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

Suelen G. Carvalho
Universidade de São Paulo
Rua do Matão, 1010
São Paulo, SP 05508-090
suelengc@ime.usp.br

Marco Aurélio Gerosa
Northern Arizona University
E Runke Dr
Flagstaff, Arizona 86011
marco.gerosa@nau.edu

Maurício Aniche
Delft University of Technology
Mekelweg 2
Delft, The Netherlands 2628 CD
m.f.aniche@tudelft.nl

ABSTRACT

Existem muitas práticas, ferramentas e padrões que ajudam desenvolvedores na busca por produzir código com qualidade, dentre elas podemos citar catálogos de maus cheiros, que indicam possíveis problemas no código. Esses catálogos possibilitam a implementação de ferramentas de detecção automática de trechos de códigos problemáticos ou mesmo a inspeção manual. Apesar de já existirem diversos cheiros de código catalogados, pesquisas sugerem que tecnologias diferentes apresentaram cheiros de código específicos, e uma tecnologia que tem chamado a atenção de muitos pesquisadores é o Android. Neste artigo, nós investigamos a existência de cheiros de código em projetos Android. Por meio de um *survey* com 45 desenvolvedores, descobrimos que além de cheiros de código tradicionais, o uso de algumas estruturas específicas da plataforma são amplamente percebidas como más práticas, portanto, possíveis cheiros de código específicos. Identificamos 23 más práticas específicos ao *front-end* Android e validamos a percepção sobre quatro delas com 20 desenvolvedores. Descobrimos que fato há boas e más práticas específicas ao *front-end* Android. Elaboramos um catálogo com 23, das quais investigamos a percepcão de desenvovlvedores sobre quatro e validamos com sucesso a percepção sobre duas.

KEYWORDS

Android, cheiros de código, qualidade de código

ACM Reference format:
Suelen G. Carvalho, Marco Aurélio Gerosa, and Maurício Aniche. 2017. Maus cheiros de código em aplicativos Android: Um estudo sobre a percepção dos desenvolvedores. In *Proceedings of SBES 2017: 31st Brazilian Symposium on Software Engineering, Fortaleza, Ceará Brasil, Setembro 2017 (SBES'17),* 12 pages.
DOI: 10.475/123_4

SBES'17, Fortaleza, Ceará Brasil

DOI: 10.475/123_4

1 INTRODUÇÃO

Escrever código com qualidade tem se tornado cada vez mais importante com o aumento da complexidade de tecnologias e anseio dos usuários por novas funcionalidade e atualizações [30, 50]. Existem diferentes práticas, padrões e ferramentas que auxiliam os desenvolvedores a escreverem código com qualidade, incluindo *design patterns* [25] e cheiros de código [23]. A falta de qualidade resulta em defeitos de software que custam a empresas quantias significativas, especialmente quando conduzem a falhas de software [15, 39]. Evolução e manutenção de software também já se provaram como os maiores gastos com aplicações [49].

Uma das formas de aumentar a qualidade de software é identificar trechos de códigos ruins e refatorá-los, ou seja, alterar o código sem alterar o comportamento [23]. Desta forma, temos que cheiros de código são aliados importantes na busca por qualidade de código pois, representam sintomas que podem indicar problemas mais profundos no software, não necessariamente, sendo o problema em si [24]. Seu mapeamento possibilita a definição de heurísticas que, por sua vez, possibilitam a implementação de ferramentas que os identificam de modo automático no código. PMD [8], Checkstyle e FindBugs são exemplos de ferramentas que identificam automaticamente alguns tipos de cheiros de código em projetos Java.

Enquanto que maus cheiros de código em projetos Java já foram extensivamente estudados [23, 32, 38], ainda há muito a se pesquisar sobre cheiros de código em projetos Android.

No entanto, determinar o que é ou não um mau cheiro de código é subjetivo e pode variar de acordo com a tecnologia, desenvolvedor, metodologia de desenvolvimento dentre outros aspectos [9]. Em particular, Aniche et al. [12, 14] mostraram que a arquitetura do software é um fator importante e que deve ser levada em conta ao analisar a qualidade de um sistema. Outros estudos identificaram cheiros de código específicos Android relacionados ao consumo inteligente de recursos do dispositivo, como bateria e memória, usabilidade, dentre outros [28, 42].

Nossa pesquisa complementa as anteriores no sentido de que também buscamos por cheiros de código Android. E se difere delas pois buscamos cheiros de código relacionados à qualidade, em termos de manutenabilidade e legibilidade, específico dessa plataforma. Por exemplo Activities, Fragments e Adapters são classes usadas na construção de telas e listeners são responsáveis pelas interações com os

usuários. Buscamos entender então quais são as *boas e más* práticas no desenvolvimento da interface visual Android.

Por meio de um questionário online com perguntas sobre boas e más práticas no desenvolvimento do *front-end* Android respondido por 45 desenvolvedores, derivamos 23 más práticas. Validamos a percepção dos desenvolvedores sobre essas más práticas através de um experimento com outros 20 desenvolvedores. Os resultados mostram que de fato existem boas e más práticas específicas ao *front-end* Android. Ainda que tenhamos validado com sucesso a percepção dos desenvolvedores sobre apenas duas, esperamos que este catálogo de más práticas possa contribuir com ideias iniciais para outras pesquisas sobre qualidade de codigo em projetos Android.

As contribuições deste trabalho são:

(1) Catálogo com 23 más práticas e sugestões de soluções no desenvolvimento do *front-end* Android, derivadas a partir dos resultados obtidos com a aplicação de um questionário online respondido por 45 desenvolvedores.
(2) A percepção de desenvolvedores sobre as quatro más práticas de alta recorrência através de um questionário online com 20 desenvolvedores. Sendo que obtivemos um resultado positivo e estatisticamente válido sobre duas delas.
(3) Apêndice online [16] com roteiros dos questionários e outras informações da pesquisa para que outros pesquisadores possam replicar nosso estudo.

As seções seguintes deste artigo estão organizadas da seguinte forma: a Seção 2 aborda a metodologia de pesquisa. A Seção 3 apresenta o catálogo de más práticas a percepção de desenvovledores sobre as quatro mais recorrêntes. A Seção **??** discute pontos relevantes. A Seção 4 aborda as ameaças à validade do estudo. A Seção 5 discute os trabalhos relacionados e o estado da arte sobre Android e cheiros de código. E por fim, a Seção 6 conclui.

2 METODOLOGIA

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

QP1. O que desenvolvedores consideram boas e más práticas no desenvolvimento Android?
QP2. Códigos afetados por estas más práticas são percebidos pelos desenvolvedores como problemáticos?

Para alcançar nosso objetivo, conduzimos um estudo qualitativo e exploratório no qual coletamos dados através de um questionário e, na sequência, conduzimos um estudo online para capturar a percepção de desenvolvedores Android em relação à classes afetadas pelos maus cheiros.

Esta Seção está organizada da seguinte forma: na Seção 2.1 abordamos o processo de descoberta das boas e más práticas Android e na Seção 2.2 abordamos o processo de validação da percepção dos desenvolvedores sobre as boas e más práticas definidas.

2.1 QP1: Boas e Más Práticas Android

Para responder a **QP1**, buscamos entender o que desenvolvedores consideram boas e más práticas no desenvolvimento do *front-end* Android. Para tal, optamos por um estudo qualitativo. Segundo Strauss e Corbin [18], há muitos motivos válidos para se fazer pesquisa qualitativa, dentre eles a natureza do problema de pesquisa e para obter mais informações sobre áreas cujo ainda se sabe pouco, como é o caso de cheiros de código Android. Muitas vezes, a pesquisa qualitativa, constitui-se na primeira etapa de uma investigação mais ampla na qual se busca o entendimento de um assunto específico por meio de descrições, comparações e interpretações dos dados [40, 41].

Algumas características básicas da pesquisa qualitativa, como *a)* o foco na interpretação que os participantes possuem quanto à situação investigada e *b)* o fato de enfatizar a subjetividade e a flexibilidade, orientando-se para o processo e não para o resultado, justificam seu uso neste artigo [35, 40].

Para definirmos quais elementos representam o *front-end* Android, fizemos umas extensa revisão da documentação oficial e chegamos nos seguintes itens: Activities, Fragments, Listeners, Adapters e os recursos do aplicativo, que são arquivos XML ou imagens utilizados na interface visual, como por exemplo Drawables, Layouts, Styles e Colors. Como existem muitos tipos de recursos do aplicativo [48], selecionamos quatro: Layout, Styles, String e Drawable. Optamos por esses recursos pois estão presentes no template padrão do Android Studio [47], IDE oficial para desenvolvimento de projetos da plataforma Android [2].

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

A primeira seção é composta por questões cujo objetivo era traçar o perfil demográfico do participante (idade, estado de residência, experiência em desenvolvimento de software e em desenvolvimento Android). A segunda seção é composta por 16 questões opcionais e dissertativas sobre boas e más práticas relacionadas ao *front-end* Android. Para cada elemento do *front-end* Android foram feitas duas perguntas abertas, onde a participante deveria discutir sobre boas e más práticas percebidas por ela naquele elemento. A terceira seção é composta por 3 perguntas opcionais e dissertativas, 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.

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

Para análise dos dados seguimos a abordagem de *Ground Theory* (GT), um método de pesquisa exploratória originada nas ciências sociais [18, 27], mas cada vez mais popular em pesquisas de engenharia de software [10]. A GT é uma abordagem indutiva, pelo qual dados provindos, por exemplo de, entrevistas ou questionários, são analisadas para derivar uma teoria. O objetivo é descobrir novas perspectivas mais do que confirmar alguma já existente. Realizamos um processo de codificação aberta sobre os dados, resultando num conjunto com 23 categorias de boas e más práticas Android. Essas categorias foram agrupadas de acordo com sua recorrência nas respostas, ou seja, a quantidade de respostas que determinada categoria é percebidas, quanto mais respostas, maior a recorrência. Explicamos o processo de análise com mais detalhes na Seção 2.1.2.

2.1.1 Participantes. 80% dos participantes responderam pelo menos 3 perguntas sobre boas e más práticas no *front-end* Android (7 responderam de 3 a 6, 6 responderam de 8 a 10 e 23 responderam 13 ou mais, sendo que desses, 14 responderam todas) e apenas 20% responderam uma (2 participantes) ou nenhuma (7 participantes). A pergunta solicitando o email foi respondida por 53% dos participantes, o que pode indicar um interesse legítimo da comunidade de desenvolvedores Android pelo tema, reforçando a relevância do estudo. A pergunta mais respondida foi a Q1 e a menos respondida foi a Q18, é possível ver detalhes desta análise na Tabela 1 em nosso apêndice online.

Com a análise das questões demográficas, notamos que atingimos com sucesso *desenvolvedores Android com variados níveis de experiência e de diversas regiões* pois: 1) 100% dos participantes indicaram possuir alguma experiência com desenvolvimento Android, 2) menos de 14% indicaram possuir 1 ano ou menos de experiência com Android e mais de 86% indicaram 2 anos ou mais (15,5% 2 anos, 13,3% 4 anos, 6,5% 5 anos, 15,5% 6 anos, 4,4% 7 anos e 4,4% 8 anos), 4) 36 respostas foram do Brasil, 7 de países europeus e 1 dos Estados Unidos (Califórnia). Vale lembrar que a plataforma Android completa 10 anos em 2017, ou seja, 5 anos de experiência nessa plataforma representa 50% do tempo de vida dela desde seu anúncio em 2007. Os dados sobre a experiência dos participantes são apresentados na Figura **??**.

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

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

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

Em seguida, realizamos a *codificação* sobre as boas e más práticas [18, 43]. Codificação é o processo pelo qual são extraídos categorias de um conjunto de afirmações através da abstração de ideias centrais e relacões entre as afirmações [18]. Durante esse processo, cada resposta recebeu uma ou mais categorias. Também durante esse processo, houveram 30 respostas que não eram triviais de identificar uma categoria ou mesmo de dizer se essas respostas deveriam ser consideradas. Essas respostas foram marcadas como *"talvez"* e reavaliadas ao final. Para toda resposta desconsiderada nesse passo, foi indicado um motivo que pode ser conferido nos arquivos em nosso apêndice online.

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

Ao final da análise constavam 389 respostas individualmente categorizadas sobre boas e más práticas no *front-end* Android.

2.2 QP2: Percepção dos Desenvolvedores

Para responder a **QP2**, buscamos entender a percepção dos desenvolvedores sobre as quatro boas e más práticas classificadas com alta recorrência. As opiniões foram coletadas

através de um estudo online (*S2*) respondido por 20 desenvolvedores Android. Nossas análises demonstram que de fato, códigos afetados pelas más práticas são percebidos pelos desenvolvedores como códigos problemáticos.

O experimento foi composto por duas seções principais. A primeira objetivou coletar informações básicas sobre os antecedentes dos participantes e, em particular, sobre sua experiência. Na segunda seção, os participantes foram solicitados a examinar seis códigos-fonte Android e, para cada uma deles, responder as seguintes perguntas:

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

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

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

No teste piloto realizado com 2 desenvolvedores não foram identificados ponto a otimizar, essas respostas foram desconsideradas para reduzir viés. O questionário do experimento esteve disponível por 8 dias, de 27 de Abril a 4 de Maio de 2017. Sua divulgação foi feita em duas etapas, na primeira, foi enviada ao grupo do Slack Android Dev Br, uma chamada a desenvolvedores com mais de 3 anos e experiência em Android, desta forma, desenvolvedores que tinham interesse de responder o questionário e, atendiam ao requisito de experiência, entravam em contato e enviávamos um email convite. Na segunda etapa, abrimos o questionário para participação de qualquer desenvolvedor Android. A divulgação nesta etapa, foi a mesma utilizada na divulgação em *S1*.

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

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

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

Para comparar as distribuições da severidade indicada pelos participantes, utilizamos o teste de Mann-Whitney não pareado [51] para analisar a significância estatística das diferenças entre a severidade atribuída pelos participantes aos problemas que observam em códigos Android cheirosos e códigos limpos. Os resultados são considerados estatisticamente significativos em $\alpha \leq = 0{,}05$. Também estimamos a magnitude das diferenças medidas usando o Delta de Cliff (ou d), uma medida de tamanho do efeito não paramétrico [29] para dados ordinais. Seguimos diretrizes bem estabelecidas para interpretar os valores do tamanho do efeito: insignificante para $| d | < 0{,}14$, baixo para $0{,}14 \leq | d | < 0{,}33$, médio para $0{,}33 \leq | d | < 0{,}474$, e alto para $| d | \geq 0{,}474$ [49]. Finalmente, relatamos achados qualitativos derivados das respostas abertas dos participantes.

3 RESULTADOS

Durante o processo de codificação emergiram 52 boas e más práticas. Classificamos essas práticas de acordo com sua recorrência: alta (acima de 20 respostas), média (de 8 a 20 respostas), baixa (de 3 a 7 respostas) ou baixíssima (menos de 3 respostas).

A Tabela 1 apresenta o total de ocorrências de cada prática (*S1*). A última linha da tabela, #Categorias, apresenta quantas categorias emergiram de cada questão, como cada questão está diretamente ligada a um elemento do *front-end* Android, podemos interpretá-la da seguinte forma: *quais são os pontos de atenção a serem analisados em determinado elemento Android?* A última coluna da tabela, #Q, apresenta em quantas questões cada categoria surgiu, podemos interpretá-la da seguinte forma: *com base na categoria, quais elementos devem ser investigados?* Duas práticas, Classe Deus/Longa [7, 32] e Herança [52], são conceitos pré-existentes e portanto não são tratados neste artigo.

Esta seção está organizada em 4 subseções onde, nas três primeiras, definimos as práticas de alta, média e baixa recorrência, *totalizando 23 más práticas*. Na última subseção apresentamos os resultados obtidos no estudo sobre a percepção de desenvolvedores com relação às más práticas de alta recorrência. As práticas de baixíssima recorrência são brevemente discutidas na Seção **??**.

3.1 Más Práticas de Alta Recorrência

3.1.1 Lógica Em Classes de UI (LCUI). Indica como má prática haver regras de negócio nos elementos como Activities, Fragments, Listeners e Adapters e indicam como boas práticas que esses mesmos elementos contenham apenas códigos relacionados a interface com o usuário. Para isso sugerem o uso de padrões como: *Model-View-Presenter* (MVP) [4, 5], *Model-View-ViewModel* (MVVM) [6] e *Clean Architecture* [37]. Exemplos de frases que indicaram más práticas são: P16 sobre Activities diz *"Fazer lógica de negócio"* (tradução livre)[1], P19 diz *"Colocar regra de negócio no adapter"* e P11 diz *"Manter lógica de negócio em Fragments"*. Exemplos de frases que indicaram boas prática são: P16 diz *"Elas [*Activities*] representam uma única tela e apenas interagem com a UI, qualquer lógica deve ser delegada para outra classe"*, P23 diz *"Apenas código relacionado à Interface de Usuário nas Activities"*, P40 diz *"Adapters devem apenas se preocupar sobre como mostrar os dados, sem trabalhá-los"*, P2 diz *"As activities que eu crio normalmente tem um propósito único e estado básico [...] eu uso MVP a maior parte do tempo, então minhas activities normalmente representam uma view no MVP"*. Os elementos afetados por essa categoria são: Activities, Fragments, Listeners e Adapters.

3.1.2 Nome de Recurso Despadronizado (NRD). Indica como má prática o não uso de um padrão de nomenclatura a ser usado nos recursos da aplicação. De modo similar, respostas indicam como boas práticas o uso de um padrão de nomenclatura a ser usados nos recursos. Exemplos de frases que indicaram más práticas são: P8 sobre Style Resources diz *"[...] o nome das strings sem um contexto"*, P37 também sobre Style Resources diz *"Nada além de ter uma boa convenção de nomes"*, ainda P37, porém sobre Layout Resources diz *"Mantenha uma convenção de nomes da sua escolha [...]"*. Exemplos de frases que indicaram boas prática são: P27 diz sobre String Resources *"Iniciar o nome de uma string com o nome da tela onde vai ser usada"*, P43 sobre Layout Resources diz *"Ter uma boa convenção de nomeação"*, P11 diz sobre Style Resources *"[...] colocar um bom nome [...]"*. Os elementos que entraram nessa categoria foram: Activities, Layout Resources, String Resources, Style Resources e Drawable Resources.

Dentre as respostas, algumas indicaram padrões de preferência. P11 indica usar prefixos nos Layout Resources: `activity_`, `fragment_`, `ui_` (para UI customizadas). P12 sugeriu usar sufixos em Activities: `_Activity`. Os padrões indicados para String Resources foram: P27 indicou *"Iniciar o nome da string com o nome da tela onde vai ser usada"*, P6 sugeriu a convenção `[screen]_[type]_[text]` e citou como exemplo `welcome_message_title`. P34 indicou que deve-se usar como prefixo o recurso usando a string, por exemplo `dialog.STRING_NAME` ou `hint.STRING_NAME`. De modo similar porém sem sugerir um exemplo, P4 sugeriu basear o nome da string no nome do recurso que a esta usando. Não foram sugeridos nenhum padrão para Styles Resources e Drawable Resources.

3.1.3 Recurso Mágico (RM). Indica como má prática o uso direto de valores como, por exemplo, strings, números e cores, sem a criação um recurso. De modo similar, respostas indicam como boas práticas o uso de um padrão de nomenclatura a ser usados nos recursos. O nome dessa categoria foi inspirado no cheiro de código *Magic Number* [38] que trata sobre números usados diretamente no código. Exemplos de frases que indicaram más práticas são: P23 diz *"Strings diretamente no código"*, P31 e P35 falam respectivamente sobre não extrair as strings e sobre não extrair os valores dos arquivos de layout. Exemplos de frases que indicaram boas prática são: P7 diz *"Sempre pegar valores de string ou dp de seus respectivos resources para facilitar"*, P36 diz para *"sempre adicionar as strings em resources para traduzir em diversos idiomas [...]"*. Os elementos que entraram nessa categoria foram: Layout Resources, String Resources e Style Resources.

3.1.4 Layout Profundamente Aninhado (LPA). Indica como má prática o uso de profundos aninhamentos na construção de layouts. De modo similar, respostas indicam como boas práticas evitar ao máximo o aninhamento de *views*. Exemplos de frases que indicaram más práticas são: P26 diz *"Hierarquia de views longas"*, P4 aborda a mesma ideia ao dizer *"Estruturas profundamente aninhadas"*, P39 diz *"Hierarquias desnecessárias"* e P45 diz *"Criar muitos ViewGroups dentro de ViewGroups"*. Exemplos de frases que indicaram boas prática são: P4 diz *"tento usar o mínimo de layout aninhado"*, P19 diz *"Utilizar o mínimo de camadas possível"*, P8 diz *"[...] não fazer uma hierarquia profunda de ViewGroups [...]"*. Apenas o elemento Layout Resources recebeu esta categoria. O site oficial do Android conta com informações e ferramentas automatizadas para lidar com esse sintoma [46].

[1] Todo texto em inglês foi traduzido livremente ao longo do artigo.

Recorrência/Prática	#T	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	#R
ALTA RECORRÊNCIA (4)																				
Lógica Em Classes de UI	60	14	15	8	8	6	8	–	1	–	–	–	–	–	–	–	–	–	–	7
Nome de Recurso Despadronizado	24	1	–	–	–	–	–	–	–	3	2	3	2	8	2	3	–	–	–	8
Recurso Mágico	23	–	–	–	–	–	–	–	–	4	2	1	1	9	6	–	–	–	–	6
Layout Profundamente Aninhado	21	–	–	–	–	1	–	–	–	9	9	–	–	–	–	–	–	1	1	5
MÉDIA RECORRÊNCIA (15)																				
Classes de UI Acopladas	18	–	2	4	6	–	3	1	2	–	–	–	–	–	–	–	–	–	–	6
Comportamento Suspeito	18	2	2	–	–	1	2	7	3	–	–	–	–	–	–	–	–	1	–	4
Mau Uso do Ciclo de Vida	15	4	3	3	5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5
Layout Longo ou Repetido	15	–	–	–	–	–	–	–	–	12	2	–	–	–	–	–	–	1	–	3
Não Uso de Padrão View Holder	13	–	–	–	–	11	2	–	–	–	–	–	–	–	–	–	–	–	–	2
Classe Deus/Longa*	13	2	4	2	2	–	1	–	1	–	–	–	1	–	–	–	–	–	–	6
Não Uso de Arquiteturas Conhecidas	13	4	–	2	–	–	–	–	–	–	–	–	–	–	–	–	–	6	1	4
Tamanho Único de Imagem	12	–	–	–	–	–	–	–	–	1	1	–	–	–	–	4	6	–	–	7
Uso Excessivo de Fragment	11	–	–	8	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	4
Não Uso de Imagens Vetoriais	10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	10	–	–	–	1
Não Uso de Fragment	9	3	2	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	2
Classes de UI Fazendo IO	9	1	4	1	2	–	1	–	–	–	–	–	–	–	–	–	–	–	–	4
Longo Recurso de Estilo	8	–	–	–	–	–	–	–	–	–	–	5	3	–	–	–	–	–	–	2
Recurso de String Bagunçado	8	–	–	–	–	–	–	–	–	–	–	–	–	4	4	–	–	–	–	2
Atributos de Estilo Repetidos	8	–	–	–	–	–	–	–	–	1	2	2	2	–	–	–	–	1	–	5
BAIXA RECORRÊNCIA (6)																				
Activity Inexistente	7	2	4	–	–	–	–	–	1	–	–	–	–	–	–	–	–	–	–	3
Imagem Dispensável	6	–	–	–	–	–	–	–	–	1	–	–	–	–	–	3	2	–	–	3
Excessivo Reúso de String	6	–	–	–	–	–	–	–	–	–	–	–	–	2	4	–	–	–	–	4
Adapter Complexo	6	–	–	–	–	3	2	–	–	–	1	–	–	–	–	–	–	–	–	2
Herança**	5	2	–	2	–	1	–	–	–	–	–	–	–	–	–	–	–	–	–	3
Listener Escondido	5	–	–	–	–	–	–	2	3	–	–	–	–	–	–	–	–	–	–	3
#Totais		35	36	34	26	23	19	10	11	31	19	11	9	23	16	20	8	10	2	
#Categorias		10	8	9	6	6	7	3	6	7	7	4	5	4	4	4	2	5	2	

* Classe Deus [32] e Classe Longa [7] são cheiros de código tradicionais previamente definidos na literatura.

** Herança é um conceito da Programação Orientada a Objetos [52].

Coluna #**T**: recorrência geral da prática.

Coluna #**R**: total de respostas distintas por prática.

Linha #**Totais**: total de respostas obtidas em cada questão.

Linha #**Categorias**: total de categorias distinstas de cada questão.

Tabela 1: Lista de práticas de alta, média e baixa recorrência.

3.2 Más Práticas de Média Recorrência

3.2.1 Classes de UI Acopladas (CUIA). Indica como má prática o acoplamento entre Activities, Fragments, Adapters e Listeners, ou seja, a existência de referências diretas entre elas. De modo similar, respostas indicam como boas práticas que estas classes não se conheçam diretamente. Com base nas respostas, identificamos 3 situações onde essa má prática é percebida.

O primeira situação é quando o Fragment está acoplado à Activities, outros Fragments ou componentes. Sobre o acoplamento de Fragments com Activities, P19 diz *"Acoplar o fragment a activity ao invés de utilizar interfaces é uma prática ruim"*. P10, P31 e P45 indicam como má prática *"acoplar o* Fragment *com a* Activity*"*. Sobre o acoplamento de Fragments com outros Fragments, P37 diz que *"Fragments nunca devem tentar falar uns com os outros diretamente"* e P45 diz *"[é uma má prática] integragir com outro Fragment diretamente"*. Sobre o Fragments

serem acoplados a outros componentes, P6 diz *"Seja um componente de UI reutilizável. Então evite dependência de outros componentes da aplicação"*. Como boa prática, para a comunicação entre essas classes, são indicados: o uso de *interfaces*, o método onAttach existente em Fragments (este método é disparado pelo Android ao associar um Fragment a uma Activity) ou a biblioteca *EventBus* [3]. P36 diz *"Criar uma interface para a comunicação entre Activity e Fragment, ou utilizar o EventBus."* e P44 diz *"Use e abuse do método onAttach para se comunicar com Activity"*.

A segunda situação é quando o Listener está acoplado a Activities. P40 diz que é uma má prática *"[o* Listener*] conter uma referência forte à Activities"*, P4 exprime a mesma ideia com uma frase um pouco diferente.

A terceira situação é quando o Adapter está acoplado à Activities ou Fragments. P10 indicou como má prática em *Adapters* o *"alto acoplamento com a Activity"* e P45 exprime a mesma ideia ao dizer *"Acessar Activities ou Fragments diretamente"*.

3.2.2 Comportamento Suspeito (CS). Indicam boas e más práticas ao implementar comportamento para responder a eventos do usuário, como por exemplo, um toque na tela. Um evento do usuário é representado através de listeners, que são interfaces Java onde cada uma representa um tipo de evento, por exemplo, a interface `OnClickListener` representa o evento de clique. Através das respostas dos participantes identificamos como boas práticas o uso de classes concretas e ferramentas de injeção de eventos e como más práticas o uso de classes anônimas e classes internas.

Classes anônimas são consideradas como má prática por todos os participantes que comentaram sobre ela, sendo que a maioria sugeriu o uso de classes concretas. Por exemplo, P9 diz *"Usar muitos anônimos pode ser complicado. Às vezes nomear coisas torna mais fácil para depuração"*, P4 diz *"Mantenha-os [listeners] em classes separadas (esqueça sobre classes anônimas)"*, P32 diz *"Prefiro declarar os listeners com "implements"e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto"* e P8 diz *"Muitas implementações de listener com classes anônimas"*.

O uso de classes internas também foi considerado como má prática. Exemplos de frases que indicam más práticas são: P42 diz *"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."*.

A implementação do Listener através do polimorfismo, por exemplo uma Activity se torna o Listener através do uso de `implements` do Java, foi considerado uma má prática por alguns participantes, que sugeriram o uso de classes concretas ou de ferramentas de injeção de eventos e dependência como *Butter Knife* [33] e *Dagger2*. Por exemplo: P44 diz *"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."*. P6 diz *"Tome cuidade se a activity/fragment é um listener uma vez que eles são destruídos quando as configurações mudam. Isso causa vazamentos de memória."*, P10 diz *"Use carregamento automático de view como ButterKnife e injeção de dependência como Dagger2"*. Para outros, esta forma de implementação é uma boa prática, por exemplo P32 diz *"Prefiro declarar os listeners com "implements"e sobrescrever os métodos (onClick, por exemplo) do que fazer um set listener no próprio objeto."*. Os elementos que entraram nesta categoria foram: Activity, Fragment ou Adapter.

3.2.3 Mau Uso do Ciclo de Vida (MUCV). Indica como má prática o uso incorreto do ciclo de vida Activities e Fragments. De modo similar, respostas indicam como boas práticas respeitar o ciclo de vida desses elementos e não confundir o ciclo de vida de ambos. Exemplos de frases que indicaram más práticas são: P23 diz *"código que depende de estado e não se adapta ao ciclo de vida das Activities."*, P28 diz *"Erros ao interpretar o ciclo de vida"*, P8 diz *"considerar o ciclo de vida de fragments como os de activities"*. Exemplos de frases que indicaram boas prática são: P43 diz *"Conhecer seu [da activity] ciclo de vida"*, P28, P31 e P15 falam *"tomar cuidado e respeitar o ciclo de vida de* fragments *e* activities*"*. Os elementos que entraram nessa categoria foram: Activities e Fragments.

3.2.4 Layout Longo ou Repetido (LLR). Indica como má prática código de *layout* repetido ou muito grandes. De modo similar, respostas indicam como boas práticas extrair layout repetidos para reutilizá-los através da tag include ou extrair apenas com o objetivo de manter arquivos pequenos. Exemplos de frases que indicaram más práticas são: P41 diz *"copiar e colar layouts parecidos sem usar includes"*, P23 diz *"[...] colocar muitos recursos no mesmo arquivo de layout."*, P8 diz *"considerar o ciclo de vida de fragments como os de activities"*. Exemplos de frases que indicaram boas prática são: P32 diz *"Sempre quando posso, estou utilizando includes para algum pedaço de layout semelhante"*, P36diz *"Criar layouts que possam ser reutilizados em diversas partes"* e P42 diz *"Separe um grande layout usando include ou merge"*.

3.2.5 Longos Recursos de Layout (LRL). Indica como má prática arquivos Layout Resources grandes e complexos. Respostas indicam como boas práticas a quebra desses arquivos em vários outros e o uso da *tag* de layout `include` para uni-los. Exemplos de frases que indicaram más práticas são: P41 diz *"Copiar e colar trechos similares de tela, sem usar includes"*, P23 diz *"[...] utilizar muitos recursos no mesmo arquivo de layout"*. Exemplos de frases que indicaram boas prática são: P34 diz *"eu apenas tento reusá-los através do uso de includes"*, P40 diz *"modularize-os"*, P9 diz *use includes para simplificar multiplas configurações [de tela]*.

3.2.6 Não Uso de Padrão View Holder (NUVH). Indica como má prática o não uso do padrão *View Holder* [45] para melhorar o desempenho de listagens. De modo similar, respostas indicam como boas práticas evitar o uso do padrão *View Holder*. Exemplo de frase que indicou má prática é P8 ao dizer *"[...] evitar o padrão ViewHolder"*. Exemplos de frases que indicaram boas prática são: P36 diz *"Reutilizar a view*

utilizando ViewHolder.", de modo similar P39 diz *"Usar o padrão ViewHolder"*. P45 sugere o uso do RecyclerView, um elemento Android para a construção de listas que já implementa o padrão *ViewHolder* [45]. Apenas o elemento Adapter entrou nessa categoria.

3.2.7 Não Uso de Arquiteturas Conhecidas (NUAC). Indica como má prática o não uso de algum padrão para desacoplar código de apresentação com código de lógica. De modo similar, respostas indicam como boas práticas o uso de algum padrão conhecido e sugerem MVP, MVC, MVVM e Clen Architecture. Exemplos de frases que indicaram más práticas são: P45 diz *"Sobre não usar um design pattern"*. Exemplos de frases que indicaram boas prática são: P28 diz *"Usar algum modelo de arquitetura para garantir apresentação desacoplada do framework (MVP, MVVM, Clean Architecture, etc)"*, P45 diz *"Sobre MVP. Eu acho que é o melhor padrão de projeto para usar com Android"*. Os elementos que entraram nessa categoria foram: Activities e textscFragments.

3.2.8 Tamanho Único de Imagem (TUI). Indica como má prática ter apenas uma imagem para atender a todas as resoluções. De modo similar, respostas indicam como boas práticas ter a mesma imagem em diversos tamanhos para atender a resoluções diferentes. Exemplos de frases que indicaram más práticas são: P31 diz *"ter apenas uma imagem para multiplas densidades"*, P4 diz *"Baixar uma imagem muito grande quando não é necessário. Há melhores formas de usar memória"*, P44 diz *"Não criar [versões da] imagem para todos as resoluções"*. Exemplos de frases que indicaram boas prática são: P34 diz *"Nada especial, apenas mantê-las em seus respectivos diretórios e ter variados tamanhos delas"*, P36 diz *"Criar as pastas para diversas resoluções e colocar as imagens corretas"*. O único elemento que entrou nessa categoria é o que representa imagens, Drawable Resource.

3.2.9 Uso Excessivo de Fragment (UEF). Indica como má prática o uso de Fragments quando se pode usar Activities e o uso excessivo de Fragments. De modo similar, respostas indicam como boas práticas evitar o uso de Fragments sempre que possível. Exemplos de frases que indicaram más práticas são: P2 diz *"Usar muitos Fragments é uma má prática"*. Exemplos de frases que indicaram boas prática são: P16 diz *"Eu tento evitá-los"*. Os elementos que entraram nessa categoria foram: Fragments.

3.2.10 Não Uso de Imagens Vetoriais (NUIV). I10 participantes) que indicam como boa prática o uso de Drawable Resources vetoriais. é recomendado sempre que possível usar imagens vetorias sobre outros tipos. Exemplos de frases que indicaram boas prática são: P28 diz *"Utilizar o máximo de Vector Drawables que for possível"*, P40 diz *"evite muitas imagens, use imagens vetoriais sempre que possível"*. O único elemento que entrou nessa categoria é o que representa imagens, Drawable Resource.

3.2.11 Não Uso de Fragment (NUF). Indica como má prática o não uso de Fragments. De modo similar, respostas indicam como boas práticas o uso de Fragments sempre que possível. Exemplos de frases que indicaram más práticas são: P22 diz *"Não usar Fragments"*. Exemplos de frases que indicaram boas prática são: P19 diz *"[...] Utilizar fragments sempre que possível."*, P45 diz *"Use Fragments para cada tela. Uma Activity para cada app/apk."*. Os elementos que entraram nessa categoria foram: Fragments e Activities.

3.2.12 Classes de UI Fazendo IO (CUIFIO). Indica como má prática realizar operações de IO, como consulta a banco de dados ou acesso a internet, a partir das classes Activities, Fragments e Adapters. Respostas indicam como boas práticas que esses elementos lidem apenas com a interface com o usuário e sugerem para isso, o padrão MVP [4, 5]. Exemplos de frases que indicaram más práticas são: P26 sobre Activities e Fragments diz *"fazer requests e consultas a banco de dados"* e sobre Adapter diz *"Fazer operações longas e requests de internet"*. P37 sobre Activities diz *"Elas nunca devem fazer acesso a dados [...]"*. Exemplos de frases que indicaram boas prática são: P26 diz *"Fazer activities e fragments apenas lidar com ações da view, faça isso usando o [padrão] MVP"*. Os elementos que entraram nessa categoria foram: Activities, Fragments e Adapters.

3.2.13 Longo Recurso de Estilo (LRE). Indica como má prática o uso de apenas um arquivo para todos os Styles Resources. De modo similar, respostas indicam como boas práticas separar os estilos em mais de um arquio. Exemplos de frases que indicaram más práticas são: P28 diz *"Deixar tudo no mesmo arquivo styles.xml"*, P8 diz *"Arquivos de estilos grandes"*. Exemplos de frases que indicaram boas prática são: P28 diz *"Se possível, separar mais além do arquivo styles.xml padrão, já que é possível declarar múltiplos arquivos XML de estilo para a mesma configuração"*. P40 diz *"Divida-os. Temas e estilos é uma escolha racional"*. O único elemento que entrou nessa categoria foi o Style Resource.

3.2.14 Recurso de String Bagunçado (RSB). Indica como má prática arquivos String Resources desorganizados ou o uso de apenas um arquivo para todos os String Resources. De modo similar, respostas indicam como boas práticas separar as *strings* em mais de um arquivo. Exemplos de frases que indicaram más práticas são: P28 diz *"Usar o mesmo arquivo strings.xml para tudo"*, P42 diz *"Não orgaizar as strings quando o strings.xml começa a ficar grande"*. Exemplos de frases que indicaram boas prática são: P28 diz *"Separar strings por tela em arquivos XML separados. Extremamente útil para identificar quais strings pertencentes a quais telas em projetos grandes"*. P32 diz *"Sempre busco separar em blocos, cada bloco representa uma activity e nunca aproveito uma String pra outra tela"*. O único elemento que entrou nessa categoria foi o String Resource.

3.2.15 Atributos de Estilo Repetidos (AER). Indica como má prática a repetição de atributos de estilo nos Layout Resource. De modo similar, respostas indicam como boas práticas sempre que identificar atributos repetidos, extraí-los para um estilo. Exemplos de frases que indicaram más práticas são: P32 diz *"Utilizar muitas propriedades em*

um único componente. Se tiver que usar muitas, prefiro colocar no arquivo de styles.". Exemplos de frases que indicaram boas prática são: P34 diz *"Sempre que eu noto que tenho mais de um recurso usando o mesmo estilo, eu tento movê-lo para o meu style resource."*. Os elementos nessa categoria foram: Layout Resources e Style Resources.

3.3 Más Práticas de Baixa Recorrência

3.3.1 Activity Inexistente (AI). Indica como má prática classes manterem referência a uma Activity, pois como ela possui ciclo de vida, quando a classe tentar acessá-la, a Activity pode não existir mais, resultando em possíveis erros na aplicação. De modo similar, respostas indicam como boas práticas, elementos com ciclo de vida independente, não manter referência à Activities. Exemplos de frases que indicaram más práticas são: P28 diz *"Fazer Activities serem callbacks de processos assíncronos gerando memory leaks. Erros ao interpretar o ciclo de vida"*, P31 diz *"[...] ter referência estática para Activities, resultando em vazamento de memória"*. Exemplos de frases que indicaram boas prática são: P31 diz *"Não manter referências estáticas para Activities (ou classes anônimas criadas dentro delas)"*, P4 diz *"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."*. Os elementos que entraram nessa categoria foram: Activities e Listeners.

3.3.2 Imagem Dispensável (ID). Indica como má prática o uso de imagens quando poderia ser usado um Drawable Resource em XML. De modo similar, respostas indicam como boas práticas o uso de Drawable Resources, criado com XML, sempre que possível. Exemplos de frases que indicaram más práticas são: P23 diz *"Uso de modotos não otimizados, uso de drawables onde recursos padrão do Android seriam preferíveis"*, P37 diz *usar jpg ou png para formas simples é ruim, apenas as desenhe [através de* Drawable Resources*]*. Exemplo de frase que indicou boa prática é P36 que diz *"[...] Quando possível, criar resources através de xml."*. Apenas o elemento Drawable Resoirce entrou nessa categoria.

3.3.3 Excessivo Reúso de String (ERS). Indica como má prática reutilizar o mesmo String Resource em muitos lugares no aplicativo, apenas porque o texto coincice, pois caso seja necessário alterar em um lugar, todos os outros serão afetados. De modo similar, respostas indicam como boas práticas considerar a semântica ou contexto ao nomear um String Resource, para mesmo que o valor seja o mesmo, os recursos sejam diferentes. Exemplos de frases que indicaram más práticas são: P32 diz *"Utilizar uma String pra mais de uma activity, pois se em algum momento, surja a necessidade de trocar em uma, vai afetar outra."*, P6 diz *"Reutilizar a string em várias telas"* e P40 diz *"Reutilizar a string apenas porque o texto coincide, tenha cuidado com a semântica"*. Exemplos de frases que indicaram boas prática são: P32 diz *"Sempre busco separar em blocos, cada bloco representa uma activity e nunca aproveito uma String pra outra tela."* e P9 diz *"Não tenha medo de repetir strings [...]"*. Apenas o elemento String Resource entrou nessa categoria.

3.3.4 Adapter Complexo (AC). Indica como má prática Adapters que contém muitos condicionais. Respostas indicam como boas práticas que um Adapter deve adaptar apenas um tipo de classe e trabalhar com um ou mais views, de modo a evitar lógicas para esconder ou mostrar view expecíficas. Exemplos de frases que indicaram más práticas são: P23 diz *"Reutilizar um mesmo adapter para várias situações diferentes, com "ifs"ou "switches". Código de lógica importante ou cálculos em Adapters."*. Exemplos de frases que indicaram boas prática são: P2 diz *"Um Adapter deve adaptar um único tipo de item ou delegar a Adapters especializados"*. O único elemento que entrou nessa categoria foi o Adapter.

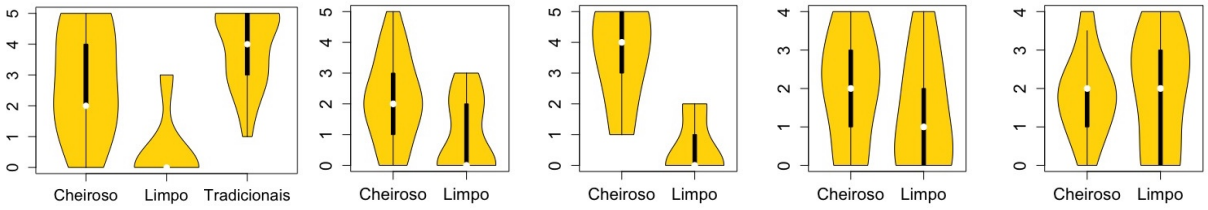
3.3.5 Listener Escondido (LE). Indica como má prática o uso de métodos de eventos do usuário no XML de layout, como por exemplo o método onClick. Respostas indicam como boas práticas o uso da biblioteca *ButterKnife* [33]. Exemplos de frases que indicaram más práticas são: P34 diz *"Nunca crie um listener dentro do XML. Isso esconde o listener de outros desenvolvedores e pode causar problemas até que ele seja encontrado"*, P39 e P41 expressam a mesma opinião, P41 ainda complementa dizendo que *"XML de layout deve lidar apenas com a view e não com ações"*. Exemplos de frases que indicaram boas prática são: P39 diz apenas *"Uso ButterKnife"*, P34 também expressa sua preferência por essa biblioteca. Apenas Listeners entraram nessa categoria.

3.4 Percepção dos Desenvolvedores

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

3.4.1 Más práticas que afetam classes Java. Na Figura 1a apresentamos três gráficos violinos, respectivamente: percepção dos desenvolvedores sobre códigos afetados pela má prática LCUI, a percepção sobre códigos limpos e por último, a percepção sobre códigos afetados por maus cheiros tradicionais como por exemplo Classe Longa.

A mediana de classes limpas tem severidade igual a 0 (Q3=0). Isso indica que, como esperado, desenvolvedores não percebem essas classes como problemáticas. Em comparação, as classes afetadas por LCUI tem mediana igual a 2 (Q3=4) logo, são percebidas como classes problemáticas. A diferença entre classes afetadas por LCUI e classes limpas é estatisticamente significante (p-value < 0.004) com alto tamanho de efeito (d = 0,72). Com relação aos maus cheiros tradicionais, a mediana de severidade é igual a 4 (Q3=5). Isso significa que classes afetadas por esses maus cheiros são percebidas pelos



(a) LCUI **(b) Integrados** **(c) LPA** **(d) RM** **(e) NRD**

Figura 1: Gráficos violino individuais das más práticas que afetam recursos (LPA, RM e NRD).

desennvolvedores como muito problemáticas, ainda mais que classes afetadas pela má prática em questão. Ainda que essa diferença de percepção esteja clara no gráfico violino, esta diferença não é estatisticamente significante (p-value = 0.077). Acreditamos que isso ocorra devido ao número limitado de dados (20 participantes).

3.4.2 Más práticas que afetam recursos. A Figura 1 reúne 4 diferentes pares de gráficos violinos. O primeiro compara recusos afetados por quaisquer das 3 más práticas, LPA, RM e NRD, com recursos limpos. O segundo, terceiro e quarto, trata de cada má prática individualmente, ou seja, recursos afetadas por aquela má prática em comparação com recursos limpos.

A Figura 1b mostra a percepção dos desenvolvedores com relação a recursos afetados pelas más práticas LPA, RND e RM, com mediana de severidade igual a 2 (Q3=3), em comparação com recursos limpos (mediana igual a 0). Isso indica que, como esperado, desenvolvedores percebem recursos afetados pelas más práticas como problemáticos. Essa diferença também é estatisticamente significante (p-value $<$ 0.008) com médio tamanho de efeito (d = 0.43).

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

A má prática NRD foi a menos percebida por desenvolvedores, com medianas iguais para códigos afetados por ela e códigos limpos (mediana = 2). Entretando, os desenvolvedores que indicaram códigos afetados por ela como problemáticos, indicaram como o problema descrições muito próximas a definição dada para essa má prática. Por exemplo, S2P15 disse "*Os nomes das strings são formados por um prefixo e um número, o que prejudica a legibilidade, é impossível saber o que este número indica*" (pontuou severiadade como 3), S2P16 disse "*Atributos não seguem convenção de nomes. Nomes não são descritivos*" (pontuou severiadade como 2), S2P11 disse "*Não segue uma boa prática de nomenclatura de recursos.*" (pontuou severiadade como 2) e S2P19 disse "*Os nomes das strings não estão seguindo um padrão (algumas em camelCase, outras lowercase, outras snakecase) [...]*" (pontuou severiadade como 2). Algumas possíveis ideias que justificariam esse resultado são discutidas na Seção **??**.

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

4 AMEAÇAS À VALIDADE

Uma limitação deste artigo é que os dados foram coletados apenas a partir de questionários online e o processo de codificação foi realizado apenas por um dos autores. Alterativas a esses cenários seria 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 possúveis 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. 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 [13] ao investigar cheiros de código no framework Spring MVC. Entretando, nos deparamos com situações diferentes, das quais, após a execução nos questionamos se aquele método seria o mais adequado para o contexto deste 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 deifnição da má prática RM. Desta forma, levantamos que de todos os recursos avaliados, 74% receberam severidade igual ou inferior a 3, contra 30% com os mesmo níveis de severidade em código Java. Desta forma, uma alternativa seja repensar a forma de avaliar a percepção dos desenvolvedores sobre más práticas que afetem recursos do aplicativo.

5 TRABALHOS RELACIONADOS

Muitas pesquisas têm sido realizadas sobre a plataforma Android, muitas delas focam em vulnerabilidades [17, 19, 20, 34, 53, 56, 57], autenticação [21, 54, 55] e testes [11, 31]. Diferentemente dessas pesquisas, nossa pesquisa tem foco na percepção dos desenvolvedores sobre boas e más práicas de desenvolvimento na plataforma Android.

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

Verloop [50] conduziu um estudo no qual avaliou por meio de 4 ferramentas de detecção automatizada de cheiros de código (JDeodorant, Checkstyle, PMD e UCDetector) a presença de 5 cheiros de código (Long Method, Large Class, Long Parameter List, Feature Envy e Dead Code) em 4 projetos Android. Nossa pesquisa se relaciona com a de Verloop no sentido de que também estamos buscando por cheiros de código, entretanto, em vez de de buscarmos por cheiros de código já definidos, realizamos uma abordagem inversa na qual, primeiro buscamos entender a percepção de desenvolvedores sobre boas e más práticas em Android, e a partir dessa percepção, relacionamos com algum cheiro de código pré-existente ou derivamos algum novo.

Gottschalk et al [28] conduziram um estudo sobre formas de detectar e refatorar cheiros de código relacionados ao uso eficiênte de energia. Os autores compilaram um catálogo com 8 cheiros de código e trabalharam sob um trecho de código Android para exemplificar um deles, o "binding resource too early", quando algum recurso é alocado muito antes de precisar ser utilizado. Essa pesquisa é relacionada à nossa por ambas considerarem a tecnologia Android e se diferenciam pois focamos na busca por cheiros de código relacionados a qualidade de código, no sentido de legibilidade e manutenablidade.

Aplicativos Android são escritos na linguagem de programação Java [44]. Então a primeira questão é: por que buscar por *smells* Android sendo que já existem tantos *smells* Java? Pesquisas têm demonstrado que tecnologias diferentes podem apresentar *code smells* específicos, como por exemplo Aniche et al. identificaram 6 *code smells* específicos ao framework Spring MVC, um framework Java para desenvolvimento web. Outras pesquisas concluem que projetos Android possuem características diferentes de projetos java [30, 36, 42], por exemplo, o *front-end* é representado por arquivos XML e o ponto de entrada da aplicação é dado por *event-handler* [1] como o método onCreate. Encontramos também diversas pesquisas sobre *code smells* sobre tecnologias usadas no desenvolvimento de *front-end* web como CSS [26] e JavaScript [22]. Essas pesquisas nos inspiraram a buscar entender se existem *code smells* no *front-end* Android.

6 CONCLUSÃO

Neste artigo investigamos a existência de boas e más práticas no *front-end* de projetos Android. Fizemos isso através de um estudo exploratório qualitativo com 45 desenvolvedores, onde mapeamos 23 más práticas Android. Após, validamos a percepção de desenvolvedores Android sobre as quatro más práticas mais recorrêntes. Fizemos isso através de um experimento online respondido com 20 desenvolvedores Android. Respondemos a **QP1** com um catálogo com 23 más práticas no *front-end* Android. Respondemos a **QP2** através da validação com sucesso da percepção de desenvolvedores sobre 2 das más práticas de alta recorrência.

REFERÊNCIAS

[1] Activities. https://developer.android.com/guide/components/activities.html. (????). Last accessed at 29/08/2016.

[2] Android Studio. https://developer.android.com/studio/index.html. (????). Last accessed at 30/08/2016.

[3] Green Robot - Event Bus). http://greenrobot.org/eventbus. (????). Last accessed at 11/04/2017.

[4] GUI Architectures). https://www.martinfowler.com/eaaDev/uiArchs.html. (????). Last accessed at 11/04/2017.

[5] Wikipédia - Model-View-Presenter. https://pt.wikipedia.org/wiki/Model-view-presenter. (????). Last accessed at 11/04/2017.

[6] Wikipédia - Model-View-ViewModel. https://en.wikipedia.org/wiki/Model\T1\textendashview\T1\textendashviewmodel. (????). Last accessed at 11/04/2017.

[7] 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] 2016. PMD (2016). https://pmd.github.io. (2016). Last accessed at 29/08/2016.

[9] 2016. Wikipedia Code Smell. https://en.wikipedia.org/wiki/Code_smell. (2016). Last accessed at 14/11/2016.

[10] Steve Adolph, Wendy Hall, and Philippe Kruchten. 2011. Using grounded theory to study the experience of software development. Empirical Software Engineering. (2011), 27.

[11] Domenico Amalfitano, Anna Fasolino, Porfirio Tramontana, Salvatore Carmine, and Atif Memon. 2012. Using GUI ripping for automated testing of Android applications. (2012).

[12] Maurício Aniche, Bavota G., Treude C., Van Deursen A., and Gerosa M. 2016. A Validated Set of Smells in Model-View-Controller Architectures. (2016).

[13] Maurício Aniche and Marco Gerosa. 2016. Architectural Roles in Code Metric Assessment and Code Smell Detection. (2016).

[14] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. 2016. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE, 41–50.

[15] Lionel C Briand, William M Thomas, and Christopher J Hetmanski. 1993. Modeling and managing risk early in software development. In *Software Engineering, 1993. Proceedings., 15th International Conference on*. IEEE, 55–65.

[16] Suelen Carvalho. Apêndice. (????).

[17] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. (2011).

[18] Juliet Corbin and Anselm Strauss. 2007. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded*

Theory (3 ed.). SAGE Publications Ltd.
[19] Enck, William, and Patrick Drew McDaniel Machigar Ongtang. 2009. Understanding Android Security. (2009).
[20] Enck, William, and Patrick McDaniel Machigar Ongtang. 2008. Mitigating Android software misuse before it happens. (2008).
[21] Zheran Fang and Yingjiu Li Weili Han. 2014. Permission Based Android Security: Issues and Countermeasures. (2014).
[22] A. Milani Fard and A. Mesbah. 2013. JSNOSE: Detecting javascript code smells. (2013).
[23] Martin Fowler. 1999. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.
[24] Martin Fowler. 2006. Code Smell. http://martinfowler.com/bliki/CodeSmell.html. (Feb. 2006).
[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston.
[26] Golnaz Gharachorlu. 2014. *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model*. Ph.D. Dissertation. The University of British Columbia.
[27] Barney G. Glaser and Anselm L. Strauss. 1999. *The Discovery of Grounded Theory: Strategies for Qualitative Research* (1 ed.). Aldine.
[28] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing Energy Code Smells with Reengineering Services. (????). Maus cheiros relacionados ao consumo de energia.
[29] Robert J Grissom and John J Kim. 2005. Effect Sizes for Research: Univariate and Multivariate Applications. Routledge, 272.
[30] Geoffrey Hecht. 2015. An Approach to Detect Android Antipatterns. (2015).
[31] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. (2011).
[32] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company. https://books.google.com.br/books?id=oHkhAQAAIAAJ
[33] Wharthon Jake. Butter Knife. http://jakewharton.github.io/butterknife/. (????). Last accessed at 11/04/2017.
[34] K Kavitha, P Salini, and V Ilamathy. 2016. Exploring the Malicious Android Applications and Reducing Risk using Static Analysis. (2016).
[35] Nigel King. 1994. In Qualitative methods in organizational research - A practical guide. (1994), 253.
[36] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding Code Smells in Android Applications. (????). DOI:https://doi.org/10.1145/2897073.2897094
[37] Robert Martin. 8thlight Blog - The Clean Architecture. https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html. (????). Last accessed at 11/04/2017.
[38] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
[39] Nachiappan Nagappan and Thomas Ball. 2005. Static Analysis Tools As Early Indicators of Pre-release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 580–586. DOI:https://doi.org/10.1145/1062455.1062558
[40] Luciana Carla Lins Prates. 2015. *Aplicando Síntese Temática em Engenharia de Software*. Ph.D. Dissertation. Universidade Federal da Bahia e Universidade Estadual de Feira de Santana.
[41] Rafael Prikladnicki. 2013. *MuNDDoS - Um Modelo de Referência Para Desenvolvimento Distribuído de Software*. Ph.D. Dissertation. Pontifíca Universidade Católica do Rio Grande do Sul - PUCRS.
[42] Jan Reimann and Martin Brylski. 2013. A Tool-Supported Quality Smell Catalogue For Android Developers. (2013).
[43] Johnny Saldaña. 2012. *The Coding Manual for Qualitative Researchers* (2 ed.). SAGE Publications Ltd.
[44] Android Developer Site. Andriod Fundamentals. https://developer.android.com/guide/components/fundamentals.html. (????). Last accessed at 04/09/2016.
[45] Android Developers Site. Android RecyclerView. https://developer.android.com/training/material/lists-cards.html. (????). Last accessed at 12/04/2017.
[46] Android Developer Site. Optimizing View Hierarchies). https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html). (????). Last accessed at 09/04/2017.
[47] Android Developer Site. 2016. Building Your Firt App. https://developer.android.com/training/basics/firstapp/creating-project.html. (2016). Last accessed at 31/03/2017.
[48] Developer Android Site. 2016. Resources Overview. https://developer.android.com/guide/topics/resources/overview.html. (2016). Last accessed at 08/09/2016.
[49] Nikolaos Tsantalis. 2010. *Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings*. Ph.D. Dissertation. University of Macedonia.
[50] Daniël Verloop. 2013. *Code Smells in the Mobile Applications Domain*. Ph.D. Dissertation. TU Delft, Delft University of Technology.
[51] Conover J W. 1999. *Practical Nonparametric Statistics* (3 ed.). Wiley. 584 pages.
[52] Wikipédia. Inheritance (object-oriented programming). https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming). (????). Last accessed at 08/04/2017.
[53] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. 2016. Eficient Fingerprinting-based Android Device Identication with Zero-permission Identiers. (2016).
[54] A. Yamashita and L. Moonen. 2012. Do code smells reflect important maintainability aspects?. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 306–315. DOI:https://doi.org/10.1109/ICSM.2012.6405287
[55] S. Yu. 2016. Big privacy: Challenges and opportunities of privacy study in the age of big data. (2016).
[56] Yuan Zhang, Min Yang, Zhemin Yang, Guofei GU, and Binyu Zang Peng Ning. 2004. Exploring Permission Induced Risk in Android–Applications for Malicious Detection. (2004).
[57] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. 2014. Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps. (2014).