Maus cheiros no front-end Android: Um estudo sobre a percepção dos desenvolvedores

Suelen Goularte Carvalho

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

Programa: Ciência da Computação

Orientador: Prof. Dr. Marco Aurélio Gerosa

São Paulo, 19 Janeiro de 2018

Maus cheiros no front-end Android: Um estudo sobre a percepção dos desenvolvedores

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. **Maus cheiros no front-end Android: Um estudo sobre a percepção dos desenvolvedores**. 2018. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Não há dúvidas de que bons códigos importam, mas como saber quando um código não está bom? Maus cheiros de código nos auxiliam na identificação de trechos de código problemáticos, porém a maioria dos maus cheiros catalogados se baseiam em tecnologias tradicionais, criadas dentre as décadas de 70 a 90, como 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 a camada de apresentação, conhecida por *front-end*, como *Activities* e *Adapters*. Notou-se também que em alguns aplicativos, códigos do *front-end* representam a maior parte. Vale ressaltar que o *front-end* Android também é composto 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 ao *front-end* Android considerando inclusive os recursos. Fizemos isso por meio de 2 questionários online e 3 experimentos totalizando a participação de 3XX desenvolvedores. Nossos resultados mostram a existência de uma percepção comum entre desenvolvedores Android praticantes sobre más práticas no desenvolvimento do *front-end* Android. Nossas principais contribuições são um catálogo com 13 maus cheiros do *front-end* Android e uma análise estatística sobre a percepção de desenvolvedores sobre os 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 praticantes como auxílio na identificação de códigos problemáticos para serem melhorados, ainda que de forma manual.

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

Abstract

CARVALHO, G. S. **Code smells on Android front-end: A study on the developers' perception**. 2018. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

There is no question that good codes matter, but how do you know when a code is not good? Code smells help us identify problematic code snippets, but most of the code smells cataloged are based on traditional technologies, created from the 1970s through the 90s, such as 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 eficience and usability. Other researches conclude 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 codes represent the larger part. 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. We did this through two online surveys and three experiments summing 3XX developers. Our results showed that there is a common perception among practicing Android developers about bad practices on Android front-end. Our main contributions are a catalog of 13 code smells about Android front-end and a statistical analysis of the perceptions of practicing developers about the code smells catalog. Our contributions will serve researchers as a starting point for the definition of heuristics and implementation of automated tools and to practicing developers as an aid in identifying problematic codes to be improved, even manually.

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

Sumário

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

Lista de Tabelas

Capítulo 1

Introdução

"Estamos cientes de que um bom código importa, pois já tivemos que lidar com a falta dele por muito tempo" [37]. De fato, não há mais dúvida de que bons códigos importam. Mas como saber se um código está bom ou não? Uma das formas que temos de responder a essa pergunta é buscando por *maus cheiros* no código. Maus cheiros de código auxiliam desenvolvedores na identificação de trechos de códigos problemáticos, de forma que eles possam ser melhorados e a qualidade do software incrementada [21].

Existem diversos maus cheiros catalogados [21, 37, 72, 76]. Muitos desses maus cheiros foram definidos baseados em conceitos e tecnologias tradicionais, como orientação a objetos e Java [31], que surgiram durante as décadas de 70 à 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 "haveriam 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 [23], o Javascript [20], o arcabouço Spring MVC [11] e inclusive fórmulas de planilhas do Google [44].

Android é uma plataforma móvel que foi lançada em 2008 pelo Google em parceria com diversas empresas [10]. Em 2011 se tornou mundialmente a principal plataforma móvel e desde então vem aumentando sua fatia de mercado, tendo em 2017 alcançado 86% [5].

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 [27, 35, 74]. 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) [25, 46]. Outros pesquisadores focaram em entender características do desenvolvimento Android que os diferenciam do desenvolvimento de software tradicional [40].

Dentre as descobertas realizadas pelos pesquisadores, notou-se que maus cheiros específicos são muito mais frequentes em aplicativos Android do que maus cheiros tradicionais [27]. Os componentes Android mais afetados por maus cheiros tradicionais fazem parte da camada de apresentação, também conhecida por *front-end*, como *Activities* e *Adapters* [27, 40, 74] e que em alguns aplicativos Android, códigos relacionados ao *front-end* são maioria, em termos de linhas de código (*Lines of Code* - LoC) [40]. Vale ressaltar que o *front-end* Android também é composto por arquivos `XML`, chamados de recursos da aplicação, que são usados para a construção da interface com o usuário (*User Interface* - UI) [59]. Nenhuma das pesquisas mencionadas considerou estes arquivos em suas análises.

Nesta dissertação, investigamos a existência de maus cheiros de código relacionados ao *front-end* Android. Outras pesquisas investigaram maus cheiros em termos de eficiência e usabilidade, diferente delas, nós buscamos por maus cheiros relacionados a manutenibilidade, que trata da facilidade do software de ser modificado ou aprimorado. Complementamos as pesquisas anteriores pois focamos no *front-end* Android considerando inclusive os recursos da aplicação. Nossos dados foram obtidos por meio de dois questionários online e três experimentos presenciais dirigidos submetidos a desenvolvedores Android praticantes totalizando a participação de 3XX desenvolvedores.

Nossos resultados mostraram que existe uma percepção comum entre desenvolvedores Android praticantes sobre más práticas no desenvolvimento Android. Mostrou também, que essas más práticas são frequentes e consideradas como problemáticas no desenvolvimento do *front-end* Android e que, desenvolvedores as percebem ao se depararem com elas no código. Concluímos com: (i) um catálogo com 13 novos maus cheiros relacionados ao *front-end* Android, (ii) uma análise estatística da percepção de desenvolvedores praticantes sobre os maus cheiros catalogados e (iii) um apêndice *online* [16] com as informações necessárias para que outros pesquisadores possam replicar 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 praticantes. Aos pesquisadores serve como ponto de partida para a definição de heurísticas de identificação dos maus cheiros e implementação de ferramentas que os identifiquem de forma automática. Aos desenvolvedores praticantes serve como auxílio na identificação de códigos problemáticos para serem melhorados, ainda que de forma manual.

1.1 Desafios no desenvolvimento Android

Activity é um dos principais componentes de aplicativos Android. Ela representa uma tela com *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 1.1 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 1.1: *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 1.2 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 1.2: *Exemplo de recurso de layout com um campo de entrada de texto e um botão organizados um abaixo do outro.*

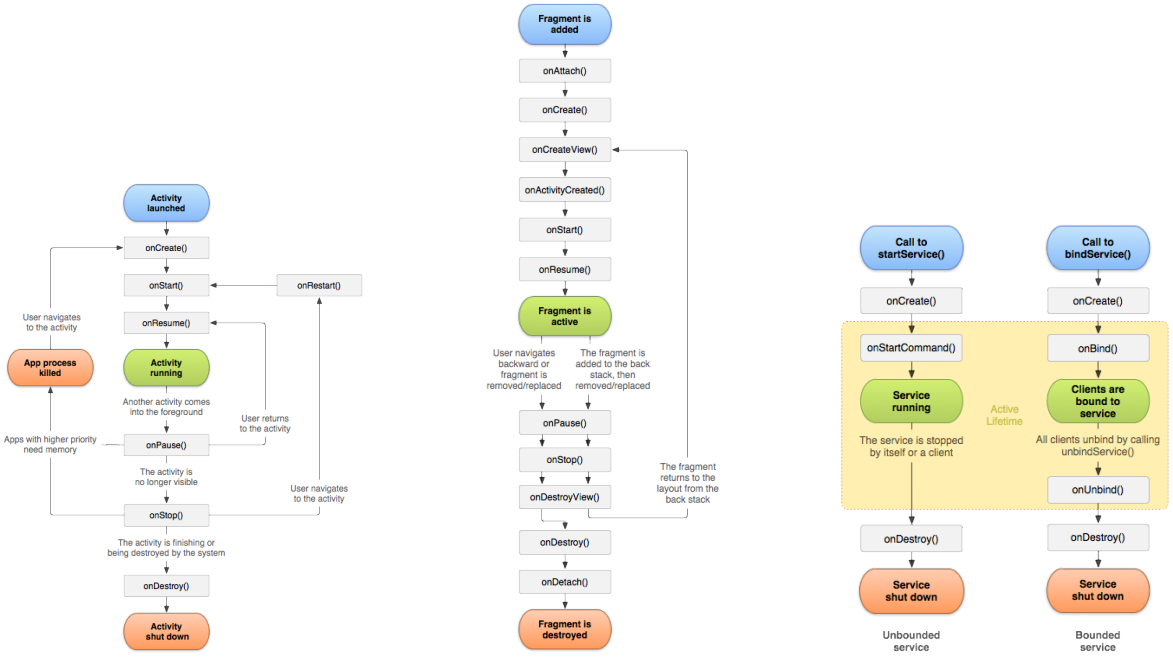
Os códigos apresentados são bem simples, mas comumente, estas telas e UIs tendem a ser bem mais robustas e ricas de informações e interatividade. São em contextos como esses que os desafios no desenvolvimento do *front-end* Android surgem. UIs ricas e robustas podem significar muitas *Views* e *ViewGroups*, 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*.

1.1.1 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 invocados pelo Android em uma ordem específica [65]. Os componente do *front-end* Android possuem ciclos de vida mais extensos, e portanto, mais complexos.

Como exemplo, na Figura 2.2 apresentamos o ciclo de vida de três componentes Android: *Activities* e *Fragments*, ambos relacionados ao *front-end* 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.



(a) *Activitiy (front-end).* **(b)** *Fragment (front-end).* **(c)** *Services (processamento).*

Figura 1.1: *Comparativo do ciclo de vida de componentes Android que pertencem e não pertencem ao front-end.*

1.2 Questões de Pesquisa

Maus cheiros são sintomas que podem indicar um problema mais profundo no código [21]. Geralmente são derivados da experiência e opinião de desenvolvedores [19] ou seja, são por natureza subjetivos [20]. Há ainda evidências na literatura que sugerem que maus cheiros

são percebidos por desenvolvedores [43]. Desta forma, dividimos a pesquisa em três questões principais, que apresentamos a seguir.

QP1: Existem maus cheiros que são específicos ao front-end Android?

Como principal questão de pesquisa, objetiva investigar a existência de maus cheiros no desenvolvimento do *front-end* Android e caso existam, documentá-los. Para isso precisamos explorar o conhecimento empírico de desenvolvedores Android [19, 20], tarefa que exige alguns passos, portanto dividimos esta questão em outras duas a seguir:

QP1.1: O que desenvolvedores consideram boas e más práticas no desenvolvimento do front-end Android?

A percepção de desenvolvedores é sempre importante quando lidamos com manutenção de software [13, 43, 79]. Esta questão nos fornece as primeiras ideias sobre maus cheiros no *front-end* Android a partir da percepção de desenvolvedores praticantes. Buscamos respondê-la por meio de um questionário online submetido a desenvolvedores Android perguntando-os sobre o que eles consideram boas e más práticas no desenvolvimento do *front-end* Android e recebemos um total de 45 respostas.

QP1.2. Desenvolvedores compartilham uma percepção comum sobre o que são más práticas no desenvolvimento do front-end Android?

Com o resultado de QP1.1, realizamos um processo de codificação pelo qual buscamos 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 [41], para serem consideradas na derivação dos maus cheiros. A partir dessas 21 categorias, extraímos 21 candidatos a maus cheiros no *front-end* 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 ao desenvolvedores Android com qual frequência os maus cheiros eram percebidos no seu dia a dia e o quão importante era mitigá-los. Obtivemos 201 respostas dos 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 [43]. A fim de validar a percepção dos maus cheiros pelos desenvolvedores Android, realiza-

mos 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.***

1.3 Organização do Trabalho

Os próximos capítulos desta dissertação estão organizadas da seguinte forma:

- O Capítulo 2 introduz os conceitos chave para este trabalho, são eles: Android, Qualidade de Software e Maus Cheiros de Código.
- O Capítulo 3 discute trabalhos relacionados 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 o catálogo com 13 maus cheiros derivados relacionados ao *front-end* Android.
- 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, de forma que a leitura se torne mais fluída.

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 busca contextualizar o leitor sobre os conceitos e subconceitos 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 [10, 48]. No início de 2011 tomou o liderança, se tornando a plataforma móvel com maior participação no mercado global. Desde então se mantém líder aumentando sua participação a cada ano, tendo em 2017 atingido mais de 86% de participação de mercado. Seus principais concorrentes são iOS da Apple, com participação de mercado de aproximadamente 13% e o Windows Phone, oficialmente descontinuado pela Microsoft em 2017[1]. A Figura 2.1 apresenta a participação de mercado das principais plataformas móveis de 2009 a 2017.

Enquanto que o iOS só é utilizado por iPhones e iPads, que são fabricados pela Apple, totalizando aproximadamente 30 dispositivos diferentes [77], o Android é utilizado por mais de 24 mil dispositivos diferentes segundo levantamento realizado em 2015 [2]. Em termos de desenvolvimento de software, essa grande variedade de dispositivos trás grandes desafios no desenvolvimento Android, desde desafios relacionados a desempenho, por conta das diferentes

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

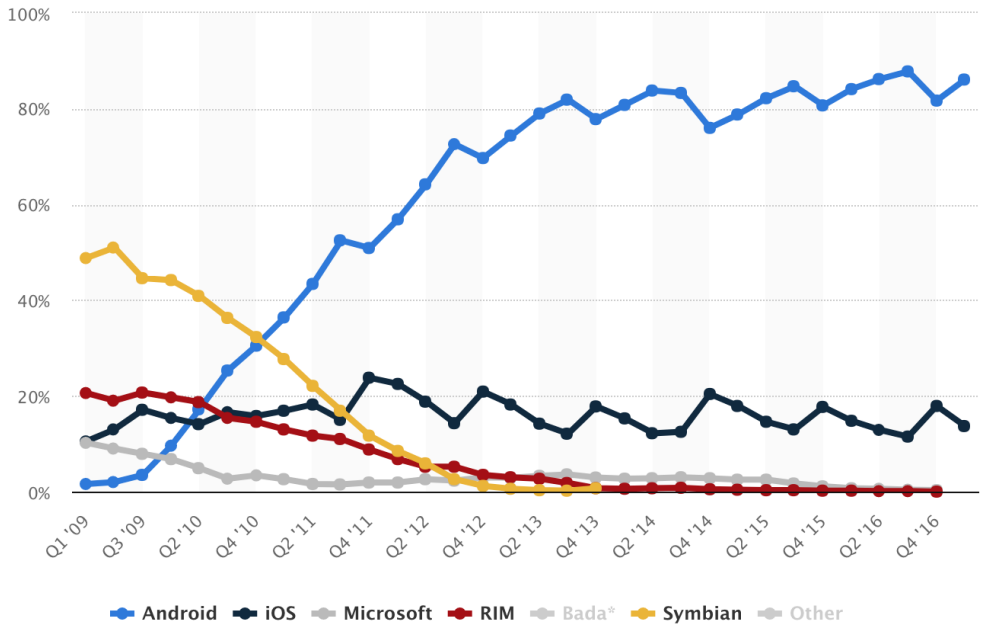


Figura 2.1: *Participação de mercado global de sistemas operacionais móveis do Q1 (1º quadrimestre) de 2009 até o Q1 de 2017.*

configurações de hardware, e desafios relacionados ao *front-end*, pelas diversas configurações de tamanhos de telas e resoluções.

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 Android[2]. 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, os 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.

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 [50] 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 desen-

[2]https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official

volvimento Android (*Software Development Kit* - SDK). O Android *Software Development Kit* (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* [59].

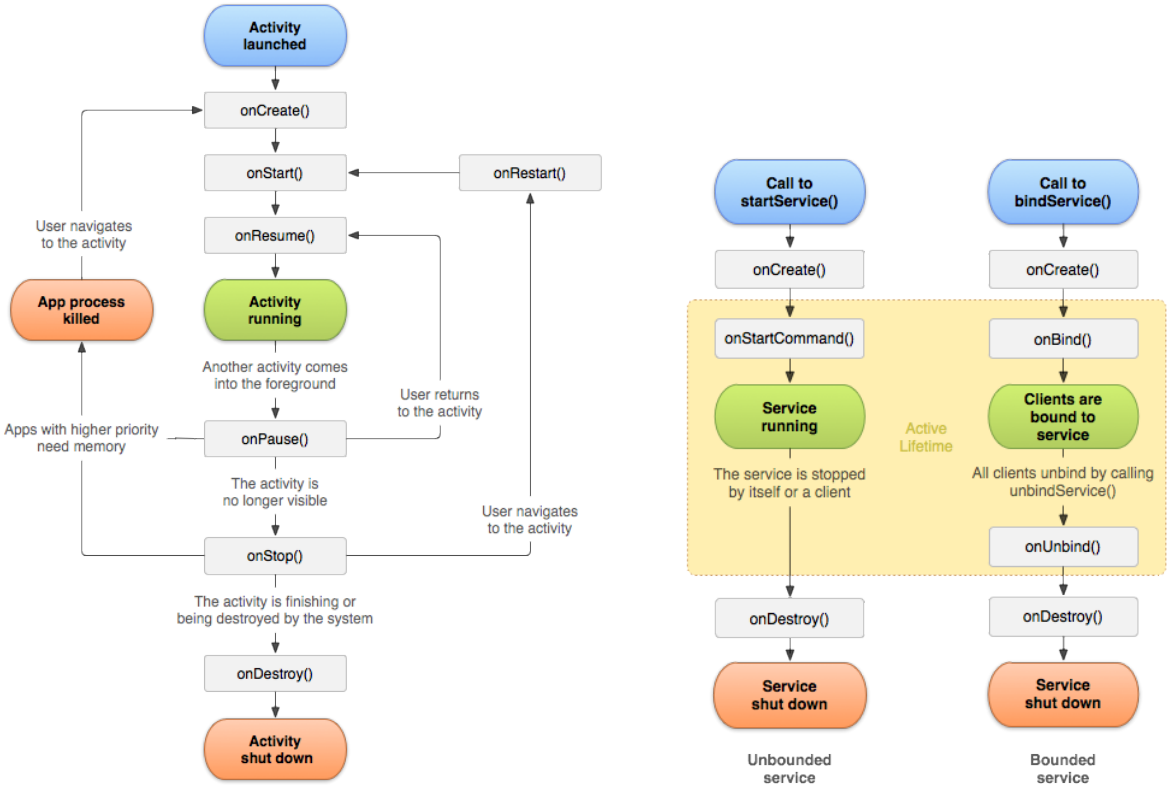
Na estrutura de um projeto Android existem dois diretórios e um arquivo principais. O diretório `src` (de *source*) contém todo o código Java, o diretório `res` (de *resource*) contém todos os recursos da aplicação, que são códigos *"não java"* como imagens, XMLs de layout, dentre outros. Por fim, o arquivo `AndroidManifest.xml` contém configurações gerais do aplicativo como permissões, declarações de componentes dentre outras [63].

O Android SDK provê diversos componentes base para o desenvolvimento Android que não estão disponíveis no desenvolvimento Java tradicional [42]. Alguns componentes são utilizados no *front-end*, 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 [65]. *Services* são usados para longos processamentos em segundo plano e não possuem UI [64]. *BroadcastReceivers* são usados como ouvintes, registrando interesse por mensagens enviadas pelo sistema operacional Android, como bateria baixa, ou por outros componentes [67]. *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 [68, 62].

Cada componente do Android SDK possui um conjunto de métodos de retorno que podem ou devem ser sobrescritos pelo desenvolvedor e que são invocados pelo sistema operacional Android. Ciclo de vida de um componente diz respeito a como ele é criado, mantido e destruído. O ciclo de vida de um componente é composto por um conjunto de métodos de retorno que são sempre invocados e em uma sequência específica [65, 60]. Componentes de UI como *Activities* costumam ter ciclos de vida mais extensos, e portanto mais complexos, que componentes que não lidam com o *front-end*. A Figura 2.2 apresenta como exemplo os ciclos de vida de *Activities*, que estão diretamente relacionadas ao *front-end*, e *Services*, que não estão relacionados ao *front-end*.

O ciclo de vida de *Activities* é complexo, com mais de dez métodos de retorno [66]. A Figura 2.2a 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 2.2b 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



(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.*

segundo o *onBind* e *onUnbind*.

Recursos são arquivos "não Java", utilizados na construção da UI e necessários para a criação de aplicativos Android [59]. 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 [56].

2.1.2 Elementos de Interface Android

São muitos os componentes e recursos disponibilizados pelo Android SDK. Nosso estudo objetiva analisar o que pertencem ao *front-end* Android. Em termos de componentes, fizemos uma extensa revisão na documentação oficial do Android [52] e chegamos em quatro, são eles: *Activities*, *Fragments*, *Adapters* e *Listeners*.

Em termos de recursos, todos são por natureza relacionados ao *front-end*. Com o objetivo de focar a pesquisa, optamos por selecionar os principais com base no modelo padrão de projeto do Android Studio [51], são eles: recursos de *Layout*, recursos de *Strings*, recursos de *Style* e recursos *Drawable*. De modo genérico os chamaremos de "elementos" do *front-end* Android.

A seguir introduzimos brevemente cada um dos oito elementos do *front-end* 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 [65, 66].

Fragments representam parte da UI de uma *Activity* e também devem indicar seu recurso de *layout* correspondente. 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* [69].

Adapters são utilizados para popular 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 [55].

Listeners são interfaces que representam eventos do usuário, por exemplo o *OnClickListener* que captura o clique pelo usuário. Essas interfaces costumam ter apenas um método, onde é implementado o comportamento desejado para responder a interação do usuário [49].

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 [54, 55].

Recursos de String são XMLs utilizados para definir textos, conjunto de textos e plurais usados no aplicativo. A principais vantagens de se usar recursos de *String* é o reaproveitamento dos textos em diferentes UI e a facilidade para internacionalizar [57].

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 [58].

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 [53].

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 *International Organization for Standardization* (ISO) 9000 [29], 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 [32], 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".

Stefan [75, p 6] também examinou diversas definições e identificou que na maioria delas é possível identificar uma mesma ideia central sobre qualidade, todas elas se apontam para "requisitos que precisam ser satisfeitos para ter qualidade".

Nome do Modelo	**Descrição**
ISO/IEC 9126 [7]	Teve sua primeira publicação em 1991, foi revisado em 2001 e em 2011 substituído pela ISO/IEC 25010, conhecida por *Systems and software Quality Requirements and Evaluation* (SQuaRE). Decompõe qualidade de software em 6 áreas: manutenibilidade, eficiência, portabilidade, confiabilidade, funcionalidade e usabilidade.
ISO/IEC 25010 (SQuaRE) [6]	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.
CISQ [4]	Fundado em 2009 [73], decompõe qualidade de software em 4 características: segurança, confiabilidade, desempenho/eficiência e manutenibilidade. Se baseia nas definições do SQuaRE.
FURPS [26]	FURPS é um acrônimo para: Funcionalidade, Usabilidade, Confiabilidade (do inglês *Reliability*), Desempenho (do inglês *Performance*) e Suportabilidade.

Tabela 2.1: *Modelos de qualidade de software baseados no modelo de decomposição hierárquica de Boehm et al. [14] e McCall et al. [38].*

As primeiras contribuições referente a qualidade em termos de software foram publicadas

no fim da década de 70. Boehm et al. [14] e McCall et al. [38] 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 [75, 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 em um nível em que é possível medir e estimar [75].

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" [75, 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 2.3.

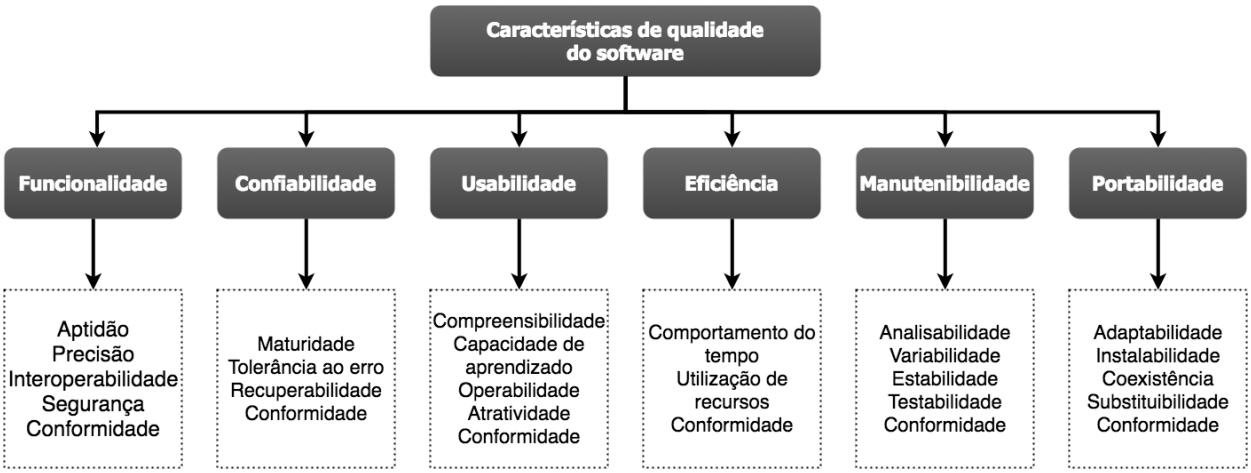


Figura 2.3: *Características de Qualidade de Software segundo norma ISO/IEC 9126 [7].*

A seguir introduzimos brevemente cada um dos conceitos e subconceitos do SQuaRE.

2.2.1 Funcionalidade

A capacidade de um software prover funcionalidades que satisfaçam o usuário em suas necessidades declaradas e implícitas, dentro de um determinado contexto de uso. Seus sub-conceitos são:

- *Adequação*, que mede o quanto o conjunto de funcionalidades é adequado às necessidades do usuário.
- *Acurácia* (ou precisão) representa a capacidade do software de fornecer resultados precisos ou com a precisão dentro do que foi acordado/solicitado.

- *Interoperabilidade* que trata da maneira como o software interage com outro(s) sistema(s) especificados.
- *Segurança* mede a capacidade do sistema de proteger as informações do usuário e fornecê-las apenas (e sempre) às pessoas autorizadas. Segurança também pode estar dirigida em, processar gerar e armazenar as informações.
- *Conformidade* trata da padronização, políticas e normas de um projeto.

2.2.2 Confiabilidade

O produto se mantém no nível de desempenho nas condições estabelecidas. Seus sub-conceitos são:

- *Maturidade*, entendida como sendo a capacidade do software em evitar falhas decorrentes de defeitos no software.
- *Tolerância a Falhas* representando a capacidade do software em manter o funcionamento adequado mesmo quando ocorrem defeitos nele ou nas suas interfaces externas.
- *Recuperabilidade* que foca na capacidade de um software se recuperar após uma falha, restabelecendo seus níveis de desempenho e recuperando os seus dados.
- *Conformidade* tempo ou utilização de recursos.

2.2.3 Usabilidade

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

- *Inteligibilidade* que representa a 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* identifica a facilidade de aprendizado do sistema para os seus potenciais usuários.
- *Operacionalidade* é 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* refere-se a 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.

2.2.4 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* que avalia se os tempos de resposta (ou de processamento) estão dentro das especificações.
- *Utilização de Recursos* que mede tanto os recursos consumidos quanto a capacidade do sistema em utilizar os recursos disponíveis.

2.2.5 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* identifica a facilidade em se diagnosticar eventuais problemas e identificar as causas das deficiências ou falhas.
- *Modificabilidade* caracteriza a facilidade com que o comportamento do software pode ser modificado.
- *Estabilidade* avalia a capacidade do software de evitar efeitos colaterais decorrentes de modificações introduzidas.
- *Testabilidade* representa a capacidade de se testar o sistema modificado, tanto quanto as novas funcionalidades quanto as não afetadas diretamente pela modificação.

2.2.6 Portabilidade

A capacidade do sistema ser transferido de um ambiente para outro. Seus sub-conceitos são:

- *Adaptabilidade*, representando a capacidade do software se adaptar a diferentes ambientes sem a necessidade de ações adicionais (configurações).

- *Instalabilidade* identifica a facilidade com que pode se instalar o sistema em um novo ambiente.
- *Coexistência* mede o quão facilmente um software convive com outros instalados no mesmo ambiente.
- *Substituibilidade* representa a capacidade que o sistema tem de substituir outro sistema especificado, em um contexto de uso e ambiente específicos. Este atributo interage tanto com adaptabilidade quanto com instalabilidade.

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

Muito embora tenha se provado 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 plano [3]. Frases como "depois eu testo" ou "depois eu refatoro" são comuns no dia a dia de desenvolvimento. Manutenibilidade está relacionada as "necessidades implícitas do software" [7].

Focado nesse conceito, ao longo dos últimos anos diversas boas práticas de software vem 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 [15]. Um dos maiores exemplos de anti-padrão é o *Singleton*, ***todo: breve descrição de singleton***.

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* [39].

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

[3] http://www.ifsq.org/resources/level-2/booklet.pdf

ajudar a melhorar a qualidade do software [39]. A seguir introduzimos os conceitos das boas práticas: padrões de projetos, anti-padrões e *maus cheiros de código*.

2.3.1 Padrões de Projeto

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

Para engenharia de software, a principal definição sobre *padrões* provém do livro Uma Linguagem de Padrões do arquiteto Christopher Alexander (1977) [9] onde ele define um *padrão* como sendo uma regra de *três partes* que expressa a *relação* entre um certo **contexto**, um **problema** e uma **solução**. Martin Fowler apresenta uma definição mais simples que diz que *um padrão é uma ideia que foi útil em algum contexto prático e provavelmente será útil em outros* [9].

Inspirados por Alexander [9], Kent Beck e Ward Cunnigham fizeram alguns experimentos do uso de padrões na área de desenvolvimento de software e apresentaram estes resultados na OOPSLA em 1987. Apoiando-se na definição de padrões de Alexander [9], Design Pattern - GoF (1994) foi o primeiro livro sobre padrões de projeto de software a ser lançado, documentando X padrões.

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 outros informações além destes três bases de todos padrão. Ao longo dos anos, alguns formatos foram se destacando, a Figura 2.4 apresenta a tradução de um gráfico criado por Joshua Kerievsky [33] contendo quatro formatos existentes de se escrever padrões. Os formatos são nomeados como Portland, Coplien, *Gang of Four* (GoF) e Alexandrino, e foram plotadas no gráfico de acordo com seu nível de maturidade e clareza.

É possível observar na Figura 2.4 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.

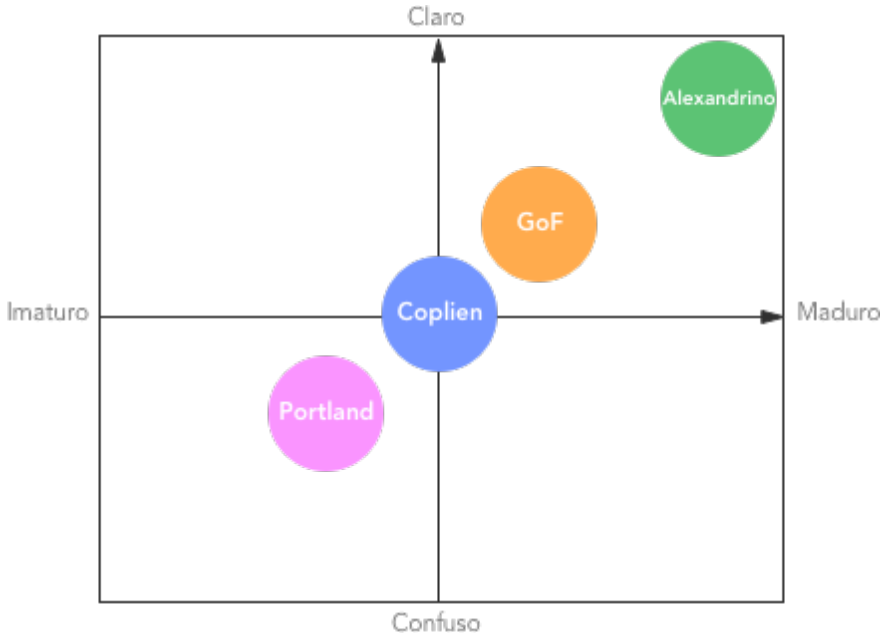


Figura 2.4: *Formato de padrões com relação a maturidade e clareza definido por Joshua Kerievzky [33].*

2.3.2 Anti-Padrões

Um anti-padrão é uma resposta comum a um problema recorrente que geralmente é ineficaz e corre o risco de ser altamente contraproducente. O termo foi cunhado por Andrew Koenig em um artigo publicado em 1995 [???], inspirado pelo livro GoF [22].

O termo se popularizou 3 anos após, no livro de Brown (1998) [15]. Segundo Brown [15], para diferenciar um anti-padrão de um mal hábito, má prática ou ideia ruim um anti-padrão deve apresentar dois elementos: 1) um processo, estrutura ou padrão de ação comumente usado que, apesar de inicialmente parecer ser uma resposta adequada e efetiva a um problema, tem mais consequências ruins do que as boas, 2) existe outra solução que é documentada, repetitiva e provada ser eficaz.

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 um mau cheiro não é, nem poderá se tornar um padrão, condição necessária para se tornar um anti-padrão. E ainda, segundo Brown [15], um anti-padrão deve apresentar os dois elementos acima citados.

2.4 Maus Cheiros de Código

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 [72], 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."* [72, 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 [80, 71, 78, 34].

Há indícios informais de que o termo foi cunhado em meados da década de 90 por Kent Beck [3, 1]. Webster (1995) [76] 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* [76, p. 180]. Apesar do conceito já permear em livros e publicações sobre desenvolvimento de software desde o começo da década de 90, o termo só se popularizou após o livro *Refatoração: Aperfeiçoando o Projeto de Código Existente* (Fowler, M 1999) [21].

No livro são apresentados 22 maus cheiros e mais de 70 técnicas de refatoração. *Código Duplicado* [21, 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* [21, p. 89], ou seja, extrair a expressão duplicada para um novo método e substituí-lo nos lugares onde a expressão era usada.

Maus cheiros provém 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 [1]. Ou seja, é por natureza subjetivo, baseado em opiniões e experiências [19, 20].

Nos anos seguintes o termo mau cheiro se tornou frequente em livros [37, 72] e pesquisas acadêmicas. No livro *Clean Code* (Martin, R 2008) [37], que se tornou muito popular entre desenvolvedores de software, são definidos novos maus cheiros além de citar alguns já apresentados por Fowler [21]. 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 [21] 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* [21, 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 [37] 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* [37, 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 [11, 23].

2.4.2 Formato Adotado

Como apresentado na seção anterior, são muitos os formatos usados na literatura e em pesquisas para se catalogar maus cheiros. Com a evolução de pesquisas sobre o assunto, cada vez mais é possível refiná-los criando heurísticas, ferramentas automatizadas de detecção e inclusive propondo soluções padrões.

Entretanto, este trabalho é uma pesquisa exploratória sobre maus cheiros relacionados a manutenibilidade do *front-end* Android dando os primeiros passos nesse sentido, portanto optamos pelo formato mais simples de mau cheiro, contendo:

- *título*,
- *descrição dos sintomas*,
- e algumas *frases de participantes* que nos levaram a conclusão dos sintomas definidos.

***todo:* ler melhor este parágrafo** Acreditamos que essas definições iniciais são passos importantes e poderão servir de base para pesquisas futuras buscarem por heurísticas, automatizações de ferramentas de detecção e, inclusive propôr soluções para estes e outros maus cheiros Android.

Capítulo 3

Trabalhos Relacionados

Aplicativos Android são desenvolvidos, em sua maioria, utilizando a linguagem de programação Java [59]. 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 3.1 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 software tradicionais.
- A Seção 3.2 apresenta pesquisas que têm demonstrado que diferentes tecnologias podem apresentar maus cheiros específicos. Inclusive que o *front-end* de softwares tradicionais apresentam maus cheiros específicos, e reforça nossa hipótese de que o desenvolvimento do *front-end* Android pode seguir por este mesmo comportamento.
- A Seção 3.3 apresenta pesquisas que já vem investigando (i) a presença de maus cheiros tradicionais em aplicativos Android, (ii) a existência de maus cheiro específicos ao Android e também (iii) comparando 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 ao *front-end* Android e também dar uma visão sólida do estado da arte sobre o assunto.

3.1 Projetos Android vs. projetos de software tradicionais

Minelli e Lanza [40] 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 core e não core, similar ao realizado por [74], onde o código core 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.

3.2 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 [11, 17], a linguagem CSS (*Cascading Style Sheets*) [23] e fórmulas em planilhas [44].

Chen et al. [17] 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. [11] investigaram maus cheiros de código relacionado ao arcabouço Spring MVC, usado para o desenvolvimento do *front-end* 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 [23] investigou maus cheiros em código CSS, linguagem amplamente utilizada no *front-end* 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 *front-end* 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 no *front-end* de aplicações web [20, 23, 11] o que reforça nossa hipótese de que aplicativos Android podem seguir o mesmo comportamento possivelmente apresentando maus cheiros específicos ao *front-end* não encontrados necessariamente nos demais códigos da aplicação.

3.3 Maus cheiros em aplicativos Android

Pesquisas em torno de maus cheiros em aplicativos Android ainda são poucas. Umme et al. [36] recentemente levantaram que, das principais conferências de manutenção de soft-

ware, 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.

3.3.1 Presença de maus cheiros tradicionais em aplicativos Android

Linares-Vásquez et al. [35] 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 [74] investigou a presença de maus cheiros de códigos tradicionais propostos por Fowler [21] (*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 tragam 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 sobre-

cargas 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 [74] conclui propondo 5 refatorações com o objetivo de mitigar o mau cheiro *Long Method* que se apresentou por diversos motivos em `Activities` e `Adapters`. Dentre estas 5 propostas de refatoração, ele implementou e experimentou 3 [explicar melhor estes resultados - chapter 6], 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 com de desempenho e consumo de energia [explicar], por todos estes benefícios, hoje [em qual versão do Android?] já existe um componente no SDK Android com esta implementação de forma nativa.

- 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 do *front-end* 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.

É interessante notar que dentro da definição de Verloop [74] 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 BACKGROUND ANDROID, essas classes são responsáveis por lidar com o *front-end* Android, o que reforça nossa hipótese de que o *front-end* Android

tende a ser mais problemático que o restante dos códigos da aplicação e por isso vale a pena ser estudado mais a fundo.

3.3.2 Maus cheiros específicos Android

Gottschalk et al [25] 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. [46] 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 [45].

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. [15] e Fowler [21]. Todo o catálogo pode ser encontrado online[1] e os mesmos também foram implementados no arcabouço Refactory [45].

Os requisitos de qualidade tratados por Reimann et al [46] 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. [46] cita que o problema no desenvolvimento móvel é que os desenvolvedores estão cientes de cheiros de qualidade apenas indiretamente porque suas definições são informais (melhores práticas, problemas de rastreamento de bugs, discussões de fórum etc.) e os recursos onde encontrá-los são distribuídos pela web e que é difícil coletar e analisar todas essas fontes sob um ponto de vista comum e fornecer suporte de ferramentas para desenvolvedores.

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. (segundo o paper de Geoffrey Hetch - abaixo)

[1] http://www.modelrefactoring.org/smell_catalog

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

Hetch [28] utilizou a ferramenta de detecção de maus cheiros Páprika[2] para identificar 8 maus cheiros, sendo 4 tradicionais (*Blob Class* [15], *Swiss Army Knife* [15], *Complex Class* [21] e *Long Method* [21]) e 4 Android (*Internal Getter/Setter* [46], *No Low Memory Resolver* [46], *Member Ignoring Method* [46] e *Leaking Inner Class* [46]), 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 [15] como abordamos na Seção X.X.X e não há evidências que de que os itens tratados por Hetch atendem as características mencionadas.

Hetch [28] 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* [15] (muito similar a *God Class* [30] e *Large Class* [21]) que reforça a conclusão de Verloop [74]. Esta conclusão reforça nossa hipótese que códigos pertencentes ao *front-end* 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-

[2] https://github.com/geoffreyhecht/paprika

cionais ou específicos Android, sentem a necessidade de delimitar o código em análise como pôde ser visto nos trabalhos [74, 40] através do termo *classes núcleos* (ou *código núcleo*), em ambas as pesquisas significando *classes que herdam 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 a 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 do *front-end* Android.

Essas semelhanças reforçaram nossa curiosidade em focar nossa pesquisa no código relacionado ao *front-end* 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

Pesquisa

Este trabalho trata-se de uma pesquisa exploratória de gênero empírico e de abordagem mista. Nosso objetivo é, por meio do conhecimento empírico de desenvolvedores Android, coletar e identificar más práticas comumente percebidas no desenvolvimento do *front-end* Android e documentá-las na forma de maus cheiros.

A percepção desempenha um importante papel na definição de maus cheiros de código relacionados a uma tecnologia específica [13, 43, 79]. Principalmente considerando a natureza subjetiva intrínseca a maus cheiros [19, 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 Método de Pesquisa

Esta pesquisa foi divida em 3 etapas conforme apresentado na Figura 4.1.

A primeira etapa objetiva responder a **QP1: Existem maus cheiros que são específicos ao front-end Android?** Para obtermos os dados iniciais, buscamos entender o que desenvolvedores Android consideram boas e más práticas ao desenvolver aplicativos Android. Estes dados foram obtidos através de um questionário online respondido por 45 desenvolvedores. A partir desses dados foi realizado um processo de codificação pelo qual

derivamos 21 possíveis maus cheiros de código Android. Apresentamos na Seção 4.1.1 os detalhes desta etapa.

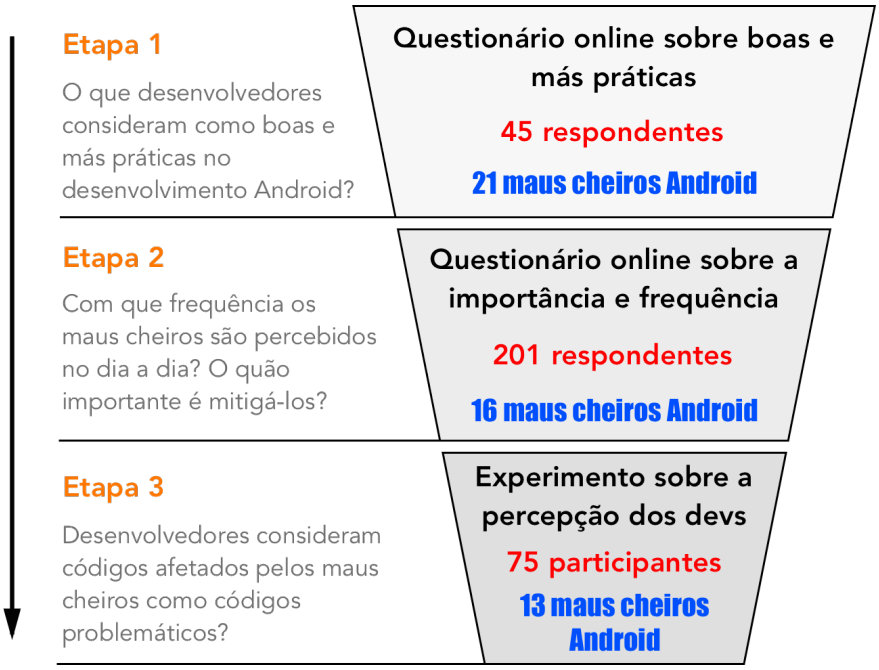


Figura 4.1: *Etapas da pesquisa.*

Na segunda etapa, objetivamos responder a **QP2. Com qual frequência os maus cheiros são percebidos e o quão importante são considerados pelos desenvolvedores?**. Fizemos isso a partir de outro questionário online respondido por 201 desenvolvedores Android. Nessa etapa, 5 dos maus cheiros inicialmente derivados foram removidos, visto que não eram considerados importantes para os desenvolvedores, restando 16. Apresentamos os detalhes desta etapa na Seção 4.1.2.

Na terceira e última etapa, objetivamos responder a **QP3. Desenvolvedores Android percebem os códigos afetados pelos maus cheiros como problemáticos?**. Para isso fizemos um experimento com 75 desenvolvedores Android do qual pudemos colher dados estatísticos sobre a percepção sobre os maus cheiros. Nesta etapa mais 3 maus cheiros foram descartados e concluímos com 13 maus cheiros do *front-end* Android percebidos e considerados relevantes a desenvolvedores Android. Os detalhes desta etapa são apresentados na Seção 4.1.3.

4.1.1 Etapa 1 - Boas e más práticas no *front-end* Android

Iniciamos nossa pesquisa com algumas hipóteses de quais práticas poderiam ser consideradas más práticas, e portanto exalando um mau cheiro no desenvolvimento do *front-end* Android. Uma dessa situações hipotéticas foi, por exemplo, o uso de textos, cores e tamanhos diretamente no XML de layout, sem a criação de um novo recurso, respectivamente nos arquivos `strings.xml`, `colors.xml` ou `dimens.xml`.

Para validarmos essa e outras hipóteses e responder a **QP1. Quais são possíveis maus cheiros do *front-end* Android?**, perguntamos através de um questionário online respondido por 45 desenvolvedores o que eles consideravam como boas e más práticas no desenvolvimento do *front-end* Android. Com o resultado extraímos 46 categorias que resultaram em 21 maus cheiros, dentre os quais, estavam maus cheiros que confirmaram nossas hipóteses iniciais.

A seguir apresentamos detalhes sobre a concepção do questionário (Seção 4.1.1), sobre os participantes (Seção 4.1.1) e sobre o processo de análise dos dados (Seção 4.1.1).

Questionário

O questionário (*S1*) 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 do *front-end* Android. Foi composta por 16 questões opcionais e abertas, 8 sobre boas práticas em cada um dos 8 elementos do *front-end* Android e 8 sobre más práticas. Por exemplo, para o elemento Activity foram feitas as seguintes perguntas:

> **Você tem alguma boa prática para lidar com `Activities`?** (Resposta aberta)
>
> **Você considera alguma coisa uma má prática ao lidar com `Activities`?** (Resposta 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 realizarmos 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 para efeitos de viés.

O questionário foi divulgado em redes sociais como Facebook, Twitter e Linkedin, em grupos de discussão sobre Android como *Android Dev Brasil*, *Android Brasil Projetos* e o grupo do *Slack Android Dev Br*, maior grupo de desenvolvedores Android do Brasil com

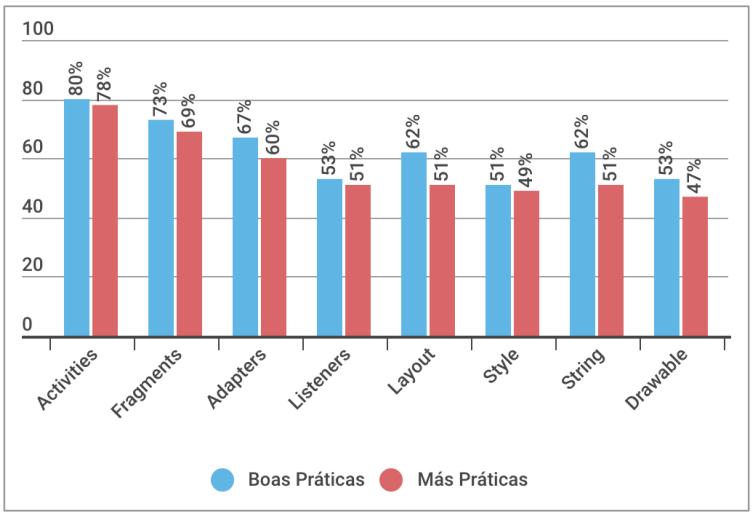


Figura 4.2: *Percentual de respostas sobre boas e más práticas nos elementos do front-end Android.*

2622 participantes até o momento desta escrita. O questionário esteve aberto por aproximadamente 3 meses e meio, de 9 de Outubro de 2016 até 18 de Janeiro de 2017. Ao final, o questionário foi respondido por 45 desenvolvedores.

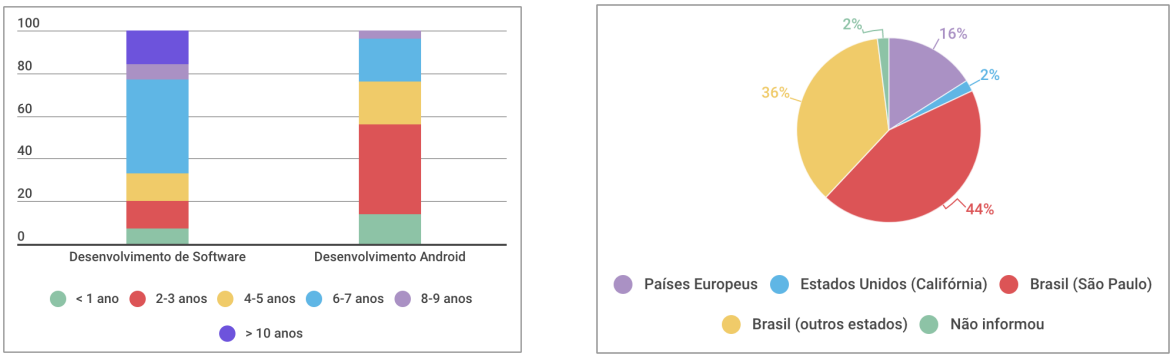
Recebemos um total de 45 respostas. 80% dos participantes responderam pelo menos 3 perguntas, apenas 20% responderam uma (2 participantes) ou nenhuma (7 participantes) pergunta. A questão do email era opcional, mas foi respondida por 53% dos participantes, o que pode indicar um interesse legítimo da comunidade de desenvolvedores Android pelo tema, reforçando a relevância do estudo.

Conforme apresentado na Figura 4.2, todos os elementos do *front-end* Android receberam respostas sobre boas e más práticas. O elemento que recebeu menos respostas foi o recurso Style, com 51% sobre boas práticas, e o que menos recebeu respostas foi o recurso Drawable, com 47% sobre más práticas. Esse resultado pode ser entendido como um indício de que desenvolvedores Android de fato possuem um arsenal pessoal de práticas de desenvolvimento Android que consideram boas e más. Resta-nos entender quais dessas práticas são compartilhadas e quais são apenas preferências pessoais. Fazemos isso na etapa 2.

Podemos observar que classes Java, no geral, receberam mais respostas do que recursos da aplicação. Dentre os elementos Java, os mais respondidos foram Activity e Fragment. Dentre os recursos da aplicação, os mais respondidos foram Layout e String. Notamos também que mais boas práticas foram indicadas do que más práticas em todos os elementos.

Participantes

Conforme apresentado na Figura 4.4a, 87% dos participantes indicaram ter 2 anos ou mais de experiência com desenvolvimento Android. Vale lembrar 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.



(a) *Experiência* **(b)** *Distribuição geográfica*

Figura 4.3: *Experiência com desenvolvimento de software, desenvolvimento Android e distribuição geográfica dos participantes de* Q1.

Conseguimos atingir 3 continentes porém, a maior representatividade dos dados é originada do Brasil com 80% dos participantes. Desse total, os estados com maior representatividade foram, na ordem: São Paulo com 44%, Rio de Janeiro e Goiás cada um com 7% e Rio Grande do Sul e Santa Catarina, cada um com 4%. 16% dos participantes são originados de países Europeus, sendo Irlanda com 8% e Alemanha, Polônia, Suíça e Reino Unido com 2% (1 participante) cada. Nos Estados Unidos tivemos apenas 2%.

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.

Análise dos Dados

Para análise dos dados seguimos a abordagem de *Ground Theory* (GT), um método de pesquisa exploratória originada nas ciências sociais [18, 24], mas cada vez mais popular em pesquisas de engenharia de software [8]. 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*.

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

O passo seguinte foi realizar a *limpeza dos dados*. Esse passo consistiu em remover respostas obviamente não úteis como respostas em branco, 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 praticas 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 [18, 47]. 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 [18]. 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 23 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 [41]. Dessas categorias, duas foram desconsideradas por se tratarem de: um mau cheiro tradicional (*Large Class*) e um aspecto da orientação a objetos (Herança).

O interessante é que por acaso nos deparamos com as mesmas conclusões que algumas pesquisas anteriores, de que aplicativos Android são fortemente afetados pelo mau cheiro *Large Class* [74] e que é pouco ou quase nada usado herança para estruturar o *design* do código [40]. Como nosso foco não estava em avaliar a presença de maus cheiro tradicionais ou práticas de orientações a objetos em aplicativos Android, desconsideramos esses resultados. A Tabela 4.1 apresenta as 21 categorias consideradas para derivação dos maus cheiros.

Cada categoria considerada deu origem a um mau cheiro de código no *front-end* Android sendo 11 relacionadas a elementos Java e 10 relacionadas a recursos da aplicação.

Categoria	Ocorrências	Tipo
Classes de UI Inteligentes	60	Java
Nome de Recurso Despadronizado	23	Recurso
Recursos Mágicos	23	Recurso
Layout Profundamente Aninhado	18	Recurso
Classes de UI Acopladas	18	Java
Imagens Dispensável	18	Recurso
Comportamento Suspeito	17	Java
Layout Longo ou Repetido	14	Recurso
Adapter Consumista	13	Java
Imagem Faltante	12	Recurso
Uso Excessivo de Fragments	9	Java
Classes de UI Fazendo IO	9	Java
Não Uso de Fragments	8	Java
Longo Recurso de Estilo	8	Recurso
Recurso de String Bagunçado	8	Recurso
Atributos de Estilo Repetidos	7	Recurso
Referência Morta	10	Java
Arquitetura Não Identificada	6	Java
Reuso inadequado de String	6	Recurso
Adapter Complexo	5	Java
Listener Escondido	5	Java

Tabela 4.1: *Total de ocorrências e tipo de elemento Android afetado por categoria.*

4.1.2 Etapa 2 - Frequência e importância dos maus cheiros

Para responder a **QP2**, buscamos entender a percepção dos desenvolvedores com relação a frequência e importância dos maus cheiros derivados em seu dia a dia. 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 diversos países. Com o resultado conseguimos validar que os 16 maus cheiros derivados são percebidos no dia a dia de desenvolvimento e são considerados importantes.

A seguir apresentamos detalhes sobre a concepção do questionário (Seção 4.1.2), sobre os participantes (Seção 4.1.2) e sobre o processo de análise dos dados (Seção 4.1.2).

Questionário

Foram criadas duas versões do questionário, inglês e português, de forma que um apontava para o outro, possibilitando o respondente escolher o idioma mais apropriado a ele. O questionário (*Q2*) continha 3 seções, a primeira seção, como em *Q1*, 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 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 frases, onde cada frase descrevia o sintoma de um dos maus cheiros, e pedimos para o respondente **indicar com qual frequência ele percebia cada uma das situações listadas no seu dia a dia**. Para cada frase o respondente podia escolher uma dentre os cinco opções da escala *likert* de frequência: muito frequente, frequente, as vezes, raramente e nunca.

As frases foram baseadas nas respostas em *Q1*, 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 nas respostas, extraímos a seguinte 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 27 frases, essa diferença do número total de maus cheiros vs. o número de frases se dá pois, para os maus cheiros Comportamento Suspeito, Entenda o Ciclo de Vida, Layout Longo ou Repetido, Longo Recurso de Estilo e Atributos de Estilo Repetidos identificamos nas respostas em *Q1*, 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 frase para eles, sendo que cada frase abordou um dos sintomas. Por exemplo, para o mau cheiro **Comportamento Suspeito** apresentamos três frases, onde cada uma representou uma forma diferente, considerada má prática pelos respondentes de *Q1*, sobre como implementar a resposta a eventos do usuário:

- ***`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 quão importante ele considerava as frases apresentadas**. Para contemplar os 21

maus cheiros apresentamos 25 frases que basicamente negavam a frase relacionada a percepção de frequência do mau cheiro. A divergência do total de maus cheiros vs. o total de frases apresentadas se dá pelo mesmo motivo encontrado na seção anterior do questionário, sobre percepção de frequência. Como exemplo, novamente sobre o mau cheiro CLASSES DE UI ACOPLADAS, apresentamos a seguinte frase:

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

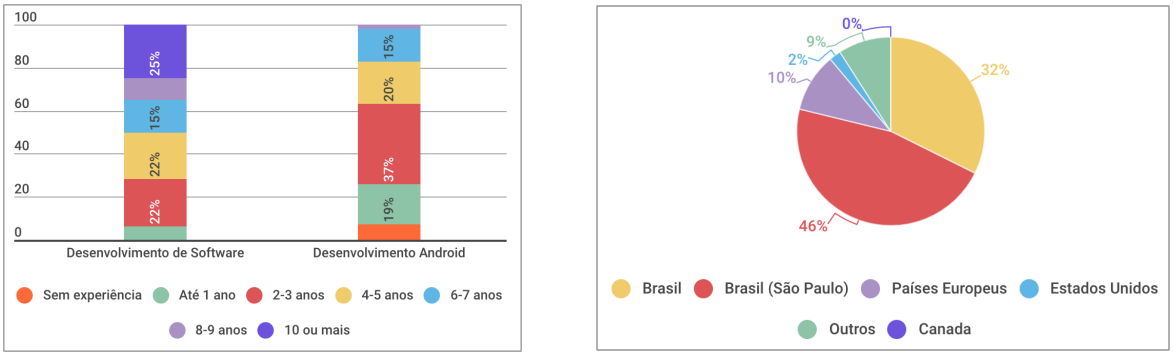
Para cada frase o respondente podia escolher uma dentre as cinco opções da escala *likert* de importância: muito importante, importante, razoavelmente importante, pouco importante, não é importante.

Antes da divulgação do questionário realizamos a validação de cada uma das frases, 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 ***todo: confirmar com o Cauê***. Ambos os desenvolvedores trabalham atualmente em *startups* brasileiras com desenvolvimento mobile.

A entrevista seguiu de forma que o desenvolvedor validador 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* do validador. Após a discussão ele respondia, considerando a conclusão das discussões, e seguia para a próxima. Esta dinâmica segue até passar pro todas as frases e ao final o validador era incentivado a dar qualquer outro *feedback* que considerasse relevante. Alguns importantes ajustes foram realizados após o primeiro validador.

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 frases ficavam muito maiores de ler e difíceis de serem interpretadas então trocamos para escalas de frequência e importância e ajustamos as frases. Também foram sugeridas melhorias em algumas poucas frases. Com o segundo validador tivemos apenas dois ajustes em frases. Para efeitos de viés, as respostas dos validadores não foram consideradas.

Após a validação das frases, executamos um teste piloto com dois desenvolvedores, do qual, não solicitaram alterações no questionário, considerando-o apropriado para lançamento.



(a) *Experiência* **(b)** *Distribuição geográfica*

Figura 4.4: *Experiência com desenvolvimento de software, desenvolvimento Android e distribuição geográfica dos participantes de* Q2.

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 Brasil*, *Android Brasil Projetos* e o grupo do *Slack Android Dev Br*, maior grupo de desenvolvedores Android do Brasil com 2622 participantes até o momento desta escrita. Para engajar os respondentes ofertamos o sorteio de um Google Chromecast Áudio. O questionário esteve aberto por aproximadamente 3 semanas em meados de Setembro de 2017. As questões 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.

Participantes

Análise dos Dados

Para análise dos dados extraímos a moda e distribuição relativa de cada frase das seções dois e três. Consideramos como maus cheiros *percebidos* todos aqueles com *moda de importância* (I_{Moda}) maior igual a 4 e *moda de frequência* (F_{Moda}) maior igual a 3. Ou seja:

$$F_{Moda} > 3 \cup I_{Moda} \geq 4 \implies MauCheiro$$

As Tabelas 4.2 e 4.3 apresentam respectivamente a moda e distribuição relativa de frequência e importância para cada um dos 21 maus cheiros derivados. Em vermelho são os que não atendem a condição mencionada acima e portanto, são desconsiderados para a etapa seguinte.

Mau Cheiro	Moda	1	2	3	4	5
Classe de UI Inteligente	4	7%	19%	27%	28%	19%
Classe de UI Acoplada	3	7%	20%	33%	24%	15%
Comportamento Suspeito	3	11%	14%	33%	22%	19%
Referência Morta	3	14%	17%	29%	26%	13%
Classes de UI Fazendo IO	3	17%	20%	26%	23%	14%
Arquitetura Não Identificada	3	13%	20%	27%	20%	18%
Adapter Complexo	3	7%	21%	32%	25%	14%
Recursos Mágicos	4	10%	22%	23%	28%	16%
Nome de Recurso Despadronizado	3	11%	22%	26%	25%	16%
Layout Profundamente Aninhado	4	2%	11%	22%	36%	28%
Imagem Dispensável	4	11%	19%	24%	30%	14%
Layout Longo ou Repetido	4	4%	13%	29%	37%	17%
Imagem Faltante	4	15%	24%	21%	29%	11%
Longo Recurso de Estilo	5	5%	11%	23%	28%	32%
Recurso de String Bagunçado	5	6%	8%	20%	32%	33%
Atributos de Estilo Repetidos	4	5%	13%	31%	35%	15%
Reuso Inadequadro de String	4	5%	9%	22%	34%	29%
Listener Escondido	2	24%	30%	19%	16%	10%
Adapter Consumista	2	25%	29%	23%	17%	5%
Uso Excessivo de Fragment	3	8%	20%	30%	27%	15%
Não Uso de Fragment	2	15%	32%	26%	16%	10%

1 = Nunca, 2 = Raramente, 3 = As vezes, 4 = Frequente e 5 = Muito frequente.

Tabela 4.2: *Moda e distribuição relativa sobre percepção da frequência dos maus cheiros por desenvolvedores Android.*

4.1.3 Etapa 3 - Percepção dos maus cheiros

Experimento

todo: ***Ainda não escrito pois estamos finalizando como será feito.***

Participantes

todo: ***Ainda não escrito pois estamos finalizando como será feito.***

Análise dos Dados

todo: ***Aguardando dois anteriores para ser escrito. Mas basicamente extrairemos dados estatísticos cliff e wilcoxon.***

Mau Cheiro	Moda	1	2	3	4	5
Classe de UI Inteligente	5	3%	3%	15%	25%	53%
Classe de UI Acoplada	5	2%	6%	21%	33%	37%
Comportamento Suspeito	4	7%	17%	27%	28%	20%
Referência Morta	5	1%	2%	11%	26%	59%
Classes de UI Fazendo IO	5	2%	6%	13%	19%	60%
Arquitetura Não Identificada	5	1%	2%	9%	18%	69%
Adapter Complexo	5	1%	3%	14%	35%	46%
Recursos Mágicos	5	1%	6%	15%	30%	47%
Nome de Recurso Despadronizado	5	1%	3%	11%	25%	59%
Layout Profundamente Aninhado	4	5%	10%	25%	35%	23%
Imagem Dispensável	5	1%	5%	13%	33%	47%
Layout Longo ou Repetido	5	2%	3%	16%	33%	45%
Imagem Faltante	5	2%	4%	10%	27%	57%
Longo Recurso de Estilo	4	2%	15%	26%	35%	20%
Recurso de String Bagunçado	4	9%	21%	23%	31%	16%
Atributos de Estilo Repetidos	4	1%	2%	17%	41%	38%
Reuso Inadequadro de String	3	16%	20%	26%	23%	14%
Listener Escondido	5	7%	9%	26%	24%	33%
Adapter Consumista	5	1%	4%	9%	26%	58%
Uso Excessivo de Fragment	3	21%	14%	28%	20%	16%
Não Uso de Fragment	4	19%	15%	24%	26%	15%

1 = Não é importante, 2 = Pouco importante, 3 = Razoavelmente importante, 4 = Importante e 5 = Muito.

Tabela 4.3: *Moda e distribuição relativa sobre percepção de importância dos maus cheiros por desenvolvedores Android.*

Capítulo 5

Maus Cheiros do *Front-End* Android

Nesta seção apresentamos as definições dos 13 maus cheiros derivados após as três etapas de pesquisa.

5.1 Maus Cheiros em Componentes Android

Nesta seção apresentamos o catálogo de maus cheiros que afetam componentes do *front-end* Android, podendo ser: *Activities*, *Fragments*, *Adapters* ou *Listeners*.

5.1.1 Classe de UI Inteligente (SML-J1)[20⋆]

Activities[10⋆], Fragments[6⋆], Adapters[5⋆] e Listeners[1⋆] devem conter apenas códigos responsáveis por apresentar, interagir e atualizar a UI. São indícios do mau cheiro a existência de códigos relacionados a lógica de negócio, operações de IO[1], conversão de dados ou campos estáticos nesses elementos.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *"Fazer lógica de negócio [em Activities]"*[2] (P16). *"Colocar regra de negócio no Adapter"* (P19). *"Manter lógica de negócio em Fragments"* (P11). E frases sobre **boas 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).

[1] ver mau cheiro Classes de UI Fazendo IO 5.1.5.
[2] Todo texto em inglês foi traduzido livremente ao longo da dissertação

5.1.2 Classes de UI Acopladas (SML-J2)[6⋆]

Fragments[4⋆], Adapters[1⋆] e Listeners[1⋆] não devem ter referência direta para quem os utiliza. São indícios do mau cheiro a existência de referência direta para Activities ou Fragments nesses elementos.

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

5.1.3 Comportamento Suspeito (SML-J3)[6⋆]

Activities[3⋆], Fragments[2⋆] e Adapters[1⋆] não devem ser responsáveis pela implementação do comportamento dos eventos. São indícios do mau cheiro o uso de classes anônimas, classes internas ou polimorfismo (através de `implements`) para implementar Listeners de modo a responder a eventos do usuário.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *"Usar muitos anônimos pode ser complicado. Às vezes nomear coisas torna mais fácil para depuração"* (P9). *"Mantenha-os [Listeners] em classes separadas (esqueça sobre classes anônimas)"* (P4). *"Muitas implementações de Listener com classes anônimas"* (P8). *"Declarar como classe interna da Activity ou Fragment ou outro componente que contém um ciclo de vida. Isso pode fazer com que os aplicativos causem vazamentos de memória."* (P42). *"Eu não gosto quando os desenvolvedores fazem a activity implementar o Listener porque eles [os métodos] serão expostos e qualquer um pode chamá-lo de fora da classe. Eu prefiro instanciar ou então usar ButterKnife para injetar cliques."* (P44). E sobre **boas práticas**: *"Prefiro declarar os listeners com implements e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto"* (P32). *"Tome 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).

5.1.4 Entenda o Ciclo de Vida (SML-J4)[5⋆]

O Ciclo de Vida de Activities[3⋆] e Fragments[3⋆] é bem delicado e elaborado, logo o uso dele exige um conhecimento mais profundo, caso contrário pode resultar em *memory leaks* e

outros problemas. São indícios do mau cheiro ter estes elementos como *callbacks* de processos assíncronos, efetivar a transação de FRAGMENTS (através do `FragmentTransaction.commit`) após o `onPause` da ACTIVITY ou o não tratamento da restauração do estado de ACTIVITIES e FRAGMENTS após por exemplo, rotação da tela.

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

5.1.5 Classes de UI Fazendo IO (SML-J8)[3⋆]

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

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

5.1.6 Activity Inexistente (SML-J9)[2⋆]

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

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *"Fazer Activities serem callbacks de processos assíncronos gerando memory leaks. Erros ao interpretar o ciclo de vida"* (P28). *"Ter referência estática para Activities, resultando em vazamento de memória"* (P31). E sobre **boas práticas**: *"Não manter referências estáticas para Activities (ou classes anônimas criadas dentro delas)"* (P31). *"Deus mata um cachorro toda vez que alguém passa o contexto da Activity para um componente que tem um ciclo de vida independente dela. Vaza memória e deixa todos tristes."* (P4).

5.1.7 Arquitetura Não Identificada (SML-J10)[2⋆]

São indícios do mau cheiro quando diferentes ACTIVITIES[2⋆] e FRAGMENTS[1⋆] no projeto apresentam fluxos de código complexos, possivelmente são CLASSE DE UI INTELIGENTE

5.1.1, onde não é possível identificar uma organização padronizada entre eles que aponte para algum padrão arquitetural, como por exemplo, MVC (*Model View Controller*), MVP (*Model View Presenter*), MVVM (*Model View ViewModel*) ou *Clean Architecture*.

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

5.1.8 Adapter Complexo (SML-J11)[2⋆]

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

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

5.2 Maus Cheiros Em Recursos

Nesta seção apresentamos o catálogo com oito maus cheiros que afetam recursos do *front-end* Android, podendo ser: recursos de *layout*, recursos de *string*. recursos de *style* ou recursos *drawable*.

5.2.1 Nome de Recurso Despadronizado (SML-R2)[8⋆]

São indícios do mau cheiro quando Recursos de Layout[2⋆], Recursos de Texto[4⋆], Recursos de Estilo[2⋆] e Recursos Gráficos[1⋆] não possuem um padrão de nomenclatura.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *"O nome das strings sem um contexto"* (P8). *"[Sobre Style Resources] Nada além de ter uma boa convenção de nomes"* (P37). *"[Sobre Layout Resource] Mantenha uma convenção de nomes da sua escolha"* (P37). E sobre **boas práticas**: *"Iniciar o nome de uma string com o nome da tela onde vai ser usada"* (P27). *"[Sobre Layout Resource] Ter uma boa convenção de nomeação"* (P43). *"[Sobre Style Resource] colocar um bom nome"* (P11).

5.2.2 Layout Profundamente Aninhado (SML-R3)[7⋆]

São indícios desse mau cheiro o uso de profundos aninhamentos na construção de Recursos de Layout[7⋆], ou seja, `ViewGroups` contendo outros `ViewGroups` sucessivas vezes. O site oficial do Android conta com informações e ferramentas automatizadas para lidar com esse sintoma [61].

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

5.2.3 Imagem Dispensável (SML-R4)[6⋆]

O Android possui diversos tipos de Recursos Gráficos[6⋆] que podem substituir imagens tradicionais como `.png`, `.jpg` ou `.gif` a um custo menor em termos de tamanho do arquivo e sem a necessidade de haver versões de diferentes tamanhos/resoluções. São indícios do mau cheiro a existência de imagens com, por exemplo, cores sólidas, degradês ou estado de botões, que poderiam ser substituídas por Recursos Gráficos de outros tipos como Shapes, State Lists ou Nine-Patch File ou a não existência de imagens vetoriais, que podem ser redimensionadas sem a perda de qualidade.

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

5.2.4 Layout Longo ou Repetido (SML-R5)[5⋆]

São indícios do mau cheiro quando Recursos de Layout[5⋆] é muito grande ou possue trechos de layout muito semelhantes ou iguais a outras telas.

Um exemplo de frase sobre **má prática** que embasou esse mau cheiro é: *"Copiar e colar layouts parecidos sem usar includes"* (P41). *"Colocar muitos recursos no mesmo arquivo de layout."* (P23). E sobre **boas práticas**: *"Sempre quando posso, estou utilizando includes para algum pedaço de layout semelhante"* (P32). *"Criar layouts que possam ser reutilizados em diversas partes"* (P36). *"Separe um grande layout usando include ou merge"* (P42)

5.2.5 Imagem Faltante (SML-R6)[4⋆]

As imagens devem ser disponibilizadas em mais de um tamanho/resolução para que o Android possa realizar otimizações. São indícios do mau cheiro haver apenas uma versão de algum Recursos Gráfico[4⋆] do tipo png, jpg ou gif ou ainda, ter imagens em diretórios incorretos em termos de `dpi`.

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

5.2.6 Longo Recurso de Estilo (SML-R7)[3⋆]

É indício do mau cheiro haver apenas um Recurso de Estilo[4⋆] ou conter Recursos de Estilo muito longos.

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

5.2.7 Recurso de String Bagunçado (SML-R8)[3⋆]

É indício do mau cheiro o uso de apenas um arquivo para todos os Recursos de Texto[3⋆] do aplicativo e a não existência de um padrão de nomenclatura e separação para os Recursos de Texto de uma mesma tela.

Alguns exemplos de frases sobre **más práticas** que embasaram esse mau cheiro são: *"Usar o mesmo arquivo strings.xml para tudo"* (P28). *"Não orgaizar as strings quando o strings.xml começa a ficar grande"* (P42). E sobre **boas práticas**: *"Separar strings por tela em arquivos XML separados. Extremamente útil para identificar quais strings pertencentes a quais telas em projetos grandes"* (P28). *"Sempre busco separar em blocos, cada bloco representa uma Activity e nunca aproveito uma String pra outra tela"* (P32).

5.2.8 Atributos de Estilo Repetidos (SML-R9)[3⋆]

É indício do mau cheiro haver Recursos de Layout[1⋆] ou Recursos de Estilo[2⋆] com blocos de atributos de estilo repetidos.

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

Capítulo 6

Conclusão

todo:* *Capítulo não concluído, favor desconsiderar na revisão

6.1 Resumo

Nesta dissertação investigamos a existência de boas e más práticas em elementos usados para implementação de *front-end* de projetos Android: Activities, Fragments, Listeners, Adapters, Layout, Styles, String e Drawable. Fizemos isso através de um estudo exploratório qualitativo onde coletamos dados por meio de um questionário online respondido por 45 desenvolvedores Android. A partir deste questionário mapeamos 23 más práticas e sugestões de solução, quando mencionado por algum participante. Após, validamos a percepção de desenvolvedores Android sobre as quatro mais recorrêntes dessas más práticas: Lógica em Classes de UI, Nome de Recurso Despadronizado, Recurso Mágico e Layout Profundamente Aninhado. Fizemos isso através de um experimento online respondido por 20 desenvolvedores Android, onde os participantes eram convidados a avaliar 6 códigos com relação a qualidade.

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

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

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

Validamos a percepção de desenvolvedores sobre as quatro más prática mais recorrêntes. Concluímos que desenvolvedores de fato as percebem como más práticas. Duas das más práticas, Lógica em Classes de UI e Layout Profundamente Aninhado foram possíveis confirmar com dados estatísticos. Outras duas, Nome de Recurso Despadronizado e Recurso Mágico, embora os dados estatísticos não tenham confirmado, notamos por meio das respostas abertas que existe esta percepção.

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

6.2 Limitações

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

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

Nossa pesquisa tenta replicar o método utilizado por Aniche [12] ao investigar cheiros de código no arcabouço Spring MVC. Entretanto, nos deparamos com situações diferentes, das quais, após a execução nos questionamos se aquele método seria o mais adequado para todos os contextos neste artigo. Por exemplo, nosso resultado com a má prática RM nos levou a conjecturar se desenvolvedores consideram problemas em códigos Java mais severos que problemas em recursos do aplicativo. O que nos levou a pensar sobre isso foi que, apesar do resultado, obtivemos muitas respostas que se aproximavam da definição da má prática RM. Desta forma, levantamos que de todos os recursos avaliados, 74% receberam severidade igual ou inferior a 3, contra apenas 30% com os mesmo níveis de severidade em código Java. Desta forma, uma alternativa é repensar a forma de avaliar a percepção dos desenvolvedores sobre más práticas que afetem recursos do aplicativo.

6.3 Trabalhos Futuros

6.4 Propostas de soluções

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

todo: ***Como os achados do estudo ajudam a melhorar o desenvolvimento de aplicações para Android, ou apoiar a identificação, priorização e minimização de más práticas?***

todo: ***Relacionar as más práticas identificadas com os maus cheiros tradicionais. O que há de semelhança, faz sentido definir um diferente/específico ao android ou basta aplicar o conceito do tradicional a plataforma (exemplo de style grande pode ser aplicado o conceito de large class?)?***

Apêndice A

Questionário sobre boas e más práticas

Android Good & Bad Practices

Help Android Developers around the world in just 15 minutes letting us know what you think about good & bad practices in Android native 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 throught this session were not mandatory.

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

- Do you have anything you consider a bad practice when dealing with Styles Resources?

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

Last thoughts Only 3 more final questions.

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

Apêndice B

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). "Integragir com outro Fragment diretamente" (P45). "[Listener] conter uma referência direta à Activities" (P4, P40). "[Adapters] alto acoplamento com a Activity" (P10). "Acessar Activities ou Fragments diretamente" (P45). **Boa prática:** "Seja um componente de UI reutilizável. Então evite dependência de outros componentes da aplicação" (P6).
SML-J3	**Más práticas:** "Usar muitos anônimos pode ser complicado. Às vezes nomear coisas torna mais fácil para depuração" (P9). "Mantenha-os [Listeners] em classes separadas (esqueça sobre classes anônimas)" (P4). "Muitas implementações de Listener com classes anônimas" (P8). "Declarar como classe interna da Activity ou Fragment ou outro componente que contém um ciclo de vida. Isso pode fazer com que os aplicativos causem vazamentos de memória." (P42). "Eu não gosto quando os desenvolvedores fazem a activity implementar o Listener porque eles [os métodos] serão expostos e qualquer um pode chamá-lo de fora da classe. Eu prefiro instanciar ou então usar ButterKnife para injetar cliques." (P44). **Boas práticas:** "Prefiro declarar os listeners com `implements` e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto" (P32). "Tome 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).

[1]Todo 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 [70].
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:** "Sobreescrever o comportamento do botão voltar" (P43). "Lidar com a pilha do app manualmante" (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 as desenhe [através de Drawable Resources]" (P37). **Boas práticas:** "Quando possível, criar resources através de xml" (P36). "Utilizar o máximo de Vector Drawables que for possível" (P28). "Evite muitas imagens, use imagens vetoriais sempre que possível" (P40).
SML-R5	**Má prática:** "Copiar e colar layouts parecidos sem usar includes" (P41). "Colocar muitos recursos no mesmo arquivo de layout." (P23). **Boas práticas:** "Sempre quando posso, estou utilizando includes para algum pedaço de layout semelhante" (P32). "Criar layouts que possam ser reutilizados em diversas partes" (P36). "Separe um grande layout usando include ou merge" (P42)
SML-R6	**Más práticas:** "Ter apenas uma imagem para multiplas densidades" (P31). "Baixar uma imagem muito grande quando não é necessário. Há melhores formas de usar memória" (P4). "Não criar imagens para todas as resoluções" (P44). **Boas prática:** "Nada especial, apenas mantê-las em seus respectivos diretórios e ter variados tamanhos delas" (P34). "Criar as pastas para diversas resoluções e colocar as imagens corretas" (P36).
SML-R7	**Más práticas:** "Deixar tudo no mesmo arquivo styles.xml" (P28). "Arquivos de estilos grandes" (P8). **Boas práticas:** "Se possível, separar mais além do arquivo styles.xml padrão, já que é possível declarar múltiplos arquivos XML de estilo para a mesma configuração" (P28). "Divida-os. Temas e estilos é uma escolha racional" (P40).
SML-R8	**Más práticas:** "Usar o mesmo arquivo strings.xml para tudo" (P28). "Não orgaizar as strings quando o strings.xml começa a ficar grande" (P42). **Boas práticas:** "Separar strings por tela em arquivos XML separados. Extremamente útil para identificar quais strings pertencentes a quais telas em projetos grandes" (P28). "Sempre busco separar em blocos, cada bloco representa uma Activity e nunca aproveito uma String pra outra tela" (P32).
SML-R9	**Má prática:** "Utilizar muitas propriedades em um único componente. Se tiver que usar muitas, prefiro colocar no arquivo de styles." (P32). **Boa prática:** "Sempre que eu noto que tenho mais de um recurso usando o mesmo estilo, eu tento movê-lo para o meu style resource." (P34).
SML-R10	**Más práticas:** "Utilizar uma String pra mais de uma activity, pois se em algum momento, surja a necessidade de trocar em uma, vai afetar outra." (P32). "Reutilizar a string em várias telas" (P6) "Reutilizar a string apenas porque o texto coincide, tenha cuidado com a semântica" (P40). **Boas prática:** "Sempre busco separar em blocos, cada bloco representa uma activity e nunca aproveito uma String pra outra tela." (P32). "Não tenha medo de repetir strings [...]" (P9).

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

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 detre as opções Não conheço, Iniciante, Intermediário, Avançado e Especialista). As tecnologias apresentadas foram: Java, Ruby, C/C++, Kotlin, Objective-C, Swift, Android, C#, Ptyhon, 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 Frequente, Frequente, 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 LinerarLayout.

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.

Referências Bibliográficas

[1] Ward's wiki - code smell. http://wiki.c2.com/?CodeSmell, 1995. Acessado em 05/10/2017.

[2] Android fragmentation visualized. http://opensignal.com/reports/2015/08/android-fragmentation, August 2015. Acessado em 12/09/2016.

[3] Wikipedia code smell. https://en.wikipedia.org/wiki/Code_smell, 2016. Acessado em 14/11/2016.

[4] Consortium for it software quality, 2017.

[5] 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. Acessado em 24/07/2017.

[6] ISO/IEC 25010:2011. Systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models. https://www.iso.org/standard/35733.html, 2011. Acessado em 23/10/2017.

[7] ISO/IEC TR 9126-1:2001. Software engineering - product quality - part 1: Quality model. https://www.iso.org/standard/22749.html, 1991. Acessado em 23/10/2017.

[8] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. empirical software engineering. 2011.

[9] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A pattern language: Towns, buildings, construction (center for environmental structure). 1977.

[10] Open Handset Alliance. Open handset alliance releases android sdk. https://www.openhandsetalliance.com/press_111207.html, 2007. Acessado em 10/11/2017.

[11] MaurÃcio Aniche, Bavota G., Treude C., Van Deursen A., and Gerosa M. A validated set of smells in model-view-controller architectures. 2016.

[12] Maurício Aniche and Marco Gerosa. Architectural roles in code metric assessment and code smell detection. 2016.

[13] Roberta Arcoverde, Alessandro Garcia, and 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.

[14] Boehm Berry W., Brown J.R., Kaspar H., Lipow M., Macleod G.J., and Merrit M.J. *Characteristics of software quality*. TRW series of software technology. North-Holland Pub. Co., 1978.

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

[16] Suelen G. Carvalho. Apêndice online. http://suelengc.com/android-code-smells-article/, 2017. Acessado em 06/05/2017.

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

[18] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications Ltd, 3 edition, 2007.

[19] E. Van Emden and L. 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.

[20] A. Milani Fard and A. Mesbah. Jsnose: Detecting javascript code smells. 2013.

[21] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

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

[24] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1 edition, 1999.

[25] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. 2012.

[26] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, 1987.

[27] G. Hecht. An approach to detect android antipatterns. 2:766–768, May 2015.

[28] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, May 2015.

[29] ISO. Iso 9000:2000. https://www.iso.org/standard/29280.html. Acessado em 23/10/2017.

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

[31] Java. What is java technology and why do i need it? https://www.java.com/en/download/faq/whatis_java.xml. Acessado em 21/11/2017.

[32] Juran Joseph M. and Godfrey A. Blanton. *Juran's Quality Handbook*. McGraw-Hill, 5ª edition, 1998.

[33] Kerievzky Joshua. A timeless way of communicating. https://www.slideshare.net/JoshuaKerievsky/a-timeless-way-of-communicating-alexandrian-pattern-languages, 2017. Acessado em 24/11/2017.

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

[35] Mario Linares-Vásquez, Sam Klock, Collin Mcmillan, Aminata Sabanè, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 2017.

[36] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding code smells in android applications. 2017.

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

[38] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in software quality: Concept and definitions of software quality. I:188, 1977.

[39] Steve McConnell. *Code Complete*. Microsoft Press, Redmond, WA, USA, 2004.

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

[41] J. Nielsen. Why you only need to test with 5 users. 2000. Acessado em 05/11/2017.

[42] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and 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.

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

[44] Martin Pinzger, Felienne Hermans, and 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.

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

[46] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2014.

[47] Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications Ltd, 2 edition, 2012.

[48] Android Developer Site. Plataform architecture. https://developer.android.com/guide/platform/index.html. Acessado em 04/09/2016.

[49] Android Developer Site. Ui events. https://developer.android.com/guide/topics/ui/ui-events.html. Acessado em 25/11/2016.

[50] Android Developer Site. Android studio. https://developer.android.com/studio/index.html, 2016. Acessado em 30/08/2016.

[51] Android Developer Site. Building your firt app. https://developer.android.com/training/basics/firstapp/creating-project.html, 2016. Acessado em 31/03/2017.

[52] Android Developer Site. Documentação site android developer. https://developer.android.com, 2016. Acessado em 27/10/2016.

[53] Android Developer Site. Drawable resources. https://developer.android.com/guide/topics/resources/drawable-resource.html, 2016. Acessado em 20/11/2017.

[54] Android Developer Site. Layout resources. https://developer.android.com/guide/topics/resources/layout-resource.html, 2016. Acessado em 20/11/2017.

[55] Android Developer Site. Layouts. https://developer.android.com/guide/topics/ui/declaring-layout.html, 2016. Acessado em 23/10/2016.

[56] Android Developer Site. Providing resources. https://developer.android.com/guide/topics/resources/providing-resources.html, 2016. Acessado em 08/09/2016.

[57] Android Developer Site. String resources. https://developer.android.com/guide/topics/resources/string-resource.html, 2016. Acessado em 20/11/2017.

[58] Android Developer Site. Style resources. https://developer.android.com/guide/topics/resources/style-resource.html, 2016. Acessado em 20/11/2017.

[59] Android Developer Site. Android fundamentals. https://developer.android.com/guide/components/fundamentals.html, 2017. Acessado em 04/09/2016.

[60] Android Developer Site. Handling lifecycles with lifecycle-aware components. https://developer.android.com/topic/libraries/architecture/lifecycle.html, 2017. Acessado em 23/11/2017.

[61] Android Developer Site. Optimizing view hierarchies). https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html), 2017. Acessado em 09/04/2017.

[62] Android Developer Site. Processes and threads. https://developer.android.com/guide/components/processes-and-threads.html#Threads, 2017. Acessado em 23/11/2017.

[63] Android Developer Site. Project overview. https://developer.android.com/studio/projects/index.html, 2017. Acessado em 19/11/2017.

[64] Android Developer Site. Services. https://developer.android.com/guide/components/services.html, 2017. Acessado em 19/11/2017.

[65] Android Developers Site. Activities. https://developer.android.com/guide/components/activities.html, 2016. Acessado em 29/08/2016.

[66] Android Developers Site. Activitiy reference. https://developer.android.com/reference/android/app/Activity.html, 2016. Acessado em 29/08/2016.

[67] Android Developers Site. Async task. https://developer.android.com/guide/components/broadcasts.html, 2016. Acessado em 29/08/2016.

[68] Android Developers Site. Async task. https://developer.android.com/reference/android/os/AsyncTask.html, 2016. Acessado em 29/08/2016.

[69] Android Developers Site. Fragments. https://developer.android.com/guide/components/fragments.html, 2016. Acessado em 20/11/2017.

[70] Android Developers Site. Android recyclerview. https://developer.android.com/training/material/lists-cards.html, 2017. Acessado em 12/04/2017.

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

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

[73] 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.

[74] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[75] Stefan Wagner. *Software Product Quality Control*. Springer-Verlag Berlin Heidelberg, 2013.

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

[77] Wikipédia. ios. https://en.wikipedia.org/wiki/IOS#Devices, 2017. Acessado em 23/11/2017.

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

[79] Aiko Yamashita and 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.

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