas em Android, e a partir dessa percepção, relacionamos com algum cheiro de código pré-existente ou derivamos algum novo.

Gottschalk et al [21] conduziram um estudo sobre formas de detectar e refatorar cheiros de código relacionados ao uso eficiente de energia. Os autores compilaram um catálogo com 8 cheiros de código e trabalharam sob um trecho de código Android para exemplificar um deles, o "binding resource too early", quando algum recurso é alocado muito antes de precisar ser utilizado. Essa pesquisa é relacionada à nossa por ambas considerarem a tecnologia Android e se diferenciam pois focamos na busca por cheiros de código relacionados a qualidade de código, no sentido de legibilidade e manutenabilidade.

Aplicativos Android são escritos na linguagem de programação Java [40]. Então a pri-

meira questão é: por que buscar por *smells* Android sendo que já existem tantos *smells* Java? Pesquisas têm demonstrado que tecnologias diferentes podem apresentar *code smells* específicos, como por exemplo Aniche et al. identificaram 6 *code smells* específicos ao framework Spring MVC, um framework Java para desenvolvimento web. Outras pesquisas concluem que projetos Android possuem características diferentes de projetos java [23, 28, ?], por exemplo, o *front-end* é representado por arquivos XML e o ponto de entrada da aplicação é dado por *event-handler* [42] como o método ONCREATE. Encontramos também diversas pesquisas sobre *code smells* sobre tecnologias usadas no desenvolvimento de *front-end* web como CSS [19] e JavaScript [16]. Essas pesquisas nos inspiraram a buscar entender se existem *code smells* no *front-end* Android.

Capítulo 4

Metodologia

O *objetivo* deste estudo é investigar a existência de boas e más práticas no desenvolvimento do *front-end* Android. Neste trabalho, respondemos às seguintes questões de pesquisa:

QP1. O que desenvolvedores consideram boas e más práticas no desenvolvimento Android?

QP2. Códigos afetados por estas más práticas são percebidos pelos desenvolvedores como problemáticos?

Para alcançar nosso objetivo, conduzimos um estudo qualitativo e exploratório no qual coletamos dados através de um questionário e derivamos más práticas Android e, na sequência, conduzimos um estudo online, para capturar a percepção de desenvolvedores Android em relação às classes afetadas por essas más práticas.

Esta Seção está organizada da seguinte forma: na Seção 4.1 abordamos o processo de descoberta das boas e más práticas Android e na Seção 4.2 abordamos o processo de validação da percepção dos desenvolvedores sobre as más práticas definidas.

4.1 QP1: Boas e Más Práticas Android

Para responder a **QP1**, buscamos entender o que desenvolvedores consideram boas e más práticas no desenvolvimento do *front-end* Android. Para tal, optamos por um estudo qualitativo.

Segundo Strauss e Corbin [12], há muitos motivos válidos para se fazer pesquisa qualitativa, dentre eles a natureza do problema de pesquisa e para obter mais informações sobre áreas cujo ainda se sabe pouco, como é o caso de cheiros de código Android. Muitas vezes, a pesquisa qualitativa, constitui-se na primeira etapa de uma investigação mais ampla na qual se busca o entendimento de um assunto específico por meio de descrições, comparações

e interpretações dos dados [?, 31]. Algumas características básicas da pesquisa qualitativa, como *a)* o foco na interpretação que os participantes possuem quanto à situação investigada e *b)* o fato de enfatizar a subjetividade e a flexibilidade, orientando-se para o processo e não para o resultado, justificam seu uso neste artigo [31, 26].

Para definirmos quais elementos representam o *front-end* Android, fizemos uma extensa revisão da documentação oficial e chegamos nos seguintes itens: ACTIVITIES, FRAGMENTS, LISTENERS, ADAPTERS e os recursos do aplicativo, que são arquivos XML ou imagens utilizados na interface visual, como por exemplo DRAWABLES, LAYOUTS, STYLES e COLORS. Como existem muitos tipos de recursos do aplicativo [38], selecionamos quatro: LAYOUT, STYLES, STRING e DRAWABLE. Optamos por esses recursos pois estão presentes no template padrão do Android Studio [35], IDE oficial para desenvolvimento de projetos da plataforma Android [34].

O questionário (*S1*) é composto por 25 questões divididas em três seções. A primeira seção é composta por 6 perguntas demográficas, a segunda seção é composta por 16 perguntas sobre boas e más práticas relacionadas ao *front-end* Android e a terceira seção é composta por 3 perguntas, 2 para obter últimos pensamentos sobre boas e más práticas e 1 solicitando email caso o participante tivesse interesse em etapas futuras da pesquisa. O questionário é escrito em inglês, porém informava o participante que respostas em inglês ou português eram aceitas. Antes da divulgação, realizamos um piloto com 3 desenvolvedores Android onde os *feedbacks* nos fizeram configurar as perguntas da segunda e terceira seção como opcionais. As respostas dos participantes piloto foram desconsideradas para efeitos de viés. O questionário completo pode ser encontrado em nosso apêndice online.

A primeira seção é composta por questões cujo objetivo foi traçar o perfil demográfico do participante (idade, estado de residência, experiência em desenvolvimento de software e em desenvolvimento Android). A segunda seção é composta por 16 questões opcionais e dissertativas sobre boas e más práticas relacionadas ao *front-end* Android. Para cada elemento do *front-end* Android foram feitas duas perguntas abertas, onde a participante deveria discutir sobre boas e más práticas percebidas por ele naquele elemento. Por exemplo, para o elemento ACTIVITY foram feitas as seguintes perguntas:

Q1 Você tem alguma boa prática para lidar com Activities? (Resposta aberta)

Q2 Você considera alguma coisa uma má prática ao lidar com Activities? (Resposta aberta)

A terceira seção é composta por 3 perguntas opcionais e abertas, 2 para captar qualquer última ideia sobre boas e más práticas não captadas nas questões anteriores e 1 solicitando o email do participante caso o mesmo tivesse interesse em participar de etapas futuras da pesquisa.

Antes da divulgação, realizamos um piloto com 3 desenvolvedores Android e o feedback nos mostrou que nem sempre o desenvolvedor teria algo a comentar sobre todos os componentes Android que estávamos questionando, com este feedback removemos a obrigatoriedade das perguntas da segunda seção do questionário, onde todas tornaram-se opcionais. As respostas dos participantes piloto foram desconsideradas para efeitos de viés.

O questionário foi divulgado em redes sociais como Facebook, Twitter e LinkedIn, em grupos de discussão sobre Android como *Android Dev Brasil*, *Android Brasil Projetos* e o grupo do *Slack Android Dev Br*, maior grupo de desenvolvedores Android do Brasil com 2622 participantes até o momento da escrita deste artigo. O questionário esteve aberto por aproximadamente 3 meses e meio, de 9 de Outubro de 2016 até 18 de Janeiro de 2017. Ao final, o questionário foi respondido por 45 desenvolvedores.

Para análise dos dados seguimos a abordagem de *Ground Theory* (GT), um método de pesquisa exploratória originada nas ciências sociais [12, 20], mas cada vez mais popular em pesquisas de engenharia de software [5]. A GT é uma abordagem indutiva, pelo qual dados provindos, por exemplo de, entrevistas ou questionários, são analisadas para derivar uma teoria. O objetivo é descobrir novas perspectivas mais do que confirmar alguma já existente. Realizamos um processo de codificação aberta sobre os dados, resultando num conjunto com 23 más práticas Android. Essas más práticas foram agrupadas de acordo com sua recorrência nas respostas, ou seja, a quantidade de respostas em que cada má prática é percebida. Explicamos o processo de análise com mais detalhes na Seção 4.1.

Todas as 18 questões sobre boas e más práticas (16 na segunda seção e 2 da terceira) são apresentadas na Tabela 4.1.

Participantes

Recebemos um total de 45 respostas. 80% dos participantes responderam pelo menos 3 perguntas sobre boas e más práticas no *front-end* Android, sendo que 7 responderam de 3 a 6, 6 responderam de 8 a 10 e 23 responderam 13 ou mais, sendo que desses, 14 responderam todas. Apenas 20% responderam uma (2 participantes) ou nenhuma (7 participantes) questão. A pergunta solicitando o email foi respondida por 53% dos participantes, o que pode indicar um interesse legítimo da comunidade de desenvolvedores Android pelo tema, reforçando a relevância do estudo. A pergunta mais respondida foi a Q1 e a menos respondida foi a Q18, é possível ver detalhes desta análise na Tabela 4.1.

Com a análise das questões demográficas, notamos que atingimos com sucesso *desenvolvedores Android com variados níveis de experiência e de diversas regiões* pois: 1) 100% dos participantes indicaram possuir alguma experiência com desenvolvimento Android, 2) 14% indicaram possuir 1 ano ou menos de experiência com Android e 86% indicaram 2 anos ou mais (15,5% 2 anos, 13,3% 4 anos, 6,5% 5 anos, 15,5% 6 anos, 4,4% 7 anos e 4,4% 8 anos), 4)

Id	Questões	Respostas		Participantes
		Total	%	
Q1	Você tem alguma boa prática para lidar com Activities?	36	80%	P1, P2, P4-P12, P14-P17, P19, P22, P23, P25-P32, P34-P37, P39-P43, P45
Q2	Você considera alguma coisa uma má prática ao lidar com Activities?	35	78%	P2, P4-P11, P14-P17, P19, P22, P23, P25-P32, P34-P37, P39-P45
Q3	Você tem alguma boa prática para lidar com Fragments?	33	73%	P4-P11, P14-P17, P19, P22, P23, P25-P28, P30-P32, P34-P37, P39-P45
Q4	Você considera alguma coisa uma má prática ao lidar com Fragments?	31	69%	P2, P4-P11, P14, P15, P17, P19, P22, P23, P25-P28, P31, P32, P34-P37, P39-P43, P45
Q5	Você tem alguma boa prática para lidar com Adapters?	30	67%	P2, P4-P11, P14, P15, P17-P19, P22, P23, P26, P28, P29, P31, P32, P34-P37, P39-P43, P45
Q6	Você considera alguma coisa uma má prática ao lidar com Adapters?	27	60%	P2, P4-P8, P10, P11, P14, P18, P19, P22, P23, P26, P28, P31, P34-P37, P39-P45
Q7	Você tem alguma boa prática para lidar com Listeners?	24	53%	P2, P4-P6, P8, P9, P11, P14, P22, P23, P26, P28, P29, P31, P32, P34, P36, P37, P39-P43, P45
Q8	Você considera alguma coisa uma má prática ao lidar com Listeners?	23	51%	P2, P4, P5, P8, P9, P11, P14, P18, P19, P22, P23, P26, P28, P31, P32, P34, P36, P37, P39-P44
Q9	Você tem alguma boa prática para lidar com Layout Resources?	28	62%	P4-P9, P11, P14, P19, P22, P23, P26-P29, P31, P32, P34-P37, P39-P45
Q10	Você considera alguma coisa uma má prática ao lidar com Layout Resources?	23	51%	P4, P5, P7-P9, P11, P22, P23, P26, P28, P31, P32, P34-P37, P39-P45
Q11	Você tem alguma boa prática para lidar com Styles Resources?	23	51%	P4-P9, P11, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q12	Você considera alguma coisa uma má prática ao lidar com Styles Resources?	22	49%	P4-P8, P11, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q13	Você tem alguma boa prática para lidar com String Resources?	28	62%	P4-P6, P8-P11, P14, P18, P22, P23, P26-P29, P31, P32, P34-P37, P39-P45
Q14	Você considera alguma coisa uma má prática ao lidar com String Resources?	23	51%	P4-P6, P8, P9, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34-P37, P40-P43, P45
Q15	Você tem alguma boa prática para lidar com Drawable Resources?	24	53%	P4-P6, P8-P11, P14, P18, P22, P23, P26, P28, P31, P32, P34-P37, P39-P43
Q16	Você considera alguma coisa uma má prática ao lidar com Drawable Resources?	21	47%	P4-P6, P8, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34, P36, P37, P40-P44
Q17	Existem outras *BOAS* práticas sobre a Camada de Apresentação Android que nós não perguntamos ou que você não disse ainda?	22	49%	P2, P4, P8, P10, P11, P14, P18, P22, P23, P26, P28, P31, P32, P34, P36, P37, P39-P43, P45
Q18	Existem outras *MÁS* práticas sobre a Camada de Apresentação Android que nós não perguntamos ou que você não disse ainda?	20	44%	P2, P4, P8, P10, P11, P18, P22, P23, P28, P31, P32, P34, P36, P37, P40-P45

* Os participantes P3, P13, P20, P21, P24, P33 e P38 não responderam nenhuma das questões da segunda e terceira seção.

Tabela 4.1: Total de respostas obtidas por cada questão sobre boas e más práticas no front-end Android.

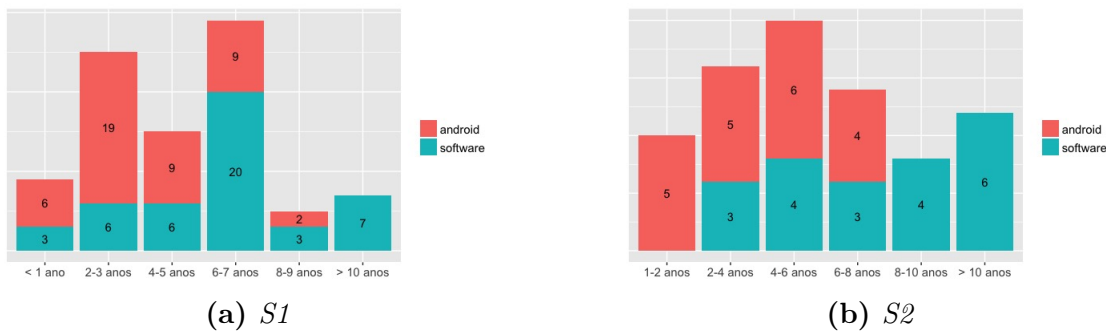


Figura 4.1: Experiência com desenvolvimento de software e android dos desenvolvedores.

36 respostas foram do Brasil, 7 de países europeus e 1 dos Estados Unidos (Califórnia). Vale lembrar que a plataforma Android completa 10 anos em 2017, ou seja, 5 anos de experiência nessa plataforma representa 50% do tempo de vida dela desde seu anúncio em 2007. Os dados sobre a experiência dos participantes são apresentados na Figura 4.1a.

Análise dos Dados

O processo de análise partiu da listagem das 45 respostas do questionário e se deu em 4 passos: *verticalização*, *limpeza dos dados*, *codificação* e *divisão*.

O processo que denominamos de *verticalização* consistiu em considerar cada resposta de

boa ou má prática como um registro individual a ser analisado. Ou seja, cada participante respondeu 18 perguntas sobre boas e más práticas no *front-end* Android (2 perguntas para cada elemento e mais duas perguntas genéricas). Com o processo de *verticalização*, cada uma dessas respostas se tornou um registro, ou seja, cada participante resultava em 18 respostas a serem analisadas, totalizando 810 respostas (18 perguntas multiplicado por 45 participantes) sobre boas e más práticas.

O passo seguinte foi realizar a *limpeza dos dados*. Esse passo consistiu em remover respostas obviamente não úteis como respostas em branco, que foi composto por frases como "*Não*", "*Não que eu saiba*", "*Eu não me lembro*" e similares, as consideradas vagas como "*Eu não tenho certeza se são boas praticas mas uso o que vejo por ai*", as consideradas genéricas como "*Como todo código java...*" e as que não eram relacionadas a boas práticas de código. Das 810 boas e más práticas, 352 foram consideradas e 458 desconsideradas. Das 352, 44,6% foram apontadas como más práticas e 55,4% como boas práticas.

Em seguida, realizamos a *codificação* sobre as boas e más práticas [12, 32]. Codificação é o processo pelo qual são extraídos categorias de um conjunto de afirmações através da abstração de ideias centrais e relações entre as afirmações [12]. Durante esse processo, cada resposta recebeu uma ou mais categorias. Toda resposta desconsiderada nesse passo, foi indicado um motivo que pode ser conferido nos arquivos em nosso apêndice online.

Por último realizamos o passo de *divisão*. Esse passo consistiu em dividir as respostas que receberam mais de uma categoria em duas ou mais respostas, de acordo com o número de categorias identificadas, de modo a resultar em uma categoria por resposta. Por exemplo, a resposta "*Não fazer Activities serem callbacks de execuções assíncronas. Herdar sempre das classes fornecidas pelas bibliotecas de suporte, nunca diretamente da plataforma*" indica na primeira oração uma categoria e na segunda oração, outra categoria. Ao dividi-la, mantivemos apenas o trecho da resposta relativo à categoria, como se fossem duas respostas distintas e válidas. Em algumas divisões realizadas, a resposta completa era necessária para entender ambas as categorizações, nesses casos, mantivemos a resposta original, mesmo que duplicada, e categorizamos cada uma de modo diferente.

Ao final da análise constavam 389 respostas individualmente categorizadas sobre boas e más práticas. A partir dessas respostas derivamos 23 más práticas no *front-end* Android agrupadas por recorrência: alta (acima de 20 respostas), média (de 8 a 20 respostas) e baixa (de 3 a 7 respostas).

4.2 QP2: Percepção dos Desenvolvedores

Para responder a **QP2**, buscamos entender a percepção dos desenvolvedores sobre as más práticas derivadas. Como primeiro passo, focamos nas más práticas que apresentaram alta recorrência. As opiniões foram coletadas através de um estudo online (*S2*) respondido

por 20 desenvolvedores Android. Nossas análises demonstram que de fato, códigos afetados pelas más práticas são percebidos pelos desenvolvedores como códigos problemáticos.

O experimento foi composto por duas seções principais. A primeira objetivou coletar informações básicas sobre os antecedentes dos participantes e, em particular, sobre sua experiência. Na segunda seção, os participantes foram solicitados a examinar seis códigos-fonte Android e, para cada uma deles, responder 5 perguntas onde perguntamos se considerava o código como problemático, e se sim, pedíamos para avaliar a severidade do problema a partir de uma escala Likert de 1 a 5. Abaixo listamos essas 5 perguntas.

- Q1.** Na sua opinião, este código apresenta algum problema de design e/ou implementação? (Sim/Não)
- Q2.** Se SIM, por favor explique quais são, na sua opinião, os problemas que afetam este código. (Resposta aberta)
- Q3.** Se SIM, por favor avalie a severidade do problema de design e/ou implementação selecionando dentre as opções a seguir um ponto. (Escala *Likert* de 5 pontos indo de 1 – muito baixo – a 5 – muito alto)
- Q4.** Na sua opinião, este código precisa ser refatorado? (Sim/Não)
- Q5.** Se SIM, como você faria esta refatoração? (Resposta aberta)

Os seis códigos apresentadas foram selecionadas randomicamente para cada participante de um conjunto de 58 códigos, contendo 24 códigos afetadas por uma das quatro más práticas Android de alta recorrência (seis para cada má prática), 10 códigos afetadas por cheiros de códigos tradicionais e 24 códigos limpos. Para possibilitar que os códigos fossem apresentados dessa forma, desenvolvemos um software específico. Para reduzir viés, selecionamos apenas códigos relacionados ao *front-end* Android definido no contexto deste artigo, ou seja: ACTIVITIES, FRAGMENTS, ADAPTERS, LISTENERS, STYLES, STRINGS, DRAWABLES e LAYOUTS. Cada participante avaliou dois códigos selecionados randomicamente de cada um desses três grupos, totalizando 6 códigos avaliados por participante. Os 58 códigos foram aleatoriamente coletados de projetos Android de código aberto no GitHub.

Para também reduzir viés de aprendizado, cada participante recebeu os seis códigos selecionados aleatoriamente em uma ordem aleatória. Além disso, os participantes não estavam cientes de quais classes pertenciam a qual grupo (más práticas Android, cheiros de código tradicionais e limpo). Apenas foi dito que estávamos estudando qualidade de código em aplicações Android. Nenhum limite de tempo foi imposto para que eles concluíssem a tarefa.

No teste piloto realizado com 2 desenvolvedores não foram identificados ponto a otimizar, essas respostas foram desconsideradas para reduzir viés. O questionário do experimento esteve disponível por 8 dias, de 27 de Abril a 4 de Maio de 2017. Sua divulgação foi feita em

duas etapas, na primeira, foi enviada ao grupo do Slack Android Dev Br, uma chamada a desenvolvedores com mais de 3 anos e experiência em Android, desta forma, desenvolvedores que tinham interesse de responder o questionário e, atendiam ao requisito de experiência, entravam em contato e enviávamos um email convite. Na segunda etapa, abrimos o questionário para participação de qualquer desenvolvedor Android. A divulgação nesta etapa, foi a mesma utilizada na divulgação em *S1*.

Participantes

Todos os participantes exceto 1, são atualmente desenvolvedores Android profissionais, ou seja, atuam profissionalmente com a plataforma Android. 70% responderam com seu email para receber resultados da pesquisa, tal como em *S1*, pode ser um indicativo de interesse legítimo da comunidade sobre esta temática. Questionamos da experiência com desenvolvimento de software, bem como com desenvolvimento Android, notamos que 55% relataram ter mais de 5 aplicativos publicados e 47% tinham mais de 4 anos de experiência com Android. A experiência dos desenvolvedores pode ser visualizada na Figura 4.1b.

Análise dos Dados

Nossa análise consistiu em investigar a percepção dos desenvolvedores sobre códigos limpos, códigos afetados pelas más práticas e códigos afetados por maus cheiros tradicionais, exemplo, Classe Deus/Longa ou Método Longo. Dividimos a análise da percepção em dois grupos: más práticas que afetam apenas códigos Java, no caso apenas a LCUI, e más práticas que afetam apenas recursos da aplicação, no caso LPA, NRD e RM.

Para o grupo de más práticas que afetam código Java, analisamos a percepção dos desenvolvedores a partir de três comparações: classes afetadas pela má prática vs. classes limpas, classes afetadas pela má prática vs. classes afetadas por maus cheiros tradicionais e por fim, classes afetadas por maus cheiros tradicionais vs. classes limpas. Para o grupo de más práticas que afetam apenas recursos da aplicação, analisamos a percepção dos desenvolvedores a partir da comparação entre códigos afetados pela má prática vs. códigos limpos.

Para comparar as distribuições da severidade indicada pelos participantes, utilizamos o teste de Mann-Whitney não pareado [47] para analisar a significância estatística das diferenças entre a severidade atribuída pelos participantes aos problemas que observam em códigos Android cheirosos e códigos limpos. Os resultados são considerados estatisticamente significativos em $\alpha \leq 0,05$. Também estimamos a magnitude das diferenças medidas usando o Delta de Cliff (ou d), uma medida de tamanho do efeito não paramétrico [22] para dados ordinais. Seguimos diretrizes bem estabelecidas para interpretar os valores do tamanho do efeito: insignificante para $|d| < 0,14$, baixo para $0,14 \leq |d| < 0,33$, médio para $0,33 \leq |d| < 0,474$, e alto para $|d| \geq 0,474$ [49]. Finalmente, relatamos achados qualitativos derivados

das respostas abertas dos participantes.

Capítulo 5

Maus Cheiros de Interface Android

Durante o processo de codificação emergiram 25 maus cheiros de interface Android dos quais 15 afetam classes Java (ACTIVITIES, FRAGMENTS, ADAPTERS ou LISTENERS) e 10 afetam recursos da aplicação (RECURSOS DE LAYOUT, RECURSOS DE TEXTO, RECURSOS DE ESTILO ou RECURSOS GRÁFICOS).

A Tabela ?? apresenta o total de ocorrências de cada mau cheiro. A última linha da tabela, **#Maus Cheiros**, apresenta quantos maus cheiros emergiram de cada questão, como cada questão está diretamente ligada a um elemento de interface Android, podemos interpretá-la da seguinte forma: *quais são os pontos de atenção a serem analisados em determinado elemento Android?* A última coluna da tabela, **#Elementos**, apresenta em quantas questões cada mau cheiro surgiu, podemos interpretá-la da seguinte forma: *com base no mau cheiro, quais elementos devem ser investigados?*

Esta seção está organizada em três subseções onde, nas duas primeiras, definimos as práticas de alta e média recorrência, *totalizando 19 más práticas*. Na última subseção apresentamos os resultados obtidos no estudo sobre a percepção de desenvolvedores com relação às más práticas de alta recorrência.

5.1 Maus Cheiros Em Código Java

5.1.1 CLASSE DE UI INTELIGENTE (SML-J1)^{20*}

ACTIVITIES^{10*}, FRAGMENTS^{6*}, ADAPTERS^{5*} e LISTENERS^{1*} devem conter apenas códigos responsáveis por apresentar, interagir e atualizar a UI. São indícios do mau cheiro a existência de códigos relacionados a lógica de negócio, operações de IO¹, conversão de dados ou campos estáticos nesses elementos.

¹ver mau cheiro CLASSES DE UI FAZENDO IO 5.1.8.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “Fazer lógica de negócio [em Activities]”² (P16). “Colocar regra de negócio no Adapter” (P19). “Manter lógica de negócio em Fragments” (P11). E frases sobre **boas prácticas**: “Elas [Activities] representam uma única tela e apenas interagem com a UI, qualquer lógica deve ser delegada para outra classe” (P16). “Apenas código relacionado à Interface de Usuário nas Activities” (P23). “Adapters devem apenas se preocupar sobre como mostrar os dados, sem trabalhá-los” (P40).

5.1.2 CLASSES DE UI ACOPLADAS (SML-J2)^{6*}

FRAGMENTS^{4*}, ADAPTERS^{1*} e LISTENERS^{1*} não devem ter referência direta para quem os utiliza. São indícios do mau cheiro a existência de referência direta para ACTIVITIES ou FRAGMENTS nesses elementos.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “Acoplar o fragment a activity ao invés de utilizar interfaces é uma prática ruim” (P19). “Acoplar o Fragment com a Activity” (P10, P31 e P45). “Fragments nunca devem tentar falar uns com os outros diretamente” (P37). “Integrar com outro Fragment diretamente” (P45). “[Listener] conter uma referência direta à Activities” (P4, P40). “[Adapters] alto acoplamento com a Activity” (P10). “Acessar Activities ou Fragments diretamente” (P45). E sobre **boa prática**: “Seja um componente de UI reutilizável. Então evite dependência de outros componentes da aplicação” (P6).

5.1.3 COMPORTAMENTO SUSPEITO (SML-J3)^{6*}

ACTIVITIES^{3*}, FRAGMENTS^{2*} e ADAPTERS^{1*} não devem ser responsáveis pela implementação do comportamento dos eventos. São indícios do mau cheiro o uso de classes anônimas, classes internas ou polimorfismo (através de implements) para implementar LISTENERS de modo a responder a eventos do usuário.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “Usar muitos anônimos pode ser complicado. Às vezes nomear coisas torna mais fácil para depuração” (P9). “Mantenha-os [Listeners] em classes separadas (esqueça sobre classes anônimas)” (P4). “Muitas implementações de Listener com classes anônimas” (P8). “Declarar como classe interna da Activity ou Fragment ou outro componente que contém um ciclo de vida. Isso pode fazer com que os aplicativos causem vazamentos de memória.” (P42). “Eu não gosto quando os desenvolvedores fazem a activity implementar o Listener porque eles [os métodos] serão expostos e qualquer um pode chamá-lo de fora da classe. Eu prefiro instanciar ou então usar ButterKnife para injetar cliques.” (P44). E sobre **boas prácticas**: “Prefiro

²Todo texto em inglês foi traduzido livremente ao longo da dissertação

declarar os listeners com implements e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto” (P32). “Tome cuidado se a Activity/Fragment é um Listener uma vez que eles são destruídos quando as configurações mudam. Isso causa vazamentos de memória.” (P6). “Use carregamento automático de view como ButterKnife e injeção de dependência como Dagger2” (P10).

5.1.4 ENTENDA O CICLO DE VIDA (SML-J4)^{5*}

O Ciclo de Vida de ACTIVITIES^{3*} e FRAGMENTS^{3*} é bem delicado e elaborado, logo o uso dele exige um conhecimento mais profundo, caso contrário pode resultar em *memory leaks* e outros problemas. São indícios do mau cheiro ter estes elementos como *callbacks* de processos assíncronos, efetivar a transação de FRAGMENTS (através do `FragmentManager.commit()`) após o `onPause` da ACTIVITY ou o não tratamento da restauração do estado de ACTIVITIES e FRAGMENTS após por exemplo, rotação da tela.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: “Não conhecer o enorme e complexo ciclo de vida de Fragment e não lidar com a restauração do estado” (P42). “Não commitar fragmentos após o `onPause` e aprender o ciclo de vida se você quiser usá-los” (P31). “Fazer Activities serem callbacks de processos assíncronos gerando *memory leaks*. Erros ao interpretar o ciclo de vida” (P28).

5.1.5 ADAPTER CONSUMISTA (SML-J5)^{5*}

São indícios do mau cheiro quando ADAPTERS^{5*} não reutilizam instâncias das *views* que representam os campos a serem populados para cada item da coleção através do padrão *View Holder* ou quando os mesmos possuem classes internas para reaproveitamento das *views* porém não são estáticas.

Alguns exemplos de frases sobre **boas práticas** que embasaram esse mau cheiro são: “Reutilizar a view utilizando *ViewHolder*.” (P36). “Usar o padrão *ViewHolder*” (P39). P45 sugere o uso do *RecyclerView*, um elemento Android para a construção de listas que já implementa o padrão *ViewHolder* [43].

5.1.6 USO EXCESSIVO DE FRAGMENT (SML-J6)^{3*}

FRAGMENTS^{3*} devem ser evitados. São indícios do mau cheiro quando o aplicativo não é utilizado em Tablets ou não possuem *VIEWPAGERS* e ainda assim faz o uso de FRAGMENTS ou quando existem FRAGMENTS no projeto que não são utilizados em mais de uma tela do aplicativo.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: “Usar muitos

Fragments é uma má prática” (P2). E frases sobre **boas práticas**: *“Evite-os. Use apenas com View Pagers”* (P7). *“Eu tento usar o Fragment para lidar apenas com as visualizações, como a Activity, e eu o uso apenas quando preciso deles em um layout de Tablet ou para reutilizar em outra Activity. Caso contrário, eu não uso”* (P41).

5.1.7 NÃO USO DE FRAGMENT (SML-J7)^{3*}

FRAGMENTS^{2*} devem ser usados sempre que possível em conjunto com ACTIVITIES^{2*}. É indício do mau cheiro a não existência de FRAGMENTS na aplicação ou o uso de EDITTEXTS, SPINNERS ou outras *views* por ACTIVITIES.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Não usar Fragments”* (P22). *“Usar todas as view (EditTexts, Spinners, etc...) dentro de Activities e não dentro de Fragments”* (P45). E sobre **boas práticas**: *“Utilizar fragments sempre que possível.”* (P19), *“Use um Fragment para cada tela. Uma Activity para cada aplicativo.”* (P45).

5.1.8 CLASSES DE UI FAZENDO IO (SML-J8)^{3*}

ACTIVITIES^{2*}, FRAGMENTS^{1*} e ADAPTERS^{1*} não devem ser responsáveis por operações de IO. São indícios do mau cheiro implementações de acesso a banco de dados ou internet a partir desses elementos.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“[Activities e Fragments] fazerem requests e consultas a banco de dados”* (P26). *“[Adapters] fazerem operações longas e requests de internet”* (P26). E sobre **boa prática**: *“Elas [Activities] nunca devem fazer acesso a dados”* (P37).

5.1.9 ACTIVITY INEXISTENTE (SML-J9)^{2*}

ACTIVITIES^{2*} podem deixar de existir a qualquer momento, tenha cuidado ao referenciá-las. São indícios do mau cheiro a existência de referências estáticas a ACTIVITIES ou classes internas a ela ou referências não estáticas por objetos que tenham o ciclo de vida independente dela.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Fazer Activities serem callbacks de processos assíncronos gerando memory leaks. Erros ao interpretar o ciclo de vida”* (P28). *“Ter referência estática para Activities, resultando em vazamento de memória”* (P31). E sobre **boas práticas**: *“Não manter referências estáticas para Activities (ou classes anônimas criadas dentro delas)”* (P31). *“Deus mata um cachorro toda vez que alguém passa o contexto da Activity para um componente que tem um ciclo de*

vida independente dela. Vaza memória e deixa todos tristes.” (P4).

5.1.10 ARQUITETURA NÃO IDENTIFICADA (SML-J10)^{2*}

São indícios do mau cheiro quando diferentes ACTIVITIES^{2*} e FRAGMENTS^{1*} no projeto apresentam fluxos de código complexos, possivelmente são CLASSE DE UI INTELIGENTE 5.1.1, onde não é possível identificar uma organização padronizada entre eles que aponte para algum padrão arquitetural, como por exemplo, MVC (*Model View Controller*), MVP (*Model View Presenter*), MVVM (*Model View ViewModel*) ou *Clean Architecture*.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: “*Não usar um design pattern*” (P45). E frases sobre **boas práticas**: “*Usar algum modelo de arquitetura para garantir apresentação desacoplada do framework (MVP, MVVM, Clean Architecture, etc)*” (P28). “*Sobre MVP. Eu acho que é o melhor padrão de projeto para usar com Android*” (P45).

5.1.11 ADAPTER COMPLEXO (SML-J11)^{2*}

ADAPTERS^{2*} devem ser responsáveis por popular uma view a partir de um único objeto, sem realizar lógicas ou tomadas de decisão. São indícios desse mau cheiro quando ADAPTERS contém muitos condicionais (if ou switch) ou cálculos no método getView, responsável pela construção e preenchimento da view.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: “*Reutilizar um mesmo adapter para várias situações diferentes, com ifs ou switches. Código de lógica importante ou cálculos em Adapters.*” (P23). E sobre **boa prática**: “*Um Adapter deve adaptar um único tipo de item ou delegar a Adapters especializados*” (P2).

5.1.12 FRAGMENT ANINHADO (SML-J12)^{1*}

O FRAGMENT^{1*} deve representar a menor unidade de tela. São indícios do mau cheiro quando um RECURSO DE LAYOUT associado a algum FRAGMENT contém a tag <Fragment> ou quando o código do Fragment faz uso da classe FragmentTransaction para colocar outro Fragment em seu RECURSO DE LAYOUT.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: “*De preferência, eles não devem ser aninhados*” (P37). “*Fragments aninhados!*” (P4).

5.1.13 CONTROLE MANUAL DA PILHA DE ACTIVITIES (SML-J13)^{1*}

São indícios do mau cheiro a implementação do método `onBackPressed` de `ACTIVITIES`^{1*} ou códigos que tenham o intuito de controlar a pilha de `ACTIVITIES` do Android.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “*Sobreescrever o comportamento do botão voltar*” (P43). “*Lidar com a pilha do app manualmente*” (P41).

5.1.14 ESTRUTURA DE PACOTES (SML-J14)^{1*}

É indício do mau cheiro quando `ACTIVITIES`^{1*} estão em pacotes que não `activity` ou `view`.

Alguns exemplos de frases sobre **boas prácticas** que embasaram esse mau cheiro são: “*Apenas separe estes arquivos no diretório view no padrão MVC*” (P12). “*Eu sempre coloco minhas Activities em um pacote chamado activities*” (P11).

5.2 Maus Cheiros Em Recursos

5.2.1 RECURSO MÁGICO (SML-R1)^{8*}

Todo recurso de cor, tamanho, texto ou estilo deve ser criado em seu respectivo arquivo e então ser usado. São indícios do mau cheiro quando `RECURSOS DE LAYOUT`^{2*}, `RECURSOS DE TEXTO`^{5*} ou `RECURSOS DE ESTILO`^{1*} usam alguma dessas informações diretamente no código ao invés de fazer referência para um recurso.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “*Strings diretamente no código*” (P23). “*Não extrair as strings e sobre não extrair os valores dos arquivos de layout*” (P31 e P35). E sobre **boas prácticas**: “*Sempre pegar valores de string ou dp de seus respectivos resources para facilitar*” (P7). “*Sempre adicionar as strings em resources para traduzir em diversos idiomas*” (P36).

5.2.2 NOME DE RECURSO DESPADRONIZADO (SML-R2)^{8*}

São indícios do mau cheiro quando `RECURSOS DE LAYOUT`^{2*}, `RECURSOS DE TEXTO`^{4*}, `RECURSOS DE ESTILO`^{2*} e `RECURSOS GRÁFICOS`^{1*} não possuem um padrão de nomenclatura.

Alguns exemplos de frases sobre **más prácticas** que embasaram esse mau cheiro são: “*O nome das strings sem um contexto*” (P8). “[*Sobre Style Resources*] Nada além de ter uma boa

convenção de nomes” (P37). “[*Sobre Layout Resource*] Mantenha uma convenção de nomes da sua escolha” (P37). E sobre **boas práticas**: “Iniciar o nome de uma string com o nome da tela onde vai ser usada” (P27). “[*Sobre Layout Resource*] Ter uma boa convenção de nomeação” (P43). “[*Sobre Style Resource*] colocar um bom nome” (P11).

5.2.3 LAYOUT PROFUNDAMENTE ANINHADO (SML-R3)^{7*}

São indícios desse mau cheiro o uso de profundos aninhamentos na construção de RECURSOS DE LAYOUT^{7*}, ou seja, ViewGroups contendo outros ViewGroups sucessivas vezes. O site oficial do Android conta com informações e ferramentas automatizadas para lidar com esse sintoma [41].

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: “Hierarquia de views longas” (P26). “Estruturas profundamente aninhadas” (P4). “Hierarquias desnecessárias” (P39). “Criar muitos ViewGroups dentro de ViewGroups” (P45). E sobre **boas práticas**: “Tento usar o mínimo de layout aninhado” (P4). “Utilizar o mínimo de camadas possível” (P19). “Não fazer uma hierarquia profunda de ViewGroups” (P8).

5.2.4 IMAGEM DISPENSÁVEL (SML-R4)^{6*}

O Android possui diversos tipos de RECURSOS GRÁFICOS^{6*} que podem substituir imagens tradicionais como .png, .jpg ou .gif a um custo menor em termos de tamanho do arquivo e sem a necessidade de haver versões de diferentes tamanhos/resoluções. São indícios do mau cheiro a existência de imagens com, por exemplo, cores sólidas, degradês ou estado de botões, que poderiam ser substituídas por RECURSOS GRÁFICOS de outros tipos como SHAPES, STATE LISTS ou NINE-PATCH FILE ou a não existência de imagens vetoriais, que podem ser redimensionadas sem a perda de qualidade.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: “Uso de formatos não otimizados, uso de drawables onde recursos padrão do Android seriam preferíveis” (P23). “Usar jpg ou png para formas simples é ruim, apenas as desenhe [através de Drawable Resources]” (P37). E sobre **boas práticas**: “Quando possível, criar resources através de xml” (P36). “Utilizar o máximo de Vector Drawables que for possível” (P28). “Evite muitas imagens, use imagens vetoriais sempre que possível” (P40).

5.2.5 LAYOUT LONGO OU REPETIDO (SML-R5)^{5*}

São indícios do mau cheiro quando RECURSOS DE LAYOUT^{5*} é muito grande ou possui trechos de layout muito semelhantes ou iguais a outras telas.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: “Copiar e colar

layouts parecidos sem usar includes” (P41). *“Colocar muitos recursos no mesmo arquivo de layout.”* (P23). E sobre **boas práticas**: *“Sempre quando posso, estou utilizando includes para algum pedaço de layout semelhante”* (P32). *“Criar layouts que possam ser reutilizados em diversas partes”* (P36). *“Separe um grande layout usando include ou merge”* (P42)

5.2.6 IMAGEM FALTANTE (SML-R6)^{4*}

As imagens devem ser disponibilizadas em mais de um tamanho/resolução para que o Android possa realizar otimizações. São indícios do mau cheiro haver apenas uma versão de algum RECURSOS GRÁFICO^{4*} do tipo png, jpg ou gif ou ainda, ter imagens em diretórios incorretos em termos de dpi.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Ter apenas uma imagem para multiplas densidades”* (P31). *“Baixar uma imagem muito grande quando não é necessário. Há melhores formas de usar memória”* (P4). *“Não criar imagens para todas as resoluções”* (P44). E sobre **boas prática**: *“Nada especial, apenas mantê-las em seus respectivos diretórios e ter variados tamanhos delas”* (P34). *“Criar as pastas para diversas resoluções e colocar as imagens corretas”* (P36).

5.2.7 LONGO RECURSO DE ESTILO (SML-R7)^{3*}

É indício do mau cheiro haver apenas um RECURSO DE ESTILO^{4*} ou conter Recursos de Estilo muito longos.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Deixar tudo no mesmo arquivo styles.xml”* (P28). *“Arquivos de estilos grandes”* (P8). E sobre **boas práticas**: *“Se possível, separar mais além do arquivo styles.xml padrão, já que é possível declarar múltiplos arquivos XML de estilo para a mesma configuração”* (P28). *“Divida-os. Temas e estilos é uma escolha racional”* (P40).

5.2.8 RECURSO DE STRING BAGUNÇADO (SML-R8)^{3*}

É indício do mau cheiro o uso de apenas um arquivo para todos os RECURSOS DE TEXTO^{3*} do aplicativo e a não existência de um padrão de nomenclatura e separação para os RECURSOS DE TEXTO de uma mesma tela.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Usar o mesmo arquivo strings.xml para tudo”* (P28). *“Não orgaizar as strings quando o strings.xml começa a ficar grande”* (P42). E sobre **boas práticas**: *“Separar strings por tela em arquivos XML separados. Extremamente útil para identificar quais strings pertencentes a quais telas em projetos grandes”* (P28). *“Sempre busco separar em blocos, cada bloco representa uma*

Activity e nunca aproveito uma String pra outra tela” (P32).

5.2.9 ATRIBUTOS DE ESTILO REPETIDOS (SML-R9)^{3*}

É indício do mau cheiro haver RECURSOS DE LAYOUT^{1*} ou RECURSOS DE ESTILO^{2*} com blocos de atributos de estilo repetidos.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: *“Utilizar muitas propriedades em um único componente. Se tiver que usar muitas, prefiro colocar no arquivo de styles.” (P32).* E sobre **boa prática**: *“Sempre que eu noto que tenho mais de um recurso usando o mesmo estilo, eu tento movê-lo para o meu style resource.” (P34).*

5.2.10 REÚSO INADEQUADO DE STRING (SML-R10)^{2*}

Cada tela deve ter seu conjunto de RECURSOS DE TEXTO^{2*}. É indício do mau cheiro reutilizar o mesmo RECURSO DE TEXTO em diferentes telas do aplicativo apenas porque o texto coincide.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Utilizar uma String pra mais de uma activity, pois se em algum momento, surja a necessidade de trocar em uma, vai afetar outra.” (P32).* *“Reutilizar a string em várias telas” (P6)* *“Reutilizar a string apenas porque o texto coincide, tenha cuidado com a semântica” (P40).* E sobre **boas práticas**: *“Sempre busco separar em blocos, cada bloco representa uma activity e nunca aproveito uma String pra outra tela.” (P32).* *“Não tenha medo de repetir strings” (P9).*

5.2.11 LISTENER ESCONDIDO (SML-R11)^{2*}

RECURSOS DE LAYOUT^{2*} devem ser responsáveis apenas por apresentar informações. É indício do mau cheiro o uso de atributos diretamente em RECURSOS DE LAYOUT, como por exemplo o atributo `onClick`, para configurar o LISTENER que responderá ao evento.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *“Nunca crie um listener dentro do XML. Isso esconde o listener de outros desenvolvedores e pode causar problemas até que ele seja encontrado” (P34, P39 e P41).* E sobre **boa prática**: *“XML de layout deve lidar apenas com a view e não com ações” (P41).*

Capítulo 6

Percepção dos Desenvolvedores

6.1 Percepção dos Desenvolvedores

A Figura 6.1 apresenta gráficos de violino sobre a percepção dos desenvolvedores com relação as quatro más práticas de alta recorrência (LCUI, LPA, RM e NRD) e um gráfico de violino com as 3 más práticas que afetam apenas recursos do aplicativo. No eixo y, 0 (zero) indica códigos não percebidos pelos desenvolvedores como problemáticos (ou seja, responder *não* à pergunta: este código apresenta algum problema de design e/ou implementação?), enquanto que valores de 1 a 5 indicam o nível de severidade para o problema percebido pelo desenvolvedor.

6.1.1 Más práticas que afetam classes Java

Na Figura 6.1a apresentamos três gráficos violinos, respectivamente: percepção dos desenvolvedores sobre códigos afetados pela má prática LCUI, a percepção sobre códigos limpos e por último, a percepção sobre códigos afetados por maus cheiros tradicionais como por exemplo Classe Longa.

A mediana de classes limpas tem severidade igual a 0 ($Q3=0$). Isso indica que, como esperado, desenvolvedores não percebem essas classes como problemáticas. Em comparação, as classes afetadas por LCUI tem mediana igual a 2 ($Q3=4$) logo, são percebidas como

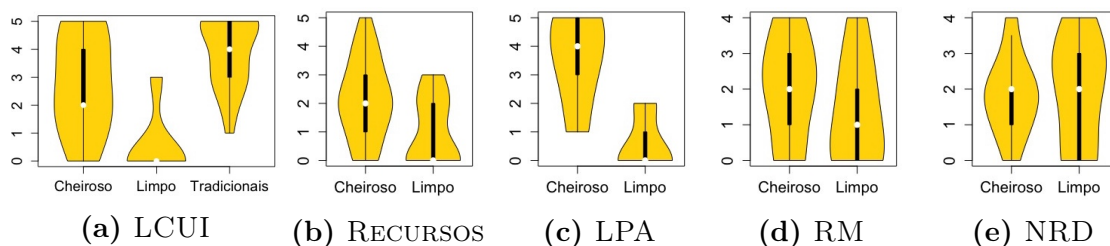


Figura 6.1: Gráficos violino das más práticas LCUI, LPA, RM e NRD.

classes problemáticas. A diferença entre classes afetadas por LCUI e classes limpas é estatisticamente significativa ($p\text{-value} < 0.004$) com alto tamanho de efeito ($d = 0,72$). Com relação aos maus cheiros tradicionais, a mediana de severidade é igual a 4 ($Q3=5$). Isso significa que classes afetadas por esses maus cheiros são percebidas pelos desenvolvedores como muito problemáticas, até mais que classes afetadas pela má prática LCUI. Ainda que essa diferença de percepção esteja clara no gráfico violino, esta diferença não é estatisticamente significativa ($p\text{-value} = 0.077$). Acreditamos que isso ocorra devido ao número limitado de dados (20 participantes).

6.1.2 Más práticas que afetam recursos

As Figuras 6.1b até 6.1e reúnem 4 diferentes pares de gráficos violinos. O primeiro compara recursos afetados por quaisquer das 3 más práticas, LPA, RM e NRD, com recursos limpos. O segundo, terceiro e quarto, tratam de cada má prática individualmente, ou seja, recursos afetados por aquela má prática em comparação com recursos limpos.

A Figura 6.1b mostra a percepção dos desenvolvedores com relação a recursos afetados pelas más práticas LPA, RND e RM, com mediana de severidade igual a 2 ($Q3=3$), em comparação com recursos limpos (mediana=0). Isso indica que, como esperado, desenvolvedores percebem recursos afetados pelas más práticas como problemáticos. Essa diferença também é estatisticamente significativa ($p\text{-value} < 0.008$) com médio tamanho de efeito ($d = 0.43$).

Ao avaliarmos os gráficos violinos das más práticas individualmente, podemos notar que duas, LPA (Figura 6.1c) e RM (Figura 6.1d), se mostram percebidas como problemáticas, sendo a primeira mais percebida do que a segunda. Códigos afetados pela má prática LPA tem mediana de severidade igual a 4 ($Q3=5$) logo, desenvolvedores percebem códigos afetados por ela como problemáticos. Em comparação, códigos limpos apresentam mediana igual a 0 ($Q3=1$). A diferença entre códigos afetados por LPA e códigos limpos também é estatisticamente significativa ($p\text{-value} < 0.02$) com alto tamanho de efeito ($d = 0,89$). Em contrapartida, ainda que o gráfico de violino da má prática RM apresente também uma diferença visual entre códigos limpos e afetados pela má prática, essa diferença não é estatisticamente significativa ($p\text{-value} = 0,34$).

A má prática NRD (Figura 6.1e) foi a menos percebida por desenvolvedores, com medianas iguais para códigos afetados por ela e códigos limpos (mediana = 2). Entretanto, os desenvolvedores que indicaram códigos afetados por ela como problemáticos, indicaram como o problema descrições muito próximas a definição dada para essa má prática. Por exemplo, S2P15 disse “*Os nomes das strings são formados por um prefixo e um número, o que prejudica a legibilidade, é impossível saber o que este número indica*” (pontuou severidade como 3), S2P16 disse “*Atributos não seguem convenção de nomes. Nomes não são descritivos*” (pontuou severidade como 2), S2P11 disse “*Não segue uma boa prática de no-*

menclatura de recursos.” (pontuou severidade como 2) e S2P19 disse “*Os nomes das strings não estão seguindo um padrão (algumas em camelCase, outras lowercase, outras snakecase) [...]*” (pontuou severidade como 2).

De forma geral, os desenvolvedores conseguiram identificar corretamente a má prática em questão, colocando em suas respostas descrições muito próximas as definições dadas a elas. Por exemplo S2P11 ao confrontar um código afetado por LCUI disse “[...] *Não há nenhuma arquitetura implementada, o que causa a classe fazendo muito mais do que é de sua alçada. O método onItemClick está muito complexo, contendo 7 condições [...]*”, S2P5 ao confrontar um código afetado por RM disse “*Valores de cores, tamanhos, animações e distancias, nao estao extraídos fazendo que muitos deles estejam repetidos dificultando uma posterior manutenção ou reusabilidade*”, S2P7 ao confrontar um código afetado por LPA disse “*Os sucessivos aninhamentos de view groups provavelmente irá causar uma performance ruim*”.

Capítulo 7

Discussão

7.1 Propostas de soluções

Notamos que muitas vezes as respostas para as questões sobre sobre boas práticas apresentada no 1o questionário, sobre boas e más práticas em elementos Android, vieram na forma de sugestões de como solucionar o que o participante indicou como má prática para aquele elemento. Como não foi o foco desta pesquisa validar se a sugestões dadas como solução ao mau cheiro de fato se aplica, não exploramos a fundo estas informações. Entretanto, disponibilizamos uma tabela que indica a boa prática sugerida para cada mau cheiro definido no apêndice [B](#).

Capítulo 8

Ameaças à Validade

Uma limitação deste artigo é que os dados foram coletados apenas a partir de questionários online e o processo de codificação foi realizado apenas por um dos autores. Alternativas a esses cenários seriam realizar a coleta de dados de outras formas como entrevistas ou consulta a especialistas, e que o processo de codificação fosse feito por mais de um autor de forma a reduzir possíveis enviesamentos.

Outra possível ameaça é com relação a seleção de códigos limpos. Selecionar códigos limpos é difícil. Sentimos uma dificuldade maior ao selecionar códigos de recursos do Android pois quando achamos que isolamos um problema, um participante mencionava sobre outro o qual não havíamos removido. Um alternativa seria investigar a existência de ferramentas que façam esta seleção, validar os códigos selecionados com um especialista ou mesmo estender o teste piloto.

Nossa pesquisa tenta replicar o método utilizado por Aniche [9] ao investigar cheiros de código no framework Spring MVC. Entretanto, nos deparamos com situações diferentes, das quais, após a execução nos questionamos se aquele método seria o mais adequado para todos os contextos neste artigo. Por exemplo, nosso resultado com a má prática RM nos levou a conjecturar se desenvolvedores consideram problemas em códigos Java mais severos que problemas em recursos do aplicativo. O que nos levou a pensar sobre isso foi que, apesar do resultado, obtivemos muitas respostas que se aproximavam da definição da má prática RM. Desta forma, levantamos que de todos os recursos avaliados, 74% receberam severidade igual ou inferior a 3, contra apenas 30% com os mesmo níveis de severidade em código Java. Desta forma, uma alternativa é repensar a forma de avaliar a percepção dos desenvolvedores sobre más práticas que afetem recursos do aplicativo.

Capítulo 9

Conclusão

Neste artigo investigamos a existência de boas e más práticas em elementos usados para implementação de *front-end* de projetos Android: ACTIVITIES, FRAGMENTS, LISTENERS, ADAPTERS, LAYOUT, STYLES, STRING e DRAWABLE. Fizemos isso através de um estudo exploratório qualitativo onde coletamos dados por meio de um questionário online respondido por 45 desenvolvedores Android. A partir deste questionário mapeamos 23 más práticas e sugestões de solução, quando mencionado por algum participante. Após, validamos a percepção de desenvolvedores Android sobre as quatro mais recorrentes dessas más práticas: LÓGICA EM CLASSES DE UI, NOME DE RECURSO DESPADRONIZADO, RECURSO MÁGICO E LAYOUT PROFUNDAMENTE ANINHADO. Fizemos isso através de um experimento online respondido por 20 desenvolvedores Android, onde os participantes eram convidados a avaliar 6 códigos com relação a qualidade.

QP1. O que desenvolvedores consideram boas e más práticas no desenvolvimento Android?

Questionamos 45 desenvolvedores sobre o que eles consideravam boas e más práticas em elementos específicos do Android. Com base nesses dados, consolidamos um catálogo com 23 más práticas onde, para cada uma delas apresentamos uma descrição textual e exemplos de frases usadas nas respostas que nos levaram a sua definição.

QP2. Códigos afetados por estas más práticas são percebidos pelos desenvolvedores como problemáticos?

Validamos a percepção de desenvolvedores sobre as quatro más prática mais recorrentes. Concluímos que desenvolvedores de fato as percebem como más práticas. Duas das más práticas, LÓGICA EM CLASSES DE UI e LAYOUT PROFUNDAMENTE ANINHADO foram possíveis confirmar com dados estatísticos. Outras duas, NOME DE RECURSO DESPADRONIZADO e RECURSO MÁGICO, embora os dados estatísticos não tenham confirmado, notamos por meio

das respostas abertas que existe esta percepção.

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 through 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.

Apêndice B

Exemplos de Respostas que Embasaram os Mau Cheiros

Mau Cheiro	Respostas sobre boas e más práticas
SML-J1	Más práticas: “Fazer lógica de negócio [em Activities]” ¹ (P16). “Colocar regra de negócio no Adapter” (P19). “Manter lógica de negócio em Fragments” (P11). Boas práticas: “Elas [Activities] representam uma única tela e apenas interagem com a UI, qualquer lógica deve ser delegada para outra classe” (P16). “Apenas código relacionado à Interface de Usuário nas Activities” (P23). “Adapters devem apenas se preocupar sobre como mostrar os dados, sem trabalhá-los” (P40).
SML-J2	Más práticas: “Acoplar o fragment a activity ao invés de utilizar interfaces é uma prática ruim” (P19). “Acoplar o Fragment com a Activity” (P10, P31 e P45). “Fragments nunca devem tentar falar uns com os outros diretamente” (P37). “Integrar com outro Fragment diretamente” (P45). “[Listener] conter uma referência direta à Activities” (P4, P40). “[Adapters] alto acoplamento com a Activity” (P10). “Acessar Activities ou Fragments diretamente” (P45). Boa prática: “Seja um componente de UI reutilizável. Então evite dependência de outros componentes da aplicação” (P6).
SML-J3	Más práticas: “Usar muitos anônimos pode ser complicado. Às vezes nomear coisas torna mais fácil para depuração” (P9). “Mantenha-os [Listeners] em classes separadas (esqueça sobre classes anônimas)” (P4). “Muitas implementações de Listener com classes anônimas” (P8). “Declarar como classe interna da Activity ou Fragment ou outro componente que contém um ciclo de vida. Isso pode fazer com que os aplicativos causem vazamentos de memória.” (P42). “Eu não gosto quando os desenvolvedores fazem a activity implementar o Listener porque eles [os métodos] serão expostos e qualquer um pode chamá-lo de fora da classe. Eu prefiro instanciar ou então usar ButterKnife para injetar cliques.” (P44). Boas práticas: “Prefiro declarar os listeners com implements e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto” (P32). “Tome cuidado se a Activity/Fragment é um Listener uma vez que eles são destruídos quando as configurações mudam. Isso causa vazamentos de memória.” (P6). “Use carregamento automático de view como ButterKnife e injeção de dependência como Dagger2” (P10).

¹Todo texto em inglês foi traduzido livremente ao longo da dissertação

Mau Cheiro	Respostas sobre boas e más práticas
SML-J4	Más prácticas: “Não conhecer o enorme e complexo ciclo de vida de Fragment e não lidar com a restauração do estado” (P42). “Não commitar fragmentos após o onPause e aprender o ciclo de vida se você quiser usá-los” (P31). “Fazer Activities serem callbacks de processos assíncronos gerando memory leaks. Erros ao interpretar o ciclo de vida” (P28).
SML-J5	Boas prácticas: “Reutilizar a view utilizando ViewHolder.” (P36). “Usar o padrão ViewHolder” (P39). P45 sugere o uso do RecyclerView, um elemento Android para a construção de listas que já implementa o padrão ViewHolder [43].
SML-J6	Má prática: “Usar muitos Fragments é uma má prática” (P2). Boas prácticas: “Evite-os. Use apenas com View Pagers” (P7). “Eu tento usar o Fragment para lidar apenas com as visualizações, como a Activity, e eu o uso apenas quando preciso deles em um layout de Tablet ou para reutilizar em outra Activity. Caso contrário, eu não uso” (P41).
SML-J7	Más prácticas: “Não usar Fragments” (P22). “Usar todas as view (EditTexts, Spinners, etc...) dentro de Activities e não dentro de Fragments” (P45). Boas prácticas: “Utilizar fragments sempre que possível.” (P19), “Use um Fragment para cada tela. Uma Activity para cada aplicativo.” (P45).
SML-J8	Más prácticas: “[Activities e Fragments] fazerem requests e consultas a banco de dados” (P26). “[Adapters] fazerem operações longas e requests de internet” (P26). Boa práctica: “Elas [Activities] nunca devem fazer acesso a dados” (P37).
SML-J9	Más prácticas: “Fazer Activities serem callbacks de processos assíncronos gerando memory leaks. Erros ao interpretar o ciclo de vida” (P28). “Ter referência estática para Activities, resultando em vazamento de memória” (P31). Boas prácticas: “Não manter referências estáticas para Activities (ou classes anônimas criadas dentro delas)” (P31). “Deus mata um cachorro toda vez que alguém passa o contexto da Activity para um componente que tem um ciclo de vida independente dela. Vaza memória e deixa todos tristes.” (P4).
SML-J10	Más prácticas: “Não usar um design pattern” (P45). Boas prácticas: “Usar algum modelo de arquitetura para garantir apresentação desacoplada do framework (MVP, MVVM, Clean Architecture, etc)” (P28). “Sobre MVP. Eu acho que é o melhor padrão de projeto para usar com Android” (P45).
SML-J11	Má práctica: “Reutilizar um mesmo adapter para várias situações diferentes, com ifs ou switches. Código de lógica importante ou cálculos em Adapters.” (P23). Boa práctica: “Um Adapter deve adaptar um único tipo de item ou delegar a Adapters especializados” (P2).
SML-J12	Más prácticas: “De preferência, eles não devem ser aninhados” (P37). “Fragments aninhados!” (P4).
SML-J13	Más prácticas: “Sobreescrever o comportamento do botão voltar” (P43). “Lidar com a pilha do app manualmente” (P41).
SML-J14	Boas prácticas: “Apenas separe estes arquivos no diretório "Visualizar" no padrão MVC” (P12). “I always put my activities in a package called activities” (P11).
SML-R1	Más prácticas: “Strings diretamente no código” (P23). “Não extrair as strings e sobre não extrair os valores dos arquivos de layout” (P31 e P35). Boas prácticas: “Sempre pegar valores de string ou dp de seus respectivos resources para facilitar” (P7). “Sempre adicionar as strings em resources para traduzir em diversos idiomas” (P36).

Mau Cheiro	Respostas sobre boas e más práticas
SML-R2	Más práticas: “O nome das strings sem um contexto” (P8). “[Sobre Style Resources] Nada além de ter uma boa convenção de nomes” (P37). “[Sobre Layout Resource] Mantenha uma convenção de nomes da sua escolha” (P37). Boas práticas: “Iniciar o nome de uma string com o nome da tela onde vai ser usada” (P27). “[Sobre Layout Resource] Ter uma boa convenção de nomeação” (P43). “[Sobre Style Resource] colocar um bom nome” (P11).
SML-R3	Más práticas: “Hierarquia de views longas” (P26). “Estruturas profundamente aninhadas” (P4). “Hierarquias desnecessárias” (P39). “Criar muitos ViewGroups dentro de ViewGroups” (P45). Boas práticas: “Tento usar o mínimo de layout aninhado” (P4). “Utilizar o mínimo de camadas possível” (P19). “Não fazer uma hierarquia profunda de ViewGroups” (P8).
SML-R4	Más práticas: “Uso de formatos não otimizados, uso de drawables onde recursos padrão do Android seriam preferíveis” (P23). “Usar jpg ou png para formas simples é ruim, apenas as desenhe [através de Drawable Resources]” (P37). Boas práticas: “Quando possível, criar resources através de xml” (P36). “Utilizar o máximo de Vector Drawables que for possível” (P28). “Evite muitas imagens, use imagens vetoriais sempre que possível” (P40).
SML-R5	Má prática: “Copiar e colar layouts parecidos sem usar includes” (P41). “Colocar muitos recursos no mesmo arquivo de layout.” (P23). Boas práticas: “Sempre quando posso, estou utilizando includes para algum pedaço de layout semelhante” (P32). “Criar layouts que possam ser reutilizados em diversas partes” (P36). “Separe um grande layout usando include ou merge” (P42)
SML-R6	Más práticas: “Ter apenas uma imagem para multiplas densidades” (P31). “Baixar uma imagem muito grande quando não é necessário. Há melhores formas de usar memória” (P4). “Não criar imagens para todas as resoluções” (P44). Boas prática: “Nada especial, apenas mantê-las em seus respectivos diretórios e ter variados tamanhos delas” (P34). “Criar as pastas para diversas resoluções e colocar as imagens corretas” (P36).
SML-R7	Más práticas: “Deixar tudo no mesmo arquivo styles.xml” (P28). “Arquivos de estilos grandes” (P8). Boas práticas: “Se possível, separar mais além do arquivo styles.xml padrão, já que é possível declarar múltiplos arquivos XML de estilo para a mesma configuração” (P28). “Divida-os. Temas e estilos é uma escolha racional” (P40).
SML-R8	Más práticas: “Usar o mesmo arquivo strings.xml para tudo” (P28). “Não orgaizar as strings quando o strings.xml começa a ficar grande” (P42). Boas práticas: “Separar strings por tela em arquivos XML separados. Extremamente útil para identificar quais strings pertencentes a quais telas em projetos grandes” (P28). “Sempre busco separar em blocos, cada bloco representa uma Activity e nunca aproveito uma String pra outra tela” (P32).
SML-R9	Má prática: “Utilizar muitas propriedades em um único componente. Se tiver que usar muitas, prefiro colocar no arquivo de styles.” (P32). Boa prática: “Sempre que eu noto que tenho mais de um recurso usando o mesmo estilo, eu tento movê-lo para o meu style resource.” (P34).
SML-R10	Más práticas: “Utilizar uma String pra mais de uma activity, pois se em algum momento, surja a necessidade de trocar em uma, vai afetar outra.” (P32). “Reutilizar a string em várias telas” (P6) “Reutilizar a string apenas porque o texto coincide, tenha cuidado com a semântica” (P40). Boas prática: “Sempre busco separar em blocos, cada bloco representa uma activity e nunca aproveito uma String pra outra tela.” (P32). “Não tenha medo de repetir strings [...]” (P9).

Mau Cheiro	Respostas sobre boas e más práticas
SML-R11	Más práticas: “Nunca crie um listener dentro do XML. Isso esconde o listener de outros desenvolvedores e pode causar problemas até que ele seja encontrado” (P34, P39 e P41). Boa prática: “XML de layout deve lidar apenas com a view e não com ações” (P41).

Referências Bibliográficas

- [1] Wikipedia android. [https://en.wikipedia.org/wiki/Android_\(operating_system\)](https://en.wikipedia.org/wiki/Android_(operating_system)). Acessado em 27/08/2017. 4
- [2] Android fragmentation visualized. <http://opensignal.com/reports/2015/08/android-fragmentation>, August 2015. Acessado em 12/09/2016. 9
- [3] Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016. <http://www.gartner.com/newsroom/id/3323017>, May 2016. Acessado em 23/07/2016. 2
- [4] 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. Acessado em 23/07/2016. 2
- [5] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. empirical software engineering. 2011. 23
- [6] Domenico Amalfitano, Anna Fasolino, Porfirio Tramontana, Salvatore Carmine, and Atif Memon. Using gui ripping for automated testing of android applications. 2012. 19
- [7] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. Satt: Tailoring code metric thresholds for different software architectures. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 41–50. IEEE, 2016. 2
- [8] Maurício Aniche, Bavota G., Treude C., Van Deursen A., and Gerosa M. A validated set of smells in model-view-controller architectures. 2016. 2, 18
- [9] Maurício Aniche and Marco Gerosa. Architectural roles in code metric assessment and code smell detection. 2016. 2, 42
- [10] Suelen G. Carvalho. ApÃndice online. <http://suelengc.com/android-code-smells-article/>, 2017. Acessado em 06/05/2017. 3
- [11] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. 2011. 19
- [12] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications Ltd, 3 edition, 2007. 21, 23, 25
- [13] Enck, William, and Patrick Drew McDaniel Machigar Ongtang. Understanding android security. 2009. 19

- [14] Enck, William, and Patrick McDaniel Machigar Ongtang. Mitigating android software misuse before it happens. 2008. 19
- [15] Zheran Fang and Yingjiu Li Weili Han. Permission based android security: Issues and countermeasures. 2014. 19
- [16] A. Milani Fard and A. Mesbah. Jsnose: Detecting javascript code smells. 2013. 2, 20
- [17] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. 2, 16, 17
- [18] Martin Fowler. Code smell. <http://martinfowler.com/bliki/CodeSmell.html>, February 2006. 16
- [19] Golnaz Gharachorlu. *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model*. PhD thesis, The University of British Columbia, 2014. 2, 18, 20
- [20] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1 edition, 1999. 23
- [21] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. 2012. 2, 19
- [22] Robert J Grissom and John J Kim. Effect sizes for research: Univariate and multivariate applications. page 272. Routledge, Mar 2005. 27
- [23] G. Hecht. An approach to detect android antipatterns. 2:766–768, May 2015. 20
- [24] Cuixiong Hu and Iulian Neamtii. Automating gui testing for android applications. 2011. 19
- [25] K Kavitha, P Salini, and V Ilamathy. Exploring the malicious android applications and reducing risk using static analysis. 2016. 19
- [26] Nigel King. In qualitative methods in organizational research - a practical guide. 1994. 22
- [27] Mario Linares-Vásquez, Sam Klock, Collin Mcmillan, Aminata Sabanè, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 2017. 2
- [28] Umme Mannan, Danny Dig, Iftexhar Ahmed, Carlos Jensen, Rana Abdullah, and M Al-murshed. Understanding code smells in android applications. 2017. 2, 20
- [29] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. 17, 18
- [30] Steve McConnell. *Code Complete*. Microsoft Press, Redmond, WA, USA, 2004. 15
- [31] Luciana Carla Lins Prates. *Aplicando Síntese Temática em Engenharia de Software*. PhD thesis, Universidade Federal da Bahia e Universidade Estadual de Feira de Santana, 2015. 22

- [32] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications Ltd, 2 edition, 2012. 25
- [33] Android Developer Site. Platform architecture. <https://developer.android.com/guide/platform/index.html>. Acessado em 04/09/2016. 4
- [34] Android Developer Site. Android studio. <https://developer.android.com/studio/index.html>, 2016. Acessado em 30/08/2016. 22
- [35] Android Developer Site. Building your first app. <https://developer.android.com/training/basics/firstapp/creating-project.html>, 2016. Acessado em 31/03/2017. 22
- [36] Android Developer Site. Layouts. <https://developer.android.com/guide/topics/ui/declaring-layout.html>, 2016. Acessado em 23/10/2016. 12
- [37] Android Developer Site. Resource type. <https://developer.android.com/guide/topics/resources/available-resources.html>, 2016. Acessado em 12/09/2016. 9
- [38] Android Developer Site. Resources overview. <https://developer.android.com/guide/topics/resources/overview.html>, 2016. Acessado em 08/09/2016. 22
- [39] Android Developer Site. Ui overview. <https://developer.android.com/guide/topics/ui/overview.html>, 2016. Acessado em 23/10/2016. 10
- [40] Android Developer Site. Android fundamentals. <https://developer.android.com/guide/components/fundamentals.html>, 2017. Acessado em 04/09/2016. 6, 9, 19
- [41] Android Developer Site. Optimizing view hierarchies). <https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html>), 2017. Acessado em 09/04/2017. 35
- [42] Android Developers Site. Activities. <https://developer.android.com/guide/components/activities.html>, 2016. Acessado em 29/08/2016. 20
- [43] Android Developers Site. Android recyclerview. <https://developer.android.com/training/material/lists-cards.html>, 2017. Acessado em 12/04/2017. 31, 49
- [44] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison Wesley, 2006. xi, 13, 14
- [45] G. Suryanarayana, G. Samarthayam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014. 17
- [46] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 2, 19
- [47] Conover J W. *Practical Nonparametric Statistics*. Wiley, 3 edition, Dec 1999. 27
- [48] Bruce Webster F. *Pitfalls of Object-Oriented Development*. M & T Books, 1995. 16
- [49] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. Efficient fingerprinting-based android device identification with zero-permission identifiers. 2016. 19

- [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. [19](#)
- [51] S. Yu. Big privacy: Challenges and opportunities of privacy study in the age of big data. 2016. [19](#)
- [52] Yuan Zhang, Min Yang, Zhemin Yang, Guofei GU, and Binyu Zang Peng Ning. Exploring permission induced risk in android-applications for malicious detection. 2004. [19](#)
- [53] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. Permission use analysis for vetting undesirable behaviors in android apps. 2014. [19](#)