

Anomalias na Camada de Apresentação de Aplicativos Android

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

São Paulo, 19 Janeiro de 2018

Anomalias na Camada de Apresentação de Aplicativos Android

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

Agradecimentos

(Só para a versão final.)

Resumo

CARVALHO, G. S. **Anomalias na Camada de Apresentação de Aplicativos**. 2018. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Estamos cientes de que bons códigos importam, mas como saber quando a qualidade está baixa? Maus cheiros de código auxiliam desenvolvedores na identificação de trechos de código problemáticos, porém a maioria dos maus cheiros catalogados se baseia em práticas e tecnologias tradicionais, criadas entre as décadas de 70 a 90, como orientação a objetos e Java. Ainda há dúvidas sobre maus cheiros em tecnologias que surgiram na última década, como o Android, principal plataforma móvel em 2017 com mais de 86% de participação de mercado. Alguns pesquisadores derivaram maus cheiros Android relacionados a eficiência e usabilidade. Outros notaram que maus cheiros específicos ao Android são muito mais frequentes nos aplicativos do que maus cheiros tradicionais. Diversas pesquisas concluíram que os componentes Android mais afetados por maus cheiros tradicionais pertencem à camada de apresentação, como *Activities* e *Adapters*. Notou-se também que em alguns aplicativos, códigos da camada de apresentação representam a maior parte do código do projeto. Vale ressaltar que a camada de apresentação Android também é composta por arquivos XML, chamados de recursos, usados na construção da interface do usuário (*User Interface - UI*), porém nenhuma das pesquisas citadas os considerou em suas análises. Nesta dissertação, investigamos a existência de maus cheiros relacionados à camada de apresentação Android considerando inclusive os recursos. Fizemos isso por meio de dois questionários online e três experimentos totalizando a participação de 3XX desenvolvedores. Nossos resultados mostram a existência de uma percepção comum entre desenvolvedores sobre más práticas no desenvolvimento da camada de apresentação Android. Nossas principais contribuições são um catálogo com 21 maus cheiros da camada de apresentação Android e uma análise estatística da percepção de desenvolvedores sobre os 8 principais maus cheiros catalogados. Nossas contribuições servirão a pesquisadores como ponto de partida para a definição de heurísticas e implementação de ferramentas automatizadas e a desenvolvedores como auxílio na identificação de códigos problemáticos, ainda que de forma manual.

Palavras-chave: engenharia de software, android, maus cheiros, qualidade de código, ma-

nutenção de software, anomalias de software.

Abstract

CARVALHO, G. S. **Anomalies in the Presentation Layer of Android Applications.** 2018. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

We are aware that good codes matter, but how do you know when quality is low? Code smells help us identify problematic code snippets, but most of the code smells cataloged are based on traditional practices and technologies, created from the 1970s through the 90s, such as object oriented and Java. There are still doubts about code smells in technologies that have emerged in the last decade, such as Android, the main mobile platform in 2017 with more than 86% market share. Some researchers have defined code smells related to Android efficiency and usability. Other research concludes that the components most affected by traditional code smells are related to the front-end, such as *Activities* and *Adapters*. Also noticed in some applications, front-end's codes represent the larger part of the project's code. It is noteworthy that the Android front-end is also composed of XML files, called application resources, used to build user interface (UI), but these files were not considered in their analyzes. In this dissertation, we investigate existence of code smells related to the Android front-end considering even application resources. To aim that performed two online surveys and three experiments summing 3XX developers. Our results show that there is a common perception among practicing Android developers about bad practices on Android front-end. Our main contributions are a catalog of 21 code smells about Android front-end and a statistical analysis of the perceptions of practitioners developers about the main 8 code smells cataloged. Our contributions will serve researchers as a starting point for the definition of heuristics and implementation of automated tools and to practitioners developers as an aid in identifying problematic codes, even manually.

Palavras-chave: software engineering, android, code smells, code quality, software maintenance, software anomalies.

Sumário

Lista de Abreviaturas	viii
Lista de Figuras	ix
Lista de Tabelas	x
1 Introdução	1
1.1 Questões de Pesquisa	3
1.2 Principais Contribuições	4
1.3 Organização do Trabalho	4
2 Fundamentação Conceitual	6
2.1 Android	6
2.1.1 Fundamentos do Desenvolvimento Android	7
2.1.2 Elementos da Camada de Apresentação Android	9
2.1.3 Desafios no Desenvolvimento da Camada de Apresentação Android . .	11
2.2 Qualidade de Software	12
2.3 Boas Práticas de Software	16
2.3.1 Padrões de Projeto	16
2.3.2 Anti-Padrões	18
2.4 Maus Cheiros de Código	18
2.4.1 Formato dos Maus Cheiros	19
3 Trabalhos Relacionados	22
3.1 Maus Cheiros Tradicionais	23

3.2	Projetos Android vs. projetos de software tradicionais	23
3.3	Maus cheiros específicos a uma tecnologia	23
3.4	Maus cheiros em aplicativos Android	25
3.4.1	Presença de maus cheiros tradicionais em aplicativos Android	25
3.4.2	Maus cheiros específicos ao Android	27
3.4.3	Presença de maus cheiros tradicionais vs. maus cheiros específicos em aplicativos Android	28
4	Metodologia de Pesquisa	30
4.1	Introdução	30
4.2	Etapa 1 - Boas e más práticas na camada de apresentação Android	32
4.2.1	Questionário	32
4.2.2	Participantes	33
4.2.3	Análise dos Dados	34
4.3	Etapa 2 - Frequência e importância dos maus cheiros	36
4.3.1	Questionário	36
4.3.2	Participantes	39
4.3.3	Análise dos Dados	41
4.4	Etapa 3 - Percepção dos maus cheiros	42
4.4.1	Experimento	42
4.4.2	Participantes	45
4.4.3	Análise dos Dados	45
4.5	Ameaças à Validade	45
4.5.1	Internas	45
4.5.2	Externas	46
5	Resultados	47
5.1	QP1: Existem maus cheiros que são específicos a camada de apresentação Android?	47
5.1.1	Resultados gerais e descobertas	47
5.1.2	Recorrência dos Maus Cheiros	49

5.1.3 Maus Cheiros Propostos	50
5.2 QP2. Com qual frequência os maus cheiros são percebidos e o quanto importante são considerados pelos desenvolvedores?	53
5.2.1 Resultados gerais	54
5.2.2 Importância dos Maus Cheiros	55
5.2.3 Frequência dos Maus Cheiros	56
5.3 QP3. Desenvolvedores Android percebem os códigos afetados pelos maus chei- ros como problemáticos?	57
6 Conclusão	58
6.1 Discussões	58
A Exemplos de respostas que embasaram os maus cheiros	60
B Questionário sobre boas e más práticas	64
C Questionário sobre frequência e importância dos maus cheiros	67
D Afirmações sobre frequência dos maus cheiros e respectivos dados estatís- ticos	72
E Afirmações sobre importância dos maus cheiros e respectivos dados esta- tísticos	75
F Gists dos Códigos Utilizados no Experimento da Etapa 3	78
Referências Bibliográficas	80

Lista de Abreviaturas

SDK *Software Development Kit*

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

LoC *Lines of Code*

UI *User Interface*

GoF *Gang of Four*

CSS *Cascading Style Sheets*

ORM *Object-Relational Mapping*

J2ME *Java Mobile Edition*

GT *Ground Theory*

MVC *Model View Controller*

MVP *Model View Presenter*

MVVM *Model View ViewModel*

SQuaRE *Systems and software Quality Requirements and Evaluation*

CISQ *Consortium for IT Software Quality*

ISO *International Organization for Standardization*

IEC *International Electrotechnical Commission*

SWEBOK *Software Engineering Body of Knowledge*

FURPS *Functionality Usability Reliability Performance Supportability*

Lista de Figuras

2.1	Participação de mercado global de sistemas operacionais móveis do Q1 (1º quadrimestre) de 2009 até o Q1 de 2017.	7
2.2	Comparação do ciclo de vida de <i>Activities</i> e <i>Services</i>	8
2.3	Comparativo do ciclo de vida de componentes Android que pertencem e não pertencem à camada de apresentação.	13
2.4	Características de Qualidade de Software segundo norma ISO/IEC 9126. . .	14
2.5	Formatos de padrões de acordo com seu nível de maturidade e clareza segundo Joshua Kerievsky.	17
4.1	Etapas da pesquisa.	31
4.2	Escolaridade e distribuição de idade dos participantes em S1.	33
4.3	Tempo de experiência com desenvolvimento de software e desenvolvimento Android dos participantes de S1.	34
4.4	Distribuição geográfica global e brasileira dos participantes de S1.	34
4.5	Escolaridade e distribuição de idade dos participantes em S2.	40
4.6	Experiência com desenvolvimento de software e desenvolvimento Android dos participantes de S2.	40
4.7	Nível de conhecimento em diversas linguagens de programação orientada a objetos dos participantes de S2.	40
4.8	Distribuição geográfica global e brasileira dos participantes de S2.	41
5.1	Total de respostas para cada pergunta sobre boas e más práticas nos oito elementos da camada de apresentação Android, apresentadas na segunda seção de <i>S1</i>	48
5.2	Distribuição relativa de importância dos 21 maus cheiros derivados.	56
5.3	Distribuição relativa de frequência dos 21 maus cheiros derivados.	57

Lista de Tabelas

2.1	Modelos de qualidade de software baseados no modelo de decomposição hierárquica de Boehm et al. e McCall et al.	13
4.1	Sete maus cheiros avaliados no experimento de código sobre a percepção de desenvolvedores Android.	43
4.2	Listagem dos nove projetos de software livre Android usados para coletar os códigos usados no experimento.	44
5.1	Maus cheiros em componentes da camada de apresentação Android, sua ocorrência em cada componente e ocorrência total.	49
5.2	Maus cheiros em recursos de aplicativos Android, sua ocorrência em cada componente e ocorrência total.	50
5.3	Maus cheiros em componentes da camada de apresentação Android e breve descrição dos sintomas.	51
5.4	Maus cheiros em recursos Android e breve descrição dos sintomas.	52
5.5	Média, moda e desvio padrão sobre a percepção da importância dos maus cheiros relacionados a componentes da camada de apresentação Android. . .	54
5.6	Listagem dos maus cheiros da camada de apresentação Android de acordo com seu nível de importância, alta ou moderada.	55
5.7	Listagem dos maus cheiros da camada de apresentação Android de acordo com seu nível de frequência, alta, moderada ou baixa.	56

Capítulo 1

Introdução

“Estamos cientes de que um bom código importa, pois já tivemos que lidar com a falta dele por muito tempo” argumenta Robert Martin [65]. De fato, estamos cientes de que bons códigos importam. Mas como saber se a qualidade de um código está baixa? Uma das formas de responder a essa pergunta é buscando por *maus cheiros* no código. Maus cheiros são certas estruturas no código que indicam a violação de princípios fundamentais de *design* e impactam negativamente a qualidade do projeto [79, p. 258]. Maus cheiros auxiliam desenvolvedores na identificação de trechos de códigos problemáticos, de forma que possam ser melhorados e a qualidade do software incrementada [22]. Nesta dissertação anomalias de código e maus cheiros são sinônimos.

Existem diversos maus cheiros catalogados, Método Longo e Classe Deus são dois exemplos [22, 65, 79, 84]. Muitos desses maus cheiros foram definidos baseados em conceitos e tecnologias tradicionais, como orientação a objetos e Java [58], que surgiram durante as décadas de 70 a 90. Nesta dissertação os denominamos de maus cheiros tradicionais. Entretanto, na última década surgiram muitas novas tecnologias, como por exemplo o Android, que levantaram questões como: “os maus cheiros tradicionais se aplicam às novas tecnologias?” ou “existem maus cheiros específicos às novas tecnologias ainda não catalogados?”. Questões como essas instigaram a curiosidade de diversos pesquisadores que decidiram investigar maus cheiros em tecnologias específicas, como por exemplo o CSS [24], o Javascript [20], o arcabouço Spring MVC [6] e fórmulas de planilhas do Google [73].

Android é uma plataforma móvel que foi lançada em 2008 pelo Google em parceria com diversas empresas [4]. Em 2011 se tornou mundialmente a principal plataforma móvel e desde então vem aumentando sua fatia de mercado, tendo em 2017 alcançado 86% [78].

O Android também chamou a atenção de pesquisadores da área de qualidade de software. Alguns investigaram a existência de maus cheiros tradicionais em aplicativos Android [54, 63, 82]. Outros investigaram a existência de maus cheiros específicos ao Android relacionados a eficiência (boa utilização de recursos como memória e processamento) e usabilidade

(capacidade do software em ser compreendido) [50, 75]. Outros pesquisadores focaram em entender características do desenvolvimento Android que os diferenciam do desenvolvimento de software tradicional [68].

Dentre as descobertas realizadas pelos pesquisadores, notou-se que maus cheiros específicos são muito mais frequentes em aplicativos Android do que maus cheiros tradicionais [54]. Os componentes Android mais afetados por maus cheiros tradicionais fazem parte da camada de apresentação, como *Activities* e *Adapters* [54, 68, 82], e em alguns aplicativos Android, códigos relacionados à camada de apresentação são maioria, em termos de linhas de código (*Lines of Code - LoC*) [68]. Vale ressaltar que à camada de apresentação Android também é composta por arquivos XML, chamados de recursos da aplicação, que são usados para a construção da interface com o usuário (*User Interface - UI*) [43]. Nenhuma das pesquisas mencionadas considerou esses arquivos em suas análises.

Nesta dissertação, investigamos a existência de maus cheiros de código relacionados à camada de apresentação Android. Outras pesquisas investigaram maus cheiros em termos de eficiência e usabilidade, diferente delas, nós buscamos por maus cheiros relacionados à manutenibilidade, que trata da facilidade do software de ser modificado ou aprimorado. Complementamos as pesquisas anteriores pois focamos na camada de apresentação Android considerando inclusive os recursos da aplicação.

Nossos dados foram obtidos por meio de dois questionários online. O primeiro foi um questionário exploratório onde perguntamos desenvolvedores Android sobre boas e más práticas utilizadas no dia a dia do desenvolvimento da camada de apresentação e obtivemos 45 respostas, das quais derivamos 21 maus cheiros. O segundo foi um questionário confirmatório, onde validamos a percepção com 201 desenvolvedores Android sobre a frequência e importância dos 21 maus cheiros derivados. Por último, realizamos um experimento de código presencial com 75 desenvolvedores Android com objetivo de validar a percepção dos maus cheiros no código. Ao todo, participaram da pesquisa 3XX **todo: ajustar número após experimento** desenvolvedores Android.

Nossos resultados mostram que existe uma percepção comum entre desenvolvedores sobre más práticas no desenvolvimento da camada de apresentação Android. Mostram também que essas más práticas são frequentes e consideradas importantes de se mitigar e são percebidas em códigos por desenvolvedores Android. Concluímos com: (i) um catálogo com 21 novos maus cheiros relacionados à camada de apresentação Android, (ii) uma análise estatística da percepção de desenvolvedores sobre 8 dos principais maus cheiros catalogados e (iii) um apêndice *online* [11] com as informações necessárias para outros pesquisadores replicarem nossa pesquisa.

Acreditamos que nossas contribuições dão um pequeno mas importante passo na busca por qualidade de software na plataforma Android e que poderá servir a pesquisadores e desenvolvedores. Aos pesquisadores serve como ponto de partida para a definição de heurís-

ticas de identificação dos maus cheiros e implementação de ferramentas que os identifiquem de forma automática. Aos desenvolvedores serve como auxílio na identificação de códigos problemáticos para serem melhorados, ainda que de forma manual.

1.1 Questões de Pesquisa

Maus cheiros são sintomas que podem indicar um problema mais profundo no código [22]. Geralmente são derivados da experiência e opinião de desenvolvedores [81] ou seja, são por natureza subjetivos [20]. Há ainda evidências na literatura que sugerem que maus cheiros são percebidos por desenvolvedores [72]. Desta forma, dividimos a pesquisa em três questões principais, que apresentamos a seguir.

QP1: Existem maus cheiros que são específicos à camada de apresentação de aplicativos Android?

Como principal questão de pesquisa, objetiva investigar a existência de maus cheiros no desenvolvimento da camada de apresentação Android. A percepção de desenvolvedores é sempre importante quando lidamos com manutenção de software [8, 72, 89]. Portanto, explorar o conhecimento empírico de desenvolvedores Android é relevante na busca por maus cheiros de código [20, 81].

Esta questão nos fornece as primeiras ideias sobre maus cheiros na camada de apresentação Android. Os dados foram coletados a partir de um questionário online submetido a desenvolvedores onde perguntamos sobre boas e más práticas no desenvolvimento da camada de apresentação Android.

Recebemos um total de 45 respostas onde realizamos um processo de codificação buscando por más práticas recorrentes de modo a ser a base para derivação dos maus cheiros. Esse processo resultou em 46 categorias das quais, 21 apresentaram-se recorrente o suficiente, com base no número de Nielsen [69], para a derivação dos maus cheiros. Como resultado, derivamos 21 maus cheiros na camada de apresentação Android.

QP2. Com qual frequência os maus cheiros são percebidos e o quão importante são considerados pelos desenvolvedores?

Com o objetivo de validar os maus cheiros derivados em QP1, por meio de um questionário online, perguntamos a desenvolvedores Android com qual frequência os maus cheiros são percebidos no seu dia a dia e o quão importante é mitigá-los. Obtivemos 201 respostas das quais validamos positivamente que 16 maus cheiros são considerados importantes e percebidos frequentemente no dia a dia. 5 maus cheiros foram descartados nesta etapa.

QP3. Desenvolvedores Android percebem os códigos afetados pelos maus cheiros como problemáticos?

Evidências na literatura sugerem que maus cheiros são percebidos por desenvolvedores [72]. A fim de validar a percepção dos maus cheiros pelos desenvolvedores Android, realizamos um experimento de código com 75 desenvolvedores. Com os resultados pudemos validar estatisticamente a percepção dos desenvolvedores sobre XX dos 16 maus cheiros derivados. *todo: trocar XX pelo número correto após experimento de código.*

1.2 Principais Contribuições

As principais contribuições desta dissertação são:

- Um catálogo com 21 novos maus cheiros relacionados a oito elementos da camada de apresentação Android, sendo quatro componentes: *Activities, Fragments, Adapters e Listeners* e quatro recursos: *Layouts, Styles, String e Drawables*.
- A validação da percepção de frequência e importância dos 21 maus cheiros no dia a dia de desenvolvimento Android.
- Uma análise estatística confirmando a percepção em códigos por desenvolvedores Android dos 8 principais maus cheiros catalogados.
- Um apêndice online com todos os relatórios e dados produzidos durante a pesquisa.

1.3 Organização do Trabalho

Os próximos capítulos desta dissertação estão organizadas da seguinte forma:

- O Capítulo 2 introduz os conceitos chave para este trabalho, a saber: Android, Qualidade de Software e Maus Cheiros de Código.
- O Capítulo 3 discute trabalhos relacionados a características que diferem o desenvolvimento Android de projetos de software tradicionais, maus cheiros específicos a uma tecnologia, maus cheiros tradicionais em aplicativos Android e maus cheiros específicos ao Android.
- O Capítulo 4 aborda em detalhes os métodos usados nas diferentes etapas da pesquisa.
- O Capítulo 5 apresenta os resultados que responde a cada questão principal de pesquisa, sendo que a Seção 5.1 apresenta a resposta da QP1, incluindo o catálogo com

21 maus cheiros derivados relacionados à camada de apresentação Android. A Seção 5.2 apresenta a resposta da QP2 e a Seção 5.3 apresenta a resposta da QP3.

- E por fim, o Capítulo 6 é onde discutimos nossas descobertas, concluímos e sugerimos trabalhos futuros.

Capítulo 2

Fundamentação Conceitual

Neste capítulo introduzimos os principais temas envolvidos nesta pesquisa com o objetivo de ambientar o leitor sobre as questões mais relevantes de cada tema.

A Seção 2.1 introduz a plataforma Android, abordando aspectos que a tornam interessante para a pesquisa e os principais termos e conceitos da plataforma abordados durante os capítulos seguintes. A Seção 2.2 contextualiza o leitor sobre os conceitos e sub-conceitos de qualidade de software de modo a clarificar em qual deles esta dissertação está inserida. Com objetivo similar, a Seção 2.3 introduz o conceito de boas práticas de software. Por último, a Seção 2.4 introduz mais profundamente o tema maus cheiros de código.

2.1 Android

O Android é uma plataforma para desenvolvimento móvel, baseada no Kernel do Linux, lançada em 2008 pelo Google em parceria com diversas empresas [4, 26]. No início de 2011 tomou a liderança, se tornando a plataforma móvel com maior participação no mercado global. A Figura 2.1 apresenta a participação de mercado das principais plataformas móveis de 2009 a 2017 [78]. Desde 2011, o Android se mantém líder e aumenta sua participação a cada ano, tendo em 2017 atingido mais de 86% de participação de mercado [78]. Seus principais concorrentes são iOS da Apple, com participação de mercado de aproximadamente 13% e o Windows Phone, oficialmente descontinuado pela Microsoft em 2017¹.

Enquanto que o iOS só é utilizado por iPhones e iPads, que são fabricados pela Apple, totalizando aproximadamente 30 dispositivos diferentes [87], o Android é utilizado por mais de 24 mil dispositivos diferentes segundo levantamento realizado em 2015 [71]. Em termos de desenvolvimento de software, essa grande variedade de dispositivos traz grandes desafios no desenvolvimento Android, desde desafios relacionados a desempenho, por conta das diferentes configurações de hardware, e desafios relacionados à camada de apresentação, pelas diversas

¹<https://support.microsoft.com/en-us/help/4001737/products-reaching-end-of-support-for-2017>

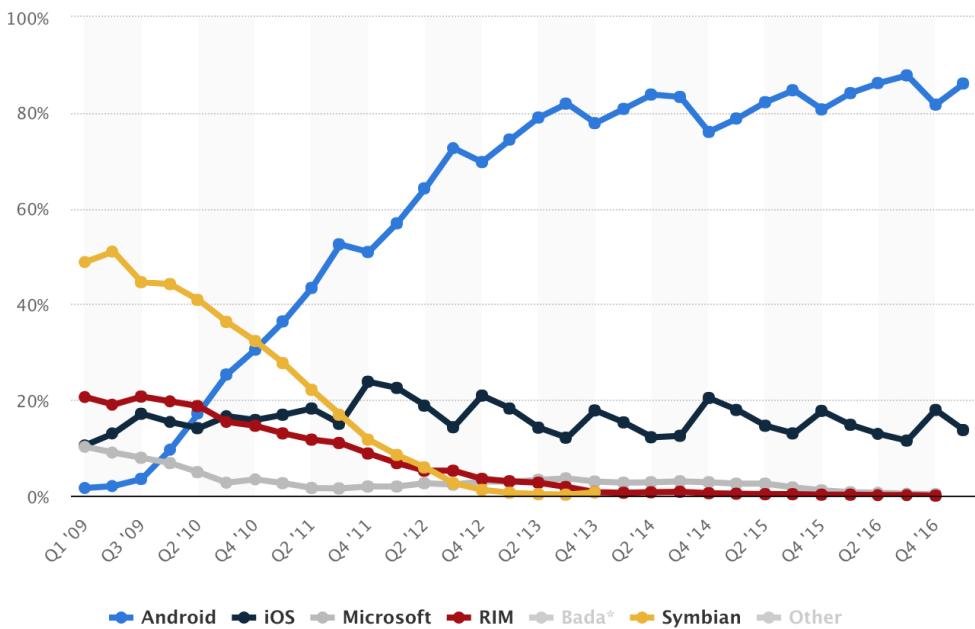


Figura 2.1: Participação global no mercado de sistemas operacionais móveis de Q1 (1º quadrimestre) de 2009 até o Q1 de 2017 [78].

configurações de tamanhos de telas e resoluções.

As seções seguintes apresentam os fundamentos e bases do desenvolvimento de aplicativos Android. Todos os códigos de exemplo na dissertação foram implementados e testados utilizando a linguagem Java.

2.1.1 Fundamentos do Desenvolvimento Android

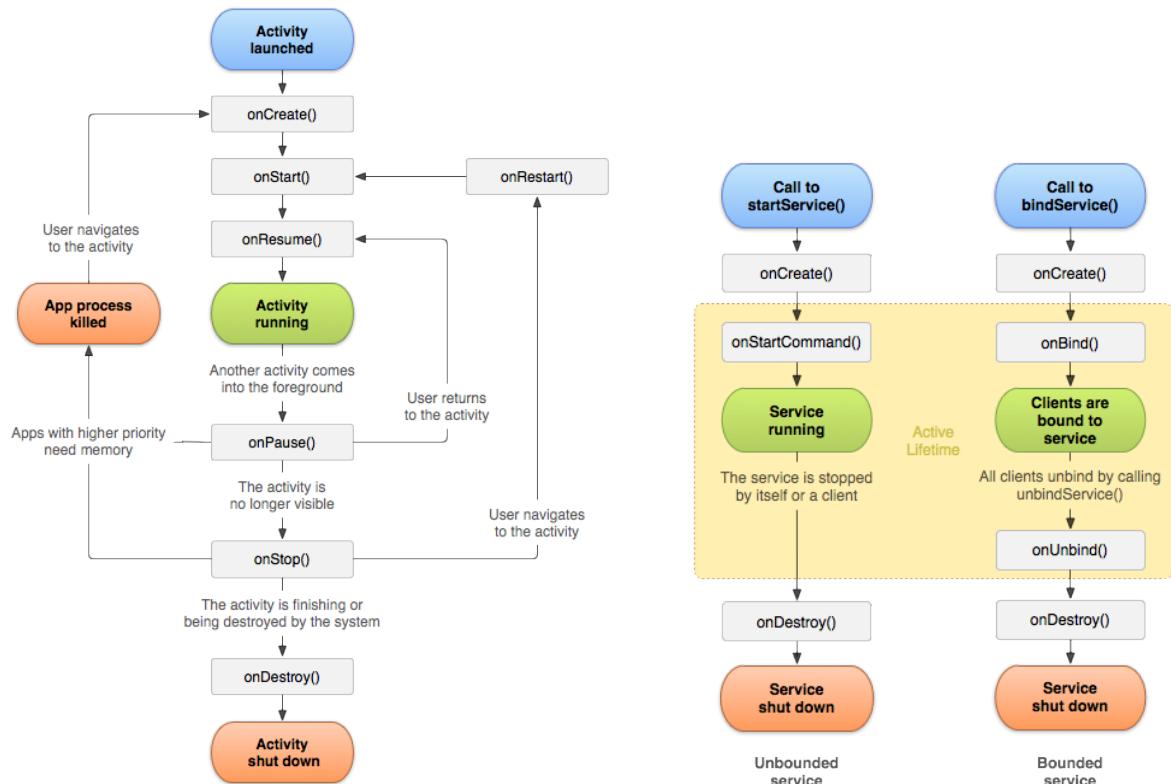
O Android Studio [41] é o ambiente integrado de desenvolvimento (*Integrated Development Environment* - IDE) oficial e gratuito, mantido pelo Google e comumente usado para o desenvolvimento Android. Juntamente com o Android Studio vem instalado o kit de desenvolvimento Android (*Software Development Kit* - SDK). O Android SDK é um conjunto de ferramentas para o desenvolvimento Android. Entre as ferramentas, podemos encontrar o compilador, depurador de código, emulador, bibliotecas de componentes base para a criação de aplicativos Android e outras. O arquivo instalável de um aplicativo Android possui a extensão .apk, acrônimo para *Android PackAge* [43].

Na estrutura de um projeto Android existem dois diretórios e um arquivo principais. O diretório `src` (de *source*) contém todo o código Java, o diretório `res` (de *resource*) contém todos os recursos da aplicação, que são códigos “*não java*”, como imagens, XMLs de layout, dentre outros. Por fim, o arquivo `AndroidManifest.xml` contém configurações gerais do aplicativo, como permissões, declarações de componentes, dentre outras [47].

O Android *Software Development Kit* (SDK) provê diversos componentes base para o

desenvolvimento Android que não estão disponíveis no desenvolvimento Java tradicional [70]. Alguns componentes são utilizados na camada de apresentação, outros para processamentos em segundo plano, outros para persistência em banco de dados local, dentre outros. Por exemplo, *Activities* são usadas para a criação de telas com UI pelo qual usuários podem interagir [28]. *Services* são usados para longos processamentos em segundo plano e não possuem UI [49]. *BroadcastReceivers* são usados como ouvintes, registrando interesse por mensagens enviadas pelo sistema operacional Android, como bateria baixa, ou por outros componentes [30]. *AsyncTasks* são usadas para curtos processamentos em segundo plano de forma assíncrona, com o objetivo de não bloquear a UI *Thread*, principal *Thread* em um aplicativo Android [31, 46].

Cada componente do Android SDK possui um conjunto de métodos de retorno que podem ou devem ser sobreescritos pelo desenvolvedor e que são chamados pelo sistema operacional Android. Ciclo de vida de um componente diz respeito a como ele é criado, mantido e destruído. O ciclo de vida é composto por um conjunto de métodos de retorno que são sempre chamados e em uma sequência específica [28, 44]. Componentes de UI como *Activities* costumam ter ciclos de vida mais extensos, e portanto mais complexos, que componentes que não lidam com a camada de apresentação. A Figura 2.3 apresenta como exemplo os ciclos de vida de *Activities*, que estão diretamente relacionadas à camada de apresentação, e *Services*, que não estão relacionados à camada de apresentação.



(a) Principais métodos de retorno do ciclo de vida de *Activities*. **(b)** Métodos de retorno do ciclo de vida de *Services*.

Figura 2.2: Comparação do ciclo de vida de *Activities* e *Services*.

O ciclo de vida de *Activities* é complexo, com mais de dez métodos de retorno [29]. A Figura 2.3b apresenta os sete principais métodos de retorno representados pelos retângulos em cinza. Além desses principais, as *Activities* possuem outros, dentre eles, *onUserLeaveHint*, *onPostCreate* e *onPostResume*.

O ciclo de vida de *Services* é bem menor se comparado com o de *Activities*, contendo até quatro métodos de retorno. A Figura 2.3c apresenta os dois possíveis ciclos de vida de *Services*, que variam de acordo com o método usado na sua criação, podendo ser *startService* ou *bindService*. Em ambos os casos, os métodos de retorno *onCreate* e *onDestroy* são chamados, a depender de como ele foi criado, no primeiro caso será chamado o *onStartCommand* e no segundo o *onBind* e *onUnbind*.

Recursos são arquivos “não Java”, utilizados na construção da UI e necessários para a criação de aplicativos Android [43]. Recursos podem ser imagens, arquivos de áudio ou arquivos XML, sendo as *tags* e atributos usados no XML, derivados do Android SDK. Recursos de *Layout* são XMLs responsáveis pela estrutura da UI como posicionamento, botões e caixas de textos. Recursos de *String* são XMLs responsáveis pelo armazenamento dos textos usados no aplicativo e possibilitam a internacionalização, ou seja, traduzir o aplicativo em outros idiomas. Recursos de *Style* são XMLs responsáveis pelo armazenamento dos estilos usados nos recursos de *layout*. Recursos *Drawable* são gráficos que podem ser imagens ou arquivos XML que criam animações ou efeitos de estado de botões, como pressionado ou desabilitado. Apesar de ser possível implementar os recursos via código Java, é fortemente recomendado pela plataforma que isso não seja feito [37].

2.1.2 Elementos da Camada de Apresentação Android

São muitos os componentes e recursos disponibilizados pelo Android SDK. Nossa estudo objetiva analisar os que estão relacionados à camada de apresentação Android. Em termos de componentes, fizemos uma extensa revisão na documentação oficial do Android [42] e chegamos a quatro, são eles: *Activities*, *Fragments*, *Adapters* e *Listeners*.

Em termos de recursos, todos são por natureza relacionados à camada de apresentação. O Android provê mais de quinze diferentes tipos de recursos [38]. Com o objetivo de focar a pesquisa, optamos por selecionar os principais com base no modelo padrão de projeto do Android Studio [32], são eles: recursos de *Layout*, recursos de *Strings*, recursos de *Style* e recursos *Drawable*. De modo genérico chamaremos os componentes e recursos mencionados aqui de “elementos” da camada de apresentação Android.

A seguir introduzimos brevemente cada um dos oito elementos da camada de apresentação Android considerados na pesquisa.

Activity é um dos principais componentes de aplicações Android e representa uma tela pelo qual o usuário pode interagir com a UI. Possui um ciclo de vida, como mencionado na seção

anterior. Toda *Activity* deve indicar no método de retorno *onCreate* o recurso de *layout* que deve ser usado para a construção de sua UI [28, 29].

Fragments representam parte de uma *Activity* e também devem indicar seu recurso de *layout* correspondente. **Fragments** só podem ser usados dentro de uma *Activity*. Podemos pensar neles como *Sub-Activities*. **Fragments** possuem um ciclo de vida extenso, com mais de dez métodos de retorno. Seu ciclo de vida está diretamente ligado ao ciclo de vida da *Activity* ao qual ele está contido. A principal uso de **Fragments** é para o reaproveitamento de trechos de UI e comportamento em diferentes *Activities* [34].

Adapters são utilizados para popular a UI com coleções de dados como por exemplo, uma lista de *e-mails*, onde o *layout* é o mesmo para cada item mas o conteúdo é diferente [36].

Listeners são interfaces que representam eventos do usuário, por exemplo o *OnClickListener* captura o clique pelo usuário. Essas interfaces costumam ter apenas um método, onde é implementado o comportamento desejado para responder a interação do usuário [27].

Recursos de Layout são XMLs utilizados para o desenvolvimento da estrutura da UI dos componentes Android. O desenvolvimento é feito utilizando uma hierarquia de *Views* e *ViewGroups*. *Views* são caixas de texto, botões, dentre outros. *ViewGroups* são *Views* especiais pois podem conter outras *Views*. Cada *ViewGroup* organiza suas *Views* filhas de uma forma específica, horizontalmente, em tabela, posicionamento relativo, dentre outros. Esta hierarquia pode ser tão simples ou complexa quanto se precisar, mas quanto mais simples melhor o desempenho [35, 36].

Recursos de String são XMLs utilizados para definir textos, conjunto de textos e plurais usados no aplicativo. As principais vantagens de se usar recursos de *String* é o reaproveitamento dos textos em diferentes UIs e a facilidade para internacionalizar [39].

Recursos de Style são XMLs utilizados para a definição de estilos a serem aplicados nos XMLs de *layout*. A principais vantagens em se utilizar recursos *Styles* é separar o código de estrutura da UI do código que define sua aparência e forma, e também possibilitar a reutilização de estilos em diferentes UIs [40].

Recursos de Drawable são arquivos gráficos utilizados na UI. Estes arquivos podem ser imagens tradicionais, .png, .jpg ou .gif, ou XMLs gráficos. A principal vantagem dos XMLs gráficos está no tamanho do arquivo que é comumente bem menor do que imagens tradicionais e, diferente das imagens tradicionais onde é recomendado que tenha mais de uma versão da mesma em resolução diferentes, XMLs gráficos só é necessário uma versão [33].

2.1.3 Desafios no Desenvolvimento da Camada de Apresentação Android

Activity é um dos principais componentes de aplicativos Android. Ela representa uma tela com interface do usuário (*User Interface - UI*) pelo qual o usuário pode interagir através de botões, listagens, caixas de entrada de textos, dentre outros. Para implementar uma *Activity* é necessário criar uma classe derivada de *Activity* e sobrescrever alguns métodos herdados, chamados de métodos de retorno. O principal método de retorno é o *onCreate*, entre suas responsabilidades está a criação da tela e configuração da UI. O Código-Fonte 2.1 apresenta o código mínimo para a criação de uma *Activity*, na linha 5 temos o código responsável pela configuração da UI que indica o recurso de *layout* “main_activity”.

```
1 public class MainActivity extends Activity {  
2     @Override  
3     public void onCreate(Bundle savedInstanceState) {  
4         super.onCreate(savedInstanceState);  
5         setContentView(R.layout.main_activity);  
6     }  
7 }
```

Código-Fonte 2.1: Código mínimo para a criação de uma *Activity*

A UI de uma *Activity* é construída por meio de recursos de *layout*, arquivos XML cujo as *tags* provém do kit de desenvolvimento Android (*Software Development Kit - SDK*) e representam *Views* ou *ViewGroups*. O Código-Fonte 2.2 apresenta um recurso de *layout* com duas *Views* e um *ViewGroup*. As *Views* são um *TextView*, que representa uma caixa de entrada de texto e um *Button*, que representa um botão. Essas duas *Views* estão contidas dentro do *ViewGroup LinearLayout*, que as organiza verticalmente.

```
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="Um TextView" />  
11  
12    <Button android:id="@+id/button"  
13        android:layout_width="wrap_content"  
14        android:layout_height="wrap_content"  
15        android:text="Um Button" />
```

```
16 </LinearLayout>
```

Código-Fonte 2.2: Exemplo de recurso de layout com um campo de entrada de texto e um botão organizados um abaixo do outro.

Os códigos apresentados são bem simples, mas comumente, estas telas e UIs tendem a ser bem mais robustas e ricas de informações e interatividade. São em contextos como esses que os desafios no desenvolvimento da camada de apresentação Android surgem. UIs ricas e robustas podem significar muitas *Views* e *ViewGroups*, resultando em recursos de *layout* grandes e complexos. E ainda, quanto mais ricas e robustas são as UIs, mais provavelmente o código das *Activities* correspondentes também serão grandes e complexos, pois são pelas *Activities* que *Views* e *ViewGroups* conseguem interagir com o usuário, também são pelas *Activities* que os dados chegam até a UI e vice-versa, dentre muitas outras responsabilidades que tendem a ficar com as *Activities*.

Ciclo de Vida

Toda *Activity*, bem como outros componentes Android, possui um ciclo de vida. O ciclo de vida de um componente é composto por um conjunto de métodos de retorno, que por sua vez, são métodos chamados pelo Android em uma ordem específica [28]. Os componentes da camada de apresentação Android possuem ciclos de vida mais extensos, e portanto, mais complexos.

Como exemplo, na Figura 2.3 apresentamos o ciclo de vida de três componentes Android: *Activities* e *Fragments*, ambos relacionados à camada de apresentação e *Services*, componente usado para longos processamento em segundo plano. Os métodos de retorno são representados pelos retângulos em cinza. É possível observar que *Activities* e *Fragments* possuem respectivamente sete e onze métodos de retorno enquanto que *Services* possuem apenas quatro.

2.2 Qualidade de Software

Antes de falarmos sobre qualidade de software, é importante falarmos brevemente sobre qualidade. Há décadas diversos autores e organizações vem trabalhando em suas próprias definições de qualidade. Segundo o ISO 9000 [56] desenvolvido pela Organização Internacional de Normalização (*International Organization for Standardization - ISO*), qualidade é “*o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes*”. Segundo Juran [59], um dos principais autores sobre o assunto, qualidade está relacionado “*as características dos produtos que atendem as necessidades dos clientes, e assim, proporcionam a satisfação do mesmo*” e a “*ausência de deficiências*”.

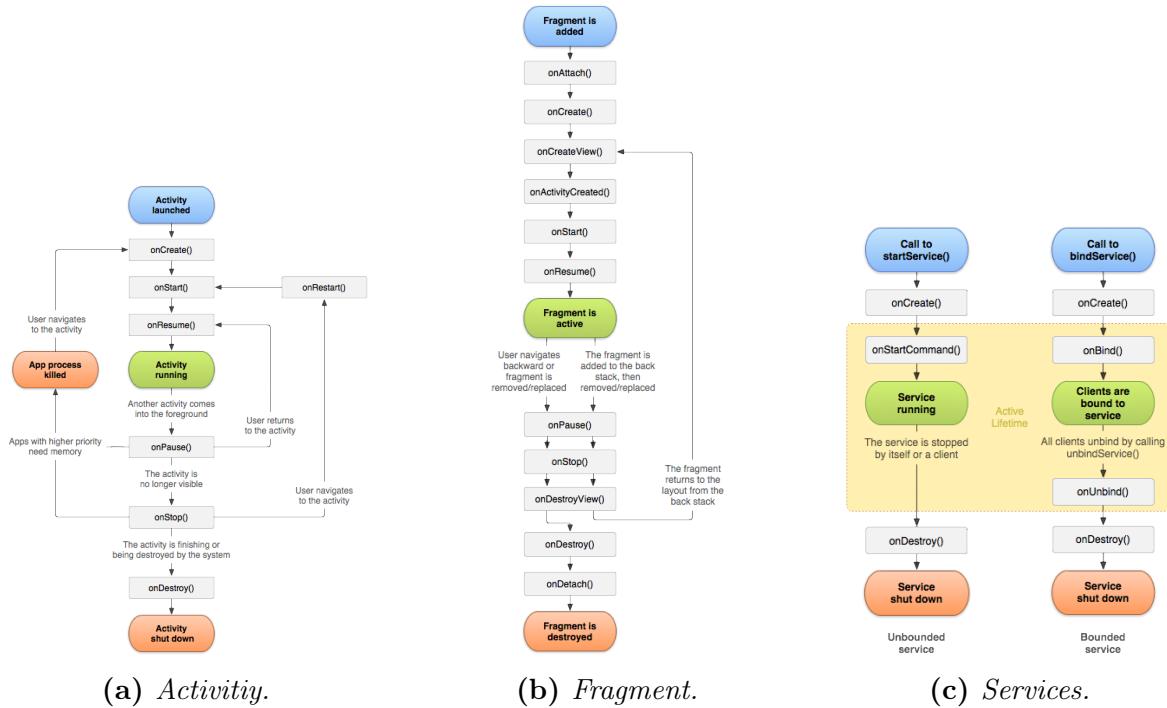


Figura 2.3: Comparativo do ciclo de vida de componentes Android que pertencem e não pertencem à camada de apresentação.

Stefan [83, p 6] também examinou diversas definições e identificou que na maioria das é possível identificar uma mesma ideia central sobre qualidade, todas elas se apontam para “requisitos que precisam ser satisfeitos para ter qualidade”.

Tabela 2.1: Modelos de qualidade de software baseados no modelo de decomposição hierárquica de Boehm et al. [9] e McCall et al. [66].

Nome do Modelo	Descrição
ISO/IEC 9126	Teve sua primeira publicação em 1991, foi revisado em 2001 e em 2011 substituído pela ISO/IEC 25010, conhecida por <i>Systems and software Quality Requirements and Evaluation</i> (SQuaRE). Decompõe qualidade de software em 6 áreas: manutenibilidade, eficiência, portabilidade, confiabilidade, funcionalidade e usabilidade [15].
ISO/IEC 25010 (SQuaRE)	Teve sua primeira publicação em 2005, revisado em 2011, quando passou a substituir a ISO/IEC 9126. Decompõe qualidade de software em 8 áreas: manutenibilidade, eficiência, portabilidade, confiabilidade, funcionalidade, usabilidade, compatibilidade e segurança [55].
CISQ	Fundado em 2009 [80], compõe qualidade de software em 4 características: segurança, confiabilidade, desempenho/eficiência e manutenibilidade. Se baseia nas definições do SQuaRE [14].
FURPS	FURPS é um acrônimo para: Funcionalidade, Usabilidade, Confiabilidade (do inglês <i>Reliability</i>), Desempenho (do inglês <i>Performance</i>) e Suportabilidade [51].

As primeiras contribuições referente a qualidade em termos de software foram publicadas no fim da década de 70. Boehm et al. [9] e McCall et al. [66] descrevem modelos de qualidade

de software através de conceitos e subconceitos. Ambos se utilizaram de uma estratégia de decomposição hierárquica do conceito de qualidade em fatores como manutenibilidade e confiabilidade [83, p 29-30]. Com o tempo, diversas variações desse modelo começaram a surgir. A Tabela 2.1 apresenta algumas dessas variações e uma breve descrição da forma como qualidade de software foi decomposta. Entretanto, o grande valor do modelo de decomposição hierárquica foi a ideia de decompor qualidade até um nível em que seja possível medir e estimar [83].

O padrão ISO/IEC 9126 é considerado ainda como principal referência para a definição de qualidade de software e a define como “a capacidade do produto de software satisfazer as necessidades implícitas e explícitas quando usado em condições específicas” [83, p 10]. O ISO/IEC 9126 subdivide qualidade de software em seis conceitos, cada qual contendo um conjunto de sub-conceitos, conforme apresentado na Figura 2.4.

Dentre os conceitos apresentados na Figura 2.4, os mais relevantes nesta dissertação são **manutenibilidade**, pois é onde esta pesquisa está inserida, e **usabilidade** e **eficiência**, pois é o contexto onde encontramos alguns trabalhos relacionados sobre maus cheiros de código Android [50, 75]. Portanto apresentamos uma breve descrição sobre cada um destes três conceitos a seguir.

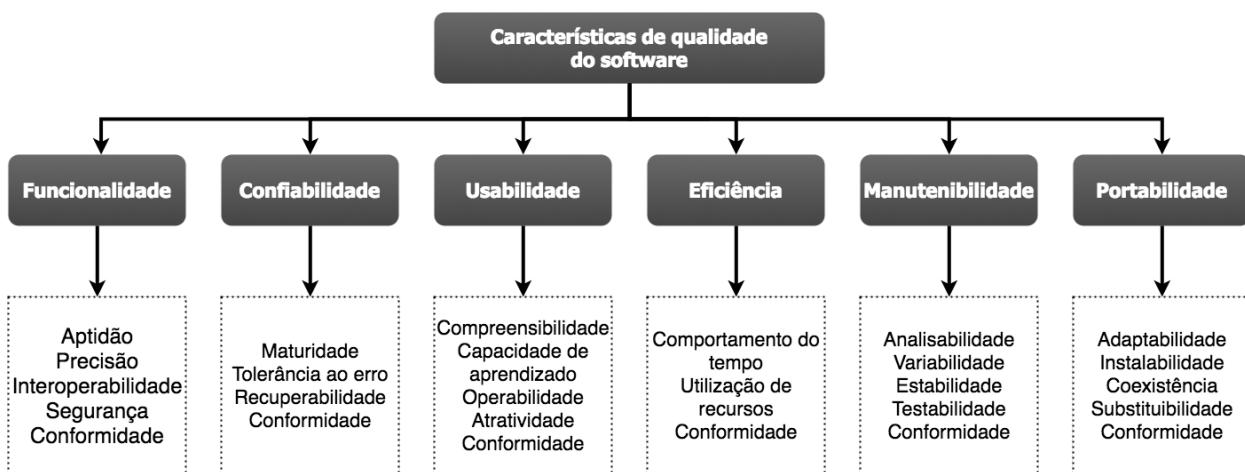


Figura 2.4: Características de Qualidade de Software segundo norma ISO/IEC 9126 [15].

Manutenibilidade A capacidade (ou facilidade) do produto de software ser modificado, incluindo tanto as melhorias ou extensões de funcionalidade quanto as correções de defeitos, falhas ou erros. Seus sub-conceitos são:

- *Analisabilidade* Facilidade em se diagnosticar eventuais problemas e identificar as causas das deficiências ou falhas.
- *Modificabilidade* Facilidade com que o comportamento do software pode ser modificado.
- *Estabilidade* Capacidade do software de evitar efeitos colaterais decorrentes de modificações introduzidas.

- *Testabilidade* Capacidade de se testar o sistema modificado, tanto quanto as novas funcionalidades quanto as não afetadas diretamente pela modificação.

Usabilidade A capacidade do produto de software ser compreendido, seu funcionamento aprendido, ser operado e ser atraente ao usuário. Seus sub-conceitos são:

- *Inteligibilidade* Facilidade com que o usuário pode compreender as suas funcionalidades e avaliar se o mesmo pode ser usado para satisfazer as suas necessidades específicas.
- *Apreensibilidade* Facilidade de aprendizado do sistema para os seus potenciais usuários.
- *Operacionalidade* Como o produto facilita a sua operação por parte do usuário, incluindo a maneira como ele tolera erros de operação.
- *Proteção frente a erros de usuários* Como produto consegue prevenir erros dos usuários.
- *Atratividade* Envolve características que possam atrair um potencial usuário para o sistema, o que pode incluir desde a adequação das informações prestadas para o usuário até os requintes visuais utilizados na sua interface gráfica.
- **Acessibilidade** Prática inclusiva de fazer softwares que possam ser utilizados por todas as pessoas que tenham deficiência ou não. Quando os softwares são corretamente concebidos, desenvolvidos e editados, todos os usuários podem ter igual acesso à informação e funcionalidades.

Eficiência O tempo de execução e os recursos envolvidos são compatíveis com o nível de desempenho do software. Seus sub-conceitos são:

- *Comportamento em Relação ao Tempo* Avalia se os tempos de resposta (ou de processamento) estão dentro das especificações.
- *Utilização de Recursos* Mede tanto os recursos consumidos quanto a capacidade do sistema em utilizar os recursos disponíveis.

Esta pesquisa está inserida no contexto de *manutenibilidade*, mais especificamente *analisabilidade* e *modificabilidade*, pois, conforme descrito na Seção 2.4, maus cheiros de código visam apontar trechos de códigos possivelmente problemáticos que podem se beneficiar de refatorações, melhorando a manutenibilidade do software.

Muito embora tenhamos ciência que investir em qualidade pode reduzir os custos de um projeto, aumentar a satisfação dos usuários e desenvolvedores, qualidade de software costuma ser esquecido ou deixado em segundo plano². Frases como “depois eu testo” ou “depois eu refatoro” são comuns no dia a dia de desenvolvimento. Manutenibilidade está relacionada às “necessidades implícitas do software” [15].

²<http://www.ifsq.org/resources/level-2/booklet.pdf>

Focado nesse conceito, ao longo dos últimos anos, diversas boas práticas de software vêm sendo documentadas objetivando servir de ferramenta a desenvolvedores menos experientes para aumentar a qualidade do software. Por exemplo, os padrões de projeto da Gangue dos Quatro (*Gang of Four - GoF*) documentam as *melhores soluções para problemas comuns* originadas a partir do conhecimento empírico de desenvolvedores de software experientes. Em contrapartida, anti-padrões são padrões antes recomendados que passaram a ser evitados, pois percebeu-se que os problemas em usá-los superavam os benefícios [10]. Um dos maiores exemplos de anti-padrão é o *Singleton*, que visa limitar a instanciação de uma classe em um único objeto [23].

É relevante destacar 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 documentos são comumente generalizados entre os desenvolvedores simplesmente pelo termo *boas práticas de software* [67].

2.3 Boas Práticas de Software

No 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 [67]. A seguir introduzimos os conceitos das boas práticas: padrões de projetos, anti-padrões e *maus cheiros de código*.

2.3.1 Padrões de Projeto

“*Cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira.*”

– Christopher Alexander, Uma Linguagem de Padrões [3]

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* [19]. Padrões não são invenção de algo novo, é uma forma de organizar o conhecimento de experiências [12].

Para engenharia de software, a principal definição sobre “padrões” provém do livro Uma Linguagem de Padrões do arquiteto Christopher Alexander (1977) [3] 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” [21].

Inspirados por Alexander [3], 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. Apoando-se na definição de padrões de Alexander [3], Design Patterns - GoF (1994) foi o primeiro livro sobre padrões de projeto de software a ser lançado, documentando 23 padrões de projetos.

Para se documentar um padrão, é comum seguir um formato específico. O formato indica onde e como cada aspecto, como problema, contexto ou solução, será encontrado. Alguns formatos incluem uma série de outras informações além destes três bases de todos padrão. Ao longo dos anos, alguns formatos foram se destacando, a Figura 2.5 apresenta a tradução de um gráfico criado por Joshua Kerievsky [60] contendo quatro formatos existentes de se escrever padrões. Os formatos são nomeados como Portland, Coplien, *Gang of Four* (GoF) e Alexandrino, e foram posicionadas no gráfico de acordo com seu nível de maturidade e clareza.

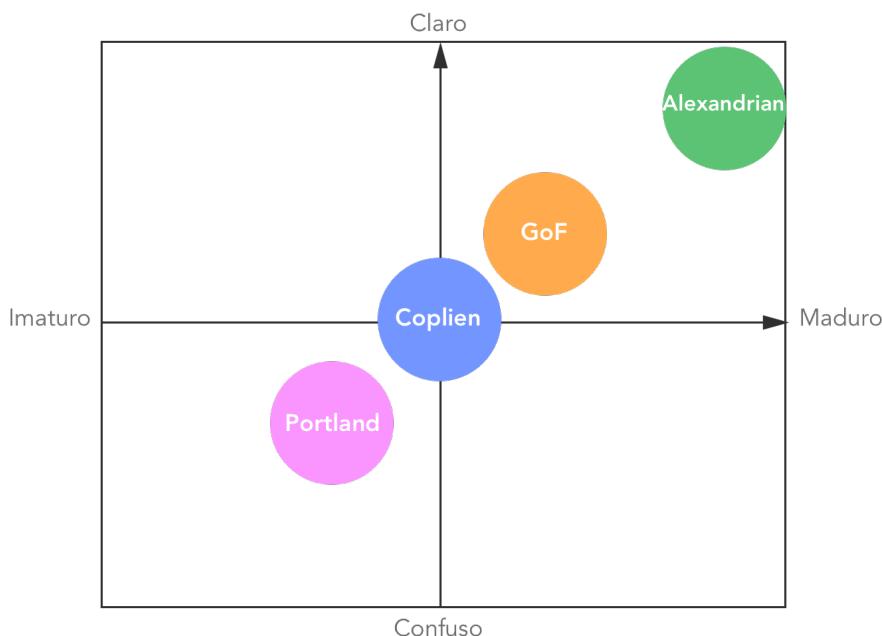


Figura 2.5: Formatos de padrões de acordo com seu nível de maturidade e clareza segundo Joshua Kerievsky [60].

É possível observar na Figura 2.5 que quanto mais para cima e mais a direita do gráfico, mais claro e maduro é o formato do padrão. Sendo assim, o mais claro e maduro é o formato Alexandrino seguido por Gof, Coplien e por último Portland.

2.3.2 Anti-Padrões

“Um anti-padrão é como um padrão, exceto que em vez de uma solução, ele dá algo que parece superficialmente como uma solução, mas não é.”

— Andrew Koenig, O Manual de Padrões: Técnicas, Estratégias e Aplicações [62]

Um anti-padrão é uma resposta comumente usada para 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 [62], inspirado pelo livro GoF [23].

O termo se popularizou três anos após, no livro de Brown (1998) [10]. Segundo Brown [10], para diferenciar um anti-padrão de um mau hábito, má prática ou ideia ruim 1) existe 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.

É muito comum ver o termo anti-padrão ser equivocadamente usado para indicar mau cheiro de código. O equívoco ocorre geralmente por ambos tratam de práticas que influenciam negativamente a manutenibilidade do software. Entretanto, anti-padrões se diferem de maus cheiros pois maus cheiros são sintomas que podem ou não indicar um problema mais profundo no código enquanto que um anti-padrão é uma solução com passos bem definidos para um problema específico, porém uma solução que deixou de ser recomendada.

2.4 Maus Cheiros de Código

*“Maus cheiros são certas estruturas no código que indicam a violação de princípios fundamentais de *design* e impactam negativamente a qualidade do projeto.”*

— Refactoring for Software Design Smells: Managing Technical Debt [79]

Um mau cheiro de código é uma sugestão de que algo não está certo no código, semelhante o dito popular “Algo não cheira bem!” quando alguém desconfia de que há algum problema em dada situação. Segundo Fowler [79], maus cheiros *“são indicações de que existe um problema que pode ser resolvido por meio de uma refatoração”*. Do ponto de vista de qualidade de software, *“são certas estruturas no código que indicam a violação de princípios fundamentais de design e impactam negativamente a qualidade do projeto.”* [79, p. 258].

Entretanto, maus cheiros não são erros — eles não estão tecnicamente incorretos e não impedem o funcionamento do software. Em vez disso, eles indicam deficiências no código que podem dificultar a manutenibilidade do software e aumentar o risco de erros ou falhas no futuro [90, 77, 88, 61].

Webster (1995) [84] foi um dos primeiros livros a usar o conceito de maus cheiros 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* [84, p. 180]. Há indícios informais de que o termo foi cunhado em meados da década de 90 por Kent Beck [86, 85]. 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) [22].

No livro são apresentados 22 maus cheiros e mais de 70 técnicas de refatoração. *Código Duplicado* [22, 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* [22, p. 89], ou seja, extrair a expressão duplicada para um novo método e substituí-lo nos lugares onde a expressão era usada.

Maus cheiros provem do profundo conhecimento e experiências de desenvolvedores experientes que ao longo dos anos desenvolveram um “intuição” para o bom *design* possibilitando-os olhar para um *design* ou código e obter imediatamente uma “intuição” sobre sua qualidade, sem ter que dar “argumentos logicamente detalhados” sobre o porquê de sua conclusão [85]. Ou seja, é por natureza subjetivo, baseado em opiniões e experiências [81, 20].

Nos anos seguintes o termo mau cheiro se tornou frequente em livros [65, 79] e pesquisas acadêmicas. No livro *Clean Code* (Martin, R. 2008) [65], que se tornou muito popular entre desenvolvedores de software, são definidos novos maus cheiros além de citar alguns já apresentados por Fowler [22]. 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 diferente do que vimos com padrões, não encontramos referências formais sobre como documentar adequadamente um mau cheiro.

Fowler [22] 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* [22, 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 [65] 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* [65, 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 [6, 24].

Capítulo 3

Trabalhos Relacionados

Aplicativos Android são desenvolvidos, em sua maioria, utilizando a linguagem de programação Java [43]. Deste modo, um provável questionamento é: “Por que investigar maus cheiros específicos ao Android quando já existem tantos maus cheiros e boas práticas documentadas para linguagens orientada a objetos como o Java?”. Para responder a esta pergunta temos as seguintes seções:

- A Seção 3.2 apresenta pesquisas que investigaram e encontraram importantes características que diferenciam projetos Android de projetos de software tradicionais, como projetos web e cliente/servidor. Estas características agregam complexidades ao desenvolvimento Android não encontradas no desenvolvimento de softwares tradicionais.
- A Seção 3.3 apresenta pesquisas que têm demonstrado que diferentes tecnologias podem apresentar maus cheiros específicos. Há pesquisas que, inclusive que a camada de apresentação de softwares tradicionais apresentam maus cheiros específicos, e reforça nossa hipótese de que o desenvolvimento da camada de apresentação Android pode seguir por este mesmo comportamento.
- A Seção 3.4 apresenta pesquisas que (i) investigaram a presença de maus cheiros tradicionais em aplicativos Android, (ii) a existência de maus cheiro específicos ao Android e também (iii) compararam a presença dos maus cheiros tradicionais vs. a presença de maus cheiros específicos em aplicativos Android. Estas pesquisas reforçam a relevância de investigarmos maus cheiros específicos ao Android pois concluem que maus cheiros específicos aparecem muito mais do que os maus cheiros tradicionais.

Ao final desta seção pretende-se esclarecer os motivos pelo qual optamos por investigar maus cheiros de código relacionados a camada de apresentação Android e também dar uma visão sólida do estado da arte sobre o assunto.

3.1 Maus Cheiros Tradicionais

todo: Em construção.

3.2 Projetos Android vs. projetos de software tradicionais

Minelli e Lanza [68] apresentam diferenças no desenvolvimento de aplicativos e softwares tradicionais em termos de métricas de código, uso de APIs terceiras e evolução. Para isso se utilizam da ferramenta SAMOA desenvolvida por eles de análise estática de código.

É interessante que para análise do código Android eles modelam o projeto em código núcleo e não núcleo, similar ao realizado por [82], onde o código núcleo está relacionado a classes que herdam do Android SDK. Apesar disso, eles dizem coletar essa informação do Android Manifesto, considerando *Activities* e *Services*, entretanto, existem diversas outras classes em um projeto Android que herdam do Android SDK e não precisam ser declaradas no Android Manifesto, ou seja, a definição usada abrange muito mais código do que de fato analisado pela pesquisa.

Os autores concluem com um conjunto de características de aplicativos Android e com um conjunto de hábitos dos desenvolvedores destes aplicativos que diferem de aplicações de software tradicionais. Dentre as características, afirmam que algumas vezes o código núcleo do app é composto por uma, ou algumas, *God Classes* e que herança, para o uso de *design* das classes, é algo quase inexistente em aplicativos Android. Estas constatações reforçam um mau cheiro por nós identificado sobre a falta de arquitetura padrão, visto que esta exige um conhecimento mais aprimorado de orientação a objetos.

3.3 Maus cheiros específicos a uma tecnologia

A constante e rápida evolução de tecnologias existentes e criação de novas tecnologias faz com que diversos temas, como manutenibilidade de software, estejam também em constante alta. Muitos pesquisadores veem pesquisando sobre a existência de maus cheiros de código específicos a uma dada tecnologia como por exemplo arcabouços Java [6, 13], a linguagem CSS (*Cascading Style Sheets*) [24] e fórmulas em planilhas [73].

Chen et al. [13] viu a necessidade de estudar maus cheiros de código em arcabouços Object-Relational Mapping (ORM) pelo grande uso pela indústria e pela desatenção de desenvolvedores com relação ao impacto dos seus códigos no desempenho do banco de dados que podiam causar *timeouts* e paradas dos sistemas. Os autores implementaram um arcabouço automatizado e sistemático para detectar e priorizar anti-padrões de desempenho em

aplicações desenvolvidas usando ORM e também mapearam dois anti-padrões específicos a arcabouços ORM.

Aniche et al. [7, 6, 5] investigaram maus cheiros de código relacionado ao arcabouço Spring MVC, usado para o desenvolvimento da camada de apresentação de aplicações web Java. Aniche et al. encontram maus cheiros específicos a cada camada do arcabouço Spring MVC, modelo, visualização e controladora, afirmando que cada papel arquitetural possui responsabilidades diferentes o que resulta em distribuições diferentes de valores de métrica de código e maus cheiros diferentes. Dentre as principais contribuições deste trabalho está um catálogo com seis maus cheiros específicos ao arcabouço Spring MVC mapeados e validados.

Gharachorlu [24] investigou maus cheiros em código CSS, linguagem amplamente utilizada na camada de apresentação de aplicações web para separar a semântica de apresentação do conteúdo HTML. De acordo com o autor, apesar da simplicidade de sintaxe do CSS, as características específicas da linguagem tornam a criação e manutenção de CSS uma tarefa desafiadora. Foi realizando um estudo empírico de larga escala onde os resultados indicaram que o CSS de hoje sofre significativamente de padrões inadequados e está longe de ser um código bem escrito. Gharachorlu propõe o primeiro modelo de qualidade de código CSS derivado de uma grande amostra de aprendizagem de modo a ajudar desenvolvedores a obter uma estimativa do número total de cheiros de código em seu código CSS. Sua principal contribuição foram oito novos maus cheiros CSS detectados com o uso da ferramenta CSSNose, também implementada e disponibilizada pelo autor.

Fard e Ali [20] investigaram maus cheiros de código no Javascript, que é uma flexível linguagem de *script* para o desenvolvimento do comportamento do lado do cliente, que faz parte da camada de apresentação de aplicações web. Os autores afirmam que devido à essa flexibilidade, o JavaScript é uma linguagem particularmente desafiadora para escrever e manter código. Alguns dos desafios citados são a questão de, diferentemente de aplicações Android que são compiladas, o Javascript é interpretado, o que significa que normalmente não há compilador no ciclo de desenvolvimento que ajudaria os desenvolvedores a detectar código incorreto ou não otimizado no momento da compilação. Outro ponto que o autor indica como problema é a natureza dinâmica, fracamente tipificada e assíncrona, além de outros desafios citados. Os autores propõem um conjunto de 13 maus cheiros de código JavaScript, sendo 7 maus cheiros tradicionais adaptados para o JavaScript e 6 maus cheiros específicos ao JavaScript derivado da pesquisa. Também é apresentada uma técnica automatizada, chamada JSNOSE, para detectar esses maus cheiros.

Um interessante relação que vemos é que muitas pesquisas buscaram por maus cheiros específicos em tecnologias usadas na camada de apresentação de aplicações web [20, 24, 6] o que reforça nossa hipótese de que aplicativos Android podem seguir o mesmo comportamento possivelmente apresentando maus cheiros específicos a camada de apresentação não encontrados necessariamente nos demais códigos da aplicação.

3.4 Maus cheiros em aplicativos Android

Pesquisas em torno de maus cheiros em aplicativos Android ainda são poucas. Umme et al. [64] recentemente levantaram que, das principais conferências de manutenção de software, dentre 2008 a 2015, apenas 10% dos artigos consideraram em suas pesquisas, projetos Android e nenhuma outra plataforma móvel foi considerada. As conferências consideradas foram as ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MRS e ESEM.

Dentre as pesquisas analisadas para efeitos deste trabalho, podemos classificar em 3 grupos: 1) aquelas que buscam por maus cheiros tradicionais em aplicativos Android, 2) as que buscam por maus cheiros específicos a aplicativos Android, grupo ao qual nossa pesquisa está inserida, 3) e as que buscam ambos os maus cheiros, específicos e tradicionais, em aplicativos Android e comparam qual deles é mais frequente.

As seções seguintes tratam do estado da arte de cada um desses grupos bem como suas semelhanças e diferenças com nossa pesquisa.

3.4.1 Presença de maus cheiros tradicionais em aplicativos Android

Linares-Vásquez et al. [63] usaram a ferramenta DECOR para realizar a detecção de 18 diferentes anti-padrões orientado a objetos em aplicativos móveis desenvolvidos com Java Mobile Edition (J2ME) e entender a relação dos maus cheiros com o domínio de negócio e métricas de qualidade. Dentre as principais conclusões do estudo está a conclusão de que existe uma grande diferença nos valores das métricas de qualidade em aplicativos afetados pelos maus cheiros e pelos que não e que enquanto há maus cheiros presentes em todos os domínios, alguns são mais presentes em domínios específicos.

Verloop [82] investigou a presença de maus cheiros de códigos tradicionais propostos por Fowler [22] (*Long Method, Large Class, Long Parameter List, Feature Envy* e *Dead Code*) em aplicativos Android para determinar se esses maus cheiros ocorrem mais frequentemente em *classes núcleo*, classes no projeto Android que precisam herdar de classes do SDK Android, como por exemplo *Activities*, *Fragments* e *Services*, comparando com classes não núcleo. Para isso, ele fez uso de 4 ferramentas de detecção automática de maus cheiros: JDeodorant, Checkstyle, PMD e UCDetector.

O autor afirma que classes núcleos tendem a apresentar os maus cheiros *God Class, Long Method, Switch Statement* e *Type Checking* pela sua natureza de muitas responsabilidades, sendo que a classe mais observada com estes maus cheiros foram *Activities*. Um ponto a se pensar é, se a natureza da Activity é de ter muitas responsabilidades, talvez estejamos analisando-a a partir de um ponto de vista inadequado ao buscarmos por *God Class* ou *Long Method*, visto que sabe-se agora de que estes maus cheiros de fato a afetam, mas que de certa forma, esse é o *modus operandi* normal dela. Chegamos ao ponto que a natureza do Android

pode implicar em maus cheiros específicos que trazem outros ponto de vista que respeitem a natureza de aplicativos Android, e proponham uma refatoração adequada.

O autor também conclui que o mau cheiro tradicional *Long Parameter List* é menos provável de aparecer em classes núcleo pois nessas classes, a maioria dos métodos são sobrecargas de métodos da classe herdada proveniente do SDK Android, e como para se realizar uma sobrecarga de método é necessário seguir a assinatura do método original, este normalmente não é afetado por este mau cheiro. Novamente voltamos ao ponto que maus cheiros tradicionais não foram pensados considerando a natureza de projetos Android, que neste caso está relacionada a herança de classes núcleo.

Verloop [82] conclui propondo cinco refatorações com o objetivo de mitigar o mau cheiro *Long Method* que se apresentou por diversos motivos em *Activities* e *Adapters*. Dentre estas cinco propostas de refatoração, ele implementou e experimentou três, sendo que:

- A primeira, uso do padrão *ViewHolder* em *Adapter*, de fato melhora a qualidade do código, no exemplo do autor, dos 12 *Adapter* afetados pelo mau cheiro *Long Method*, após a refatoração apenas 4 continuaram apresentando o mau cheiro. Este padrão trás resultados não apenas em manutenibilidade, mas também em eficiência, realizando um consumo melhor de memória, processamento e energia. Por todos estes benefícios no uso desse padrão, atualmente o Android SDK já provê um componente com esta implementação de forma nativa, o componente *RecyclerView*¹.
- A segunda, uma espécie de *ViewHolder* para *Activities*, objetivava mitigar *Long Method*, porém não trouxe bons resultados sendo que das 13 *Activities* refatoradas, nenhum deixou de ser afetada pelo *Long Method*. Dessa forma, o autor conclui que outros não trabalhados por meio da refatoração proposta influenciam na aparição deste mau cheiro em *Activities*. Estes outros fatores estão relacionados a *listeners* e classes anônimas usados para a implementação de comportamento dos elementos da camada de apresentação pois estes códigos são comumente colocados no método *onCreate* de *Activities*.
- A terceira propõe mitigar o mau cheiro *Long Method* em *Activities* trocando a atribuição de *listeners* feitas com classes anônimas pelo uso do atributo *onClick* no XML de layout respectivo. Os resultados aqui obtidos também não foram muito satisfatórios pois, das 13 classes refatoradas, 7 ainda apresentaram o mau cheiro devido ao uso de outros *listeners* que não o de *click*, que não possuem atributos correspondentes no XML de layout. Além disso, já se sabe hoje que o uso de atributos não é interessante devido ao acoplamento resultante da *Activity* com aquele determinado XML dentre outros problemas. Apesar disso, o autor considerou este resultado como positivo.

¹<https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html>

É interessante notar que dentro da definição de Verloop [82] de classes núcleo, estão incluídas classes que herdam de `Services` e todas as demais que herdam de alguma classe do SDK Android, porém as únicas classes que apresentaram maus cheiros foram `Activities` e `Adapters`. Como vimos em 2.1, essas classes são responsáveis por lidar com a camada de apresentação Android, o que reforça nossa hipótese de que a camada de apresentação Android tende a ser mais problemático que o restante dos códigos da aplicação e por isso vale a pena ser estudado mais a fundo.

3.4.2 Maus cheiros específicos ao Android

Gottschalk et al. [50] conduziram um estudo sobre formas de detectar e refatorar maus cheiros de código relacionados ao uso eficiente de energia. Os autores compilaram um catálogo com 6 cheiros de código extraídos de outros trabalhos, e trabalharam sob um trecho de código Android para exemplificar um deles, o *Carregar Recurso Muito Cedo*, 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 maus cheiros de código relacionados manutenibilidade enquanto eles tratam de eficiência, conforme conceitos de qualidade de software apresentados da Seção 2.2.

Reimann et al. [75] correlaciona os conceitos de mau cheiro, qualidade e refatoração a fim de introduzir o termo mau cheiro de qualidade (do inglês *quality smell*). Segundo os autores, um mau cheiro de qualidade é uma estrutura que influencia negativamente requisitos de qualidade específicos, que podem ser resolvidos por refatorações [74]. Os autores compilaram um catálogo de 30 cheiros de qualidade para Android. O formato dos cheiros de qualidade incluem: nome, contexto, requisitos de qualidades afetados e descrição, este formato foi baseado nos catálogos de Brown et al. [10] e Fowler [22]. Todo o catálogo pode ser encontrado online² e os mesmos também foram implementados no arcabouço Refactory [74].

Os requisitos de qualidade tratados por Reimann et al. [75] são: centrados no usuário (estabilidade, tempo de inicio, conformidade com usuário, experiência do usuário e acessibilidade), consumo inteligente de recursos de hardware do dispositivo (eficiência no uso de energia, processamento e memória) e segurança.

Reimann et al. [75] cita que o problema no desenvolvimento móvel é que os desenvolvedores estão cientes dos maus cheiros de qualidade apenas indiretamente, que suas definições são informais como melhores práticas e discussões em fóruns. Continua dizendo que os recursos para encontrá-los são distribuídos pela web e que é difícil coletar e analisar todas essas fontes sob um ponto de vista comum e fornecer suporte de ferramentas para desenvolvedores. Derivou os 30 maus cheiros de boas e más práticas documentadas online na documentação do Android e de postagens em blogs de desenvolvedores que reportaram suas experiências.

²http://www.modelrefactoring.org/smell_catalog

3.4.3 Presença de maus cheiros tradicionais vs. maus cheiros específicos em aplicativos Android

Hetch [53] utilizou a ferramenta de detecção de maus cheiros Páprika³ para identificar 8 maus cheiros, sendo 4 tradicionais (*Blob Class* [10], *Swiss Army Knife* [10], *Complex Class* [22] e *Long Method* [22]) e 4 Android (*Internal Getter/Setter* [75], *No Low Memory Resolver* [75], *Member Ignoring Method* [75] e *Leaking Inner Class* [75]), em 15 aplicações Android populares como Facebook, Skype, Twitter. Isso foi possível pois a ferramenta Páprika se utiliza do APK para extrair os dados para análise e mesmo essas aplicações não sendo de código aberto, o Páprika consegue extrair os dados a partir do instalável. Um ponto importante é que apesar do autor utilizar do termo anti-padrão, ele se baseia em outras pesquisas que definiram os “anti-padrões” por ele analisado como maus cheiros de código. Logo, seguiremos com o termo mau cheiro daqui em diante, pois entendemos que apesar da diferença, o autor se refere a ele. Vale considerar que, para se classificar como um antipattern o item deve atender as 2 características mencionadas em [10] como abordamos na Seção X.X.X e não há evidências de que os itens tratados por Hetch atendem as características mencionadas.

Hetch [53] afirma que os maus cheiros tradicionais são tão frequentes em aplicativos Android como em não Android, com exceção ao *Swiss Army Knife*. Essa afirmação nos leva a entender que ele teria comparado a presença dos maus cheiros tradicionais em software tradicionais com os mesmos maus cheiros em aplicativos Android, entretanto, não há informações de como ele chegou a informação da presença de maus cheiros em projetos de software tradicionais para compará-las com o resultado obtido em aplicativos Android.

Segundo o autor, *Activities* tendem a ser mais sensíveis ao *Blob Class* [10] (muito similar a *God Class* [57] e *Large Class* [22]) que reforça a conclusão de Verloop [82]. Esta conclusão reforça nossa hipótese que códigos pertencentes a camada de apresentação Android são mais propensos a apresentar trechos problemáticos, que, apesar de já existirem maus cheiros que os identificam, a refatoração proposta não é apropriada pois é da natureza de projetos Android apresentarem estes problemas, isso nos leva a pensar que essas situações em si não são o problema de fato, e que talvez existam outras formas de definir e lidar com esses problemas no Android.

Ainda segundo o autor, maus cheiros específicos Android são muito mais frequentes do que os maus cheiros tradicionais. Esta constatação reforça a importância de se investigar quais seriam outros possíveis maus cheiros específicos de forma que, eles tendem a se manifestar mais do que os maus cheiros tradicionais.

Podemos notar algumas semelhanças nos trabalhos acima citados. A primeira semelhança importante é que diversas pesquisas que analisam a presença de maus cheiros, sejam tradi-

³<https://github.com/geoffreyhecht/paprika>

cionais ou específicos Android, sentem a necessidade de delimitar o código em análise como pôde ser visto nos trabalhos [82, 68] através do termo *classes núcleos* (ou *código núcleo*), em ambas as pesquisas significando *classes que herdaram do SDK Android*, o que consequentemente exclui todo o código puramente Java existente no projeto Android, o que faz sentido pois, no código Java puro continua sendo possível se utilizar das diversas boas práticas já existentes na literatura.

Ainda com relação à delimitação do código em estudo, outra semelhança interessante é que, apesar de a definição de classes núcleo incluir classes como Services, AsyncTasks dentre muitas outras existentes no SDK Android, as classes que apareceram nos resultados se limitaram a Activities e Adapters, ambas classes utilizadas para a construção e resposta a eventos da camada de apresentação Android.

Essas semelhanças reforçaram nossa curiosidade em focar nossa pesquisa no código relacionado à camada de apresentação Android, partindo da hipótese de que existem maus cheiros específicos a ele. De forma que esta dissertação pretende pela primeira vez catalogar maus cheiros de código relacionados a manutenibilidade, especificamente relacionados a camada de apresentação de aplicativos do Android.

Capítulo 4

Metodologia de Pesquisa

O objetivo desta pesquisa é catalogar maus cheiros da camada de apresentação Android. Desta forma, optamos por realizar uma pesquisa exploratória de gênero empírico e de abordagem mista [18].

A percepção desempenha um importante papel na definição de maus cheiros de código relacionados a uma tecnologia específica [8, 72, 89]. Principalmente considerando a natureza subjetiva intrínseca a maus cheiros [81, 20].

Nossa abordagem mista se apresenta de forma que os maus cheiros são originados a partir de dados qualitativos coletados por meio de dois questionários online respondidos por 246 desenvolvedores Android. O dado quantitativo provém de dados coletados através um experimento pelo qual avaliamos a percepção de desenvolvedores Android sobre os maus cheiros definidos. Com esses dados foi possível extrair estatísticas relacionados a percepção dos maus cheiros.

Esta pesquisa foi realizada em três etapas. Cada qual objetivando responder a uma questão principal de pesquisa. Nesta seção apresentamos detalhes sobre cada etapa bem como a qual questão de pesquisa principal que se pretendia responder.

4.1 Introdução

Esta pesquisa foi dividida em três etapas conforme apresentado na Figura 4.1. Cada etapa objetiva responder uma questão de pesquisa.

A primeira etapa objetiva responder a **QP1: Existem maus cheiros que são específicos a camada de apresentação Android?** Visto que maus cheiros possuem uma relação direta com o conhecimento empírico de desenvolvedores, optamos por um questionário exploratório online para obter os dados inciais [8, 72, 89]. Nosso objetivo foi entender o que desenvolvedores Android consideram como boas e más práticas no desenvolvimento da ca-

mada de apresentação de aplicativos Android. Obtivemos 45 respostas das quais realizamos um processo de codificação e derivamos 21 maus cheiros de código relacionados à camada de apresentação Android. Apresentamos na Seção 4.2 os detalhes desta etapa.

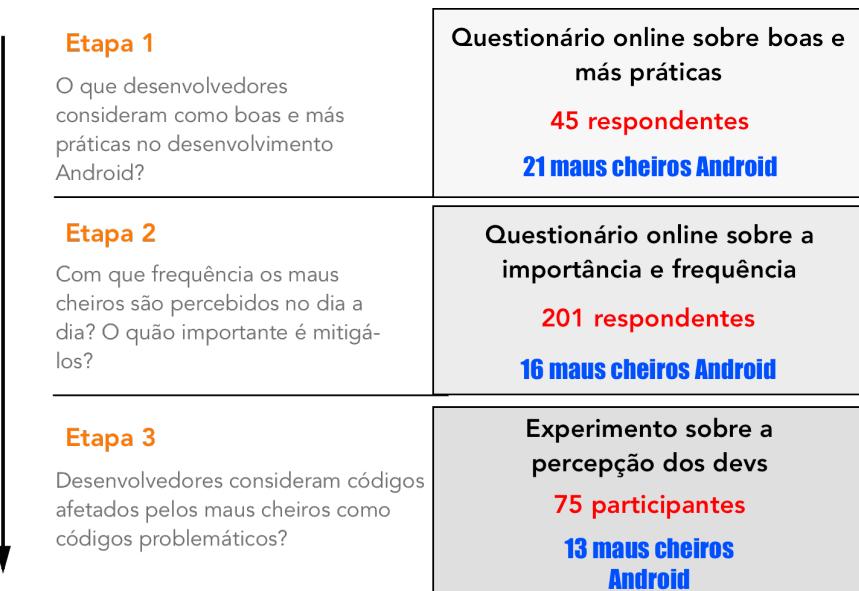


Figura 4.1: *Etapas da pesquisa.*

Na segunda etapa, objetivamos responder a **QP2. Com qual frequência os maus cheiros são percebidos e o quanto importante são considerados pelos desenvolvedores?** Os dados foram coletados a partir de um segundo questionário online respondido por 201 desenvolvedores Android. Nessa etapa foi possível validar a percepção de frequência e importância dos 21 maus cheiros no dia a dia do desenvolvimento Android, sendo todos considerados com algum nível de importância e frequência. Apresentamos os detalhes desta etapa na Seção 4.3.

Na terceira e última etapa, objetivamos responder a **QP3. Desenvolvedores Android percebem os códigos afetados pelos maus cheiros como problemáticos?** Coletamos os dados a partir de um experimento com 75 desenvolvedores Android. Com esses dados, foi possível extrair estatísticas sobre a percepção de desenvolvedores Android com relação aos 8 principais maus cheiros derivados de **QP1**. Os detalhes desta etapa são apresentados na Seção 4.4.

Por fim, concluímos com 21 maus cheiros relacionados a camada de apresentação Android percebidos e considerados importantes por desenvolvedores Android. Validamos a percepção no código dos 8 principais maus cheiros derivados e pudemos extrair dados estatísticos que comprovam essa percepção. O catálogo com esses maus cheiros é apresentado na Seção 5.1.3.

4.2 Etapa 1 - Boas e más práticas na camada de apresentação Android

Iniciamos nossa pesquisa com algumas hipóteses de quais práticas poderiam ser consideradas más práticas, e portanto exalando maus cheiros no desenvolvimento da camada de apresentação Android. Apesar de haver algumas hipóteses, com o objetivo de não limitar ou influenciar nesta primeira etapa, para responder a QP1, submetemos um questionário online com perguntas exploratórios sobre boas e más práticas em cada um dos componentes da camada de apresentação Android: *Activities*, *Fragments*, *Adapters*, *Listeners*, e recursos do aplicativo: *Layout*, *Strings*, *Styles* e *Drawables*. Obtivemos 45 respostas a partir das quais extraímos 46 categorias que resultaram em 21 maus cheiros, dentre os quais, estavam maus cheiros que confirmaram nossas hipóteses iniciais.

A seguir, na Seção 4.2.1 apresentamos detalhes sobre a concepção do questionário, na Seção 4.2.2 os detalhes sobre os participantes e na Seção 4.2.3 detalhes sobre o processo de análise dos dados. Os resultados desta etapa podem ser conferidos na Seção 5.1.2.

4.2.1 Questionário

Este questionário (*S1*) foi baseado em um estudo anterior feito por Aniche et al. [5, 7], onde os autores buscaram por maus cheiros em aplicações MVC. O questionário foi composto por 25 questões divididas em três seções.

A primeira seção teve o objetivo de traçar o perfil demográfico do participante (idade, estado de residência, experiência em desenvolvimento de software, experiência com desenvolvimento Android e escolaridade) e foi composta de 6 questões. A segunda seção teve como objetivo entender o que os desenvolvedores consideravam boas e más práticas no desenvolvimento da camada de apresentação Android. Foi composta por 16 questões opcionais e abertas, 8 sobre boas práticas em cada um dos 8 elementos da camada de apresentação Android e 8 sobre más práticas. O questionário completo pode ser visto no Apêndice B. Por exemplo, para o elemento *Activity* a segunda seção continha as seguintes perguntas:

- Você tem alguma boa prática para lidar com *Activities*? (Pergunta aberta)
- Você considera alguma coisa uma má prática ao lidar com *Activities*? (Pergunta aberta)

A terceira seção foi 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 teste piloto com 3 desenvolvedores Android. Na primeira configuração do questionário, todas as perguntas, de todas as seções com exceção do email, eram obrigatórias. Com o resultado do teste piloto, percebemos que nem sempre o desenvolvedor tem alguma boa ou má prática para comentar sobre todos os oito elementos questionados. Desta forma, removemos a obrigatoriedade das perguntas da segunda e terceira seção tornando-as opcionais e permitindo o participante responder apenas as boas e más práticas dos elementos que lhe faziam sentido. As respostas dos participantes piloto foram desconsideradas por efeitos de viés.

O questionário foi divulgado em redes sociais como Facebook, Twitter e Linkedin, em grupos de discussão sobre Android como Android Dev Brasil¹, Android Brasil Projetos² e o grupo do Slack Android Dev Br³, maior grupo de desenvolvedores Android do Brasil com 2622 participantes até o momento desta escrita⁴. O questionário esteve aberto por aproximadamente 3 meses e meio, de 9 de Outubro de 2016 até 18 de Janeiro de 2017 e foi respondido por 45 desenvolvedores.

4.2.2 Participantes

Participaram desta etapa da pesquisa 45 desenvolvedores Android. Dos participantes, 12% possui uma ou mais pós-graduações e 62% são graduados, esses dados podem ser observados na Figura 4.2a. A Figura 4.2b apresenta a distribuição de idade dos participantes, onde a maioria possui de 25 a 34 anos.

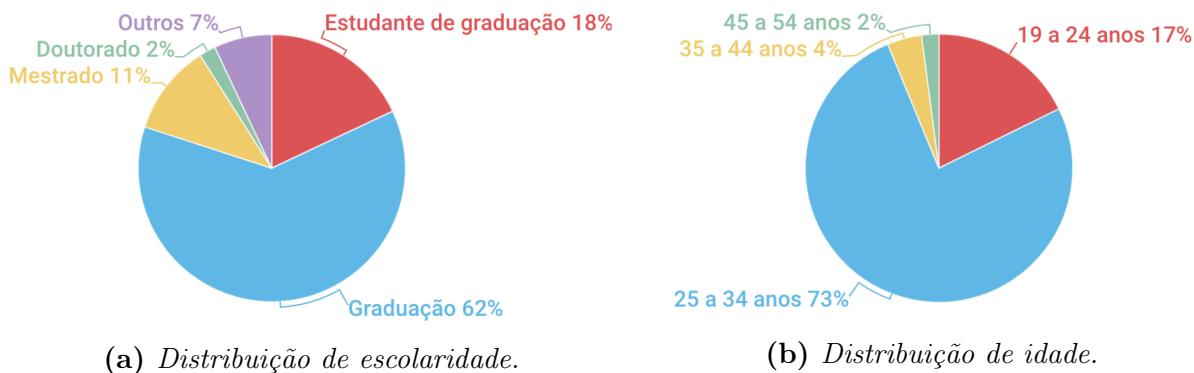


Figura 4.2: Escolaridade e distribuição de idade dos participantes em S1.

Conforme apresentado na Figura 4.3, 93% dos 45 respondentes indicaram ter 2 anos ou mais de experiência com desenvolvimento de software e 86% dos indicaram 2 anos ou mais de experiência com desenvolvimento Android. Vale ressaltar que a plataforma Android completa 10 anos em 2018, ou seja, 5 anos de experiência nessa plataforma representa 50% do tempo de existência dela desde seu anúncio em 2008.

¹<https://groups.google.com/forum/#!forum/androidbrasil-dev>

²<https://groups.google.com/forum/#!forum/android-brasil-projetos>

³<http://slack.androiddevbr.org>

⁴Dado verificado em 25/11/2017.

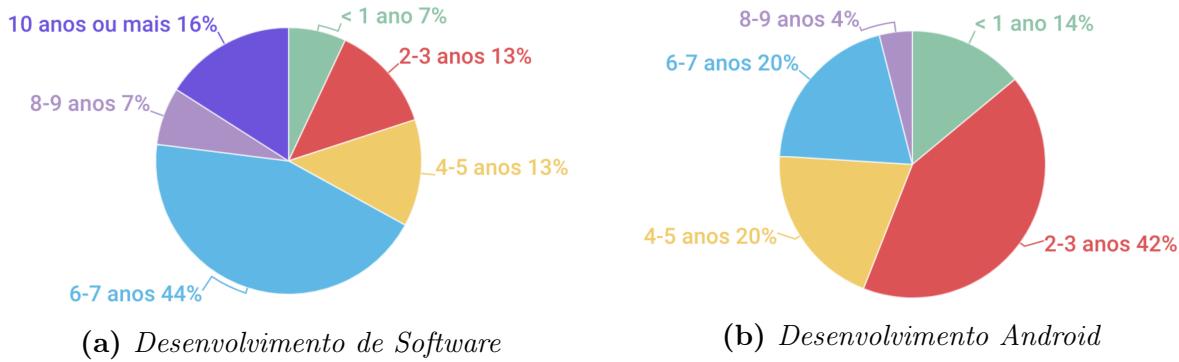


Figura 4.3: Tempo de experiência com desenvolvimento de software e desenvolvimento Android dos participantes de S1.

Nosso questionário exploratório foi respondido por profissionais de 3 continentes e mais de 7 países diferentes porém, a maior representatividade dos dados é originada do Brasil com pouco mais de 81% dos participantes de 11 estados diferentes. No Brasil, os estados com maior representatividade foram: São Paulo com 59%, Rio de Janeiro e Goiás cada um com pouco mais de 8% e Rio Grande do Sul e Santa Catarina, cada um com pouco mais de 5%. 14% dos participantes são originados de países Europeus. Nos Estados Unidos tivemos apenas 2% (1 participante) da Califórnia. Estes dados são apresentados na Figura 4.4. Podemos concluir que tivemos certa abrangência geográfica, no entanto, acreditamos que o mais apropriado seja dizer que os maus cheiros derivados das boas e más práticas representam a opinião de desenvolvedores brasileiros.

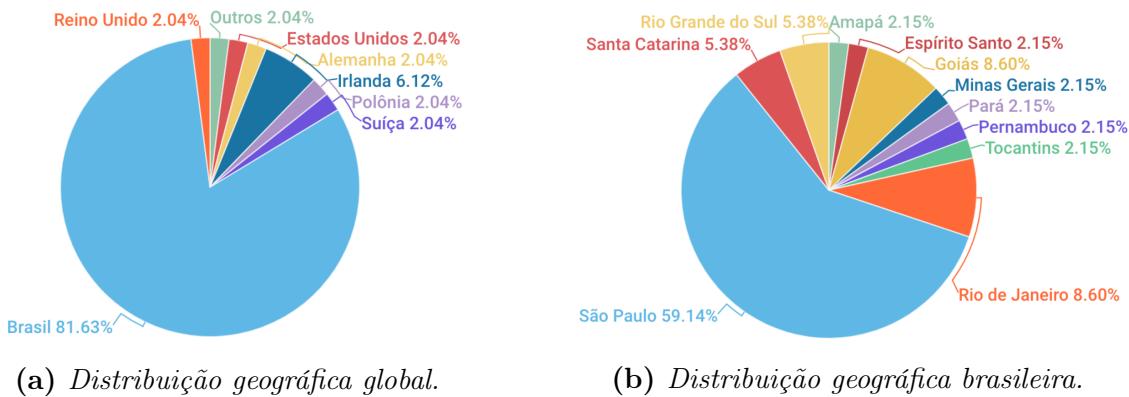


Figura 4.4: Distribuição geográfica global e brasileira dos participantes de S1.

4.2.3 Análise dos Dados

Para análise dos dados nos inspiramos na abordagem *Ground Theory* (GT), um método de pesquisa exploratória originada nas ciências sociais [17, 25], mas cada vez mais popular em pesquisas de engenharia de software [2]. 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. 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*, detalhados a seguir.

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 na camada de apresentação 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, respostas contendo frases como "*Não*", "*Não que eu saiba*", "*Eu não me lembro*" e similares, as consideradas vagas como "*Eu não tenho certeza se são boas práticas mas uso o que vejo por ai*", as consideradas genéricas como "*Como todo código java...*" e as que não eram relacionadas a boas ou más práticas de código. Das 810 boas e más práticas, 352 foram consideradas e 458 desconsideradas. Das 352, 45% foram apontadas como más práticas e 55% como boas práticas.

Em seguida, realizamos a *codificação* sobre as boas e más práticas [17, 76]. 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 [17]. Durante esse processo, cada resposta recebeu uma ou mais categorias. Neste passo desconsideramos mais algumas respostas, isso ocorreu pois não eram respostas “obviamente” descartáveis como as do passo anterior e foi necessária a análise para chegarmos a esta conclusão de desconsiderá-las. Para cada resposta desconsiderada nesse passo registramos o motivo, este 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 359 respostas sobre boas e más práticas individualmente categorizadas em 46 categorias. Para derivação dos maus cheiros foram consideradas 21 categorias que apresentaram frequência de recorrência maior ou igual a cinco. Segundo Nielsen, cinco repetições é o suficiente para se obter dados necessários para definir um problema, as

repetições seguintes tendem a não agregar novas informações relevantes, se não as já observadas [69]. Cada categoria considerada resultou em um mau cheiro de código Android, dos quais 11 são relacionados a componentes da camada de apresentação Android e 10 relacionados a recursos do aplicativo, totalizando 21 maus cheiros na camada de apresentação Android.

As descrições dos maus cheiros foram embasadas nas respostas obtidas, por exemplo, algumas respostas que embasaram o mau cheiro COMPONENTE DE UI INTELIGENTE (mais detalhes sobre os maus cheiros propostos na Seção 5.1.3) foram “*Fazer lógica de negócio /em Activities/*”⁵ (P16). “*Colocar regra de negócio no Adapter*” (P19), “*Manter lógica de negócio em Fragments*” (P11), “*Elas [Activities] representam uma única tela e apenas interagem com a UI, qualquer lógica deve ser delegada para outra classe*” (P16) onde de P1 a P45 representam cada um dos 45 respondentes. De modo a tornar a leitura dos maus cheiros mais enxuta, o catálogo com os maus cheiros é apresentado separado dos exemplos de respostas usadas para embasá-los, este por sua vez está disponível no Apêndice A.

4.3 Etapa 2 - Frequência e importância dos maus cheiros

Para responder a QP2, buscamos generalizar os maus cheiros encontrados na QP1 entendendo a percepção dos desenvolvedores com relação a frequência e importância dos maus cheiros. Coletamos esta percepção por meio de um questionário online com três seções sendo a primeira para coleta de dados demográficos da mesma forma da etapa 1, a segunda para coleta da percepção de frequência dos maus cheiros no dia a dia de desenvolvimento e a terceira para coleta da percepção de importância.

Coletamos ao todo 201 respostas de desenvolvedores Android de 3 continentes e 14 países diferentes. Nossos resultados mostraram que os 21 maus cheiros são considerados de modo relevante, frequentes e importantes. A seguir, na Seção 4.3.1 apresentamos detalhes sobre a concepção do questionário, na Seção 4.3.2 detalhes sobre os participantes e na Seção 4.3.3 detalhes sobre o processo de análise dos dados. Os resultados desta etapa podem ser conferidos na Seção 5.2.

4.3.1 Questionário

Foram criadas duas versões do questionário, inglês e português, de forma que de um era possível acessar o outro, possibilitando o participante escolher o idioma mais apropriado a ele. O questionário (S2) continha 3 seções. A primeira seção, como em S1, tinha o objetivo de traçar o perfil demográfico do participante (idade, estado de residência, experiência em desenvolvimento de software, experiência com desenvolvimento Android e escolaridade) e foi

⁵Todo texto em inglês foi traduzido livremente ao longo da dissertação

composta de 6 questões.

A segunda seção objetivou capturar a percepção dos desenvolvedores com relação a frequência em que eles percebiam os maus cheiros no seu dia a dia. Fizemos isso apresentando uma lista de afirmações, onde cada afirmação descrevia o sintoma de um dos maus cheiros, e pedimos para o participante indicar com qual frequência ele percebia cada uma das situações listadas no seu dia a dia. Para cada afirmação o participante podia escolher uma entre cinco opções da escala *likert* de frequência: muito frequente, frequente, as vezes, raramente e nunca.

As afirmações foram baseadas nas respostas em S1, por exemplo, para o mau cheiro CLASSES DE UI ACOPLADAS algumas das respostas sobre más práticas foram “*Acoplar o Fragment a Activity ao invés de utilizar interfaces é uma prática ruim*” (P19) e “*Acoplar o Fragment com a Activity*” (P10, P31 e P45), e boa prática foi “*Seja um componente de UI reutilizável. Então evite dependência de outros componentes da aplicação*” (P6). Com base nessas e outras respostas extraímos a seguinte frase para representar o sintoma do mau cheiro:

- *Fragments, Adapters ou Listeners* com referência direta para quem os usa, como *Activities* ou outros *Fragments*.

Para contemplar os 21 maus cheiros foram apresentadas 26 afirmações, essa diferença do número total de maus cheiros, que somam 21, e o número de afirmações se dá pois, para os maus cheiros COMPORTAMENTO SUSPEITO, COMPONENTE DE UI ZUMBI, LAYOUT LONGO OU REPETIDO, LONGO RECURSO DE ESTILO e ATRIBUTOS DE ESTILO REPETIDOS identificamos nas respostas em S1 mais de um sintoma que os representavam. Com o objetivo de entender qual(is) sintoma(s) era percebido no dia a dia pelo desenvolvedor, optamos por apresentar mais de uma afirmação para cada um deles, sendo que cada afirmação abordou um dos sintomas. Por exemplo, para o mau cheiro COMPORTAMENTO SUSPEITO apresentamos três afirmações, onde cada uma representou uma forma diferente, considerada como má prática pelos participantes de S1, sobre como implementar a resposta a eventos do usuário no Android:

- *Activities, Fragments ou Adapters* com classes anônimas para responder a eventos do usuário, como clique, duplo clique e outros.
- *Activities, Fragments ou Adapters* implementando algum *Listener*, através de polimorfismo (*implements*), para responder a eventos do usuário como clique, duplo clique e outros.
- *Activities, Fragments ou Adapters* com classes internas implementando algum *Listener* para responder a eventos do usuário como clique, duplo clique e outros.

A terceira seção objetivou capturar a percepção dos desenvolvedores com relação a importância em mitigar os sintomas dos maus cheiros, para isso foi solicitado que indicasse o quanto importante ele considerava as afirmações apresentadas. Para contemplar os 21 maus cheiros, apresentamos 22 afirmações que basicamente negavam as afirmações relacionadas à percepção de frequência do mau cheiro. A divergência do total de maus cheiros e o total de afirmações apresentadas se dá pelo mesmo motivo encontrado na seção anterior do questionário, sobre percepção de frequência. Para cada afirmação o participante podia escolher uma dentre as cinco opções da escala *likert* de importância: muito importante, importante, razoavelmente importante, pouco importante, não é importante. A seguir apresentamos a afirmação apresentada nesta seção para o mau cheiro CLASSES DE UI ACOPLADAS:

- *Fragments, Adapters e Listeners* não ter referência direta à quem os utiliza.

Em nenhuma das seções indicamos que seriam apresentados sintomas de maus cheiros, nem mencionamos os nomes dos maus cheiros usados nesta dissertação, foram apresentadas somente as afirmações. Optamos por fazer desta forma para abstrair a ideia de que as frases representavam maus cheiros e portanto, não exigir do participante um conhecimento prévio sobre maus cheiros de código.

Antes da divulgação do questionário realizamos a validação de cada uma das afirmações, por meio de entrevista individual, com dois desenvolvedores Android experientes, *DEV-A* e *DEV-B*. *DEV-A* possui 10 anos de experiência com desenvolvimento de software e 5 anos de experiência com desenvolvimento Android, se considera proficiente nas tecnologias Java, Objective C, Swift e Android e possui grau escolar de bacharel. *DEV-B* possui 7 anos de experiência com desenvolvimento de software e 6 anos de experiência com desenvolvimento Android, se considera proficiente nas tecnologias Java, Objective C e Android e possui grau escolar de bacharel. Ambos os desenvolvedores trabalham atualmente com desenvolvimento mobile em *startups* brasileiras.

A entrevista de validação das afirmações foi conduzida de modo que o desenvolvedor procedia respondendo o questionário e quando tinha dúvida sobre uma frase, o mesmo questionava e então ela era discutida e quando necessário, reestruturada ou reescrita, de acordo com o *feedback* dado. Após a discussão o desenvolvedor a respondia, considerando a conclusão das discussões, e seguia para a próxima. Esta dinâmica segue até passar por todas as afirmações e ao final o desenvolvedor era incentivado a dar qualquer outro *feedback* que considerasse relevante.

Alguns importantes ajustes foram realizados após a primeira entrevista de validação. O principal ajuste foi relacionado a escala *likert* utilizada, sendo que, na primeira versão do questionário, respondido por *DEV-A*, foi usada tanto para validar importância quanto para validar frequência a mesma escala *likert* de nível de concordância (concordo totalmente, concordo, não concordo nem discordo, discordo, discordo totalmente), deixando para a frase a

responsabilidade de indicar do que se tratava, frequência ou importância. Durante a validação percebemos que as afirmações ficavam muito maiores de ler e difíceis de serem interpretadas, então trocamos para escalas de frequência e importância e ajustamos as afirmações. Também foram sugeridas melhorias em algumas poucas afirmações. Com o segundo desenvolvedor, *DEV-B*, tivemos apenas dois ajustes em afirmações. Para efeitos de viés, as respostas dos desenvolvedores que participaram das entrevistas de validação foram desconsideradas.

Após a validação das afirmações, executamos um teste piloto com dois desenvolvedores, do qual, não solicitaram alterações no questionário, considerando-o apropriado para lançamento. Essas respostas foram consideradas visto que o *feedback* foi positivo e não houve necessidade de ajustes. O questionário foi divulgado em redes sociais como Reddit, Facebook, Twitter e Linkedin, grupos de discussão sobre Android como Android Dev Brasil⁶, Android Brasil Projetos⁷ e o grupo do Slack Android Dev Br⁸, maior grupo de desenvolvedores Android do Brasil com 2622 participantes até o momento desta escrita. Para engajar os participantes ofertamos o sorteio de um Google Chromecast Áudio.

O questionário esteve aberto por aproximadamente três semanas em meados de Setembro de 2017. As afirmações eram apresentadas de forma randômica. Ao final, o questionário foi respondido por 201 desenvolvedores. A versão completa do questionário pode ser conferida no Apêndice C.

4.3.2 Participantes

Participaram desta etapa da pesquisa 201 desenvolvedores. 145 respostas vieram da versão em português do questionário e 56 da versão em inglês. Dos participantes, 15% possuem uma ou mais pós-graduações e 61% são graduados, esses dados podem ser observados na Figura 4.5a. A Figura 4.5b apresenta o histograma de idade dos participantes, onde podemos observar que tivemos uma abrangência considerável de idade, onde a maior parte possui de 20 a 35 anos.

Conforme apresentado na Figura 4.6, 94% participantes indicaram ter 2 anos ou mais de experiência com desenvolvimento de software e 74% indicaram 2 anos ou mais de experiência com desenvolvimento Android. Este resultado nos indica que atingimos desenvolvedores com experiência considerável em desenvolvimento de software e Android, entendemos este resultado de forma positiva.

Perguntamos aos desenvolvedores qual seu nível de conhecimento em diversas linguagens orientadas a objetos. Na Figura 4.7 podemos observar que mais de 80% afirmam ter conhecimento de intermediário a avançado em Java e Android. De 20% a 58% dos participantes afirmam ter conhecimento de intermediário a avançado em outras linguagens orientadas a

⁶<https://groups.google.com/forum/#!forum/androidbrasil-dev>

⁷<https://groups.google.com/forum/#!forum/android-brasil-projetos>

⁸<http://slack.androiddevbr.org>

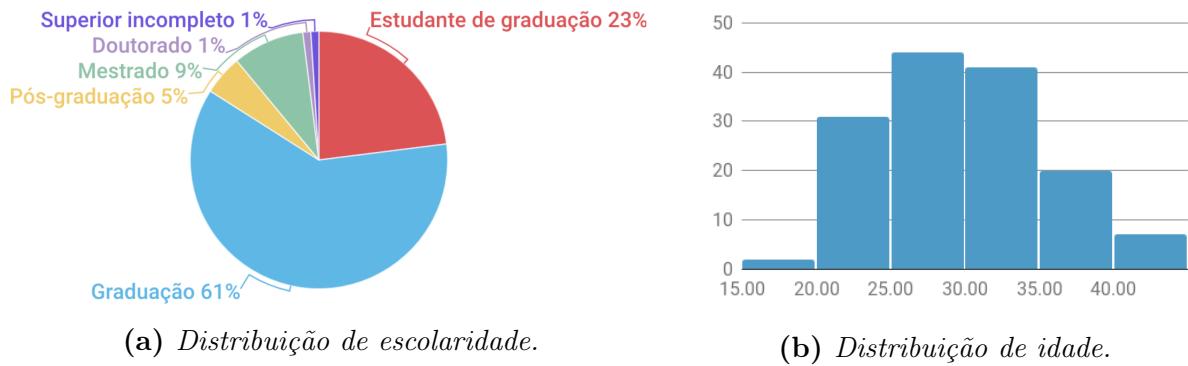


Figura 4.5: Escolaridade e distribuição de idade dos participantes em S2.

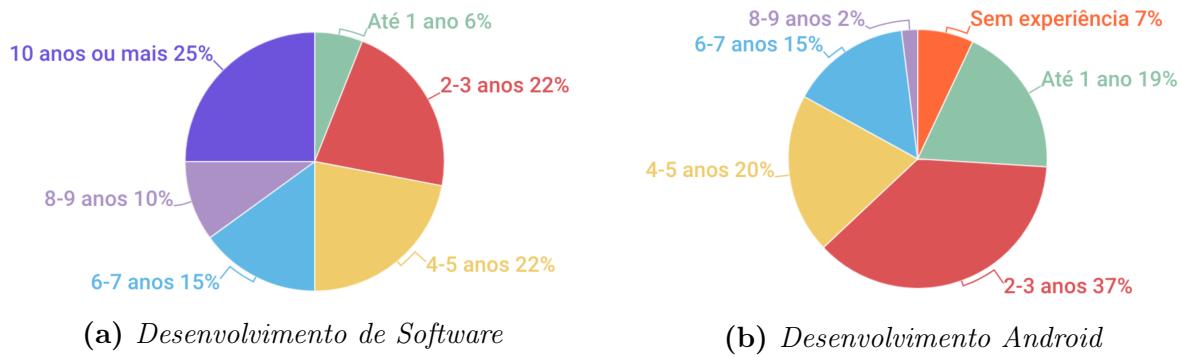


Figura 4.6: Experiência com desenvolvimento de software e desenvolvimento Android dos participantes de S2.

objetos.

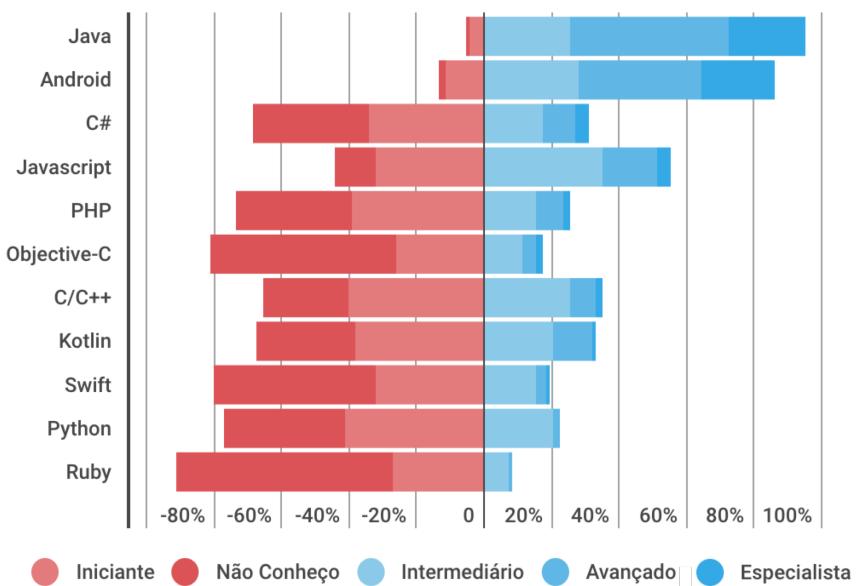
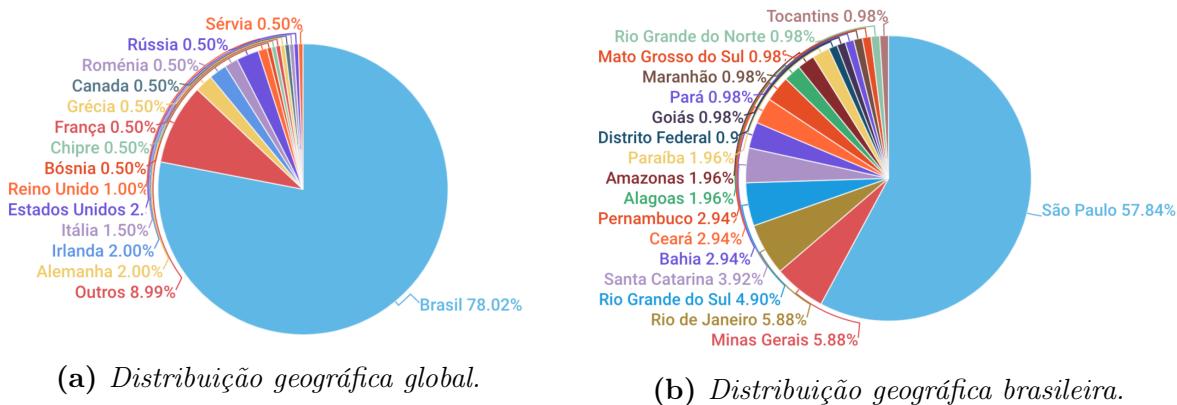


Figura 4.7: Nível de conhecimento em diversas linguagens de programação orientada a objetos dos participantes de S2.

Nosso questionário foi respondido por profissionais de 3 continentes e 14 países diferentes porém a maior representatividade dos dados é originada do Brasil, com pouco mais de 78%

dos participantes. Os brasileiros somam 157 profissionais de 18 estados diferentes. Desse total, os estados com maior representatividade foram, na ordem: São Paulo com 57%, Rio de Janeiro e Minas Gerais com quase 6% cada e Rio Grande do Sul com quase 5%. 10% dos participantes são originados de países Europeus. Nos Estados Unidos tivemos 2% provindos da Califórnia, Pensilvânia, Indiana e Utah e 0,5% (1 participante) de Ontário no Canadá. A Figura 4.8 apresenta a distribuição geográfica global e brasileira dos participantes.



(a) Distribuição geográfica global.

(b) Distribuição geográfica brasileira.

Figura 4.8: Distribuição geográfica global e brasileira dos participantes de S2.

Podemos concluir que tivemos certa abrangência geográfica porém o padrão geográfico obtido em S1 se repetiu em S2, de que apesar de termos tido esta abrangência, a maior representatividade é provinda do Brasil, logo, continuamos acreditando que o mais apropriado seja dizer que a percepção de frequência e importância dos maus cheiros derivados representam a opinião de desenvolvedores brasileiros.

4.3.3 Análise dos Dados

Para análise dos dados, as opções das escalas *likert* de **frequência** e **importância** foram codificadas em números de 1 a 5. Por exemplo, as opções “Nunca” (da escala de frequência) e “Não é importante” (da escala de importância) foram codificadas com 1, as opções “Raramente” (frequência) e “Pouco importante” (importância) com 2, e assim sucessivamente até as opções “Muito frequente” (frequência) e “Muito importante” (importância) que foram codificadas com 5.

De modo a avaliar a percepção dos desenvolvedores Android sobre a frequência e importância dos maus cheiros derivados, para análise dos dados extraímos de cada afirmação a média, moda e desvio padrão. Para os maus cheiros que apresentamos mais de uma afirmação, extraímos a média desses dados para representá-los. Por exemplo, foram apresentadas três afirmações de frequência sobre o mau cheiro COMPORTAMENTO SUSPEITO, a moda que o representa é a média das modas de cada afirmação, que neste caso foi $(3+4+4)/3 = 3,7$, que arredondamos para 4.

De modo a avaliar quais maus cheiros são considerados frequentes realizamos uma simplificação da escala *likert* de **frequência**. Classificamos de **frequência baixa** as resposta “Raramente”, de **frequência moderada** as respostas “Ás vezes” e por último, classificamos de **frequência alta** as respostas “Frequente” e “Muito frequente”. Não obtivemos nenhuma afirmação com moda (MO) igual 1, ou seja, “Nunca”, e por isso não criamos uma classificação para ela.

De modo similar, para avaliar quais maus cheiros são considerados importantes, também realizamos uma simplificação da escala *likert* de **importância**. Classificamos de **importância baixa** as resposta “Pouco importante”, de **importância moderada** as respostas “Razavelmente importante” e classificamos de **importância alta** as respostas de “Importante” e “Muito importante”. Também não foi necessário criar uma classificação para respostas “Não é importante” pois nenhuma das afirmações obteve MO igual a 1.

4.4 Etapa 3 - Percepção dos maus cheiros

Para responder a **QP3** validamos a percepção de desenvolvedores Android sobre códigos afetados por sete maus cheiros, considerados mais importantes e frequentes (vide Tabela 4.1), dos 21 propostos. Os dados foram coletados por meio de um experimento de código online onde cada participante era solicitado a avaliar a qualidade de seis códigos relativos a camada de apresentação Android. Obtivemos um total de 70 respostas. Vale ressaltar que este experimento já foi realizado em pesquisa similar, realizada por Aniche et al. [5, 6], relacionada a maus cheiros em aplicações com Spring MVC.

Nossos resultados confirmam estatisticamente que desenvolvedores Android percebem cinco dos sete maus cheiros avaliados. Não obtivemos dados suficientes para tirar conclusões estatísticas sobre os outros dois. A seguir, na Seção 4.4.1 apresentamos detalhes sobre o experimento, na Seção 4.4.2 detalhes sobre os participantes e na Seção 4.4.3 detalhes sobre o processo de análise dos dados. Os resultados desta etapa podem ser conferidos na Seção 5.3.

4.4.1 Experimento

Devido a limitações de tempo e quantidade de respondentes, foi possível coletar dados suficientes para validar estatisticamente a percepção sobre 7 dos 21 maus cheiros propostos. Para definirmos quais maus cheiros seriam avaliados no experimento, usamos o seguinte critério: todos os maus cheiros com MO de importância e frequência maior ou igual a 3 e DP de importância e frequência inferior a 1,20. Conforme obtínhamos os dados, este critério poderia ser relaxado para acomodar mais maus cheiros. A Tabela 4.1 lista os maus cheiros avaliados no experimento.

O experimento de código foi divulgado em redes sociais como Twitter e o grupo do Slack

Tabela 4.1: Sete maus cheiros avaliados no experimento de código sobre a percepção de desenvolvedores Android.

Mau Cheiro	Importância			Frequência		
	ME	MO	DP	ME	MO	DP
COMPONENTE DE UI INTELIGENTE	5	5	1,05	3	4	1,19
COMPONENTE DE UI ACOPLADO	4	5	1,02	3	3	1,15
ADAPTER COMPLEXO	4	5	0,91	3	3	1,15
LONGO RECURSO DE ESTILO	4	4	1,06	4	5	1,18
COMPORTAMENTO SUSPEITO	3	4	1,19	3	4	1,19
LAYOUT PROFUNDAMENTE ANINHADO	4	4	1,12	4	4	1,06
ATRIBUTOS DE ESTILO REPETIDOS	4	4	0,86	4	4	1,11

DP = Desvio Padrão, MO = Moda, ME = Média.

Android Dev Br⁹ e ficou aberto por 8 dias. O experimento continha duas seções principais. A primeira seção teve como objetivo coletar informações demográficas, principalmente com relação a experiência dos participantes. Na segunda seção os participantes foram solicitados a analisar seis códigos, quatro “*mau cheirosos*”, ou seja, afetados por um dos sete maus cheiros avaliados, e dois “*limpos*”, ou seja, não afetado pelos maus cheiros avaliados. E para cada um deles, responder as seguintes perguntas:

P1 Na sua opinião, este código apresenta algum problema de design e/ou implementação?

Respostas possíveis: SIM/NÃO

P2 Se SIM, por favor explique quais são, na sua opinião, os problemas que afetam este código. Pergunta aberta.

P3 Se SIM, por favor avalie a severidade do problema de design e/ou implementação selecionando dentre as opções a seguir. Respostas possíveis sendo uma escala *Likert* de 5 pontos de 1 (muito baixa) até 5 (muito alta).

Os seis códigos apresentados foram randomicamente selecionados de um conjunto de 50 códigos. Esse conjunto foi composto por 35 códigos mau cheirosos (cinco para cada mau cheiro avaliado) e 15 códigos limpos. Vale salientar que, para mitigar possíveis viés de confundimento, nos dois grupos de códigos, selecionamos apenas *Activities*, *Fragments*, *Adapters*, *Listeners*, recursos de *Layout*, *String* e *Style*, uma vez que são esses os elementos da camada de apresentação Android, cujo os maus cheiros propostos e avaliados tratam.

Os códigos usados no experimento foram extraídos de projetos de software livre Android selecionados aleatoriamente do repositório F-Droid¹⁰. A Tabela 4.2 apresenta a lista dos projetos usados, bem como informações sobre avaliação (estrelas), total de instalações e versão atual do aplicativo na Google Play Store, loja oficial de aplicativos Android, para os aplicativos que lá estão disponíveis.

⁹<http://slack.androiddevbr.org>

¹⁰f-droid.org é um repositório que lista projetos gratuitos e software livre Android (FOOS, do inglês *Free and Open Source Software*).

Tabela 4.2: Listagem dos nove projetos de software livre Android usados para coletar os códigos usados no experimento.

Código-Fonte	Google Play Store		
	Estrelas	Instalações	Versão
https://github.com/uberspot/2048-android	4.2	1.000.000 - 5.000.000	2.08
https://github.com/jerekSEL/Bucket	4.2	1.000 - 5.000	0.2.1-play
https://github.com/TeamAmaze/AmazeFileManager	4.3	500.000 - 1.000.000	3.2.1
https://gitlab.com/cfabio/AltcoinPrices			
https://github.com/pinetum/AirUnlock-for-Android			
https://github.com/SecUSo/privacy-friendly-weather	3.8	1.000 - 5.000	1.1
https://github.com/Xlythe/Calculator			
https://github.com/mkulesh/microMathematics	4.7	500 - 1.000	2.15.6
https://github.com/openfoodfacts/openfoodfacts-androidapp	4.0	1.000 - 5.000	0.7.4

A seleção dos códigos usados no experimento foi feita de forma manual, pois como os maus cheiros estão sendo propostos nesta pesquisa, ainda não existem heurísticas definidas ou ferramentas que os detectem automaticamente em projetos Android. Para cada mau cheiro a ser avaliado, buscamos nos projetos pelo tipo de código que poderia ser afetado por ele. Por exemplo, o mau cheiro ATRIBUTOS DE ESTILO REPETIDOS pode afetar recursos de *Layout* ou *Style*, mas não *Activities*, logo, para encontrar códigos afetados por ele precisamos olhar em recursos de *Layout* ou *Style*. Após encontrado, o código era analisado de modo a verificar se apresentava algum dos sintomas do mau cheiro, conforme descritos na Seção 5.1.3. Caso o código apresentasse o sintoma, o mesmo era separado para ser usado no experimento.

Repetimos esse procedimento para cada mau cheiro a ser avaliado, passando por cada um dos projetos selecionados. Quando não encontrávamos códigos nos projetos já selecionados, selecionávamos aleatoriamente outro projeto e repetíamos a busca pelo mau cheiro. Deste modo, nove projetos foram necessários para encontrarmos códigos exemplares, maus cheirosos e limpos, para os sete maus cheiros.

Para cada código separado para o experimento, foi criado um *gist*¹¹. A lista com todos os *gists* pode ser visualizada no Apêndice F. Pequenas modificações foram feitas nos códigos com o objetivo de isolar o mau cheiro a ser avaliado e reduzir o esforço cognitivo do participante. Para reduzir esforço cognitivo, removemos declarações de *packages*, *imports* e comentários. Para isolar o mau cheiro a ser avaliado, diversas e variadas pequenas mudanças foram realizadas a depender do código e do mau cheiro a ser isolado. Deste modo, cada *gist* com código mau cheiroso, apresentava apenas um dos maus cheiros avaliados.

Para mitigar possíveis viés de seleção, foram criados cinco *gists* diferentes para cada mau cheiro, totalizando 35 *gists* mau cheirosos diferentes. Sobre os *gists* com códigos limpos, selecionamos cinco códigos de componentes da camada de apresentação Android, dentre eles *Activities*, *Fragments*, *Adapters* e *Listeners*, cinco recursos de *Layout* e cinco recursos de

¹¹ *Gist* é uma forma de compartilhar conteúdo, inclusive trechos de código, pela plataforma GitHub [1].

Style, totalizando 15 *gists* com códigos limpos. Nenhum dos maus cheiros avaliados afetava recursos de *String* ou *Drawables*, por este motivo não foram selecionados códigos desses tipos.

Para reduzir efeitos de aprendizagem e viés de ordem, cada participante recebeu os seis códigos randomicamente selecionados, em ordem aleatória. Como nossas necessidades para o experimento eram bem peculiares, foi desenvolvido um software específico¹² para aplicá-lo. Além disso, os participantes não foram informados de quais códigos pertenciam a quais grupos (maus cheirosos ou limpos). Os participantes foram informados apenas de que a pesquisa tinha como objetivo estudar a qualidade de códigos da camada de apresentação de aplicativos Android nativos. Não foi imposto nenhum limite de tempo para completar a tarefa.

4.4.2 Participantes

4.4.3 Análise dos Dados

Para comparar as distribuições de severidade indicada pelos respondentes para os dois grupos de códigos, utilizamos o teste não pareado de Mann-Whitney [16]. Este teste é usado para analisar a significância estatística das diferenças entre a severidade atribuída pelos respondentes aos problemas que eles observaram nos códigos mau cheirosos e limpos. Os resultados são considerados estatisticamente significativos em $\alpha \leq 0,05$. Também estimamos a magnitude das diferenças medidas usando o Cliff's Delta (d), uma medida de tamanho de efeito não paramétrico [52] para dados ordinais. Seguimos diretrizes bem estabelecidas para interpretar os valores do tamanho do efeito, sendo: desprezível para $|d| < 0,14$, pequeno para $0,14 \leq |d| < 0,33$, médio para $0,33 \leq |d| < 0,474$, e grande para $|d| \geq 0,474$ [52]. Finalmente, relatamos resultados qualitativos derivados das respostas abertas dos participantes.

4.5 Ameaças à Validade

4.5.1 Internas

Uma limitação desta pesquisa 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

¹²Repositório do software desenvolvido para aplicação do experimento de código: github.com/suelengc/code-experiment-survey-app

forma a reduzir possíveis viés.

A seleção dos códigos usados em S3 foi feita buscando pelos sintomas que derivamos das respostas em S1. Estamos cientes de que essa seleção manual pode introduzir imprecisões. De modo a mitigar este problema, selecionamos cinco diferentes códigos maus cheirosos para cada mau cheiro analisado. Outras pesquisas precisam ser conduzidas de forma a automatizar esta detecção, removendo imprecisões de processos manuais. Entretanto, mesmo com a seleção manual pudemos validar a percepção correta do mau cheiro avaliado com base nas respostas dissertativas sobre o problema percebido no código.

Dessa forma, podemos concluir que tivemos certa abrangência geográfica no entanto, acreditamos que o mais apropriado seja dizer que os maus cheiros derivados das boas e más práticas representam a opinião de desenvolvedores brasileiros.

4.5.2 Externas

- Majoritariamente brasileiros.
- Não conseguimos garantir que desenvolvedores que participaram de uma das etapas não tenha participado das outras e vice e versa.

Capítulo 5

Resultados

5.1 QP1: Existem maus cheiros que são específicos a camada de apresentação Android?

Nossos resultados mostraram a existência de uma percepção comum entre desenvolvedores sobre más práticas no desenvolvimento da camada de apresentação Android. Considerando que maus cheiros derivam do conhecimento empírico de desenvolvedores, entendemos que faz sentido afirmar que **sim, existem maus cheiros na camada de apresentação Android.**

Recebemos um total de 45 respostas. 80% dos participantes responderam pelo menos 3 perguntas, apenas 20% responderam uma (2 participantes) ou nenhuma (7 participantes) pergunta. A questão do email era opcional, mas 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 seguir, na Seção 5.1.1 apresentamos alguns resultados gerais. Na Seção 5.1.2 abordamos o processo de definição dos maus cheiros. E por último, na Seção 5.1.3 apresentamos o catálogo derivado com 21 maus cheiros da camada de apresentação Android.

5.1.1 Resultados gerais e descobertas

Todas as perguntas da segunda seção de *S1* eram opcionais, de modo que umas receberam mais respostas do que outras. A Figura 5.1 apresenta o total de respostas recebidas por cada uma das 16 perguntas sobre boas e más práticas apresentadas na segunda seção de *S1*. Por exemplo, as duas colunas sobre *Activities* representam Q1 e Q2, respectivamente perguntando sobre boas e más práticas em *Activities*. Podemos observar que 36 dos 45 participantes responderam a pergunta sobre **boas práticas** em *Activities*, enquanto que 35 responderam sobre **más práticas**. O elemento que recebeu menos respostas sobre **boas prá-**

ticas foi o recurso *Style*, sendo respondida por 23 dos participantes. O elemento que menos recebeu respostas sobre **más práticas** foi o recurso *Drawable*, sendo respondido por 21 dos participantes. Todas as perguntas obtiveram respostas. Esse resultado é um indício de que desenvolvedores Android possuem um arsenal pessoal de práticas no desenvolvimento da camada de apresentação Android que consideram boas e más.

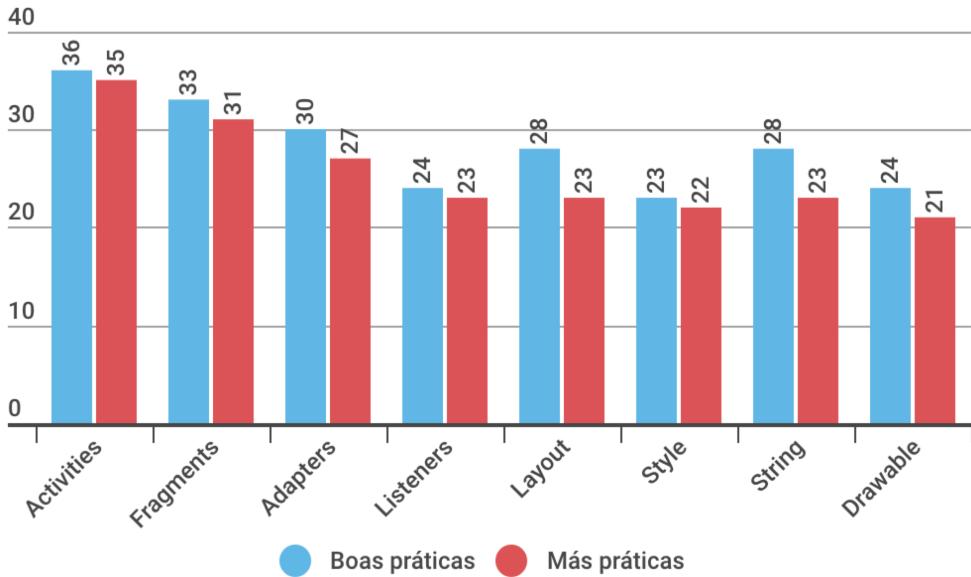


Figura 5.1: Total de respostas para cada pergunta sobre boas e más práticas nos oito elementos da camada de apresentação Android, apresentadas na segunda seção de S1.

Componentes da camada de apresentação Android (*Activities*, *Fragments*, *Adapters* e *Listeners*), receberam de modo geral, mais respostas do que recursos da aplicação (*Layouts*, *Styles*, *Strings* e *Drawables*). Dentre os componentes, os que mais receberam respostas foram *Activities* e *Fragments*, ambos tendo recebido respostas tanto de boas e más práticas de mais de 31 dos 45 participantes. Dentre os recursos da aplicação, os que mais obteve respostas foram os recursos de *Layout* e *String*, ambos tendo recebido respostas tanto de boas e más práticas de mais de 23 dos 45 participantes. Notamos também que as perguntas sobre boas práticas foram mais respondidas do que as perguntas sobre más práticas em todos os elementos Android.

O processo de codificação resultou em 46 categorias, das quais consideramos para a derivação dos maus cheiros, todas as que apresentaram ocorrência maior ou igual a cinco, com base no número de Nielsen [69]. Com base nesse critério, 23 categorias foram consideradas. Dessas, desconsideramos mais duas por se tratarem de 1) um mau cheiro tradicional (*Large Class*) e 2) um aspecto da orientação a objetos (Herança). Resultando em 21 categorias para a derivação dos maus cheiros da camada de apresentação Android.

É interessante notar que nosso processo de codificação também resultou em conclusões **similares** as de algumas pesquisas anteriores, de que aplicativos Android são fortemente afetados pelo mau cheiro tradicional *Large Class* conforme citado por Verloop [82], e que

é pouco ou quase nada usado herança para estruturar o *design* do código, conforme citado por Minelli e Lanza [68]. Como nosso foco não estava em avaliar a presença de maus cheiro tradicionais ou práticas de orientações a objetos em aplicativos Android, não trabalhamos em cima desses resultados.

5.1.2 Recorrência dos Maus Cheiros

As Tabelas 5.1 (componentes) e 5.2 (recursos) apresentam os maus cheiros derivados, o total de ocorrências por elemento da camada de apresentação Android, ou seja, de quais perguntas sobre boas e más práticas aquele mau cheiro originou, e sua ocorrência total geral, sendo a somatória das ocorrências em todos os elementos.

Na Tabela 5.1 temos os 11 maus cheiros de componentes da camada de apresentação Android. Podemos interpretar esta tabela do ponto de vista do componente. Se ele apresenta ocorrência em algum mau cheiro, significa que ele é suscetível a ser afetado por aquele mau cheiro. Ou seja, *Activities* e *Fragments* são os componentes mais suscetíveis a diferentes maus cheiros da camada de apresentação Android, totalizando 7. Já *Adapters* estão pouco menos suscetíveis, aparecendo em 6 maus cheiros. Em contrapartida, *Listeners* são os componentes da camada de apresentação menos suscetíveis a maus cheiros, aparecendo em apenas 5.

Tabela 5.1: *Maus cheiros em componentes da camada de apresentação Android, sua ocorrência em cada componente e ocorrência total.*

Mau Cheiro	Activity	Fragment	Adapter	Listener	#Total
COMPONENTE DE UI INTELIGENTE	29	16	14	1	60
COMPONENTE DE UI ACOPLADO	2	10	3	3	18
COMPORTAMENTO SUSPEITO	4	-	3	10	17
ADAPTER CONSUMISTA	-	-	13	-	13
COMPONENTE DE UI ZUMBI	7	3	-	-	10
USO EXCESSIVO DE FRAGMENTS	-	9	-	-	9
COMPONENTE DE UI FAZENDO IO	5	3	1	-	9
NÃO USO DE FRAGMENT	4	4	-	-	8
ADAPTER COMPLEXO	-	-	5	-	6
AUSÊNCIA DE ARQUITETURA	4	2	-	-	6

Vale observar que um mesmo mau cheiro pode afetar mais de um componente da camada da apresentação Android. Por exemplo, o mau cheiro COMPONENTE DE UI ZUMBI afeta os componentes *Activities* e *Fragments*.

Na Tabela 5.2 temos os 10 maus cheiros dos recursos do aplicativo. Novamente, interpretando a tabela do ponto de vista do elemento, temos que recursos de *Layout* são os recursos mais suscetíveis a diferentes maus cheiros da camada de apresentação Android, totalizando 5, seguidos de *Strings* e *Styles* suscetíveis á 4 maus cheiros e *Drawables* suscetíveis a apenas 3.

Entendemos que o total de ocorrências desempenha um papel importante na análise dos

Tabela 5.2: *Maus cheiros em recursos de aplicativos Android, sua ocorrência em cada componente e ocorrência total.*

Mau Cheiro	Layout	String	Style	Drawable	#Total
NOME DE RECURSO DESPADRONIZADO	5	10	5	3	24
RECURSO MÁGICO	6	15	2	-	23
LAYOUT PROFUNDAMENTE ANINHADO	18	-	-	-	19
IMAGEM TRADICIONAL DISPENSÁVEL	-	-	-	17	17
LAYOUT LONGO OU REPETIDO	14	-	-	-	14
IMAGEM FALTANTE	-	-	-	10	10
LONGO RECURSO DE ESTILO	-	-	8	-	8
RECURSO DE STRING BAGUNÇADO	-	8	-	-	8
ATRIBUTOS DE ESTILO REPETIDOS	3	-	4	-	7
REUSO EXCESSIVO DE STRING	-	6	-	-	6
LISTENER ESCONDIDO	5	-	-	-	5

maus cheiros pois, mais respostas indicam mais desenvolvedores com a mesma percepção, logo, maior confiabilidade de que dado mau cheiro o é de fato. Logo, **quanto maior o número de ocorrências, maior a confiabilidade do mau cheiro**. Deste modo, ao descrever os maus cheiros iremos indicar ao lado do nome um número sobreescrito indicando o número de ocorrências em respostas. Por exemplo, o mau cheiro COMPONENTE DE UI INTELIGENTE⁶⁰. Esta mesma notação será usada na ocorrência por elementos da camada de apresentação Android.

5.1.3 Maus Cheiros Propostos

Nesta seção apresentamos os 21 maus cheiros propostos sendo respectivamente 10 maus cheiros em componentes da camada de apresentação Android e 11 em recursos Android.

Maus cheiros em componentes do front-end Android

A Tabelas 5.3 apresenta a lista dos maus cheiros em componentes da camada de apresentação Android, suas respectivas estrelas indicando a confiabilidade e uma breve descrição do sintoma relacionado. Em seguida, nesta seção é apresentado a descrição completa do mau cheiro.

COMPONENTE DE UI INTELIGENTE^{12*} *Activities*^{5*}, *Fragments*^{3*}, *Adapters*^{2*} e *Listeners* 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.

COMPONENTE DE UI ACOPLADO^{3*} *Fragments*^{2*}, *Adapters* e *Listeners* 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.

¹Ver mau cheiro CLASSES DE UI FAZENDO IO.

Tabela 5.3: *Maus cheiros em componentes da camada de apresentação Android e breve descrição dos sintomas.*

Mau Cheiro	Breve descrição
COMPONENTE DE UI INTELIGENTE ^{12*}	Componentes de UI com lógicas de negócio.
COMPONENTE DE UI ACOPLADO ^{3*}	Componentes de UI com referência concretas um para o outro.
COMPORTAMENTO SUSPEITO ^{3*}	<i>Listener</i> sendo implementado dentro de outro componente de UI.
ADAPTER CONSUMISTA ^{2*}	Adapters que não usam o padrão <i>ViewHolder</i> .
COMPONENTE DE UI ZUMBI ^{2*}	Componente com referência a componentes de UI com ciclo de vida.
USO EXCESSIVO DE FRAGMENTS*	Uso <i>fragments</i> sem uma necessidade explícita.
COMPONENTE DE UI FAZENDO IO*	Componentes de UI fazendo acesso a internet ou banco de dados.
NÃO USO DE FRAGMENT*	<i>Não usar nenhum Fragment</i>
AUSÊNCIA DE ARQUITETURA*	Aplicativos sem uma arquitetura conhecida.
ADAPTER COMPLEXO*	<i>Adapters</i> com condicionais e <i>loops</i> .

COMPORTAMENTO SUSPEITO^{3*} *Activities*, *Fragments* e *Adapters* 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^{2*} de modo a responder a eventos do usuário.

ADAPTER CONSUMISTA^{2*} São indícios do mau cheiro quando *Adapters*^{2*} não reutilizam instâncias das *views* que representam os campos a serem populados para cada item da coleção através do padrão *ViewHolder* ou quando os mesmos possuem classes internas para reaproveitamento das *views* porém não são estáticas.

COMPONENTE DE UI ZUMBI^{2*} *Activities*^{*} 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.

USO EXCESSIVO DE FRAGMENT* *Fragments*^{*} 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.

COMPONENTE DE UI FAZENDO IO* *Activities*^{*}, *Fragments* e *Adapters* 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.

NÃO USO DE FRAGMENT* *Fragments*^{*} devem ser usados sempre que possível em conjunto com *Activities*. É indício do mau cheiro a não existência de *Fragments* na aplicação ou o uso de *EditTexts*, *Spinners* ou outras *views* por *Activities*.

ADAPTER COMPLEXO* *Adapters*^{*} 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*.

AUSÉNCIA DE ARQUITETURA* São indícios do mau cheiro quando diferentes *Activities* e *Fragments* no projeto apresentam fluxos de código complexos, possivelmente são CLASSE DE UI INTELIGENTE, 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*.

Maus cheiros em recursos Android

A Tabelas 5.4 apresenta a lista dos maus cheiros em recursos Android, suas respectivas estrelas indicando a confiabilidade e uma breve descrição do sintoma relacionado. Em seguida, nesta seção é apresentado a descrição completa do mau cheiro.

Tabela 5.4: *Maus cheiros em recursos Android e breve descrição dos sintomas.*

Mau Cheiro	Breve descrição
NOME DE RECURSO DESPADRONIZADO ^{4*}	Recursos com nomes despadronizados.
RECURSO MÁGICO ^{4*}	Textos, números ou cores “hardcoded”.
AYOUT PROFUNDAMENTE ANINHADO ^{3*}	Recurso de layout com mais de três níveis de <i>Views</i> aninhadas.
IMAGEM TRADICIONAL DISPENSÁVEL ^{3*}	Imagens que poderiam ser transformadas em recurso gráfico.
AYOUT LONGO OU REPETIDO ^{2*}	Recurso de <i>layout</i> muito longo ou com trechos de código similares ou repetidos.
IMAGEM FALTANTE ^{2*}	Imagen sem todas as resoluções padrões.
LONGO RECURSO DE ESTILO [*]	Recurso de estilo único e longo.
RECURSO DE STRING BAGUNÇADO [*]	Recursos de <i>string</i> sem um padrão de nomenclatura.
ATRIBUTOS DE ESTILO REPETIDOS [*]	Atributos de estilo repetidos em recursos de <i>layout</i> ou <i>string</i> .
REUSO INADEQUADO DE STRING [*]	<i>Strings</i> sendo reutilizadas indevidamente.
LISNER ESCONDIDO [*]	Atributo <i>onClick</i> em recursos de <i>layout</i> .

NOME DE RECURSO DESPADRONIZADO^{4*} São indícios do mau cheiro quando recursos de *layout*^{*}, recursos de *string*^{2*}, recursos de *style*^{*} e recursos *drawables* não possuem um padrão de nomenclatura.

RECURSO MÁGICO^{4*} 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*^{*}, recursos de *string*^{3*} ou recursos de *style* usam alguma dessas informações diretamente no código ao invés de fazer referência para um recurso.

AYOUT PROFUNDAMENTE ANINHADO^{3*} São indícios desse mau cheiro o uso de profundos aninhamentos na construção de recursos de *layout*^{3*}, ou seja, *ViewGroups* contendo outros *ViewGroups* sucessivas vezes. O site oficial do Android conta com informações e ferramentas automatizadas para lidar com esse sintoma [45].

IMAGEM TRADICIONAL DISPENSÁVEL^{3*} O Android possui diversos tipos de recursos *drawables*^{3*} 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

drawables 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.

LAYOUT LONGO OU REPETIDO^{2*} São indícios do mau cheiro quando recursos de *layout*^{2*} é muito grande ou possue trechos de layout muito semelhantes ou iguais a outras telas.

IMAGEM FALTANTE^{2*} 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 recurso **DRAWABLE^{2*}** do tipo png, jpg ou gif ou ainda, ter imagens em diretórios incorretos em termos de dpi.

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

RECURSO DE STRING BAGUNÇADO^{*} É indício do mau cheiro o uso de apenas um arquivo para todos os recursos de *string*^{*} do aplicativo e a não existência de um padrão de nomenclatura e separação para os recursos de *string* de uma mesma tela.

ATRIBUTOS DE ESTILO REPETIDOS^{*} É indício do mau cheiro haver recursos de *layout* ou recursos de *style* com blocos de atributos de estilo repetidos.

REÚSO INADEQUADO DE STRING^{*} Cada tela deve ter seu conjunto de recursos de *string*^{*}. É indício do mau cheiro reutilizar o mesmo recurso de *string* em diferentes telas do aplicativo apenas porque o texto coincide.

LISTENER ESCONDIDO^{*} recursos de *layout*^{*} 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.

5.2 QP2. Com qual frequência os maus cheiros são percebidos e o quão importante são considerados pelos desenvolvedores?

Nossos resultados mostraram que todos os 21 maus cheiros derivados são considerados de “razoavelmente importantes” a “muito importantes” e 18 deles se apresentam no dia a dia do desenvolvimento Android com frequência de “ás vezes” até “muito frequente”.

Nossos resultados mostraram que os 21 maus cheiros derivados são considerados, em diferentes níveis, como importantes e se apresentam com diferentes frequências no dia a dia do desenvolvimento Android. A distribuição relativa de frequência para cada frase pode ser vista nos Apêndices D (distribuição relativa sobre as afirmações de frequência) e E (distribuição relativa sobre as afirmações de importância).

5.2.1 Resultados gerais

Para análise dos maus cheiros extraímos dados estatísticos de moda (MO), média (ME) e desvio padrão (DP) de importância e frequência para cada um dos maus cheiros derivados. Utilizamos a MO como um classificador do mau cheiro, ou seja, se ele recebeu majoritariamente a resposta “importante”, o classificamos como importante. Estes dados são apresentados na Tabela 5.5 onde podemos observar que todos os maus cheiros apresentam MO de importância maior ou igual a 3, ou seja, de “razoavelmente importante” a “muito importante”. Em contrapartida, com relação a frequência, três maus cheiros, ADAPTER CONSUMISTA, LISTENER ESCONDIDO e NÃO USO DE FRAGMENT, apresentaram MO igual a 2, “raramente”, todos os demais apresentaram MO maior ou igual a 3, ou seja, de “ás vezes” até “muito frequente”.

Entendemos este resultado de forma positiva pois, apesar de alguns sintomas não serem tão frequentes, ainda assim são considerados com algum nível de importância de se mitigar, reforçando também a relevância desta pesquisa pois damos os primeiros passos no sentido da automatização da identificação desses maus cheiros. Com base no DP, podemos observar que, de modo geral, existe uma concordância maior sobre a importância dos maus cheiros do que sobre a frequência, pois a média do DP de importância é de 1,05, ligeiramente menor que a média do DP de frequência, que é de 1,19.

Tabela 5.5: Média, moda e desvio padrão sobre a percepção da importância dos maus cheiros relacionados a componentes da camada de apresentação Android.

Mau Cheiro	Importância			Frequência		
	ME	MO	DP	ME	MO	DP
COMPONENTE DE UI INTELIGENTE	5	5	1,05	3	4	1,19
RECURSO MÁGICO	4	5	1,00	3	4	1,24
IMAGEM TRADICIONAL DISPENSÁVEL	4	5	0,95	3	4	1,23
LAYOUT LONGO OU REPETIDO	4	5	0,95	4	4	1,07
IMAGEM FALTANTE	5	5	0,95	3	4	1,25
COMPONENTE DE UI ACOPLADO	4	5	1,02	3	3	1,15
CLASSES DE UI FAZENDO IO	5	5	1,03	3	3	1,29
COMPONENTE DE UI ZUMBI	5	5	0,88	3	3	1,16
AUSÊNCIA DE ARQUITETURA	5	5	0,82	3	3	1,30
ADAPTER COMPLEXO	4	5	0,91	3	3	1,15
NOME DE RECURSO DESPADRONIZADO	5	5	0,88	3	3	1,24
ADAPTER CONSUMISTA	5	5	0,93	2	2	1,20
LISTENER ESCONDIDO	4	5	1,23	2	2	1,29
LONGO RECURSO DE ESTILO	4	4	1,06	4	5	1,18
RECURSO DE STRING BAGUNÇADO	3	4	1,22	4	5	1,18
COMPORTAMENTO SUSPEITO	3	4	1,19	3	4	1,19
LAYOUT PROFUNDAMENTE ANINHADO	4	4	1,12	4	4	1,06
ATRIBUTOS DE ESTILO REPETIDOS	4	4	0,86	4	4	1,11
NÃO USO DE FRAGMENT	3	4	1,34	3	2	1,21
REUSO INADEQUADO DE STRING	3	3	1,29	4	4	1,12
USO EXCESSIVO DE FRAGMENT	3	3	1,36	3	3	1,17
DP Médio			1.05			1.19

DP = Desvio Padrão, MO = Moda, ME = Média.

5.2.2 Importância dos Maus Cheiros

Para análise dos dados, simplificamos a escala *likert* de importância de modo que, os maus cheiros de MO 3, “razoavelmente importante”, são classificados de **importância moderada**, os maus cheiros de MO 4 ou 5, respectivamente “importante” e “muito importante”, são classificados de **importância alta**. Nenhum mau cheiro teve MO 1 ou 2, respectivamente “não é importante” e “pouco importante”, logo, não criamos classificações para essas opções. A Tabela 5.6 apresenta a lista dos maus cheiros de acordo com seu nível de importância, alta ou moderada.

Tabela 5.6: Listagem dos maus cheiros da camada de apresentação Android de acordo com seu nível de importância, alta ou moderada.

Importância Alta	Importância Moderada
ADAPTER COMPLEXO	ATRIBUTOS DE ESTILO REPETIDOS
ADAPTER CONSUMISTA	COMPORTAMENTO SUSPEITO
AUSÊNCIA DE ARQUITETURA	LAYOUT PROFUNDAMENTE ANINHADO
CLASSES DE UI FAZENDO IO	LONGO RECURSO DE ESTILO
COMPONENTE DE UI ACOPLADO	NÃO USO DE FRAGMENT
COMPONENTE DE UI INTELIGENTE	RECURSO DE STRING BAGUNÇADO
COMPONENTE DE UI ZUMBI	IMAGEM FALTANTE
IMAGEM TRADICIONAL DISPENSÁVEL	LAYOUT LONGO OU REPETIDO
LISTENER ESCONDIDO	NOME DE RECURSO DESPADRONIZADO
RECURSO MÁGICO	
19	2

A Figura 5.2 apresenta a distribuição relativa de importância dos maus cheiros. Apresentamos negativo (em vermelho) o percentual relacionado as respostas “não é importante”. Podemos observar que 19 dos maus cheiros são de **importância alta**, com mais de 47% das respostas sendo “importante” ou “muito importante”. Apenas 2 maus cheiros são de **importância moderada**, e mesmo quando esta ocorre, a diferença para serem de **importância alta** é baixa. Os maus cheiros que tiveram maior concordância com relação a sua importância, ou seja, DP menor que 1, foram: ADAPTER COMPLEXO, ADAPTER CONSUMISTA, ATRIBUTOS DE ESTILO REPETIDOS, AUSÊNCIA DE ARQUITETURA, COMPONENTE DE UI ZUMBI, IMAGEM FALTANTE, IMAGEM TRADICIONAL DISPENSÁVEL, LAYOUT LONGO OU REPETIDO, NOME DE RECURSO DESPADRONIZADO. .

Os três maus cheiros que obtiveram mais respostas de “não é importante” foram: REUSO INADEQUADO DE STRING, NÃO USO DE FRAGMENT, USO EXCESSIVO DE FRAGMENT. São os mesmos que tiveram menor concordância com relação a sua importância, todos com DP acima de 1,28.

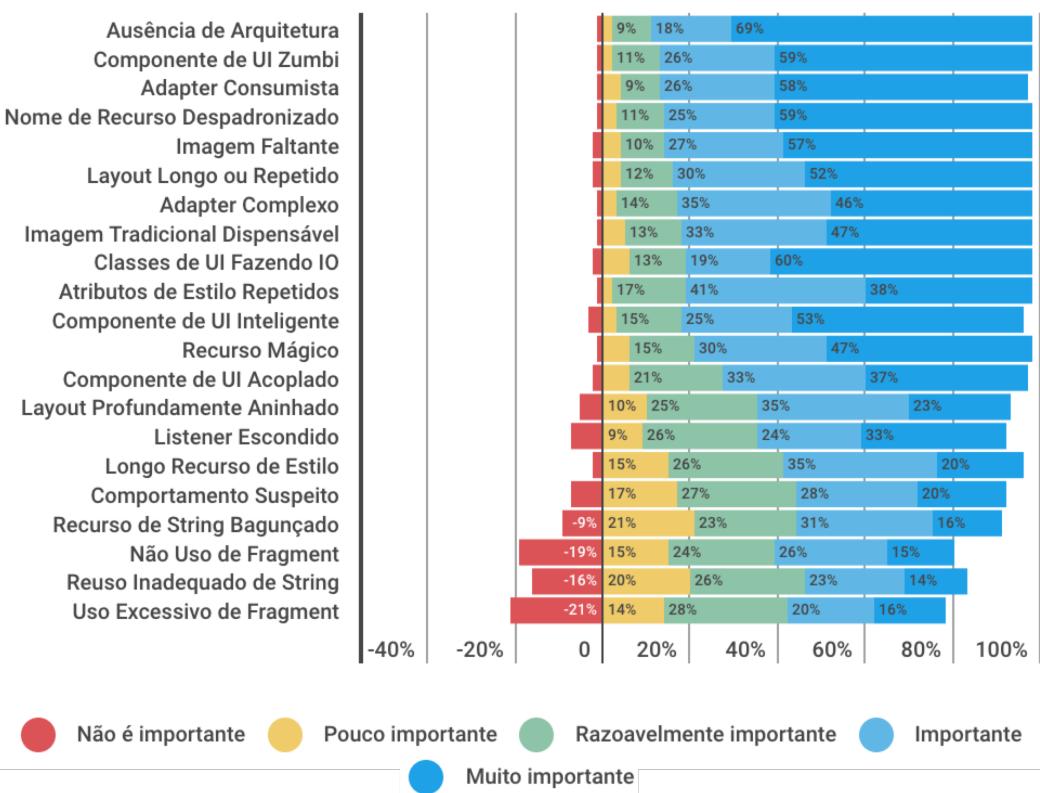


Figura 5.2: Distribuição relativa de importância dos 21 maus cheiros derivados.

5.2.3 Frequência dos Maus Cheiros

Para análise dos dados, simplificamos a escala *likert* de frequência de modo similar ao de importância, onde maus cheiros de MO 2, “raramente”, são classificados como **frequência baixa**, os maus cheiros de MO 3, “ás vezes”, são classificados como **frequência moderada** e os maus cheiros de MO 4 ou 5, respectivamente “frequente” e “muito frequente”, são classificados de **frequência alta**. Nenhum mau cheiro teve MO 1, “nunca”, e portanto não criamos classificação para essa opção. A Tabela 5.7 apresenta a lista dos maus cheiros de acordo com seu nível de frequência, alta, moderada ou baixa.

Tabela 5.7: Listagem dos maus cheiros da camada de apresentação Android de acordo com seu nível de frequência, alta, moderada ou baixa.

Frequência Alta	Frequência Moderada	Frequência Baixa
ATRIBUTOS DE ESTILO REPETIDOS	ADAPTER COMPLEXO	ADAPTER CONSUMISTA
COMPONENTE DE UI INTELIGENTE	AUSÊNCIA DE ARQUITETURA	LISTENER ESCONDIDO
IMAGEM FALTANTE	CLASSES DE UI FAZENDO IO	NÃO USO DE FRAGMENT
IMAGEM TRADICIONAL DISPENSÁVEL	COMPONENTE DE UI ACOPLADO	
AYOUT LONGO OU REPETIDO	COMPONENTE DE UI ZUMBI	
AYOUT PROFUNDAMENTE ANINHADO	COMPORTAMENTO SUSPEITO	
LONGO RECURSO DE ESTILO	NOME DE RECURSO DESPADRONIZADO	
RECURSO MÁGICO	USO EXCESSIVO DE FRAGMENT	
REUSO INADEQUADO DE STRING		
RECURSO DE STRING BAGUNÇADO		

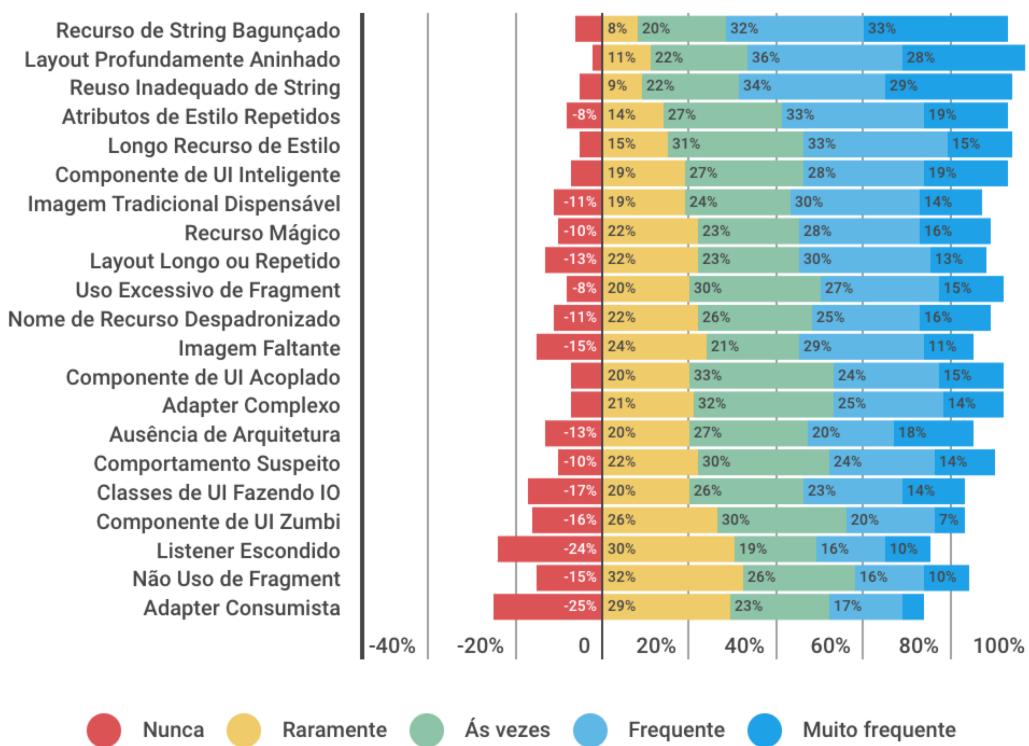


Figura 5.3: Distribuição relativa de frequência dos 21 maus cheiros derivados.

É interessante notar que, maus cheiros em recursos são percebidos mais frequentemente que os maus cheiros em componentes da camada de apresentação Android pois, 9 dentre os 10 maus cheiros de **frequência alta** são em recursos Android. Nos demais níveis de frequência, os maus cheiros em componentes são maioria, sendo 7 dentre os 8 de **frequência moderada** e 2 dentre os 3 de **frequência baixa**.

A Figura 5.2 apresenta a distribuição relativa de frequência dos maus cheiros. Apresentamos negativo (em vermelho) o percentual relacionado as respostas “nunca”. Podemos observar que 10 dos maus cheiros são de **frequência alta**, com mais de 39% das respostas sendo “frequente” ou “muito frequente”. 8 maus cheiros são de **frequência moderada**, tendo obtido mais de 27% das respostas “às vezes” e apenas 3 maus cheiros são de **frequência baixa** com mais de 29% das respostas sendo “raramente”. Nenhum dos maus cheiros obteve DP menor que 1, mas os mais próximos, indicando maior concordância sobre sua frequência são os LAYOUT PROFUNDAMENTE ANINHADO com DP de 1,06 e LONGO RECURSO DE ESTILO com DP 1,07.

5.3 QP3. Desenvolvedores Android percebem os códigos afetados pelos maus cheiros como problemáticos?

Capítulo 6

Conclusão

6.1 Discussões

Notamos que muitas vezes as respostas para as questões sobre boas práticas apresentada no primeiro 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 A.

Aplicativos Android, desde seu lançamento em 2008, são desenvolvidos utilizando a linguagem de programação Java. Recentemente, em Maio de 2017, o Google anunciou o Kotlin como linguagem oficial do Android¹. Para efeitos desta dissertação, utilizamos todos os códigos na linguagem Java, pois a pesquisa iniciou antes desse anúncio. Acreditamos que essa inclusão não interfere na relevância da pesquisa pois: 1) foram quase uma década de aplicativos Android sendo desenvolvidos em Java, 2) Kotlin é interoperável com Java, deste modo, o código antes escritos em Java não precisam necessariamente ser migrados para Kotlin, podendo continuar existindo, precisando de manutenções e evoluções, e 3) como este anúncio é muito recente, acreditamos que ainda levará algum tempo para que o mercado comece a adotar esta nova linguagem.

Mesmo os projetos em kotlin possuem recursos android iguais os projetos em java, logo, os smells aqui encontrados também são úteis para os projetos desenvolvidos em kotlin

todo: Capítulo não concluído, por favor desconsiderar na revisão por enquanto.

Nesta dissertação, investigamos a existência de maus cheiros de código relacionados a elementos da camada de apresentação Android. Os elementos investigados foram: ACTIVI-

¹<https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official>

TIES, FRAGMENTS, LISTENERS, ADAPTERS, LAYOUT, STYLES, STRING e DRAWABLE. Nossos dados foram obtidos por meio de dois questionários online e um experimento de código. Ao todo, participaram da pesquisa 3XX desenvolvedores. A seguir, apresentamos as respostas às questões de pesquisa.

QP1: Existem maus cheiros que são específicos a camada de apresentação Android?

Certamente existem diversas formas de se implementar códigos em elementos da camada de apresentação Android. Algumas destas formas são consideradas melhores e outras piores por desenvolvedores Android. Partindo desta percepção, pudemos propor 21 maus cheiros de código específicos à elementos da camada de apresentação Android. Estes elementos compreendem componentes Java derivados do Android SDK como: ACTIVITIES, FRAGMENTS, LISTENERS e ADAPTERS, e os principais recursos da aplicação: LAYOUT, STYLES, STRING e DRAWABLE.

QP2. Com qual frequência os maus cheiros são percebidos e o quão importante são considerados pelos desenvolvedores?

Todos os maus cheiros possuem alguma frequência no dia a dia de desenvolvimento, alguns mais frequentes que outros. Dentre os elementos da camada de apresentação, notamos que os desenvolvedores percebem mais frequentemente a presença de maus cheiros relacionados a recursos da aplicação do que aos componentes da camada de apresentação Android. A percepção de importância em mitigá-los é alta na maioria dos maus cheiros.

QP3. Desenvolvedores Android percebem os códigos afetados pelos maus cheiros como problemáticos?

Apêndice A

Exemplos de respostas que embasaram os maus 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 (on-Click, por exemplo) do que fazer um set listener no próprio objeto” (P32). “Tome cuidade 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 práticas: “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 práticas: “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 [48].
SML-J6	Má prática: “Usar muitos Fragments é uma má prática” (P2). 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).
SML-J7	Más práticas: “Não usar Fragments” (P22). “Usar todas as view (EditTexts, Spinners, etc...) dentro de Activities e não dentro de Fragments” (P45). Boas práticas: “Utilizar fragments sempre que possível.” (P19), “Use um Fragment para cada tela. Uma Activity para cada aplicativo.” (P45).
SML-J8	Más práticas: “[Activities e Fragments] fazerem requests e consultas a banco de dados” (P26). “[Adapters] fazerem operações longas e requests de internet” (P26). Boa prática: “Elas [Activities] nunca devem fazer acesso a dados” (P37).
SML-J9	Más práticas: “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 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).
SML-J10	Más práticas: “Não usar um design pattern” (P45). 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).
SML-J11	Má prática: “Reutilizar um mesmo adapter para várias situações diferentes, com ifs ou switches. Código de lógica importante ou cálculos em Adapters.” (P23). Boa prática: “Um Adapter deve adaptar um único tipo de item ou delegar a Adapters especializados” (P2).
SML-J12	Más práticas: “De preferência, eles não devem ser aninhados” (P37). “Fragments aninhados!” (P4).
SML-J13	Más práticas: “Sobreescriver o comportamento do botão voltar” (P43). “Lidar com a pilha do app manualmente” (P41).
SML-J14	Boas práticas: “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 práticas: “Strings diretamente no código” (P23). “Não extrair as strings e sobre não extrair os valores dos arquivos de layout” (P31 e P35). Boas práticas: “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 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 organizar 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	<p>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).</p> <p>Boa prática: “XML de layout deve lidar apenas com a view e não com ações” (P41).</p>

Apêndice B

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 aplicativos. 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 C

Questionário sobre frequência e importância dos maus cheiros

Pesquisa sobre qualidade de código em projetos Android

English version? Go to <https://goo.gl/forms/MFJjCGidSbWXFIn83>

Olá! Meu nome é Suelen e sou estudante de mestrado em Ciência da Computação pelo Instituto de Matemática e Estatísticas da USP.

Estou pesquisando sobre qualidade de código Android e a seguir tenho algumas afirmações e gostaria que você, com base no seu conhecimento e experiência, me indicasse sua opinião.

Desde já, muito obrigada pela sua contribuição! Certamente você está ajudando a termos códigos Android com mais qualidade no futuro!

Um forte abraço! – Suelen Carvalho

Seção 1 - Primeiro precisamos saber um pouco sobre você e sua experiência com desenvolvimento de software.

1. Qual sua idade? (Resposta aberta, apenas número)
2. Em que região você mora atualmente? (Uma escolha entre a lista de estados do Brasil, Estados Unidos e Europa)
3. Anos de experiência com desenvolvimento de software? (Uma escolha entre até 1 ano, 2 anos, 3 anos e assim sucessivamente até 10 anos ou mais)
4. Anos de experiência com desenvolvimento Android nativo? (Uma escolha entre, não tenho experiência, até 1 ano, 2 anos, 3 anos e assim sucessivamente até 10 anos ou mais)

5. Informe seu nível de conhecimento nas tecnologias e plataformas a seguir. (Uma escolha para cada tecnologia apresentada dentre as opções Não conheço, Iniciante, Intermediário, Avançado e Especialista). As tecnologias apresentadas foram: Java, Ruby, C/C++, Kotlin, Objective-C, Swift, Android, C#, Python, PHP e Javascript.
6. Qual sua grau escolar? (Uma escolha entre Tecnólogo, Bacharelado, Mestrado, Doutorado e outro)

Seção 2 - Nos conte um pouco o que você costuma ver nos códigos que você desenvolve ou já desenvolveu (independente se no trabalho, acadêmico ou pessoal).

Indique com qual frequência você percebe as situações abaixo no seu dia a dia (Escala Likert Muito Frequent, Frequent, As Vezes, Raramente e Nunca).

1. Activities, Fragments, Adapters ou Listeners com códigos de lógica de negócio, condicionais complexos ou conversão de dados.
2. Fragments, Adapters ou Listeners com referência direta para quem os utiliza, como Activities ou outros Fragments.
3. Activities, Fragments ou Adapters com classes anônimas para responder a eventos do usuário, como clique, duplo clique e outros.
4. Activities, Fragments ou Adapters com classes internas implementando algum listener para responder a eventos do usuário como clique, duplo clique e outros.
5. Activities, Fragments ou Adapters implementando algum listener, através de polimorfismo (implements), para responder a eventos do usuário como clique, duplo clique e outros.
6. Activities ou Fragments sendo usados como callbacks ao final de processos assíncronos, como por exemplo no onPostExecute de uma AsyncTask.
7. Transações de Fragments sendo efetivadas (FragmentTransaction#commit) após o onPause de Activities.
8. Adapters que não se utilizam do padrão ViewHolder.
9. Fragments em aplicativos que não são usados em tablets ou que não usam ViewPagers. aplicativos com Fragments que não são reutilizados em mais de uma tela do app.
10. aplicativos que não utilizam nenhum Fragment.
11. Activities, Fragments ou Adapters com códigos que fazem acesso a banco de dados, arquivos locais ou internet.

12. Projetos que não usam nenhum padrão arquitetural como MVC, MVP, MVVM, Clean Architecture ou outros.
13. Adapters com muitas responsabilidades além de popular views, com condicionais complexos ou cálculos no método getView.
14. Fragments aninhados uns aos outros.
15. Códigos que sobrescrevem o comportamento do botão voltar do Android.
16. Projetos que contém Activities em pacotes com nomes que não são nem activity nem view.
17. Recursos de cor, tamanho ou texto sendo usados “hard coded”, sem a criação de um novo recurso no arquivo de xml respectivo (colors.xml, dimens.xml ou strings.xml).
18. Projetos sem um padrão de nomenclatura para recursos de layout, texto, estilo ou gráficos (imagens e xmls gráficos).
19. Layouts com mais de 3 níveis de views aninhadas, por exemplo: um TextView dentro de um LinearLayout que está dentro de outro LinearLayout.
20. Imagens (jpg, jpeg, png e gif) sendo usadas quando podiam ser substituídas por recursos gráficos do Android (xmls), como por exemplo, background de cores sólidas ou degradês.
21. Recursos de layout muito grandes.
22. Recursos de layout com trechos que se repetem, igual ou muito semelhantes, várias vezes ao longo do arquivo.
23. Projetos que possuem as imagens usadas em apenas uma resolução/densidade.
24. Projetos que possuem apenas um arquivo de estilo (styles.xml).
25. Projetos que possuem apenas um recurso de estilo (styles.xml) e este é muito grande.
26. Projetos que possuem apenas um arquivo de strings (strings.xml) e este é muito grande.
27. Arquivos de layout com alguns atributos de estilos repetidos em mais de uma view.
28. Arquivo de estilo com alguns atributos de estilos repetidos em mais de um estilo.
29. Usar uma mesma string em diferentes telas do aplicativo.
30. Usar o atributo onClick ou outro similar, diretamente no xml de layout, para responder a eventos do usuário.

Seção 3 - Nos conte um pouco o que você considera importante no desenvolvimento Android.

Indique quão importante você considera as situações abaixo (Escala Likert Muito Importante, Importante, Razoavelmente Importante, Pouco Importante e Não é importante).

1. Activities, Fragments, Adapters e Listeners não ter códigos de lógica de negócio, condicionais complexos ou conversão de dados.
2. Fragments, Adapters e Listeners não tenham referência direta à quem os utiliza.
3. Listeners devem ser implementados em suas próprias classes ao invés de implementá-los através de classes anônimas, classes internas ou polimorfismo (uso de implements) em Activities, Fragments ou Adapters.
4. Cuidado ao usar referências diretas a objetos com ciclo de vida como Activities ou Fragments, em objetos sem ciclo de vida ou com ciclo de vida diferente.
5. Usar o padrão ViewHolder em Adapters.
6. Evitar o uso de Fragments. Usar Fragments apenas se não houver outra alternativa.
7. Usar Fragments sempre que possível. Por exemplo, pelo menos um Fragment por Activity, etc.
8. Activities, Fragments e Adapters não terem códigos de acesso a banco de dados, acesso a arquivos locais ou internet.
9. Usar padrões arquiteturais como MVC, MVP, MVVM, Clean Architecture ou outros.
10. Adapters responsáveis apenas por popular a view, sem códigos de lógicas de negócio, cálculos ou conversões de dados/objetos.
11. Não aninhar Fragments, nem pela view, nem via código.
12. Não sobrescrever o comportamento do botão voltar do Android.
13. Activities em pacotes com nomes activity ou view.
14. Criar um recurso de tamanho, cor ou texto, em seu respectivo arquivo (dimens.xml, colors.xml, strings.xml) antes de utilizá-lo.
15. Utilizar um padrão de nomenclatura para arquivos de layout, recursos de texto, estilos e gráficos (imagens ou xmls gráficos).
16. Não utilizar mais de 3 níveis de views aninhadas em xmls de layout.

17. Sempre que possível, substituir o uso de imagens (jpg, jpeg, png ou gif) por recursos gráficos do Android (xmls), no caso, por exemplo de, background de cores sólidas ou degradês.
18. Ter recursos de layouts pequenos.
19. Extrair trechos de layout que se repetem em arquivos de layout novos, e reutilizá-los com include ou merge.
20. Disponibilizar as imagens usadas no aplicativo em mais de uma resolução/densidade diferentes.
21. Separar os recursos de estilo (styles.xml) em mais de um arquivo, por exemplo: estilos e temas.
22. Separar os recursos de strings (strings.xml) em mais de um arquivo.
23. Extrair estilos e reutilizá-los, ao invés de repetir os mesmos atributos diretamente em várias views diferentes.
24. Separar as strings por tela, e não reutilizar uma string em mais de uma tela mesmo o texto sendo igual.
25. Não usar atributos de eventos, como o onClick, em xmls de layout para responder a eventos do usuário.

Apêndice D

Afirmações sobre frequência dos maus cheiros e respectivos dados estatísticos

Neste apêndice, ao final de cada afirmação, entre parênteses indicamos a qual mau cheiro ela se referia, entretanto esta informação (nome do mau cheiro) não foi apresentada no questionário online.

Afirmação	DP	MO	ME	1	2	3	4	5
Activities, Fragments, Adapters ou Listeners com códigos de lógica de negócio, condicionais complexos ou conversão de dados. (COMPONENTE DE UI INTELIGENTE)	1.19	4	3	7%	19%	27%	28%	19%
Fragments, Adapters ou Listeners com referência direta para quem os usa, como Activities ou outros Fragments. (COMPONENTE DE UI ACOPLADO)	1.15	3	3	7%	20%	33%	24%	15%
Activities, Fragments ou Adapters com classes anônimas para responder a eventos do usuário, como clique, duplo clique e outros. (COMPORTAMENTO SUSPEITO)	1.24	3	3	11%	14%	33%	22%	19%
Activities, Fragments ou Adapters com classes internas implementando algum listener para responder a eventos do usuário como clique, duplo clique e outros. (COMPORTAMENTO SUSPEITO)	1.12	4	3	6%	16%	29%	32%	17%
Activities, Fragments ou Adapters implementando algum listener, através de polimorfismo (implements), para responder a eventos do usuário como clique, duplo clique e outros. (COMPORTAMENTO SUSPEITO)	1.21	4	4	9%	12%	25%	33%	20%
Adapters que não se utilizam do padrão ViewHolder. (ADAPTER CONSUMISTA)	1.20	2	2	25%	29%	23%	17%	5%

DP = Desvio Padrão, MO = Moda e ME = Média.

1 = Nunca, 2 = Raramente, 3 = Às vezes, 4 = Frequentemente e 5 = Muito frequente.

Afirmiação	DP	MO	ME	1	2	3	4	5
Fragments em apps que não são usados em tablets ou que não usam ViewPagers. Apps com Fragments que não são reutilizados em mais de uma tela do app. (USO EXCESSIVO DE FRAGMENT)	1.17	3	3	8%	20%	30%	27%	15%
Apps que não utilizam nenhum Fragment. (NÃO USO DE FRAGMENT)	1.21	2	3	15%	32%	26%	16%	10%
Activities, Fragments ou Adapters com códigos que fazem acesso a banco de dados, arquivos locais ou internet. (CLASSES DE UI FAZENDO IO)	1.29	3	3	17%	20%	26%	23%	14%
Objetos com ciclos de vida diferentes ao de Activities ou Fragments, referenciando-os diretamente, por exemplo, uma AsyncTask referenciar diretamente uma Activity ou Fragment. (COMPONENTE DE UI ZUMBI)	1.16	3	3	16%	26%	30%	20%	7%
Projetos que não usam nenhum padrão arquitetural como MVC, MVP, MVVM, Clean Architecture ou outros. (AUSÊNCIA DE ARQUITETURA)	1.30	3	3	13%	20%	27%	20%	18%
Adapters com muitas responsabilidades além de popular views, com condicionais complexos ou cálculos no método getView. (ADAPTER COMPLEXO)	1.15	3	3	7%	21%	32%	25%	14%
Recursos de cor, tamanho ou texto sendo usados "hard coded", sem a criação de um novo recurso no arquivo de xml respectivo (colors.xml, dimens.xml ou strings.xml). (RECURSO MÁGICO)	1.24	4	3	10%	22%	23%	28%	16%
Projetos sem um padrão de nomenclatura para recursos de layout, texto, estilo ou gráficos (imagens e recursos gráficos - xmls). (NOME DE RECURSO DESPADRONIZADO)	1.24	3	3	11%	22%	26%	25%	16%
Layouts com mais de 3 níveis de views aninhadas, por exemplo: um TextView dentro de um LinearLayout que está dentro de outro LinearLayout. (LAYOUT PROFUNDAMENTE ANINHADO)	1.06	4	4	2%	11%	22%	36%	28%
Imagens (jpg, jpeg, png e gif) sendo usadas quando podiam ser substituídas por recursos gráficos do Android (xmls), como por exemplo, background de cores sólidas ou degradês. (IMAGEM TRADICIONAL DISPENSÁVEL)	1.23	4	3	11%	19%	24%	30%	14%
Recursos de layout muito grandes. (LAYOUT LONGO OU REPETIDO)	1.05	4	4	4%	13%	29%	37%	17%
Recursos de layout com trechos que se repetem, igual ou muito semelhantes, vários ao longo do arquivo. (LAYOUT LONGO OU REPETIDO)	1.09	3	3	7%	17%	34%	30%	12%

DP = Desvio Padrão, MO = Moda e ME = Média.

1 = Nunca, 2 = Raramente, 3 = Às vezes, 4 = Frequentemente e 5 = Muito frequente.

Afirmção	DP	MO	ME	1	2	3	4	5
Projetos que possuem as imagens usadas em apenas uma resolução/densidade. (IMAGEM FALTANTE)	1.25	4	3	15%	24%	21%	29%	11%
Projetos que possuem apenas um arquivo de estilo (styles.xml). (LONGO RECURSO DE ESTILO)	1.19	5	4	5%	11%	23%	28%	32%
Projetos que possuem apenas um recurso de estilo (styles.xml) e este é muito grande. (LONGO RECURSO DE ESTILO)	1.18	4	4	6%	16%	22%	33%	22%
Projetos que possuem apenas um arquivo de strings (strings.xml) e este é muito grande. (RECURSO DE STRING BAGUNÇADO)	1.18	5	4	6%	8%	20%	32%	33%
Arquivos de layout com alguns atributos de estilos repetidos em mais de uma view. (ATRIBUTOS DE ESTILO REPETIDOS)	1.07	4	4	5%	13%	31%	35%	15%
Arquivo de estilo com alguns atributos de estilos repetidos em mais de um estilo. (ATRIBUTOS DE ESTILO REPETIDOS)	1.16	4	3	9%	17%	28%	33%	13%
Usar uma mesma string em diferentes telas do aplicativo. (REUSO INADEQUADO DE STRING)	1.12	4	4	5%	9%	22%	34%	29%
Usar o atributo onClick ou outro similar, diretamente no xml de layout, para responder a eventos do usuário. (LISTENER ESCONDIDO)	1.29	2	2	24%	30%	19%	16%	10%

DP = Desvio Padrão, MO = Moda e ME = Média.

1 = Nunca, 2 = Raramente, 3 = Às vezes, 4 = Frequentemente e 5 = Muito frequente.

Apêndice E

Afirmações sobre importância dos maus cheiros e respectivos dados estatísticos

Neste apêndice, ao final de cada afirmação, entre parênteses indicamos a qual mau cheiro ela se referia, entretanto esta informação (nome do mau cheiro) não foi apresentada no questionário online.

Afirmação	DP	MO	ME	1	2	3	4	5
Activities, Fragments, Adapters e Listeners não ter códigos de lógica de negócio, condicionais complexos ou conversão de dados. (COMPONENTE DE UI INTELIGENTE)	1.05	5	5	3%	3%	15%	25%	53%
Fragments, Adapters e Listeners não tenham referência direta à quem os utiliza. (COMPONENTE DE UI ACOPLADO)	1.02	5	4	2%	6%	21%	33%	37%
Listeners devem ser implementados em suas próprias classes ao invés de implementá-los através de classes anônimas, classes internas ou polimorfismo (uso de implements) em Activities, Fragments ou Adapters. (COMPORTAMENTO SUSPEITO)	1.19	4	3	7%	17%	27%	28%	20%
Cuidado ao usar referências diretas a objetos com ciclo de vida como Activities ou Fragments, em objetos sem ciclo de vida ou com ciclo de vida diferente. (COMPONENTE DE UI ZUMBI)	0.88	5	5	1%	2%	11%	26%	59%
Usar o padrão ViewHolder em Adapters. (ADAPTER CONSUMISTA)	0.93	5	5	1%	4%	9%	26%	58%
Evitar o uso de Fragments. Usar Fragments apenas se não houver outra alternativa. (USO EXCESSIVO DE FRAGMENT)	1.36	3	3	21%	14%	28%	20%	16%

DP = Desvio Padrão, MO = Moda, ME = Média, 1 = Não é importante, 2 = Pouco importante,

3 = Razoavelmente importante, 4 = Importante e 5 = Muito importante.

Afirmiação	DP	MO	ME	1	2	3	4	5
Usar Fragments sempre que possível. Por exemplo, pelo menos um Fragment por Activity, etc. (NÃO USO DE FRAGMENT)	1.34	4	3	19%	15%	24%	26%	15%
Activities, Fragments e Adapters não terem códigos de acesso a banco de dados, acesso a arquivos locais ou internet. (CLASSES DE UI FAZENDO IO)	1.03	5	5	2%	6%	13%	19%	60%
Usar padrões arquiteturais como MVC, MVP, MVVM, Clean Architecture ou outros. (AUSÊNCIA DE ARQUITETURA)	0.82	5	5	1%	2%	9%	18%	69%
Adapters responsáveis apenas por popular a view, sem códigos de lógicas de negócio, cálculos ou conversões de dados/objetos. (ADAPTER COMPLEXO)	0.91	5	4	1%	3%	14%	35%	46%
Criar um recurso de tamanho, cor ou texto, em seu respectivo arquivo (dimens.xml, colors.xml, strings.xml) antes de utilizá-lo. (RECURSO MÁGICO)	1.00	5	4	1%	6%	15%	30%	47%
Utilizar um padrão de nomenclatura para arquivos de layout, recursos de texto, estilos e gráficos (imagens ou recursos xmls gráficos). (NOME DE RECURSO DESPADRONIZADO)	0.88	5	5	1%	3%	11%	25%	59%
Não utilizar mais de 3 níveis de views aninhadas em xmls de layout. (LAYOUT PROFUNDAMENTE ANINHADO)	1.12	4	4	5%	10%	25%	35%	23%
Sempre que possível, substituir o uso de imagens (jpg, jpeg, png ou gif) por xmls gráficos do Android, no caso, por exemplo de, background de cores sólidas ou degradês. (IMAGEM TRADICIONAL DISPENSÁVEL)	0.95	5	4	1%	5%	13%	33%	47%
Ter recursos de layouts pequenos. (LAYOUT LONGO OU REPETIDO)	0.95	4	4	1%	5%	30%	36%	27%
Extrair trechos de layout que se repetem em arquivos de layout novos, e reutilizá-los com include ou merge. (LAYOUT LONGO OU REPETIDO)	0.95	5	4	2%	3%	16%	33%	45%
Disponibilizar as imagens usadas no aplicativo em mais de uma resolução/densidade diferentes. (IMAGEM FALTANTE)	0.95	5	5	2%	4%	10%	27%	57%
Separar os recursos de estilo (styles.xml) em mais de um arquivo, por exemplo: estilos e temas ou outros. (LONGO RECURSO DE ESTILO)	1.06	4	4	2%	15%	26%	35%	20%
Separar os recursos de strings (strings.xml) em mais de um arquivo. (RECURSO DE STRING BAGUNÇADO)	1.22	4	3	9%	21%	23%	31%	16%

DP = Desvio Padrão, MO = Moda, ME = Média, 1 = Não é importante, 2 = Pouco importante,

3 = Razoavelmente importante, 4 = Importante e 5 = Muito importante.

Afirmiação	DP	MO	ME	1	2	3	4	5
Extrair estilos e reutilizá-los, ao invés de repetir os mesmos atributos diretamente em várias views diferentes. (ATRIBUTOS DE ESTILO REPETIDOS)	0.86	4	4	1%	2%	17%	41%	38%
Separar as strings por tela, e não reutilizar uma string em mais de uma tela mesmo o texto sendo igual. (REUSO INADEQUADO DE STRING)	1.29	3	3	16%	20%	26%	23%	14%
Não usar atributos de eventos, como o onClick, em xmls de layout para responder a eventos do usuário. (LISTENER ESCONDIDO)	1.23	5	4	7%	9%	26%	24%	33%

DP = Desvio Padrão, MO = Moda, ME = Média, 1 = Não é importante, 2 = Pouco importante,

3 = Razoavelmente importante, 4 = Importante e 5 = Muito importante.

Apêndice F

Gists dos Códigos Utilizados no Experimento da Etapa 3

Lista de gists de códigos maus cheirosos.

#	Mau Cheiro	Gist
1	Componente de UI Inteligente	gist.github.com/SuelenGC/66f9da1dc9ba73183f0d6eef10542ae1
2	Componente de UI Inteligente	gist.github.com/SuelenGC/e4b0197cde11d7cfe1495246a218b008
3	Componente de UI Inteligente	gist.github.com/SuelenGC/298d7b5fe349000790170270144162c8
4	Componente de UI Inteligente	gist.github.com/SuelenGC/599c9c2f9d14b39c3c1d5a1278841ccb
5	Componente de UI Inteligente	gist.github.com/SuelenGC/fdf1b57cace5e3b756d842d201b78095
6	Componente de UI Acoplado	gist.github.com/SuelenGC/5314cc1203cc12fdb0c07aad11f74f65
7	Componente de UI Acoplado	gist.github.com/SuelenGC/bd54fc8fb98f227a4a53d791df48e9bc
8	Componente de UI Acoplado	gist.github.com/SuelenGC/b1a4e52ea77ac1be174dd2bcef919dd5
9	Componente de UI Acoplado	gist.github.com/SuelenGC/fbcbe2edabf73bb259e41a1b5ab57b6e
10	Componente de UI Acoplado	gist.github.com/SuelenGC/ecbbd2cec8d0201d52a61eb72cf98e54
11	Comportamento Suspeito	gist.github.com/SuelenGC/0902b1f1103babda2b78a884bde7a616
12	Comportamento Suspeito	gist.github.com/SuelenGC/6192ef359b23fde4c8f9a3454155b604
13	Comportamento Suspeito	gist.github.com/SuelenGC/ef22d166e962d7ea6ab8233dc4dcdc2a
14	Comportamento Suspeito	gist.github.com/SuelenGC/c9ddbc1781c7227ebb6d7351f82eba4
15	Comportamento Suspeito	gist.github.com/SuelenGC/ab538ce9e16aaaf3df80c936bec07d20b
16	Adapter Complexo	gist.github.com/SuelenGC/d2f67a44ae30c69292182a8bf1e6812a
17	Adapter Complexo	gist.github.com/SuelenGC/87eb0b690056eefde7cbeaab4339c1ee
18	Adapter Complexo	gist.github.com/SuelenGC/ea5896799c99c1ab25483b36604e5984
19	Adapter Complexo	gist.github.com/SuelenGC/d95978aae64760e248841f139d7c0a8
20	Adapter Complexo	gist.github.com/SuelenGC/abe273fe3e7e616fd19863b9d848a401
21	Adapter Complexo	gist.github.com/SuelenGC/2823e7a9f25bfe878609147b99efd57d
22	Longo Recurso de Estilo	gist.github.com/SuelenGC/4b0540bc573e1e9d0277a609eba1e8e8
23	Longo Recurso de Estilo	gist.github.com/SuelenGC/468eb14bdc4cb14af3681473f501d3b1

#	Mau Cheiro	Gist
24	Longo Recurso de Estilo	gist.github.com/SuelenGC/f399d930dc04f1971debb7f4ecd55d85
25	Longo Recurso de Estilo	gist.github.com/SuelenGC/febece2da1eba052711b16212a0a27b2
26	Longo Recurso de Estilo	gist.github.com/SuelenGC/0e11bb39c8df624cf925563c78388095
27	Layout Profundamente Aninhado	gist.github.com/SuelenGC/612c179f5a50c99c35cb50297efd0766
28	Layout Profundamente Aninhado	gist.github.com/SuelenGC/8ca89e8b15d458766c0fee9209359758
29	Layout Profundamente Aninhado	gist.github.com/SuelenGC/bd40e9a9bb1f0b3647d1bd35c6d87feb
30	Layout Profundamente Aninhado	gist.github.com/SuelenGC/964f77f587cd51f1a269201499ceeb9d
31	Layout Profundamente Aninhado	gist.github.com/SuelenGC/76bc7926bebc73ff1d475e239244fa4f
32	Atributos de Estilo Repetidos	gist.github.com/SuelenGC/57335a2424ad70c8ffe97d1f53853296
33	Atributos de Estilo Repetidos	gist.github.com/SuelenGC/6aa82c50d999eab877407f5def84a936
34	Atributos de Estilo Repetidos	gist.github.com/SuelenGC/36c7c268fe4f193644d774bcda482015
35	Atributos de Estilo Repetidos	gist.github.com/SuelenGC/91380358ddcd023960dea2139537ab00

Lista de gists de códigos limpos.

#	Mau Cheiro	Gist
1	Recurso de <i>Layout</i>	gist.github.com/SuelenGC/c5c03d821057a47ba3355e735de6d8dc
2	Recurso de <i>Layout</i>	gist.github.com/SuelenGC/1411e67fba72915978aff2e38be3df80
3	Recurso de <i>Layout</i>	gist.github.com/SuelenGC/9d2b341138b0eb6a79ea776c0325f6e8
4	Recurso de <i>Layout</i>	gist.github.com/SuelenGC/c27d80b69c3bd2ebebed1adc58e4b02e
5	Recurso de <i>Layout</i>	gist.github.com/SuelenGC/068f54f6a053b0506e7b56ac982c4eff
6	Recurso de <i>Style</i>	gist.github.com/SuelenGC/b27a38d42f03aa0004cc7174f40c582f
7	Recurso de <i>Style</i>	gist.github.com/SuelenGC/d73c16b575dd0470cf936d7e5478b7f6
8	Recurso de <i>Style</i>	gist.github.com/SuelenGC/df97964a54f42e757c9532452a724f18
9	Recurso de <i>Style</i>	gist.github.com/SuelenGC/a300c158eb5caed7dfe64c28e9f6acf1
10	Recurso de <i>Style</i>	gist.github.com/SuelenGC/69ce601ad7e6e59d5d80010415118730
11	Componente de Apresentação	gist.github.com/SuelenGC/18038e100fd16d5d18d44b01942ddde7
12	Componente de Apresentação	gist.github.com/SuelenGC/f62f6ffeb78201f592307f862fc11972
13	Componente de Apresentação	gist.github.com/SuelenGC/0802fc818241f23b71925e5e5981b435
14	Componente de Apresentação	gist.github.com/SuelenGC/ed10bff8bf7eb6e33d993e79d31edfdb
15	Componente de Apresentação	gist.github.com/SuelenGC/e7f69902916ce54765d7675a0b3bfdb7

Componente de Apresentação podem ser *Activities*, *Fragments*, *Adapters* ou *Listeners*.

Referências Bibliográficas

- [1] About gists. <https://help.github.com/articles/about-gists>, 2017. [Último acesso em 25 de Dezembro de 2017].
- [2] Steve Adolph, Wendy Hall e Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.
- [3] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King e Shlomo Angel. A pattern language: Towns, buildings, construction (center for environmental structure). 1977.
- [4] Open Handset Alliance. Open handset alliance releases android SDK. https://www.openhandsetalliance.com/press_111207.html, 2007. [Último acesso em 25 de Novembro de 2017].
- [5] Maurício Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa e Arie van Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, pages 1–37, 9 2017.
- [6] Maurício Aniche, Gabriele Bavota, Christoph Treude, Arie Van Deursen e Marco Aurélio Gerosa. A validated set of smells in model-view-controller architectures. pages 233–243, 2016.
- [7] Maurício Aniche e Marco Gerosa. Architectural roles in code metric assessment and code smell detection. 2016.
- [8] Roberta Arcoverde, Alessandro Garcia e Eduardo Figueiredo. Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 33–36. ACM, 2011.
- [9] Boehm Berry W., Brown J.R., Kaspar H., Lipow M., Macleod G.J. e Merrit M.J. *Characteristics of software quality*. TRW series of software technology. North-Holland Pub. Co., 1978.
- [10] William H Brown, Raphael C Malveau, Hays W McCormick e Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.
- [11] Suelen Goularte Carvalho. Apêndice online. <http://suelengc.com/android-code-smells-article/>, 2017. [Último acesso em 25 de Novembro de 2017].
- [12] Suelen Goularte Carvalho e Eduardo Martins Guerra. Padrões para implantar métodos ágeis. Monografia, Instituto Tecnológico de Aeronáutica - ITA, 2011.

- [13] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nassar e Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. pages 1001–1012, 2014.
- [14] CISQ. Consortium for IT software quality. <http://it-cisq.org>, 2017.
- [15] Francois Coallier. Software engineering—product quality—part 1: Quality model. *International Organization for Standardization: Geneva, Switzerland*, 1991.
- [16] W.J. Conover. *Practical nonparametric statistics*. Wiley, 1999.
- [17] Juliet Corbin e Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications Ltd, 3 edition, 2007.
- [18] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications, Incorporated, 2009.
- [19] Oxford Living Dictionaries. Deffinition: Pattern. <https://en.oxforddictionaries.com/definition/pattern>, 2017. [Último acesso em 25 de Novembro de 2017].
- [20] Amin Milani Fard e Ali Mesbah. JSNOSE: Detecting javascript code smells. pages 116–125, 2013.
- [21] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1997.
- [22] Martin Fowler e Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [23] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
- [24] Golnaz Gharachorlu. *Code smells in Cascading Style Sheets: an empirical study and a predictive model*. PhD thesis, University of British Columbia, 2014.
- [25] Barney G. Glaser e Anselm L. Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.
- [26] Google. Android – plataform architecture. <https://developer.android.com/guide/platform/index.html>. [Último acesso em 25 de Novembro de 2017].
- [27] Google. Android – UI events. <https://developer.android.com/guide/topics/ui/ui-events.html>. [Último acesso em 25 de Novembro de 2017].
- [28] Google. Android – activities. <https://developer.android.com/guide/components/activities.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [29] Google. Android – activitiy reference. <https://developer.android.com/reference/android/app/Activity.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [30] Google. Android – async task. <https://developer.android.com/guide/components/broadcasts.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [31] Google. Android – async task. <https://developer.android.com/reference/android/os/AsyncTask.html>, 2016. [Último acesso em 25 de Novembro de 2017].

- [32] Google. Android – building your first app. <https://developer.android.com/training/basics/firstapp/creating-project.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [33] Google. Android – drawable resources. <https://developer.android.com/guide/topics/resources/drawable-resource.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [34] Google. Android – fragments. <https://developer.android.com/guide/components/fragments.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [35] Google. Android – layout resources. <https://developer.android.com/guide/topics/resources/layout-resource.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [36] Google. Android – layouts. <https://developer.android.com/guide/topics/ui/declaring-layout.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [37] Google. Android – providing resources. <https://developer.android.com/guide/topics/resources/providing-resources.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [38] Google. Android – resource type. <https://developer.android.com/guide/topics/resources/available-resources.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [39] Google. Android – string resources. <https://developer.android.com/guide/topics/resources/string-resource.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [40] Google. Android – style resources. <https://developer.android.com/guide/topics/resources/style-resource.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [41] Google. Android studio. <https://developer.android.com/studio/index.html>, 2016. [Último acesso em 25 de Novembro de 2017].
- [42] Google. Documentação site android developer. <https://developer.android.com>, 2016. [Último acesso em 25 de Novembro de 2017].
- [43] Google. Android – fundamentals. <https://developer.android.com/guide/components/fundamentals.html>, 2017. [Último acesso em 25 de Novembro de 2017].
- [44] Google. Android – handling lifecycles with lifecycle-aware components. <https://developer.android.com/topic/libraries/architecture/lifecycle.html>, 2017. [Último acesso em 25 de Novembro de 2017].
- [45] Google. Android – optimizing view hierarchies. <https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html>, 2017. [Último acesso em 25 de Novembro de 2017].
- [46] Google. Android – processes and threads. <https://developer.android.com/guide/components/processes-and-threads.html#Threads>, 2017. [Último acesso em 25 de Novembro de 2017].
- [47] Google. Android – project overview. <https://developer.android.com/studio/projects/index.html>, 2017. [Último acesso em 25 de Novembro de 2017].
- [48] Google. Android – recyclerview. <https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html>, 2017. [Último acesso em 25 de Novembro de 2017].

- [49] Google. Android – services. <https://developer.android.com/guide/components/services.html>, 2017. [Último acesso em 25 de Novembro de 2017].
- [50] Marion Gottschalk, Mirco Josefiok, Jan Jelschen e Andreas Winter. Removing energy code smells with reengineering services. *GI-Jahrestagung*, 208:441–455, 2012.
- [51] Robert B. Grady e Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, 1987.
- [52] Robert J Grissom e John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.
- [53] G. Hecht, R. Rouvoy, N. Moha e L. Duchien. Detecting antipatterns in android apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, May 2015.
- [54] Geoffrey Hecht. An approach to detect android antipatterns. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 766–768. IEEE Press, May 2015.
- [55] ISO. IEC25010: 2011 systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. *International Organization for Standardization*, 34:2910, 2011.
- [56] BSEN ISO9000. ISP 9000:2000 – quality management systems: Fundamentals and vocabulary. *London: British Standards Institution*, 2000.
- [57] Arthur J. Riel. *Object-Oriented Design Heuristics*, volume 335. Addison-Wesley Publishing Company, 1996.
- [58] Java. What is java technology and why do i need it? https://www.java.com/en/download/faq/whatis_java.xml. [Último acesso em 25 de Novembro de 2017].
- [59] Juran Joseph M. e Godfrey A. Blanton. *Juran's Quality Handbook*. McGraw-Hill, 5? edition, 1998.
- [60] Kerievsky Joshua. A timeless way of communicating. <https://www.slideshare.net/JoshuaKerievsky/a-timeless-way-of-communicating-alexandrian-pattern-languages>, 2010. [Último acesso em 25 de Novembro de 2017].
- [61] Foutse Khomh, Massimiliano Di Penta e Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, Washington, DC, USA, 2009. IEEE Computer Society.
- [62] Andrew Koenig. Patterns and antipatterns. *The patterns handbook: techniques, strategies, and applications*, 13:383, 1998.
- [63] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk e Yann-Gaël Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. pages 232–243, 2014.
- [64] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah e M Al-murshed. Understanding code smells in android applications. 2017.

- [65] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.
- [66] Jim A. McCall, Paul K. Richards e Gene F. Walters. Factors in software quality: Concept and definitions of software quality. I:188, 1977.
- [67] Steve McConnell. *Code Complete*. Microsoft Press, Redmond, WA, USA, 2004.
- [68] Roberto Minelli e Michele Lanza. Software analytics for mobile applications, insights & lessons learned. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013.
- [69] Jakob Nielsen. Why you only need to test with 5 users. <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users>, 2000. [Último acesso em 25 de Novembro de 2017].
- [70] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho e Fernando Castor. Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software*, 136:1 – 18, 2017.
- [71] OpenSignal. Android fragmentation visualized. <http://opensignal.com/reports/2015/08/android-fragmentation>, 2015. [Último acesso em 25 de Novembro de 2017].
- [72] Fabio Palomba, Gabriele Bavota, Massimiliano Penta, Rocco Oliveto e Andrea Lucia. Do they really smell bad? a study on developers' perception of bad code smells. pages 101–110, 2014.
- [73] Martin Pinzger, Felienne Hermans e Arie van Deursen. Detecting code smells in spreadsheet formulas. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 409–418, Washington, DC, USA, 2012. IEEE Computer Society.
- [74] Jan Reimann e Uwe Assmann. Quality-aware refactoring for early detection and resolution of energy deficiencies. 2013.
- [75] Jan Reimann e Martin Brylski. A tool-supported quality smell catalogue for android developers. 2014.
- [76] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications Ltd, 2 edition, 2015.
- [77] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus e Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.
- [78] Statista. Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2017. <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems>, 2017. [Último acesso em 25 de Novembro de 2017].
- [79] Girish Suryanarayana, Ganesh Samarthym e Tushar Sharma. *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann, 2014.

- [80] Software Engineering Institute Carnegie Mellon University. Carnegie mellon sei and omg announce the launch of cisq-the consortium for it software quality (www.it-cisq.org), 2009.
- [81] Eva Van Emden e Leon Moonen. Java quality assurance by detecting code smells. In *In Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002.
- [82] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.
- [83] Stefan Wagner. *Software Product Quality Control*. Springer-Verlag Berlin Heidelberg, 2013.
- [84] Bruce Webster F. *Pitfalls of Object-Oriented Development*. M & T Books, 1995.
- [85] Ward’s Wiki. Code smell. <http://wiki.c2.com/?CodeSmell>, 1995. [Último acesso em 05/10/2017].
- [86] Wikipedia. Code smell — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Code%20smell&oldid=810572249>, 2017. [Último acesso em 25 de Novembro de 2017].
- [87] Wikipedia. IOS — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=IOS&oldid=812046680>, 2017. [Último acesso em 25 de Novembro de 2017].
- [88] Aiko Yamashita e Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315. IEEE, Sept 2012.
- [89] Aiko Yamashita e Leon Moonen. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 242–251. IEEE, 2013.
- [90] Aiko Yamashita e Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, ICSE ’13, pages 682–691, Piscataway, NJ, USA, 2013. IEEE Press.