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

Cheiros de código são aliados na busca pela qualidade de código durante o desenvolvimento de software pois possibilitam a implementação de ferramentas de detecção automática de trechos de códigos problemáticos ou mesmo a inspeção manual. Apesar de já existirem vários cheiros de código catalogados, pesquisas sugerem que tecnologias diferentes podem apresentar 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 possíveis cheiros de código específicos ao Android dos quais validamos a percepção de 15 desenvolvedores sobre eles. Ao final, discutimos os resultados encontrados, pontos de melhoria e trabalhos futuros.

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

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 [28, 49]. Existem diferentes técnicas que auxiliam os desenvolvedores a escreverem código com qualidade, incluindo *design patterns* [24] e cheiros de código [22]. A falta de qualidade pode resultar em defeitos de software que podem custar a empresas quantias significativas, especialmente quando conduzem a falhas de software [15, 38]. Evolução e manutenção de software também já se provaram como os maiores gastos com aplicações [48].

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 [22]. 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 [23]. 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.

Determinar o que é ou não um cheiros 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. Alguns estudos têm buscado por cheiros de código tradicionais em projetos Android. Por exemplo, Verloop [49] analisou se classes derivadas do SDK Android são mais ou menos propensas a cheiros de código tradicionais do que classes puramente Java. Linares et al. [34] usaram o método DECOR para realizar a detecção de 18 *anti-patterns* orientado a objetos em aplicativos móveis. Outros estudos identificaram cheiros de código específicos Android relacionados ao consumo inteligente de recursos do dispositivo, como bateria e memória, usabilidade, dentre outros [27, 41].

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, por exemplo, *"quais são as* ***boas e más*** *práticas ao lidar com* ACTIVITIES*,* FRAGMENTS*,* ADAPTERS *e* LISTENERS*?"* ou *"quais são as* boas e más *práticas no desenvolvimento da interface visual Android?"*.

SBES'17, Fortaleza, Ceará Brasil

DOI: 10.475/123_4

Apesar dos trabalhos citados, Umme et al. [35] recentemente levantaram que, das principais conferências de manutenção de software (ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MRS e ESEM), dentre 2008 a 2015, apenas 9,6% dos artigos consideraram em suas pesquisas, projetos Android. Nenhuma outra plataforma móvel foi considerada. Enquanto que cheiros de código em projetos Java já foram extensivamente estudados [22, 30, 37], ainda há muito a se pesquisar sobre cheiros de código em projetos Android.

Para limitar nosso objeto de estudo, optamos por focar em cheiros de código relacionados ao *front-end* Android pois encontramos pesquisas com abordagem similar, porém relacionadas ao *front-end* de tecnologias web, como CSS [25], Javascript [21] e o arcabouço Spring MVC [13].

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 [47], com o objetivo de limitar o tamanho do questionário e foco da pesquisa, selecionamos quatro: Layout, Styles, String e Drawable. Optamos por esses recursos pois os mesmos estão presentes no template padrão do Android Studio [46], IDE oficial para desenvolvimento de projetos da plataforma Android [2].

Os dados iniciais foram obtidos através de um questionário online com perguntas sobre boas e más práticas no desenvolvimento do *front-end* Android. As análise do questionário resultou num com a 23 más práticas no desenvolvimento do *front-end* Android. Validamoss a percepção dos desenvolvedores sobre essas más práticas através de outro questionário online. Com isso, pretendemos responder as 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?

Esperamos que este catálogo de más práticas possa contribuir com ideias iniciais para a definição de cheiros de código e heurísticas para a detecção sistematizada das mesmas em projetos Android, além de contribuir com sugestões de como mitigá-las.

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 mais recorrentes através de um questionário online com 11 desenvolvedores.

3) Apêndice online [1] com roteiros dos quetionários 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 os resultados. A Seção 4 discute pontos relevantes. A Seção 5 aborda as ameaças à validade do estudo. A Seção 6 discute os trabalhos relacionados e o estado da arte sobre Android e cheiros de código. E por fim, a Seção 7 conclui.

2 METODOLOGIA

O *objetivo* deste estudo é investigar a existência de boas e más práticas no desenvolvimento do *front-end* Android. Portanto, conduzimos um estudo qualitativo e exploratório no qual coletamos dados através de dois questionários online com desenvoveldores Android. O primeiro questionário (*S1*) teve foco em investigar boas e más práticas no desenvolvimento Android. O segundo questionário (*S2*) teve foco em avaliar a percepção dos desenvolvedores sobre as más práticas definidas a partir de *S1*.

Segundo Strauss e Corbin [17], 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 [39, 40].

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 [33, 39].

Para análise dos dados seguimos a abordagem de *Ground Theory* (GT), um método de pesquisa exploratória originada nas ciências sociais [17, 26], 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.

Esta seção está organizada de forma que, na seção 2.1 abordamos o processo de descoberta das boas e más práticas Android e na subseção 2.2 abordamos o processo de validação da percepção dos desenvolvedores sobre as boas e más práticas definidas. Ambas as subseções apresentam de forma detalhada a estrutura do questionário, os participantes e a análise realizada.

[1] Apêndice online: http://suelengc.com/android-code-smells-article

2.1 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. Os dados foram coletados através de um questionário online (*S1*) respondido por 45 desenvolvedores. Realizamos um processo de codificação aberta sobre os dados, resultando num conjunto com 23 categorias de boas e más práticas Android. Estas 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.

2.1.1 Questionário. O questionário continha 25 questões divididas em três seções. A primeira seção continha 6 perguntas demográficas, a segunda seção continha 16 perguntas sobre boas e más práticas relacionadas ao *front-end* Android e a terceira seção continha 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 foi 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. O questionário completo pode ser encontrado em nosso apêndice online.

A primeira seção continha 6 questões demográficas obrigatórias de multipla escolha. Abordavam sobre idade (18 ou menos, 19 a 24, 25 a 34 e assim por diante até 55 ou mais), estado de residência (foi dada uma lista com estados do Brasil, Estados Unidos e Europa), anos de experiência com desenvolvimento de software, (1 anos ou menos, 2 anos, 3 anos, e assim por diante até 10 ou mais), anos de experiência com desenvolvimento Android (mesma escala de anos da questão anterior), uma questão sobre linguagens que o participante se considerava proeficiente (Java, Python, Ruby, Android, dentre outras) e sobre o último grau de escolaridade (estudante de bacharelado, bacharel, mestrado e doutorado). As questões sobre idade, região, linguagens e grau de escolaridade continham a opção "outros" para o caso de nenhuma das opções atenderem, ao selecionar "outros" era possível escrever uma resposta.

A segunda seção continha 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, uma sobre boas e outra sobre más práticas percebidas pelos participantes. Por exemplo, para o elemento ACTIVITY foram feitas as seguintes perguntas:

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

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

A terceira seção continha 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.

Antes da divulgação, realizamos um piloto com 3 desenvolvedores Android e com o feedback deles fizemos alguns ajustes relacionados a obrigatoriedade das perguntas da segunda seção do questionário, onde todas tornaram-se opcionais. As respostas dos participantes piloto foram desconsideradas para efeitos de viés.

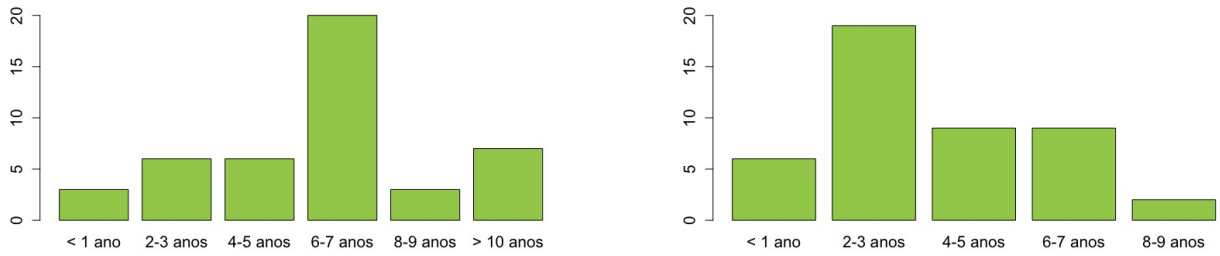
2.1.2 Participantes. O questionário foi divulgado em redes sociais como Facebook, Twitter e Linkedin, em grupos de discussão sobre Android como *Android Dev Brasil*, *Android Brasil Projetos* e o grupo do *Slack Android Dev Br*, maior grupo de desenvolvedores Android do Brasil com 2622 participantes 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. Recebemos um total de 45 respostas.

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 foi possível notar 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 anos 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 plarforma 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 1.

2.1.3 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,



(a) Desenvolvimento de Software **(b) Desenvolvimento Android**

Figura 1: Experiência dos desenvolvedores em *S1*.

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 continham 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 [17, 42]. 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 [17]. 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 forma 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 a 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 forma diferente.

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

2.2 Percepção dos Desenvolvedores

Para responder a **QP2**, buscamos entender a percepção dos desenvolvedores sobre as quatro boas e más práticas classificadas com alta recorrência. Estes dados foram coletados através de um questionário online (*S2*) respondido por 11 desenvolvedores Android. Nossas análises demonstram que de fato, códigos afetados pelas más práticas, são percebidos pelos desenvolvedores como códigos problemáticos.

2.2.1 Questionário. O questionário foi composto por duas seções principais. A primeira objetivou coletar informações básicas sobre os antecedentes dos participantes e, em particular, sobre sua experiência (dados apresentados na Figura 2). Na segunda seção, os participantes foram solicitados a examinar seis códigos-fonte Android e, para cada uma deles, responder às 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 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. Os 58 códigos foram aleatoriamente coletados de projetos Android de código aberto no GitHub.

Para reduzir o aprendizado e viés, 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.

2.2.2 Participantes.

2.2.3 Análise dos Dados. Para comparar as distribuições da gravidade indicada pelos participantes para os três grupos de classes, utilizamos o teste de Mann-Whitney não pareado [27] para analisar a significância estatística das diferenças entre a gravidade atribuída pelos participantes aos problemas que observam em MVC- Tradicional-smelly, e classes limpas. Os resultados são considerados estatisticamente significativos em alfa = 0,05. Também estimamos a magnitude das diferenças medidas usando o Delta de Cliff (ou d), uma medida de tamanho do efeito não paramétrico [49] para dados ordinais. Seguimos diretrizes bem estabelecidas para interpretar os valores do tamanho do efeito: insignificante para — d — menor 0,14, pequeno para 0,14 menor igual — d — menor 0,33, meio para 0,33 menor igual — d — menor 0,474, e grande para — d — maior igual 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). Na Figura 3 é possível observar a distribuição das práticas de acordo com sua recorrência. Nesta seção tratamos das práticas de alta, média e baixa recorrência, as práticas de baixíssima recorrência são brevemente discutidas na Seção 4.

A Tabela 1 apresenta o total de ocorrências das categorias de alta, média e baixa recorrência em cada questão sobre boas e más práticas (*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?*

Esta seção está organizada em 4 subseções. Nas três primeiras são definidas as práticas de alta, média e baixa recorrência,

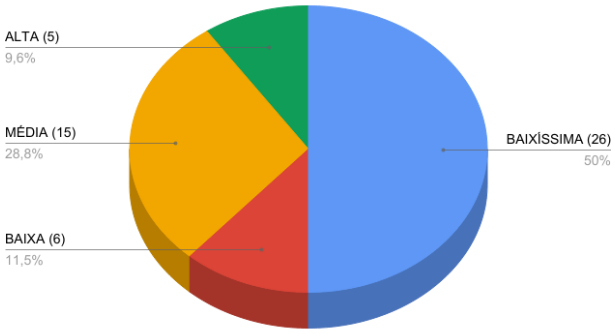


Figura 3: Distribuição das práticas vs. recorrência

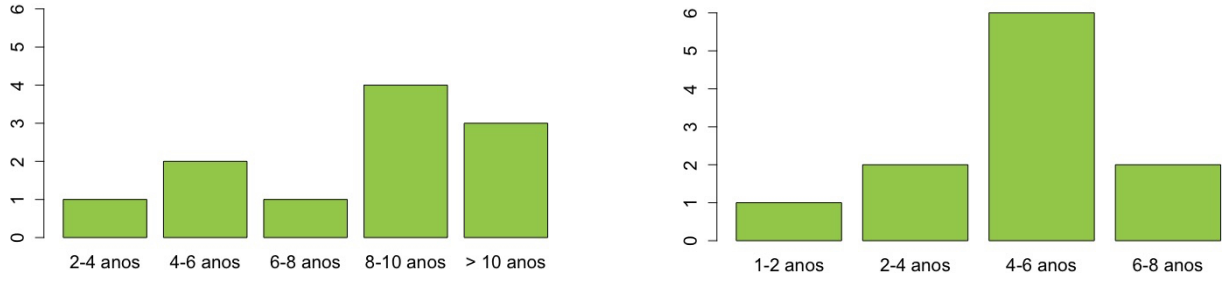
respectivamente. Na última subseção apresentamos os resultados obtidos no estudo sobre a percepção de desenvolvedores sobre as práticas definidas.

3.1 Práticas de Alta Recorrência

3.1.1 Lógica em Views. Indicam 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* [36]. Exemplos de frases que indicaram más práticas são: P16 sobre Activities diz *"Fazer lógica de negócio"* (tradução livre)[2], 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 Padrão de Nome de Recursos. Indicam como má prática o não uso de um padrão de nomenclatura a ser usado nos recursos da aplicação. De forma 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*

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



(a) Desenvolvimento de Software **(b) Desenvolvimento Android**

Figura 2: Experiência dos desenvolvedores em *S2*.

Recorrência/Prática	#T	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	#Q
ALTA RECORRÊNCIA (4)																				
Lógica em Views	60	14	15	8	8	6	8	–	1	–	–	–	–	–	–	–	–	–	–	7
Padrão de Nome de Recursos	24	1	–	–	–	–	–	–	–	3	2	3	2	8	2	3	–	–	–	8
Recursos Mágicos	23	–	–	–	–	–	–	–	–	4	2	1	1	9	6	–	–	–	–	6
Views Aninhados	21	–	–	–	–	1	–	–	–	9	9	–	–	–	–	–	–	1	1	5
MÉDIA RECORRÊNCIA (15)																				
Acoplamento Entre Views	18	–	2	4	6	–	3	1	2	–	–	–	–	–	–	–	–	–	–	6
Comportamento Suspeito	18	2	2	–	–	1	2	7	3	–	–	–	–	–	–	–	–	1	–	4
Ciclo de Vida	15	4	3	3	5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5
Use Include	15	–	–	–	–	–	–	–	–	12	2	–	–	–	–	–	–	1	–	3
Padrão View Holder	13	–	–	–	–	11	2	–	–	–	–	–	–	–	–	–	–	–	–	2
Classe Deus/Longa*	13	2	4	2	2	–	1	–	1	–	–	–	1	–	–	–	–	–	–	6
Use Arquiteturas Conhecidas	13	4	–	2	–	–	–	–	–	–	–	–	–	–	–	–	–	6	1	4
Tamanho de Imagens Importam	12	–	–	–	–	–	–	–	–	1	1	–	–	–	–	4	6	–	–	7
Fragment Apenas Se Necessário	11	–	–	8	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	4
Use Imagens Vetoriais	10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	10	–	–	–	1
Use Fragment	9	3	2	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	2
Operações de IO	9	1	4	1	2	–	1	–	–	–	–	–	–	–	–	–	–	–	–	4
Recurso de Estilo Deus	8	–	–	–	–	–	–	–	–	–	–	5	3	–	–	–	–	–	–	2
Recurso de Strings Bagunçado	8	–	–	–	–	–	–	–	–	–	–	–	–	4	4	–	–	–	–	2
Atributos de Estilos Repetidos	8	–	–	–	–	–	–	–	–	1	2	2	2	–	–	–	–	1	–	5
BAIXA RECORRÊNCIA (6)																				
Activity Inexistente	7	2	4	–	–	–	–	–	1	–	–	–	–	–	–	–	–	–	–	3
Evite Imagens	6	–	–	–	–	–	–	–	–	1	–	–	–	–	–	3	2	–	–	3
Reuso Excessivo de Strings	6	–	–	–	–	–	–	–	–	–	–	–	–	2	4	–	–	–	–	4
Adapter Flexível	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 [30] e Classe Longa [7] são cheiros de código tradicionais previamente definidos em literaturas.

** Herança é um conceito da Programação Orientada a Objetos [50].

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

Coluna **#Q**: total de questões 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.

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 forma 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 Recursos Mágicos. Indicam como má prática o uso direto de valores como, por exemplo, strings, números e cores, sem a criação um recurso. De forma 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* [37] 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 Views Aninhados. Indicam como má prática o uso de profundos aninhamentos na construção de layouts. De forma 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 [45].

3.2 Práticas de Média Recorrência

3.2.1 Acoplamento Entre Views. Indicam como má prática o acoplamento entre Activities, Fragments, Adapters e Listeners, ou seja, a existência de referências diretas entre elas. De forma 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 à 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"*.

Os elementos que entraram nessa categoria foram: Activities, Fragments, Listeners e Adapters.

3.2.2 Use Arquiteturas Conhecidas. Indicam como má prática o não uso de algum padrão para desacoplar código de apresentação com código de lógica. De forma 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.

3.2.3 Ciclo de Vida. Indicam como má prática o uso incorreto do ciclo de vida Activities e Fragments. De forma 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 Use Include. Indicam 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]*.

3.2.5 Padrão View Holder. Indicam como má prática o não uso do padrão *View Holder* [44] para melhorar o desempenho de listagens. De forma 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 forma 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* [44]. Apenas o elemento ADAPTER entrou nessa categoria.

3.2.6 Comportamento Suspeito. 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, onde 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."*.

O polimorfismo, fazendo com que o elemento Android implemente o LISTENER, foi considerado uma má prática por alguns participantes, esses sugeriram o uso de classes concretas ou de ferramentas de injeção de eventos e dependência como *Butter Knife* [31] 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.

3.2.7 Tamanho de Imagens Importam. Indicam como má prática ter apenas uma imagem para atender a todas as resoluções. De forma 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.

3.2.8 Use Imagens Vetoriais. 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.

3.2.9 Fragment Apenas se Necessário. Indicam como má prática o uso de FRAGMENTS quando se pode usar ACTIVITIES e o uso excessivo de FRAGMENTS. De forma 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 Use Fragment. Indicam como má prática o não uso de FRAGMENTS. De forma 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.

3.2.11 Operações de IO. Indicam 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.

3.2.12 Recurso de Estilo Deus. Indicam como má prática o uso de apenas um arquivo para todos os STYLES RESOURCES. De forma 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.

3.2.13 Recurso de Strings Bagunçado. Indicam como má prática arquivos STRING RESOURCES desorganizados ou o uso de apenas um arquivo para todos os STRING RESOURCES. De forma 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.

3.2.14 Atributos de Estilos Repetidos. Indicam como má prática a repetição de atributos de estilo nos LAYOUT RESOURCE. De forma 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 Práticas de Baixa Recorrência

3.3.1 Evite Imagens. Indicam como má prática o uso de imagens quando poderia ser usado um DRAWABLE RESOURCE em XML. De forma 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 formatos 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.

3.3.2 Adapter Flexível. Indicam 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 forma 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.

3.3.3 Activity Inexistente. Indicam 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 forma 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.

3.3.4 Reuso Excessivo de Strings. Indicam 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 forma 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.

3.3.5 Listener Escondido. Indicam 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* [31]. 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.

Na Figura 5.3a, apresentamos gráficos de violino da percepção dos desenvolvedores sobre os cheiros Android, tradicionais e códigos limpos. Além disso, relatamos a percepção dos desenvolvedores para cada uma das práticas em Android - Figura 5.3b. No eixo y, 0 (zero) indica classes não percebidas pelos desenvolvedores como problemáticas (ou seja, responder "não"à pergunta: este código apresenta algum problema de design e/ou implementação?), enquanto valores de 1 a 5 indicam o nível de severidade para o problema percebido pelo desenvolvedor.

Os códigos limpos têm uma mediana de gravidade igual a 1,5. Isso indica que, como esperado, os desenvolvedores no geral, não consideram essas classes como problemáticas. Já os códigos afetados por cheiros Android têm mediana = 2 (Q3 = 4,25) e, portanto, são percebidos como problemáticas na maioria dos casos por desenvolvedores. Em relação aos cheiros tradicionais, a mediana de severidade é 3. Isso mostra que as classes afetadas por esses cheiros são percebidas pelos desenvolvedores como problemáticas. No entanto, embora essa diferença na percepção seja clara, observando-se os gráficos de violino na Figura 5.3a, tal diferença não é estatisticamente significativa (valor de p = 0,21). Conjeturamos que isso pode ser devido ao número limitado de pontos de dados (21 participantes).

4 DISCUSSÃO

Maurício:: ***vc sabe o que escrever aqui? qualquer coisa grita***

[Under construction]

5 AMEAÇAS A VALIDADE

Maurício:: ***aqui veja também o paper da michaela, para ver como se escreve ameacas em trabalhos qualitativos.***

[Under construction]

6 TRABALHOS RELACIONADOS

Maurício:: ***mover para o final, depois do ameacas a validade***

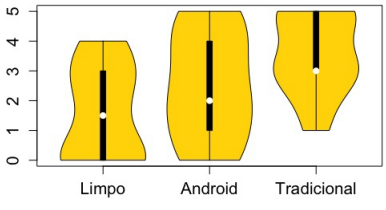
Muitas pesquisas têm sido realizadas sobre a plataforma Android, muitas delas focam em vulnerabilidades [16, 18, 19, 32, 51, 54, 55], autenticação [20, 52, 53] e testes [11, 29]. Diferentemente dessas pesquisas, nossa pesquisa tem foco na percepção dos desenvolvedores sobre boas e más práicas de desenvolvimento 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.

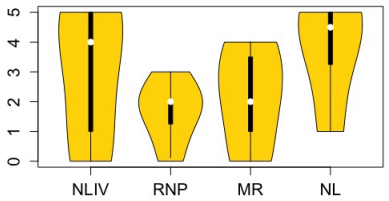
Verloop [49] conduziu um estudo no qual avaliou por meio de 4 ferramentas de detecção automatizada de cheiros de código (JDeodorant, Checkstyle, PMD e UCDetector) a presença de 5 cheiros de código (Long Method, Large Class, Long Parameter List, Feature Envy e Dead Code) em 4 projetos Android. Nossa pesquisa 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 [27] 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 [43]. 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



(a) ***Limpo=Códigos sem maus cheiros, Android=Códigos afetados pelas boas e más práticas de alta recorrência, Tradicional=Códigos afetados por maus cheiros tradicionais.***



(b) ***NLIV=No Logic In View, RNP=Resource Name Pattern, MR=Magic Resource, NL=Nested Layout.***

Figura 4: Severidade de cada boa e má prática

framework Spring MVC, um framework Java para desenvolvimento web. Outras pesquisas concluem que projetos Android possuem características diferentes de projetos java [28, 35, 41], 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 [25] e JavaScript [21]. Essas pesquisas nos inspiraram a buscar entender se existem *code smells* no *front-end* Android.

[seção não finalizada, á concluir.]

7 CONCLUSÃO

[Under construction]

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\OT1\textendashview\OT1\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] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. (2011).

[17] Juliet Corbin and Anselm Strauss. 2007. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (3 ed.). SAGE Publications Ltd.

[18] Enck, William, and Patrick Drew McDaniel Machigar Ongtang. 2009. Understanding Android Security. (2009).

[19] Enck, William, and Patrick McDaniel Machigar Ongtang. 2008. Mitigating Android software misuse before it happens. (2008).

[20] Zheran Fang and Yingjiu Li Weili Han. 2014. Permission Based Android Security: Issues and Countermeasures. (2014).

[21] A. Milani Fard and A. Mesbah. 2013. JSNOSE: Detecting javascript code smells. (2013).

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

[23] Martin Fowler. 2006. Code Smell. http://martinfowler.com/bliki/CodeSmell.html. (Feb. 2006).

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Boston.

[25] Golnaz Gharachorlu. 2014. *Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model.* Ph.D. Dissertation. The University of British Columbia.

[26] Barney G. Glaser and Anselm L. Strauss. 1999. *The Discovery of Grounded Theory: Strategies for Qualitative Research* (1 ed.). Aldine.

[27] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing Energy Code Smells with Reengineering Services. (????). Maus cheiros relacionados ao consumo de energia.

[28] Geoffrey Hecht. 2015. An Approach to Detect Android Antipatterns. (2015).

[29] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. (2011).

[30] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics.* Addison-Wesley Publishing Company. https://books.google.com.br/books?id=oHkhAQAAIAAJ

[31] Wharthon Jake. Butter Knife. http://jakewharton.github.io/butterknife/. (????). Last accessed at 11/04/2017.

[32] K Kavitha, P Salini, and V Ilamathy. 2016. Exploring the Malicious Android Applications and Reducing Risk using Static Analysis. (2016).

[33] Nigel King. 1994. In Qualitative methods in organizational research - A practical guide. (1994), 253.

[34] 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. (????).

[35] 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

[36] 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.
[37] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
[38] 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
[39] 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.
[40] 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.
[41] Jan Reimann and Martin Brylski. 2013. A Tool-Supported Quality Smell Catalogue For Android Developers. (2013).
[42] Johnny Saldaña. 2012. *The Coding Manual for Qualitative Researchers* (2 ed.). SAGE Publications Ltd.
[43] Android Developer Site. Andriod Fundamentals. https://developer.android.com/guide/components/fundamentals.html. (????). Last accessed at 04/09/2016.
[44] Android Developers Site. Android RecyclerView. https://developer.android.com/training/material/lists-cards.html. (????). Last accessed at 12/04/2017.
[45] Android Developer Site. Optimizing View Hierarchies). https://developer.android.com/topic/performance/rendering/optimizing-view-hierarchies.html). (????). Last accessed at 09/04/2017.
[46] 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.
[47] Developer Android Site. 2016. Resources Overview. https://developer.android.com/guide/topics/resources/overview.html. (2016). Last accessed at 08/09/2016.
[48] 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.
[49] Daniël Verloop. 2013. *Code Smells in the Mobile Applications Domain*. Ph.D. Dissertation. TU Delft, Delft University of Technology.
[50] Wikipédia. Inheritance (object-oriented programming). https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming). (????). Last accessed at 08/04/2017.
[51] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. 2016. Eficient Fingerprinting-based Android Device Identication with Zero-permission Identiers. (2016).
[52] 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
[53] S. Yu. 2016. Big privacy: Challenges and opportunities of privacy study in the age of big data. (2016).
[54] 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).
[55] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. 2014. Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps. (2014).