Cheiros de Código em projetos Android. Um estudo qualitativo sobre a percepção de desenvolvedores

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

Marco Aurélio Gerosa
Universidade de São Paulo
Rua do Matão, 1010
São Paulo, SP 05508-090
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. Cheiros de Código em projetos Android. Um estudo qualitativo sobre a percepção de desenvolvedores. In *Proceedings of SBES 2017: 31st Brazilian Symposium on Software Engineering, Fortaleza, Ceará Brasil, Setembro 2017 (SBES'17),* 6 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 [4, 17]. Evolução e manutenção de software também já se provaram como os maiores gastos com aplicações [22].

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 forma automática no código. PMD, Checkstyle e FindBugs por exemplo, são 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 [23] analisou se classes derivadas do SDK Android so mais ou menos propensas a code smells tradicionais do que classes puramente Java. Linares et al. [15] 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 [11, 18]. Nossa pesquisa complementa as anteriores no sentido de que também buscamos *code smells* Android, e se difere delas pois estamos buscando *smells* relacionados a qualidade do código Android no sentido de responder questões como: quais são as más práticas ao lidar com `Android resources`, ou ao lidar com `activity`s, `fragment`s, `adapter`s e `listener`s. Nossa pesquisa objetiva definir *code smells* Android baseado em o que desenvolvedores desta plataforma percebem como boas e más práticas. Optamos por focar em elementos relacionados ao *front-end* Android visto que diversar outras pesquisas também têm se focado em tecnologias de *front-end* web [9, 10].

Nas seções seguintes deste artigo, discutiremos primeiro alguns trabalhos relacionados (Seção 2) e os métodos utilizados em nosso estudo (Seção 3). A Seção 4 apresenta os resultados e as ameaças à validade do nosso estudo. Na Seção 5 discutimos e concluímos... e terminamos com uma discussão de trabalhos futuros (Seção 6).

[seção não finalizada.]

2 TRABALHOS RELACIONADOS

Muitas pesquisas têm sido realizadas sobre a plataforma Android, muitas delas focam em vulnerabilidades [5–7, 14, 24,

SBES'17, Fortaleza, Ceará Brasil

DOI: 10.475/123_4

27, 28], autenticação [8, 25, 26] e testes [2, 13]. Diferentemente destas 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 code smells, podemos chegar a heurísticas para identificá-los e com estas heurísticas, implementar ferramentas que automatizem o processo de identificar códigos maus cheirosos.

Verloop conduziu um estudo onde 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 [23]. Nossa pesquisa se relaciona com a de Verloop no sentido de que também estamos buscando por cheiros de código, entretanto, ao invés de buscarmos por cheiros de código já definidos, realizamos uma abordagem inversa onde, 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. conduzem um estudo sobre formas de detectar e refatorar cheiros de código relacionados a uso eficiênte de energia [11]. Os autores compilam um catálogo com 8 cheiros de código e trabalham 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 a 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 com a linguagem de programação Java [19]. 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 [12, 16, 18], 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 [10] e Javascript [9]. Estas pesquisas nos inspiraram a buscar entender se existem *code smells* no *front-end* Android.

[seção não finalizada, á concluir.]

3 METODOLOGIA

Nesta pesquisa objetivamos investigar a percepção de desenvolvedores Android sobre boas e más práticas de código em projetos Android. Desta forma, 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 dados iniciais para esta pesquisa, publicamos um questionário online sobre boas e más práticas em determinados elementos da plataforma Android. Perguntas dissertativas foram usadas para possibilitar respostas completas e evitar viés. O questionário foi escrito em inglês porém informava o participante que respostas em inglês ou português eram aceitas. Antes da divulgação, fizemos um piloto com 3 desenvolvedores Android, com o feedback deles fizemos alguns ajustes relacionados a obrigatoriedade de algumas perguntas. As respostas do piloto foram desconsideradas para efeitos de viés.

O questionário foi dividido em 3 seções. A primeira continham perguntas para mapeamento demográfico. A segunda continham perguntas relacionadas a boas e más práticas em elementos do *front-end* Android, que será melhor detalhado na seção 3.1. A terceira seção continham perguntas para obter alguma ideia que não tenha sido capturada nas questões anteriores, além do email do participante caso o mesmo quisesse ser contactado para participar de outras etapas da pesquisa.

Com os resultados obtidos foi possível compilar um catálogo com 21 Android *code smells*. Esse catálogo irá contribuir como base para i) a definição de heurísticas para detecção ii) automatização de ferramentas de detecção automática desses smells.

A seguir, a seção 3.1 aborda sobre como chegamos a definição dos elementos do *front-end* Android, focados nesta pesquisa. A seção 3.2 trata sobre a elaboração do questionário. Na seção 3.3 falamos sobre os participantes e a seção 3.4 aborda de forma detalhada a análise para a definição dos smells.

3.1 *Front-End* Android

Para definirmos quais seriam os elementos usados no desenvolvimento do *front-end* do Android fizemos uma extensa revisão da documentação oficial [20] e chegamos na listagem a seguir:

- classes do tipo Activity;
- classes do tipo Fragment;
- classes do tipo Listener;
- classes do tipo Adapter;
- recursos do aplicativo como drawables, layouts, styles, colors dentre outros.

Como existem muitos tipos de recursos [21], com o objetivo de limitar o tamanho do questionário, selecionamos os quatro

recursos mais frequentes em aplicativos: LAYOUT, STYLES, STRING e DRAWABLE.

3.2 Questionário

A primeira seção do questionário abordou perguntas de mapeamento demográfico. A segunda seção abordou perguntas sobre boas e más práticas sobre cada um dos elementos citados na seção 3.1. Baseamos a estrutura das nossas perguntas em uma pesquisa similar realizada por Aniche et al. [3] ao pesquisar sobre *smells* no framework Spring MVC. Por exemplo, para ACTIVITYs fizemos as seguintes perguntas:

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

A terceira seção objetivou captar qualquer última ideia sobre boas e más práticas não captadas nas questões anteriores. Para isso fizemos as seguintes perguntas:

- 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?

3.3 Participantes

Com o questionário obtivemos 45 respostas. O questionário foi divulgado em grupos de desenvolvedores Android como por exemplo o Slack AndroidDevBR, maior grupo de desenvolvedores Android do Brasil contando hoje com mais de 2500 participantes. Coletamos 36 respostas do Brasil, 7 de países europeus e 1 dos Estados Unidos, 1 participantes preferiu não responder. A tabela 2 apresenta dados sobre anos de experiência dos participantes com desenvolvimento de sofware e com desenvolvimento Android. Notamos que 67% dos participantes possuem mais de 5 anos de experiência com desenvolvimento de software e que 71% possuem 3 anos ou mais de experiência com desenvolvimento Android.

Table 1: Experiência dos participantes com desenvolvimento de software e desenvolvimento Android.

	Experiência com Software	Experiência com Android
1-2 anos	5	13
3-5 anos	10	21
6-10 anos	30	11

3.4 Categorização e Identificação dos Smells

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.

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. Esta etapa consistiu em remover respostas obviamente não úteis como respostas em branco, que continham frases como *"Não"*, *"Não que eu saiba"*, *"I don't remember"* 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 *"Like all Java code, the overkill of Utils..."* 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 consistiam 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 houveram 30 respostas que não eram triviais de identificar uma categoria ou mesmo de dizer se estas respostas deveriam ser consideradas, essas foram 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"*. Estas 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.

3.5 Análise e Definição dos Smells

Nosso objetivo agora é entender quais smells, dos 47 identificados, são de fato relevantes. Para isso, contabilizamos cada smells, em quantas respostas ele aparecia, ou seja, se duas respostas foram categorizadas com o smell "A" então diz-se que este smell tem contagem 2. Depois da contagem, elaboramos intervalos que indicam relevância ALTA (maior ou igual a 20), MéDIA (dentre 6 e 19), BAIXA (dentre 3 e 5) e IRRELEVANTE (menor ou igual a 2). E obtemos o seguinte resultado:

Table 2: Smells identificados vs. Relevância percebida pelos participantes do survey

Relevância	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,
- contexto ou exemplo,
- elementos afetados,
- solução,
- heurística para deteção.

Nossas definições foram apoiadas nas respostas dos participantes. Se os participantes não indicaram algum ponto para o smell, o mesmo não é apresentado.

3.6 Android Code Smells

3.6.1 Duplicated Styles Attributes. **Ocorre quando** mais de um view de um ou mais layouts usam o mesmo conjunto de atributos para definir sua aparncia. No estilo **ocorre quando** se v 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[1]):

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 ao invés de atributos separados

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

3.6.2 Suspicious Extra Knowledge About Behavior. **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 trs 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.

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

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 RESULTADOS

4.1 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.

4.2 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 CONCLUSÃO

Under construction

REFERENCES

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

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

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

[4] 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.

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

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

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

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

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

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

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

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

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

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

[15] 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. (????).

[16] 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

[17] 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

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

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

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

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

[22] 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.

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

[24] Wenjia Wu, Jianan Wu, Yanhao Wang, and Ming Yang Zhen Ling. 2016. Eficient Fingerprinting-based Android Device Identication with Zero-permission Identiers. (2016).
[25] 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
[26] S. Yu. 2016. Big privacy: Challenges and opportunities of privacy study in the age of big data. (2016).
[27] Yuan Zhang, Min Yang, Zhemin Yang, Guofei GU, and Binyu Zang Peng Ning. 2004. Exploring Permission Induced Risk in AndroidApplications for Malicious Detection. (2004).
[28] Yuan Zhang, Min Yang, Zhemin Yang, and Binyu Zang. 2014. Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps. (2014).