Sobre a Percepção dos Desenvolvedores Android em Relação aos Maus Cheiros de Código

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

Marco Aurélio Gerosa
Northern Arizona University
E Runke Dr
Flagstaff, Arizona 86011
gerosa@ime.usp.br

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

ABSTRACT

Cheiros de código são fortes 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 muitos 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 maus cheiros em projetos Android. Nós conduzimos um *survey* com 45 desenvolvedores e descobrimos que além de maus cheiros já mapeados, algumas estruturas específicas da plataforma são amplamente percebidas como más práticas, portanto, possíveis cheiros de código específicos. Desta percepção propômos três cheiros de código Android, validados com um especialista e em um experimento com 30 desenvolvedores. Ao final, discutimos os resultados encontrados bem como 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. Sobre a Percepção dos Desenvolvedores Android em Relação aos Maus Cheiros de Código. In *Proceedings of SBES 2017: 31st Brazilian Symposium on Software Engineering, Fortaleza, Ceará Brasil, Setembro 2017 (SBES'17),* 7 pages.
DOI: 10.475/123_4

1 INTRODUÇÃO

Escrever código com qualidade tem se tornado cada vez mais importante com o aumento da complexidade da tecnologia. Existem diferentes técnicas que auxiliam os desenvolvedores a escreverem código com qualidade incluindo *design patterns* e *code smells*. Defeitos de software, ou *bugs*, podem custar a empresas quantias significativas, especialmente quando conduzem a falhas de software [5, 18]. Evolução e manutenção de software também já se provaram como os maiores gastos com aplicações [24].

Code smells desempenham um importante papel na busca por qualidade de código. 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, Checkstyle e FindBugs são exemplos de ferramentas que identificam automaticamente alguns tipos de *code smells* em códigos Java.

Determinar o que é ou não um *code smells* é subjetivo e pode variar de acordo com tecnologia, desenvolvedor, metologia de desenvolvimento dentre outros aspectos []. Alguns estudos têm buscado por *code smells* tradicionais em projetos Android. Por exemplo, Verloop [25] analisou se classes derivadas do SDK Android são mais ou menos propensas a code smells tradicionais do que classes puramente Java. Linares et al. [16] usaram o método DECOR para realizar a detecção de 18 *anti-patterns* orientado a objetos em aplicativos móveis. Outros estudos identificaram *code smells* específicos Android, porém relacionados ao consumo inteligente de recursos do dispositivo, como bateria e memória, usabilidade, dentre outros [12, 19].

Nossa pesquisa complementa as anteriores no sentido de que também buscamos *code smells* Android, e se difere delas pois estamos buscando *smells* relacionados à qualidade do código Android, ou seja, qualidade relacionada a códigos específicos dessa plataforma. Por exemplo ACTIVITYs, FRAGMENTs e ADAPTERs são classes usadas na construção de telas e LISTENERs são responsáveis pelas interações com o usuário. Buscamos responder questões como *"quais são as **boas e más** práticas ao lidar com* ACTIVITY*s,* FRAGMENT*s,* ADAPTER*s e* LISTENER*s?"* ou *"quais são as **boas e más** práticas para a construção da interface visual?"*. Respondemos a essas e outras questões com base na percepção de desenvolvedores dessa plataforma.

Optamos por focar em elementos relacionados ao *front-end* Android pois encontramos pesquisas com uma curiosidade similar, porém relacionadas a identificação de *smells* em tecnologias usadas no *front-end* de projetos web [4, 10, 11]. E enquanto que *smells* em projetos Java já foram extensivamente estudados [], o *front-end* Android possui peculiaridades não encontradas, e portanto não investigadas, em código Java tradicional. Podemos citar alguns exemplos como o ciclo de vida definido pela plataforma que é atribuído a toda ACTIVITY ou FRAGMENT ou a criao da interface visual de ACTIVITYs

SBES'17, Fortaleza, Ceará Brasil

DOI: 10.475/123_4

que é feita através de arquivos XML chamados de LAYOUT RESOURCES.

Nesta pesquisa objetivamos investigar a percepção de desenvolvedores Android sobre boas e más práticas de código relacionadas ao *front-end* Android. Para definirmos quais elementos representam o *front-end* Android fizemos uma extensa revisão da documentação oficial e chegamos nos seguintes itens [22]:

* classes do tipo ACTIVITY,
* classes do tipo FRAGMENT,
* classes do tipo LISTENER,
* classes do tipo ADAPTER e
* os *applications resources* que são arquivos XML ou imagens utilizados na interface visual como DRAWABLEs, LAYOUTs, STYLEs e COLORs.

Como existem muitos tipos de *applications resources* [23], com o objetivo de limitar o tamanho do questionário, selecionamos quatro recursos: LAYOUT, STYLES, STRING e DRAWABLE. Esses recursos estão presentes no template padrão, selecionado automaticamente ao iniciar um novo projeto no Android Studio [21], IDE oficial para desenvolvimento de projetos da plataforma [2].

Com isso, pretendemos responder as seguintes questões de pesquisa:

RQ1 O que desenvolvedores consideram boas e más práticas no desenvolvimento Android?

RQ2 Códigos afetados pelos *code smells* propostos são percebidos pelos desenvolvedores como problemáticos?

Para coletar os dados iniciais, publicamos um questionário online questionando sobre boas e más práticas no *front-end* Android. Perguntas dissertativas foram usadas para não limitar o participante, possibilitar respostas mais completas e evitar enviesamento. 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 e com o feedback deles fizemos alguns ajustes relacionados a obrigatoriedade de algumas perguntas. As respostas dos participantes piloto foram desconsideradas para efeitos de viés.

Com os resultados obtidos foi possível compilar um catálogo com 21 Android *code smells* classificados em alta, média e baixa recorrência de percepção. Esperamos com este catálogo contribuir com as ideias iniciais para a definição de heurísticas para a detecção sistematizada desses *smells*.

As seções seguintes deste artigo esto organizadas da seguinte forma: na Seção 2 discutimos alguns trabalhos relacionados e o estado da arte sobre Android e *code smells*. Na Seção 3 falamos sobre os métodos utilizados em nosso estudo. A Seção 4 apresenta os resultados e as ameaças à validade do nosso estudo. Na Seção 5 discutimos e concluímos.

[**seção não finalizada.**]

2 TRABALHOS RELACIONADOS

Muitas pesquisas têm sido realizadas sobre a plataforma Android, muitas delas focam em vulnerabilidades [6–8, 15, 26, 29, 30], autenticação [9, 27, 28] e testes [3, 14]. 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 maus cheirosos.

Verloop [25] 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 [12] 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 [20]. 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. identificou 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 [13, 17, 19], 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 [11] e JavaScript [10]. Essas pesquisas nos inspiraram a buscar entender se

existem *code smells* no *front-end* Android.

[seção não finalizada, á concluir.]

3 METODOLOGIA

Conduzimos um estudo qualitativo e exploratório onde os dados foram coletados através de um questionário online com desenvoveldores Android. Esta seção descreve de forma detalhada a estrutura do questionário, os participantes e o análise realizada sob as respostas do questionário.

3.1 Questionário

O questionário continha 25 questões subdivididas 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 última 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. Todos estes dados estão disponíveis no nosso pacote de replicação[1].

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, bacharelado, mestrado, doutorado). As questões sobre idade, região, linguagens e grau de escolaridade continham a opção "outros" onde possibilitava que o participante inserisse sua resposta de forma manual.

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 a Activity as perguntas foram:

- Do you have any good practices to deal with Activities?
- Do you have any bad practices to deal with Activities?

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 opcional onde solicitamos o email do participante caso o mesmo tivesse interesse em participar de etapas futuras da pesquisa. As perguntas sobre boas e más práticas foram as a seguir:

- Are there any other *GOOD* practices in Android Presentation Layer we did not asked you or you did not said yet?
- Are there any other *BAD* practices in Android Presentation Layer we did not asked you or you did not said yet?

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.

3.2 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 4 meses, de 9 de Outubro de 2016 até 18 de Janeiro de 2017. Recebemos um total de 45 respostas sendo que 41 foram coletadas em Outubro de 2016, 3 no começo de Novembro de 2016 e 1 em Janeiro de 2017. Acreditamos que poucas e nenhuma respostas foram coletadas respectivamente, nos meses de Novembro e Dezembro, devido as festas comemorativas de fim de ano. 7 participantes não responderam nenhumas das questes da segunda e terceira seção, tornando suas respostas não úteis a pesquisa. 53% preencheram seus emails se disponibilizando para futuras etapas da pesquisa, acreditamos que este alto percentual pode indicar um interesse legítimo da comunidade de desenvolvedores Android pelo tema, reforçando a relevância do estudo.

A primeira seção do questionário continham questões demográficas. Com a análise dessas questões 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 30% indicaram possuir 2 anos ou menos de experiência com Android e mais de 70% indicaram 3 anos ou mais (13% 4 anos, 6,5% 5 anos, 15,5% 6 anos, 4,4% 7 anos e 4,4% 8 anos), vale considerar que a plarforma Android completa 10 anos em 2017, ou seja, 5 anos representam 50% do tempo desta tecnolgia 4) 36 respostas foram do Brasil, 7 de países europeus e 1 dos Estados Unidos (Califórnia). Estes dados estão sumarizados na Tabela 1.

3.3 Análise dos Dados

O processo de categorização e definição dos *smells* seguiu as seguintes etapas: Verticalização, Limpeza dos Dados, Iterações de Categorização, Divisões e Eliminação das Dúvidas. Essas etapas são detalhadas a seguir.

[1] https://github.com/SuelenGC/android-code-smells-article

Table 1: Experiência dos participantes com desenvolvimento Android.

Experiência com Android	Total de Participantes
até 1 ano	6 (13,33%)
2 anos	7 (15,56%)
3 anos	12 (26,67%)
4 anos	6 (13,33%)
5 anos	3 (6,67%)
6 anos	7 (15,56%)
7 anos	2 (4,44%)
8 anos	2 (4,44%)

A análise partiu da listagem das 45 respostas do questionário. A partir desta listagem realizamos o processo que denominamos como verticalização, ou seja, cada resposta de boa ou má prática se tornou um registro individual a ser analisado, resultando em 810 respostas sobre boas ou más práticas em algum elemento do front-end Android. O número 810 refere-se as 18 perguntas sobre boas e más práticas multiplicado pelos 45 participantes.

Nosso segundo passo foi realizar a limpeza dos dados. Essa etapa 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 diversas iterações nas respostas sobre boas e más práticas consideradas a fim de categorizá-las em algum novo smell Android ou algum smell pré-existente. Essas iterações consistiram em analisar resposta a resposta e atribuir uma ou mais categorias de algum possível novo smell Android ou pré-existente. Foram realizadas diversas iterações de categorização com o objetivo de normalizar as categorias, ou seja, evitar sinônimos e homônimos. Um sinônimo é o mesmo conceito com dois nomes diferentes e homônimos são dois conceitos diferentes com o mesmo nome.

Durante a categorização houve 30 respostas que não eram triviais de identificar uma categoria ou mesmo de dizer se essas respostas deveriam ser consideradas, sendo marcadas como *"talvez"* durante o processo e reavaliadas ao final, onde 6 permaneceram e 24 foram desconsideradas. Uma situação interessante é que diversas dessas respostas indicavam que não se deve usar Fragments porém não apresentavam nenhum argumento sobre o motivo, por exemplo: *"Fragments are the spawn of satan"* e *"I try to avoid them"*. Essas respostas inicialmente seriam desconsideradas, mas pela quantidade de repetições obtidas, 10 no caso, optamos por considerar.

Também durante a categorização, 9 respostas incialmente consideradas, foram desconsideradas. Para toda resposta desconsiderada foi indicado um motivo. Ao final da categorização, 313 boas e más práticas foram de fato consideradas.

Durante a categorização, uma mecânica que consideramos importante para a normalização das categorias, foi a criação de uma lista de categorias, onde a cada nova categoria atribuída, incrementávamos a lista e preenchíamos com descrições que indicavam que tipo de boa ou má prática estava recebendo aquela categoria. Esta mecânica ajudou a evitar homônimos e sinônimos e serviu como base para a definição e avaliação da relevância dos *smells* a serem trabalhados nos próximos passos.

Em seguida, passamos pela etapa de divisão, ou seja, as respostas que receberam mais de uma categoria foram divididas em duas ou mais respostas, de acordo com o número de categorias identificadas. 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 a categoria de smell que denominamos **Zumbi Referended Activity** e na segunda oração, a categoria de smell **Inherit From Support Library Always**. Ao dividí-la mantivemos o texto da resposta apenas relativo a categoria, como se fossem duas respostas válidas. Em algumas respostas que foram divididas não pudemos dividir o texto pois a resposta completa era necessário para entender ambas as categorizações, nesses casos, mantivemos a resposta original, mesmo que duplicada, e categorizamos cada uma de forma diferente. Após estas divisões, as 313 respostas iniciais se tornaram 388 sendo cada uma com apenas uma categoria de *smell*.

Ao final de todas as etapas, concluímos com 388 respostas sobre boas e más práticas categorizadas em 47 *smells*.

3.4 Análise e Definição dos Smells

Nosso objetivo nesta etapa foi entender quais *smells*, dos 47 identificados, eram mais recorrêntes. Para isso, contabilizamos cada *smell*, em quantas respostas ele aparecia, ou seja, se duas respostas foram categorizadas com o *smell* "A" então diz-se que esse *smell* tem contagem 2. Depois da contagem, elaboramos intervalos que indicam recorrência ALTA, maior ou igual a 20, MÉDIA, dentre 6 e 19 e BAIXA, dentre 3 e 5. Abaixo de 3 classificamos como IRRELEVANTE. Obtemos o seguinte resultado:

Table 2: Smells identificados vs. recorrência percebida pelos participantes do questionário

Recorrência	**Quantidade de Smells**
Alta	5
Média	17 (1 *smell* tradicional)
Baixa	5
Irrelevante	20

Os irrelevantes foram desconsiderados nesta etapa de definição. Os demais foram definidos com a ajuda das respostas dos participantes. Para cada *smell* definimos os seguintes tópicos:

* **Quando ocorre**. Indicamos os motivos pelo qual foi considerado uma má prática.

- **Contexto/exemplo**. Indicamos algum exemplo ou contexto prático.
- **Elementos afetados**. Eventualmente algum *smell* afeta mais de um elemento, nesse tópico abordamos em quais os elementos este *smell* pode estar presente.
- **Solução**. Indicamos possíveis refatorações para reduzir ou eliminar o *smell*.

É importante ressaltar que todas as definições dos *smells* foram apoiadas nas respostas dos participantes. Se os participantes não indicaram algum tópico para algum *smell*, o mesmo não foi definido.

4 RESULTADOS

4.1 Android Code Smells

4.1.1 Duplicated Styles Attributes (DSA). **Ocorre quando** mais de um view de um ou mais layouts usam o mesmo conjunto de atributos para definir sua aparência. No estilo **ocorre quando** se vê mais de um style repetindo o mesmo conjunto de atributos e valores. Os **elementos afetados** são xmls de layout ou estilo. A **heurística para detectá-lo** é identificar um conjunto de atributos que se repetem em um ou mais views de um ou mais layouts. Sua **solução** é extrair um estilo com o conjunto de atributos repetidos relacionados a características de exibição da view.

Considere o seguinte **contexto** onde o `TextView` abaixo é usado em dois momentos no mesmo layout (exemplo extraído da documentação do Android[2]):

Listing 1: `TextView` usado em mais de um layout e com estilo definido por atributos

```
<TextView
    Android:layout-width="match-parent"
    Android:layout-height="wrap-content"
    Android:textColor="#00FF00"
    Android:typeface="monospace"
    Android:text="@string/hello" />
```

Uma opção seria extrair o estilo acima para um recurso de estilos:

Listing 2: `TextView` usando estilo ao invés de atributos separados

```
<TextView
    Android:layout-width="match-parent"
    Android:layout-height="wrap-content"
    Android:textAppearance="@style/CodeFont"
    Android:text="@string/hello" />
```

Arquivo de estilo com o estilo CodeFont criado:

Listing 3: `TextView` usando estilo em vez de atributos separados

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" ...>
        <item name="Android:textColor">
            #00FF00
        </item>
        <item name="Android:typeface">
            monospace
        </item>
    </style>
</resources>
```

[2] https://developer.Android.com/guide/topics/ui/themes.html

4.1.2 Suspicious Extra Knowledge About Behavior (SEKAB). **Ocorre quando** uma view possui um conhecimento maior do que o considerado aceitável sobre os detalhes de seu comportamento. Para este smell identificamos por meio das respostas que três elementos da view **podem ser afetados** por ele, são: ACTIVITY, FRAGMENT e ADAPTER. E a depender de qual view está se tratando, o que é considerado aceitável pode mudar. Sendo assim, chegamos a seguinte escala de conhecimento aceitável:

- nível 1 - implementação de classe concreta;
- nível 2 - através de *implements* (polimorfismo);
- nível 3 - através de classe anônima;
- nível 4 - através de *inner class*.

Para ADAPTERs, apenas o nível 1 foi considerado como aceitável. Para ACTIVITYs e FRAGMENTs, até o nível 3 é considerado aceitável, sendo que houveram indicações positivas e nenhuma negativa sobre o uso do nível 1, houveram indicações positivas e negativas sobre o nível 2 e houveram apenas indicações negativas sobre o nível 3 e 4.

Como **solução**, segundo as respostas R666 e R672, *"adapters devem ser o mais estúpido possível"* (tradução livre). Podemos dizer então que se for necessário atribuir algum comportamento para alguma view que está sendo populada, isso deverá ser feito usando a método descrito para o nível 1. Já para ACTIVITYs e FRAGMENTs a tolerância de conhecimento aumenta até o nível 3, sendo 1 e 2 os mais indicados e 3 tolerável ás vezes (aqui talvez existam dois thresholds como número de linhas do método do evento ou mesmo quantidade de classes anônimas na classe). O nível 4 é inaceitável para todos os elementos afetados por este smell.

Os **elementos afetados** são: ACTIVITY, FRAGMENT e ADAPTERS. As **heurísticas** de detecção são:

- [nível 4] se existe uma ou mais inner class de listener
- [nível 3] se existe um certo número de classes anônimas
- [nível 2 e 3] se LOC dentro do método do listener - anônimo ou sobreescrito é maior que 1

4.1.3 Flex Adapter (FA). **Ocorre quando** um adapter precisa de condicionais para decidir se determinada view deve estar visivel ou não ou quando um adapter recebe mais de um tipo de objeto para ser colocado na view.

Como **solução**, R578 e R647 sugerem que, caso os condicionais sejam para decidir sobre a exibição ou não de views, separa em duas ou mais views. Sobre adaptar mais de um tipo de objeto, a resposta R577 sugere a delegação para outros adapters especialistas em lidar com cada objeto.

4.1.4 Magic Resource (MR). **Ocorre quando** se encontra uma string, integer, color ou dimension em arquivos que não sejam os seus respectivos onde é possível atribuir um nome e reutilizá-los.

4.1.5 Resource Name Pattern (RNP). **Ocorre quando** seus recursos não seguem um padrão de nomenclatura. Tornando difícil encontrá-los e reaproveitá-los. A seguir apresentamos uma pasta res desorganizada e despadronizada:

Listing 4: Recursos de aplicação sem padrão de nomenclatura

```
– res
  – layout
    – tela-de-login.xml
    – cadastro-act.xml
    – propaganda-frag.xml
    – item1.xml
    – item2.xml
  – values
    – strings.xml
    – styles.xml
    – colors.xml
  – drawables
    – ic-app.jpg
    – tela-cadastro.png
    – bg.xml
```

Agora, uma pasta res seguindo as sugestões de solução:

Listing 5: Recursos de aplicação com padrão de nomenclatura

```
– res
  – layout
    – activity-login.xml
    – activity-cadastro.xml
    – fragment-propaganda.xml
    – item-lista-propaganda-tipo1.xml
    – item-lista-propaganda-tipo2.xml
  – values
    – strings-login.xml
    – strings-cadastro.xml
    – strings-propaganda.xml
    – styles-login.xml
    – colors.xml
  – drawables
    – icone-app.jpg
    – icone-cadastro.png
    – background.xml
```

Em se falando de strings resources, segundo P27 "Iniciar o nome de uma string com o nome da tela onde vai ser usada", e P2 reforça essa ideia sugerindo ¡screen¿-¡type¿-¡text¿. Eg.: welcome-message-title, registration-field-name, registration-hint-edit-name. P34 e P4 sugere que o nome do elemento usando a string como parte do nome da string: dialog.STRING-NAME or hint.STRING-NAME

Em se falando de layout resources, P11 sugere sempre começar com "activity-", "fragment-", "ui-" (para UI customizadas) já P12 sugere o uso de sufixos, como por exemplo "xxx-activity".

Sobre drawables e estilos não foi sugerida uma maneira específica de nomenclatura, apenas enfatizaram para definir um padrão e usá-lo em toda a aplicação.

4.1.6 Messy String Resources (MStringR). **Ocorre quando** se tem um único arquivo strings.xml para guardar todas as strings da aplicação ou quando os arquivos string encontram-se desorganizados, sem seguir nenhum padrão específico. Como **solução**, P28 e P42 sugerem separar as strings por arquivos que indiquem a qual tela aquelas strings pertencem. P42 também sugere ter um arquivo de strings comuns a toda aplicação. P41 sugere ainda separar os arquivos de acordo com contextos como: urls, mensagens de erro. P32 sugere usar um único arquivo porém separando-o em blocos, onde casa bloco indica uma tela.

4.1.7 Messy Style Resources (MStyleR). **Ocorre quando** se tem um único arquivo styles.xml para guardar todos os estilos e temas da aplicação ou quando os arquivos styles encontram-se desorganizados, sem seguir nenhum padrão específico. Como **solução**, P7, P42 e P40 sugerem criar um styles para estilos de componentes e outro para temas. P8 já sugere quebrar em vários arquivos estilos e um para o tema.

4.2 Interpretação dos Dados

As vezes o que era indicado como boa prática para um elemento era um smell percebido em outro elemento, por exemplo, a R1 diz que *"Sempre que noto ter mais de um [layout] resource usando o mesmo estilo eu tento movê-lo [o estilo] para meu recurso de estilo"* (tradução livre) ao responder sobre boa prática para o style resource, porém esta resposta foi considerada para definir o smell Duplicated Styles Attributes que é percebido em recursos de layout ou styles.

4.3 Afirmação sobre o *front-end* Android

Uma opinião que foi unânime em muitas respostas foi que de fato, desenvolvedores tratam Activitys, Fragments e Adapters como elementos do front-end Android, conforme constatamos na seção 3.1. Isso pode ser observado diversas vezes com respostas por exemplo, P25 indicou como boa prática na Activity "Nenhuma lógica [de negócio] aqui" (tradução livre), o P40 afirma sobre má prática em adapter é *"Lógica de negócio em adapters é não-não"* (tradução livre), ao falarem sobre fragments, muitos indicaram *"O mesmo da Activity"*. Ou seja, primeiramente estas respostas reforçam nossa definição inicial sobre elementos que compoêm o front-end Android, e por outro lado, vimos que muitas vezes fragments são tratados como Activitys, ao se falar de boas e más práticas de código.

5 DISCUSSÃO

Under construction

6 AMEAÇAS A VALIDADE

Under construction

7 CONCLUSÃO

Under construction

REFERENCES

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

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

[5] 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.

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

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

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

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

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

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

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

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

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

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

[16] 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. (????).

[17] 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

[18] 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

[19] Jan Reimann and Martin Brylski. 2013. A Tool-Supported Quality Smell Catalogue For Android Developers. (2013).

[20] Android Developer Site. Andriod Fundamentals. https://developer.android.com/guide/components/fundamentals.html. (????). Last accessed at 04/09/2016.

[21] 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.

[22] Android Developer Site. 2016. Documentação Site Android Developer. https://developer.android.com. (2016). Last accessed at 27/10/2016.

[23] Developer Android Site. 2016. Resources Overview. https://developer.android.com/guide/topics/resources/overview.html. (2016). Last accessed at 08/09/2016.

[24] 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.

[25] Daniël Verloop. 2013. *Code Smells in the Mobile Applications Domain*. Ph.D. Dissertation. TU Delft, Delft University of Technology.

[26] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. 2016. Eficient Fingerprinting-based Android Device Identication with Zero-permission Identiers. (2016).

[27] 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

[28] S. Yu. 2016. Big privacy: Challenges and opportunities of privacy study in the age of big data. (2016).

[29] Yuan Zhang, Min Yang, Zhemin Yang, Guofei GU, and Binyu Zang Peng Ning. 2004. Exploring Permission Induced Risk in AndroidApplications for Malicious Detection. (2004).

[30] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. 2014. Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps. (2014).