

**Detecção de Anomalias na Camada de Apresentação
de Aplicativos Android Nativos**

Suelen Goularte Carvalho

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: Mestrado em Ciência da Computação

Orientador: Marco Aurélio Gerosa, Ph.D.

São Paulo, Julho de 2016

Detecção de Anomalias na Camada de Apresentação de Aplicativos Android Nativos

Esta é a versão original da dissertação elaborada
pela candidata Suelen Goularte Carvalho, tal como
submetida a Comissão Julgadora.

Comissão Julgadora:

- Marco Aurélio Gerosa, Ph.d. — IME-USP

Dedico esta dissertação de mestrado a minha mãe.

“O motivo do tempo é que tudo não acontece de uma vez só.”

— Albert Einstein

Agradecimentos

A fazer.

Resumo

Android é o sistema operacional para dispositivos móveis mais usado atualmente, com 83% do mercado mundial e mais de 2 milhões de aplicativos disponíveis na loja oficial. Aplicativos Android tem se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. Este contexto pode levar a decisões ruins de *design* de código, conhecidas como anomalias ou maus cheiros, e podem degradar a qualidade do projeto, tornando-o de difícil manutenção. Desta forma, uma tarefa frequente de desenvolvedores de software é identificar trechos de código problemáticos para que possam refatorar, sendo o objetivo final ter constantemente uma base de código que favoreça a manutenção e evolução. Para isso, desenvolvedores costumam fazer uso de estratégias de detecção de maus cheiros de código. Apesar de já existirem diversos maus cheiros catalogados, como por exemplo *God Class*, *Long Method*, dentre outros, eles não levam em consideração a natureza do projeto. Projetos Android possuem particularidades relevantes e não experimentadas até o momento, por exemplo, um diretório que armazena todos os recursos usados na aplicação, uma classe que tende a acumular diversas responsabilidades. Pesquisas específicas sobre projetos Android ainda são poucas. Nesta dissertação pretendemos identificar, validar e documentar maus cheiros de código Android com relação à camada de apresentação, onde se encontra as maiores diferenças quando se comparado a projetos tradicionais. Em outros trabalhos sobre maus cheiros Android, foram identificados maus cheiros relacionados à segurança, consumo inteligente de recursos ou que de alguma maneira influenciam a experiência ou expectativa do usuário. Diferentemente deles, nossa proposta é catalogar maus cheiros Android que influenciem a qualidade do código. Com isso, os desenvolvedores terão mais um recurso para a produção de código de qualidade.

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

Abstract

A task that constantly software developers need to do is identify problematic code snippets so they can refactor, with the ultimate goal to have constantly a base code easy to be maintained and evolved. For this, developers often make use of code smells detection strategies. Although there are many code smells cataloged, such as *God Class*, *Long Method*, among others, they do not take into account the nature of the project. However, Android projects have relevant features and untested to date, for example the `res` directory that stores all resources used in the application or an `ACTIVITY` by nature, accumulates various responsibilities. Research in this direction, specific on Android projects, are still in their infancy. In this dissertation we intend to identify, validate and document code smells of Android regarding the presentation layer, where major distinctions when compared to traditional designs. In other works on Android code smells, were identified code smells related to security, intelligent consumption of device's resources or somehow influenced the experience or user expectation. Unlike them, our proposal is to catalog Android code smells which influence the quality of the code. It developers will have another ally tool for quality production code.

Keywords: android, code smells, code quality, software engineering, software maintenance, code metrics.

Sumário

Lista de Abreviaturas	vi
Lista de Símbolos	vii
Lista de Figuras	viii
1 Introdução	1
1.1 Questões de Pesquisa	3
1.2 Contribuições	4
1.3 Organização da Tese	4
2 Fundamentação Conceitual	6
2.1 Qualidade de Código	6
2.2 Maus Cheiros	6
2.3 Android	7
2.3.1 Arquitetura da Plataforma	7
2.3.2 Fundamentos do Desenvolvimento Android	9
2.3.3 Recursos do Aplicativo	12
2.3.4 Interfaces de Usuários	13
2.3.5 Eventos de Interface	16
3 Camada de Apresentação Android	17

4	Trabalhos Relacionados	19
5	Método de Pesquisa	21
5.1	Definição dos Maus Cheiros	21
5.1.1	Coleta de Dados	21
5.1.2	Análise dos Dados	22
5.1.3	Validação com Especialistas	22
5.2	Percepção dos Desenvolvedores	22
5.2.1	Coleta de Dados	22
5.2.2	Análise dos Dados	22
5.3	Impacto na Tendência a Mudanças	22
6	Catálogo de Maus Cheiros	23
6.1	Code Smell 1	23
7	Conclusão	24
7.1	Principais contribuições	24
7.2	Trabalhos futuros	24
A	Questionário sobre Boas e Más Práticas	25
	Referências Bibliográficas	28

Lista de Abreviaturas

Software Development Kit Kit para Desenvolvimento de Software

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

Lista de Símbolos

Σ Sistema de transição de estados

Lista de Figuras

2.1	Arquitetura do sistema operacional Android.	8
2.2	Árvore hierárquica de Views e ViewGroups do Android.	14

Capítulo 1

Introdução

Em 2017 o Android completará uma década desde seu primeiro lançamento em 2007. Atualmente há disponível mais de 2 milhões de aplicativos na Google Play Store, loja oficial de aplicativos Android [7]. Mais de 83,5% dos dispositivos móveis no mundo usam o sistema operacional Android, e esse percentual vem crescendo ano após ano [6, 8]. Atualmente é possível encontrá-lo também em outros dispositivos como *smart TVs*, *smartwatches*, carros, dentre outros [3, 5].

Aplicativos Android tem se tornado complexos projetos de software que precisam ser rapidamente desenvolvidos e regularmente evoluídos para atender aos requisitos dos usuários. Este contexto pode levar a decisões ruins de *design* de código, conhecidas como anomalias ou maus cheiros, e podem degradar a qualidade do projeto, tornando-o de difícil manutenção. [15].

Desenvolvedores de software constantemente i) escrevem código fácil de ser mantido e evoluído e ii) detectam trechos de código problemáticos. Para a primeira tarefa, desenvolvedores comumente buscam se apoiar em boas práticas e *design patterns* já estabelecidos [9, 14, 23]. Para a segunda tarefa, é comum utilizar estratégias de detecção de maus cheiros de código [11], que apontam trechos de códigos que podem se beneficiar de refatoração, ou seja, melhorar o código sem alterar o comportamento [2]. Apesar de já existir um catálogo extenso de maus cheiros, eles não levam em consideração a natureza do projeto e suas particularidades.

Projetos Android possuem particularidades que ainda não foram experimentadas em projetos orientados a objetos, principalmente com relação à camada de apresentação, onde apresenta suas maiores diferenças. Conforme relatado por Hecht [15] com relação a projetos Android, “*antipatterns* específicos ao Android são mais comuns e ocorrem mais frequente-

mente do que *antipatterns* OO (Orientados a Objetos)” (tradução livre). Por exemplo, além de código Java, grande parte de um projeto Android é constituído por arquivos XML. Estes são os *recursos da aplicação* (do inglês *Application Resources*) e ficam localizados no diretório `res` do projeto. São responsáveis por apresentar algo ao usuário como uma tela, uma imagem, uma tradução e assim por diante. No início do projeto os recursos costumam ser poucos e pequenos. Conforme o projeto evolui, a quantidade e complexidade dos recursos tende a aumentar, trazendo problemas para encontrá-los, reaproveitá-los e entendê-los. Enquanto estes problemas já estão bem resolvidos em projetos orientados a objetos, ainda não é trivial encontrar uma forma sistemática de identificá-los em recursos de projetos Android para que possam ser refatorados.

Outra particularidade é sobre componentes como `ACTIVITIES`, que representam uma tela no Android. `ACTIVITIES` são classes que possuem muitas responsabilidades [31], estão vinculadas a um `LAYOUT`, que representa uma interface com o usuário, e normalmente precisam de acesso a classes do modelo da aplicação. Analogamente ao padrão MVC, `ACTIVITIES` fazem os papéis de `VIEW` e `CONTROLLER` simultaneamente. Isto posto, é razoável considerar que o mau cheiro *God Class* [22] é aplicável neste caso, no entanto, conforme bem pontuado por Aniche et al. [11] “*enquanto [God Class] se encaixa bem em qualquer sistema orientado a objetos, ele não leva em consideração as particularidades arquiteturais da aplicação ou o papel desempenhado por uma determinada classe.*” (tradução livre). `ACTIVITIES` são componentes específicos da plataforma Android, responsáveis pela apresentação e interações do usuário com a tela [1].

Na prática, desenvolvedores Android percebem estes problemas frequentemente. Muitos deles já se utilizam de práticas para solucioná-los, conforme relatado pelo Reimann et al. [21] “o problema no desenvolvimento móvel é que desenvolvedores estão cientes sobre maus cheiros apenas indiretamente porque estas definições [dos maus cheiros] são informais (boas práticas, relatórios de problemas, fóruns de discussões, etc) e recursos onde encontrá-los estão distribuídos pela internet” (tradução livre). Ou seja, não é encontrado atualmente um catálogo único de boas e más práticas, tornando difícil a detecção e sugestão de refatorações apropriadas às particularidades da plataforma.

Nas principais conferências de manutenção de software, dentre 2008 a 2015, apenas 5 artigos foram sobre maus cheiros Android, dentro de um total de 52 artigos sobre o assunto [17]. A ausência de um catálogo de maus cheiros Android resulta em (i) uma carência de conhecimento sobre boas e más práticas a ser compartilhado entre praticantes da plataforma, (ii) indisponibilidade de ter uma ferramenta de detecção de maus cheiros de forma a alertar

automaticamente os desenvolvedores da existência dos mesmos e (iii) ausência de estudo empírico sobre o impacto destas más práticas na manutenibilidade do código de projetos Android. Por estes motivos, boas e más práticas que são específicas a uma plataforma, no caso Android, tem emergido como tópicos de pesquisa sobre manutenção de código [10].

1.1 Questões de Pesquisa

Nesta dissertação objetivamos mapear **alguns maus cheiros específicos à camada de apresentação de projetos Android**. Para isso, exploramos as seguintes questões:

Q1: O que desenvolvedores consideram boas e más práticas com relação à *Camada de Apresentação* de um projeto Android?

O objetivo desta questão é obter insumos para a definição dos maus cheiros e propostas de como refatorá-los. Para responder esta questão, teremos as seguintes etapas:

- **Etapas de coleta de dados**

Elaboramos um questionário para coletar boas e más práticas utilizadas ou percebidas por desenvolvedores Android com relação à camada de apresentação. Pretende-se aplicar o questionário na comunidade de desenvolvedores Android do Brasil e exterior.

- **Etapas de identificação de maus cheiros**

Os dados coletados com o questionário serão analisados de forma a resultar numa lista de possíveis maus cheiros.

- **Etapas de validação**

A lista de maus cheiros resultante será validada com desenvolvedores Android especialistas, dos quais, alguns compõem times que lidam com Android no Google.

Estas etapas são detalhadas na seção 5.1 do capítulo 5.

Q2: Desenvolvedores Android percebem as classes afetadas pelos maus cheiros propostos como problemáticas?

Com esta questão pretende-se validar a relevância e assertividade dos maus cheiros catalogados na Q1. Para isso será conduzido um experimento com desenvolvedores Android de forma a avaliar se eles percebem os códigos afetados pelos maus cheiros como indicativos de trechos problemáticos. Este processo é detalhado na seção 5.2 do capítulo 5.

Q3: Qual a relação entre os maus cheiros propostos e a tendência a mudanças do código?

Estudos prévios mostram que maus cheiros tradicionais (e.g., Blob Classes) podem impactar na tendência a mudanças em classes do projeto [11]. Desta forma, esta questão pretende, através de um experimento com desenvolvedores Android, analisar o impacto dos maus cheiros propostos na tendência a mudanças em projetos Android. Este processo é detalhado na seção 5.3 do capítulo 5.

1.2 Contribuições

Ao final desta pesquisa, pretende-se contribuir com um catálogo validado de maus cheiros na camada de apresentação de aplicativos Android. Desta forma desenvolvedores conseguiram identificar pontos específicos a serem refatorados em projetos Android ajudando a obter um código com mais qualidade, fácil de ser mantido e evoluído.

1.3 Organização da Tese

Esta dissertação está organizada da seguinte forma:

- **Capítulo 1** Introdução

Neste capítulo é introduzido o contexto atual do desenvolvimento de aplicativos Android. Apresenta-se quais são as motivações e o problema a ser resolvido. É dado também uma breve introdução sobre como pretende-se resolvê-lo.

- **Capítulo 2** Trabalhos Relacionados

Neste capítulo pretende-se apresentar estudos relevantes já feitos em torno do tema de maus cheiros Android e o que esta dissertação se diferencia deles.

- **Capítulo 3** Fundamentação Teórica

Neste capítulo é passado ao leitor informações básicas relevantes para o entedimento do trabalho. Os assuntos aprofundados aqui são: Qualidade de Código, *Code Smells* e Android.

- **Capítulo 4** Pesquisa

Neste capítulo são apresentados detalhes de toda as etapas da pesquisa. Quais tipos de coleta de dados serão usados. Quais os critérios de escolha dos respondentes e porquê. Como foram realizadas as análises dos dados e quais resultados foram obtidos.

- **Capítulo 5** Catálogo de maus cheiros

Neste capítulo, serão catalogados os maus cheiros validados.

- **Capítulo 6** Conclusão

Neste capítulo são apresentadas as conclusões do trabalho bem como as suas limitações e sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Conceitual

Para a compreensão deste trabalho é importante ter claro a definição de 3 itens, são eles: Qualidade de Código, *Code Smells* e *Android*.

2.1 Qualidade de Código

2.2 Maus Cheiros

Um mau cheiro de código (*code smell*) é uma indicação superficial que usualmente corresponde a um problema mais profundo em um software. Por si só, um *code smell* não é algo ruim, ocorre que frequentemente ele indica um problema mas não necessariamente é o problema em si [13]. O termo *code smell* foi cunhado pela primeira vez por Kent Beck enquanto ajudava Martin Fowler com o seu livro Refactoring [12, 13].

Maus cheiros são padrões de código que estão associados com um design ruim e más práticas de programação. Diferentemente de erros de código eles não resultam em comportamentos errôneos. Maus cheiros apontam para áreas na aplicação que podem se beneficiar de refatorações. [31]. Refatoração é definido por “uma técnica para reestruturação de um código existente, alterando sua estrutura interna sem alterar seu comportamento externo” [12].

Escolher não resolver um mau cheiro pela refatoração não resultará na aplicação falhar mas irá aumentar a dificuldade de mantê-la. Logo, a refatoração ajuda a melhorar a manutenibilidade de uma aplicação [31]. Uma vez que os custos com manutenção são a maior parte dos custos envolvidos no ciclo de desenvolvimento de software [30], aumentar a manutenibilidade através de refatoração irá reduzir os custos de um software no longo prazo.

2.3 Android

2.3.1 Arquitetura da Plataforma

Android é um sistema operacional de código aberto, baseado no kernel do Linux criado para um amplo conjunto de dispositivos. Para prover acesso aos recursos específicos dos dispositivos como câmera ou *bluetooth*, o Android possui uma camada de abstração de *hardware* (HAL do inglês *Hardware Abstraction Layer*) exposto aos desenvolvedores através de um arcabouço de interfaces de programação de aplicativos (APIs do inglês *Applications Programming Interface*) Java. Estes e outros elementos explicados a seguir podem ser visualizados na figura 2.1 [25].

Cada aplicativo é executado em um novo processo de sistema que contém sua própria instância do ambiente de execução Android. A partir da versão 5 (API nível 21), o ambiente de execução padrão é o Android Runtime (ART), antes desta versão era a Dalvik. ART foi escrita para executar múltiplas instâncias de máquina virtual em dispositivos com pouca memória. Suas funcionalidades incluem: duas forma de compilação, a frente do tempo (AOT do inglês *Ahead-of-time*) e apenas no momento (JIT do inglês *Just-in-time*), o coletor de lixo, ferramentas de depuração e um relatório de diagnósticos de erros e exceções.

Muitos dos componentes e serviços básicos do Android, como ART e HAL, foram criados a partir de código nativo que depende de bibliotecas nativas escritas em C e C++. A plataforma Android provê arcabouços de APIs Java para expôr as funcionalidade de algumas destas bibliotecas nativas para os aplicativos. Por exemplo, OpenGL ES pode ser acessado através do arcabouço Android Java OpenGL API, de forma a adicionar suporte ao desenho e manipulação de gráficos 2D e 3D no aplicativo.

Todo as funcionalidades da plataforma Android estão disponíveis para os aplicativos através de APIs Java. Estas APIs compõem os elementos básicos para a construção de aplicativos Android. Dentre eles, os mais relevantes para esta dissertação são:

- Um rico e extensível **Sistema de Visualização** para a contrução de interfaces com o usuário, também chamadas de arquivos de *layout*, do aplicativo. Incluindo listas, grades ou tabelas, caixas de textos, botões, dentre outros.
- Um **Gerenciador de Recursos**, provendo acesso aos recursos “não-java” como textos, elementos gráficos, arquivos de *layout*.
- Um **Gerenciador de Activity** que gerencia o ciclo de vida dos aplicativos e provê

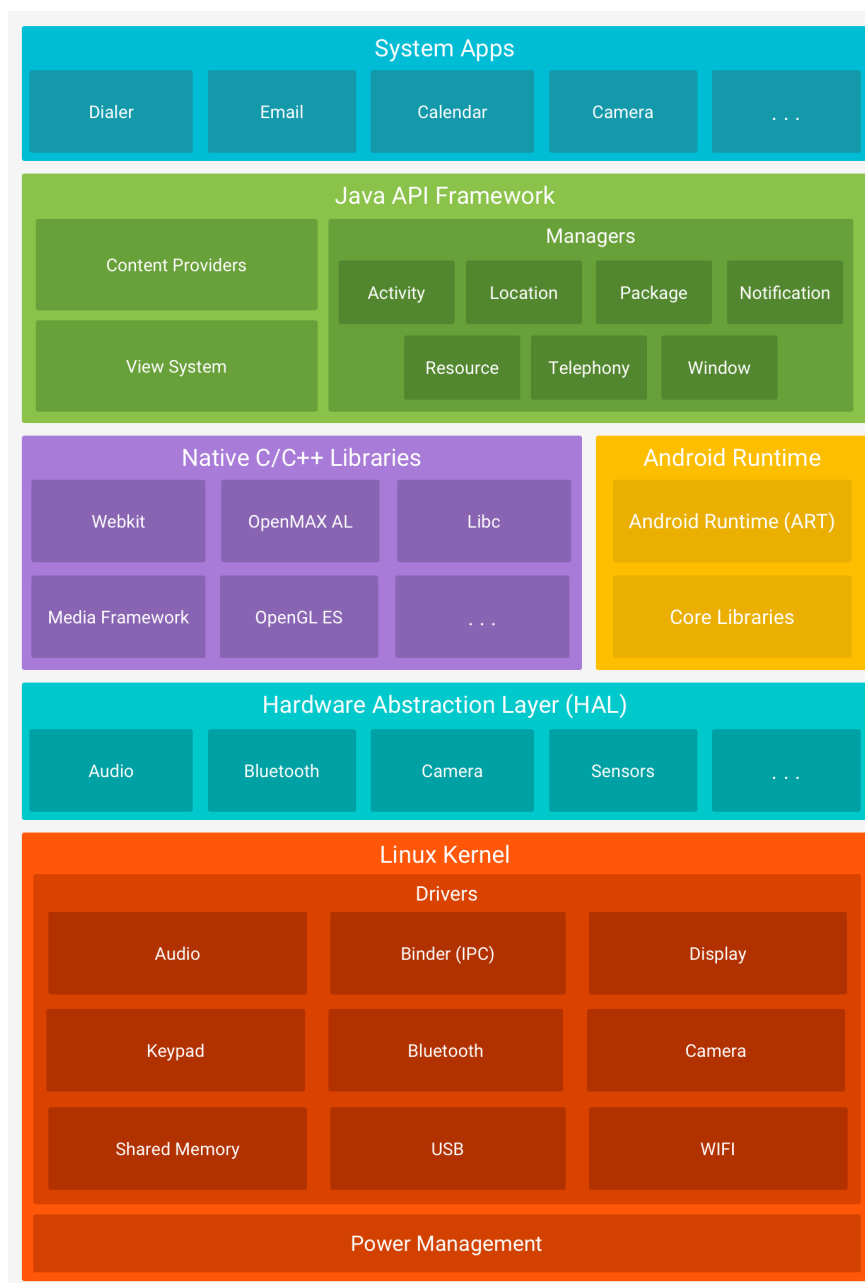


Figura 2.1: *Arquitetura do sistema operacional Android.*

uma navegação comum.

O Android já vem com um conjunto de aplicativos básicos como por exemplo, para envio e recebimento de SMS, calendário, navegador, contatos e outros. Estes aplicativos vindos com a plataforma não possuem nenhum diferencial com relação aos aplicativos de terceiros. Todo aplicativo tem acesso ao mesmo arcabouço de APIs do Android, seja ele aplicativo da

plataforma ou de terceiro. Desta forma, um aplicativo de terceiro pode se tornar o aplicativo padrão para navegar na internet, receber e enviar SMS e assim por diante.

Aplicativos da plataforma provem capacidades básicas que aplicativos de terceiros podem reutilizar. Por exemplo, se um aplicativo de terceiro quer possibilitar o envio de SMS, o mesmo pode redirecionar esta funcionalidade de forma a abrir o aplicativo de SMS já existente, ao invés de implementar por si só.

2.3.2 Fundamentos do Desenvolvimento Android

Aplicativos Android são escritos na linguagem de programação Java. O Kit para Desenvolvimento de Software (*Software Development Kit*) Android compila o código, junto com qualquer arquivo de recurso ou dados, em um arquivo Android Package (APK). Um APK, arquivo com extensão `.apk`, é usado por dispositivos para a instalação de um aplicativo [24].

Componentes Android são os elementos base para a construção de aplicativos Android. Cada componente é um diferente ponto através do qual o sistema pode acionar o aplicativo. Nem todos os componentes são pontos de entrada para o usuário e alguns são dependentes entre si, mas cada qual existe de forma autônoma e desempenha um papel específico.

Existem quatro tipos diferentes de componentes Android. Cada tipo serve um propósito distinto e tem diferentes ciclos de vida, que definem como o componente é criado e destruído. Os quatro componentes são:

- **Activities**

Uma *activity* representa uma tela com uma interface de usuário. Por exemplo, um aplicativo de email pode ter uma *activity* para mostrar a lista de emails, outra para redigir um email, outra para ler emails e assim por diante. Embora *activities* trabalhem juntas de forma a criar uma experiência de usuário (UX do inglês *User Experience*) coesa no aplicativo de emails, cada uma é independente da outra. Desta forma, um aplicativo diferente poderia iniciar qualquer uma destas *activities* (se o aplicativo de emails permitir). Por exemplo, a *activity* de redigir email no aplicativo de emails, poderia solicitar o aplicativo câmera, de forma a permitir o compartilhamento de alguma foto. Uma *activity* é implementada como uma subclasse de `Activity`.

- **Services**

Um *service* é um componente que é executado em plano de fundo para processar

operações de longa duração ou processar operações remotas. Um *service* não provê uma interface com o usuário. Por exemplo, um *service* pode tocar uma música em plano de fundo enquanto o usuário está usando um aplicativo diferente, ou ele pode buscar dados em um servidor remoto através da internet sem bloquear as interações do usuário com a *activity*. Outros componente, como uma *activity*, podem iniciar um *service* e deixá-lo executar em plano de fundo. É possível interagir com um *service* durante sua execução. Um *service* é implementado como uma subclasse de `Service`.

- **Content Providers**

Um *content provider* gerencia um conjunto compartilhado de dados do aplicativo. Estes dados podem estar armazenados em arquivos de sistema, banco de dados SQLite, servidor remoto ou qualquer outro local de armazenamento que o aplicativo possa acessar. Através de *content providers*, outros aplicativos podem consultar ou modificar (se o *content provider* permitir) os dados. Por exemplo, a plataforma Android disponibiliza um *content provider* que gerencia as informações dos contatos dos usuários. Desta forma, qualquer aplicativo, com as devidas permissões, pode consultar parte do *content provider* (como `ContactsContract.Data`) para ler e escrever informações sobre um contato específico. Um *content provider* é implementado como uma subclasse de `ContentProvider`.

- **Broadcast Receivers**

Um *broadcast receiver* é um componente que responde a mensagens enviadas pelo sistema. Muitas destas mensagens são originadas da plataforma Android, por exemplo, o desligamento da tela, baixo nível de bateria e assim por diante. Aplicativos de terceiros também podem enviar mensagens, por exemplo, informando que alguma operação foi concluída. No entanto, *broadcast receivers* não possuem interface de usuário. Para informar o usuário que algo ocorreu, *broadcast receivers* podem criar notificações. Um *broadcast receiver* é implementado como uma subclasse de `BroadcastReceiver`.

Antes de a plataforma Android poder iniciar qualquer um dos componente supramencionados, a plataforma precisa saber que eles existem. Isso é feito através da leitura do arquivo `AndroidManifest.xml` do aplicativo (arquivo de manifesto). Este arquivo deve estar no diretório raiz do projeto do aplicativo e deve conter a declaração de todos os seus componentes.

O arquivo de manifesto é um arquivo XML e pode conter muitas outras informações além das declarações dos componentes do aplicativo, por exemplo:

- Identificar qualquer permissão de usuário requerida pelo aplicativo, como acesso a internet, acesso a informações de contatos do usuário e assim por diante.
- Declarar o nível mínimo do Android requerido para o aplicativo, baseado em quais APIs são usadas pelo aplicativo.
- Declarar quais funcionalidades de sistema ou *hardware* são usadas ou requeridas pelo aplicativo, por exemplo câmera, *bluetooth* e assim por diante.
- Declarar outras APIs que são necessárias para uso do aplicativo (além do arcabouço de APIs do Android), como a biblioteca do Google Maps.

Os elementos usados no arquivo de manifesto são definidos pelo vocabulário XML do Android. Por exemplo, uma *activity* pode ser declarada conforme o *listing* 2.1.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest ... >
3     <application android:icon="@drawable/app_icon.png" ... >
4         <activity android:name="com.example.project.ExampleActivity"
5                 android:label="@string/example_label" ... >
6         </activity>
7     ...
8 </application>
9 </manifest>

```

Listing 2.1: Arquivo *AndroidManifest.xml*

No elemento `<application>` o atributo `android:icon` aponta para o ícone, que é um recurso, que identifica o aplicativo. No elemento `<activity>`, o atributo `android:name` especifica o nome completamente qualificado da *Activity*, que é uma classe que estende de *Activity*, e por fim, o atributo `android:label` especifica um texto para ser usado como título da *Activity*.

Para declarar cada um dos quatro tipos de componentes, deve-se usar os elementos a seguir:

- `<activity>` elemento para *activities*.

- `<service>` elemento para *services*.
- `<receiver>` elemento para *broadcast receivers*.
- `<provider>` elemento para *content providers*.

2.3.3 Recursos do Aplicativo

Um aplicativo Android é composto por outros arquivos além de código Java, ele requer **recursos** como imagens, arquivos de áudio, e qualquer recurso relativo a apresentação visual do aplicativo [24]. Também é possível definir animações, menus, estilos, cores e arquivos de *layout* das *activities*. Recursos costumam ser arquivos XML que usam o vocabulário definido pelo Android.

Um dos aspectos mais importantes de prover recursos separados do código-fonte é a habilidade de prover recursos alternativos para diferentes configurações de dispositivos como por exemplo idioma ou tamanho de tela. Este aspecto se torna mais importante conforme mais dispositivos são lançados com configurações diferentes. Segundo levantamento, em 2015 foram encontrados mais de 24 mil dispositivos diferentes com Android [4].

De forma a prover compatibilidade com diferentes configurações, deve-se organizar os recursos dentro do diretório `res` do projeto, usando sub-diretórios que agrupam os recursos por tipo e configuração. Para qualquer tipo de recurso, pode-se especificar uma opção padrão e outras alternativas.

- **Recursos padrões** são aqueles que devem ser usados independente de qualquer configuração ou quando não há um recurso alternativo que atenda a configuração atual. Por exemplo, arquivos de *layout* padrão ficam em `res/layout`.
- **Recursos alternativos** são todos aqueles que foram desenhados para atender a uma configuração específica. Para especificar que um grupo de recursos é para ser usado em determinada configuração, basta adicionar um qualificador ao nome do diretório. Por exemplo, arquivos de *layout* para quando o dispositivo está em posição de paisagem ficam em `res/layout-land`.

O Android irá aplicar automaticamente o recurso apropriado através da identificação da configuração corrente do dispositivo com os recursos disponíveis no aplicativo. Por exemplo, o recurso do tipo *strings* pode conter textos usados nas interfaces do aplicativo. É possível

traduzir estes textos em diferentes idiomas e salvá-los em arquivos separados. Desta forma, baseado no qualificador de idioma usado no nome do diretório deste tipo de recurso (por exemplo `res/values-fr` para o idioma francês) e a configuração de idioma do dispositivo, o Android aplica o conjunto de *strings* mais apropriado.

A seguir são listados os tipos de recursos que podem ser utilizados no Android [27]. Para cada tipo de recurso existe um conjunto de qualificadores que podem ser usados para prover recursos alternativos:

- **Recursos de animações** Definem animações pré-determinadas. Ficam nos diretórios `res/anim` ou `res/animATOR`.
- **Recursos de lista de cores de estado** Definem recursos de cores que alteram baseado no estado da *View*. Ficam no diretório `res/color`.
- **Recursos de desenhos** Definem recursos gráficos como *bitmap* ou XML. Ficam no diretório `res/drawable`.
- **Recursos de layouts** Definem a parte visual da interface com o usuário. Ficam no diretório `res/layout`.
- **Recursos de menus** Definem os conteúdos dos menus da aplicação. Ficam no diretório `res/menu`.
- **Recursos de textos** Definem textos, conjunto de textos e plurais. Ficam no diretório `res/values`.
- **Recursos de estilos** Definem os estilos e e formatos para os elementos da interface com usuário. Ficam no diretório `res/values`.
- **Outros recursos** Ainda existem outros recursos como inteiros, *booleanos*, dimensões, dentre outros. Ficam no diretório `res/values`.

2.3.4 Interfaces de Usuários

Arquivos de layout são recursos localizados na pasta `res/layout` que possuem a extensão `.xml`.

Todos os elementos de UI (Interface de Usuário, do inglês UI, *User Interface*) de um aplicativo Android são construídos usando objetos do tipo `View` ou `ViewGroup`. Uma `View`

é um objeto que desenha algo na tela do qual o usuário pode interagir. Um `ViewGroup` é um objeto que agrupa outras `Views` e `ViewGroups` de forma a desenhá-las o layout da interface [28].

A UI para cada componente do aplicativo é definida usando uma hierarquia de objetos `View` e `ViewGroup`, como mostrado na figura 2.2. Cada `ViewGroup` é um container invisível que organiza `Views` filhas, enquanto as `Views` filhas são caixas de texto, botões e outros componentes visuais que compoem a UI. Esta árvore hierárquica pode ser tão simples ou complexa quanto se precisar, mas quanto mais simples melhor o desempenho.

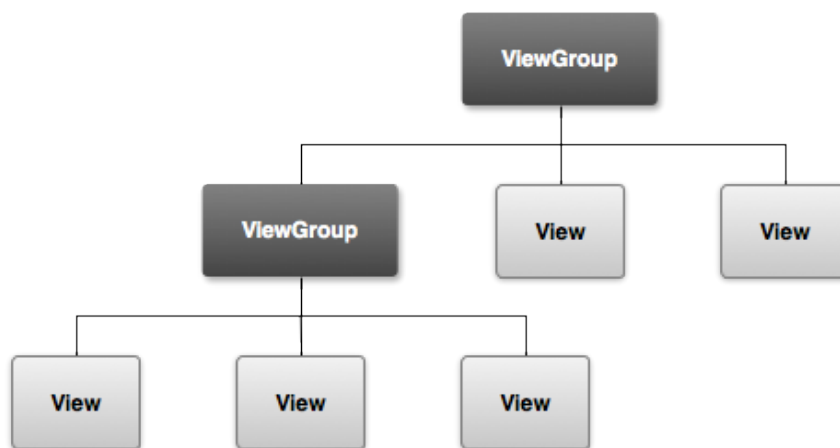


Figura 2.2: *Árvore hierárquica de Views e ViewGroups do Android.*

É possível criar um layout programaticamente, instanciando `Views` e `ViewGroups` no código e construir a árvore hierárquica manualmente, no entanto, a forma mais simples e efetiva de definir um layout é através de um XML de layout. O XML de layout oferece uma estrutura legível aos olhos humanos, similar ao HTML, podendo ser utilizados elementos aninhados.

O vocabulário XML para declarar elementos de UI segue a estrutura de nome de classes e métodos, onde os nomes dos elementos correspondem aos nomes das classes e os atributos correspondem aos nomes dos métodos. De fato, a correspondência frequentemente é tão direta que é possível adivinhar qual atributo XML corresponde a qual método de classe, ou adivinhar qual a classe correspondente para determinado elemento. No entanto, algumas classes possuem pequenas diferenças como por exemplo, o elemento `<EditText>` tem o atributo `text` que corresponde ao método `EditText.setText()`.

Um layout vertical simples com uma caixa de texto e um botão se parece com o código no listing 2.2.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout ...
3     android:layout_width="fill_parent"
4     android:layout_height="fill_parent"
5     android:orientation="vertical">
6
7     <TextView android:id="@+id/text"
8         android:layout_width="wrap_content"
9         android:layout_height="wrap_content"
10        android:text="I am a TextView" />
11
12    <Button android:id="@+id/button"
13        android:layout_width="wrap_content"
14        android:layout_height="wrap_content"
15        android:text="I am a Button" />
16
17 </LinearLayout>

```

Listing 2.2: *Arquivo exemplo de layout.*

Quando um recurso de layout é carregado pelo aplicativo, o Android inicializa um objeto para cada elemento do layout, desta forma é possível recuperá-lo programaticamente para definir comportamentos, modificar o layout ou mesmo recuperar o estado.

O Android provê uma série de elementos de UI comuns pré-prontos como: caixa de texto, botão, lista suspensa, dentre muitos outros. Desta forma, o desenvolvedor não precisa implementar do zero estes elementos básicos através de Views e ViewGroups para escrever uma interface de usuário.

Cada subclasse de ViewGroup provê uma forma única de exibir o conteúdo dentro dele. Por exemplo, o `LinearLayout` organiza seu conteúdo de forma linear horizontalmente, um ao lado do outro, ou verticalmente, um abaixo do outro. O `RelativeLayout` permite especificar a posição de uma View relativa ao posicionamento de alguma outra [26].

Quando o conteúdo é dinâmico ou não pré-determinado, como por exemplo uma lista de dados, pode-se usar um elemento que estende de `AdapterView` para popular o layout em momento de execução. Subclasses de `AdapterView` usam uma implementação de `Adapter` para carregar dados em seu layout. `Adapters` agem como um intermediador entre o conteúdo a ser exibido e o layout, ele recupera o conteúdo e converte cada item, de uma lista por exemplo, dentro de uma ou mais Views.

Os elementos comumente usados para situações de conteúdo dinâmico ou não pré-determinado são: `ListView` e `GridView`. Para fazer o carregamento dos dados nestes elementos, o Android provê alguns `Adapters` como por exemplo o `ArrayAdapter` que a partir de um array de dados popula os dados na `ListView` ou `GridView`.

2.3.5 Eventos de Interface

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific `View` object that the user interacts with. The `View` class provides the means to do so.

Within the various `View` classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a `View` (such as a `Button`) is touched, the `onTouchEvent()` method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every `View` object in order to handle such an event would not be practical. This is why the `View` class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called event listeners, are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a `View` class, in order to build a custom component. Perhaps you want to extend the `Button` class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class event handlers.

Capítulo 3

Camada de Apresentação Android

Em nossas pesquisas bibliográficas não foi encontrada uma definição única e formal de quais elementos compõem a camada de apresentação de um aplicativo Android. Sendo este um tema essencial para o entendimento deste trabalho, nesta seção temos por objetivo explicar o que foi considerado e porquê. Para definir os elementos da camada de apresentação para aplicativos Android partimos das origens de interfaces de usuário (GUI, *Graphical User Interfaces*).

Os primórdios de GUI foram em 1973 com o projeto Alto, desenvolvido pelos pesquisadores da Xerox Palo Alto Research Center (PARC), seguido do projeto Lisa da Apple em 1979. Estes dois projetos serviram de base e inspiração para o Macintosh, lançado pela Apple em 1985. As primeiras definições sobre GUI que surgiram nessa época abordavam sobre componentes de uso comum como ícones, janelas, barras de rolagem, menus suspensos, botões, caixas de entrada de texto; gerenciadores de janelas; arquivos de áudio, internacionalização e eventos. Antes deste período existiam apenas interfaces de linha de comando [20, 29].

Outra fonte define camada de apresentação como “informações gráficas, textuais e auditivas apresentadas ao utilizador, e as sequências de controle (como comandos de teclado, *mouse* ou toque) para interagir com o programa” [32].

Para efeito desta dissertação, definimos que todos os elementos do Android que são apresentados ou interagem com o usuário de alguma forma auditiva, visual ou por comando de voz ou toque são elementos da **Camada de Apresentação**, são eles:

- **Activities e Fragments** Representam uma tela ou um fragmento de tela. A exemplo temos classes Java que herdam de `Activity`, `Fragment` ou classes similares.

- **Listeners** Meio pelo qual os comandos do usuário são capturados pelo aplicativo. A exemplo temos classes Java que implementam interfaces como `View.OnClickListener`.
- **Recursos do Aplicativo** Arquivos que apresentam textos, imagens, áudios, menus, interfaces gráficas (*layout*), dentre outros. Estão incluídos neste item todos os arquivos dentro do diretório `res` ainda que em seu formato Java. A exemplo podemos citar classes que herdam da classe `View` ou `ViewGroup`.
- **Adapters** Meio pelo qual são carregados conteúdos dinâmicos ou não pré-determinados na tela. A exemplo podemos citar classes que herdam da classe `BaseAdapter`.

Capítulo 4

Trabalhos Relacionados

Diversas pesquisas em torno de maus cheiros de código vem sendo realizadas ao longo dos últimos anos. Já existem inclusive diversos maus cheiros mapeados, porém poucos deles são específicos da plataforma Android [17]. Segundo Hecht [15] estudos sobre maus cheiros de código sobre aplicações Android ainda estão em sua infância. Outro ponto que reafirma esta questão são os trabalhos de Linares-Vásquez [16] e Hecht [15] onde concluem que, em projetos Android, é mais comum maus cheiros específicos do que maus cheiros orientados a objetos.

O trabalho de Verloop [31] avalia a presença de maus cheiros definidos por Fowler [12] e Minelli e Lanza [19] em projetos Android. Apesar das relevantes contribuições feitas, a conclusão sobre a incidência de tais maus cheiros não é plenamente conclusiva, visto que dos 6 maus cheiros analisados (*Large Class*, *Long Method*, *Long Parameter List*, *Type Checking*, *Feature Envy* e *Dead Code*) apenas dois deles, *Long Method* e *Type Checking*, se apresentam com maior destaque (duas vezes mais provável) em projetos Android. Os demais apresentam uma diferença mínima em classes Android quando se comparados a classes não específicas do Android. Por fim, acaba por não ser conclusivo quanto a maior relevância deles em Android ou não.

Desta forma, Verloop [31] conclui com algumas recomendações de refatoração de forma a mitigar a presença do mau cheiro *Long Method*. Estas recomendações são o uso do atualmente já reconhecido padrão *ViewHolder* em classes do tipo *Adapters*. Ele também sugere um *ActivityViewHolder* de forma a extrair código do método `onCreate` e deixá-lo menor. Sugere também o uso do atributo `onClick` em XMLs de `LAYOUT` e `MENU`.

Diferentemente de validar a presença de maus cheiros previamente catalogados conforme feito por Verloop [31], esta dissertação objetiva identificar, validar e catalogar, com base na

experiência de desenvolvedores, boas e más práticas específicas à camada de apresentação de projetos Android.

Outro trabalho muito relevante realizado neste tema é o de Reimann et al. [21] que, baseado na documentação do Android, documenta 30 *quality smells* específicos para Android. No texto *quality smells* são definidos como “*uma determinada estrutura em um modelo, indicando que influencia negativamente a requisitos específicos de qualidade, que podem ser resolvidos por refatorações particulares ao modelo*” (tradução livre). Estes requisitos de qualidade são centrados no usuários (estabilidade, tempo de início, conformidade com usuário, experiência do usuário e acessibilidade), consumo inteligente de recursos (eficiência geral e no uso de energia e memória) e segurança.

Esta dissertação se difere do trabalho Reimann et al. [21] pois pretende-se encontrar maus cheiros em termos de qualidade de código, ou seja, que influenciam na legibilidade e manutenabilidade do código do projeto.

Capítulo 5

Método de Pesquisa

5.1 Definição dos Maus Cheiros

Para a definição dos maus cheiros será realizada uma pesquisa descritiva [18] de forma a obter boas e más práticas utilizadas ou evitadas, respectivamente, por desenvolvedores no dia-a-dia do desenvolvimento de aplicativos Android.

A seguir são detalhadas a forma de coleta de dados, critério de seleção dos respondentes, análise e validação dos dados.

5.1.1 Coleta de Dados

Pretende-se coletar, através de um questionário, boas e más práticas utilizadas ou percebidas por desenvolvedores Android. Para cada elemento da camada de apresentação, serão feitas as seguintes perguntas (“X” indica o elemento em pauta na questão):

- Você conhece ou utiliza alguma boa prática para lidar com X?
- Você considera algo uma má prática para lidar com X?

Ainda, ao final são inseridas as seguintes perguntas abertas de forma a capturar algum último pensamento sobre boas e más práticas com relação a camada de apresentação.

- Existe alguma outra *BOA* prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?
- Existe alguma outra *MÁ* prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?

Este questionário será aberto para a comunidade de desenvolvimento Android do Brasil e exterior. Para tanto, será elaborado versões em inglês e português do questionário.

5.1.2 Análise dos Dados

Os dados serão analisados manualmente e categorizados. O objetivo é identificar maus cheiros e possíveis soluções.

5.1.3 Validação com Especialistas

Os maus cheiros definidos no passo anterior serão validados com desenvolvedores Android especialistas através de uma entrevista conduzida. A fim de ter um controle maior com relação a credibilidade de conhecimento, estes especialistas serão pessoas que são referência na comunidade Android, já escreveram livros sobre o tema ou que trabalhem com Android no Google.

5.2 Percepção dos Desenvolvedores

5.2.1 Coleta de Dados

Após a definição de um catálogo de maus cheiros, pretende-se validar se desenvolvedores os percebem de fato como indicativos de trechos de códigos ruins. Para isso pretende-se conduzir um experimento apresentando aos desenvolvedores códigos com e sem os maus cheiros propostos, para cada código, será solicitado que ele indique o código como bom ou ruim.

5.2.2 Análise dos Dados

5.3 Impacto na Tendência a Mudanças

Capítulo 6

Catálogo de Maus Cheiros

6.1 Code Smell 1

A fazer.

Capítulo 7

Conclusão

A fazer.

7.1 Principais contribuições

A fazer.

7.2 Trabalhos futuros

A fazer.

Apêndice A

Questionário sobre Boas e Más Práticas

Boas e más práticas na Camada de Apresentação de aplicativos Android nativos.

Esta pesquisa é focada em boas e más práticas na camada de apresentação de aplicativos Android nativos. Nosso objetivo é melhorar a qualidade no desenvolvimento de aplicativos Android. Ajude-nos a identificar quais são estas boas e más práticas. Este questionário deve durar em torno de 10 a 15 minutos e compreendo que aparentemente parece muito tempo, mas perceberá que é pouco para um material que poderá lhe ser útil num futuro. Obrigada!

Um pouco sobre você:

1. Qual sua posição profissional atual?
2. Anos de experiência com desenvolvimento de software? (1-10+)
3. Anos de experiência com desenvolvimento de software com Java? (1-10+)
4. Anos de experiência com desenvolvimento de aplicativos Android nativo? (1-10+)
5. Quantos aplicativos Android que foram para produção você colaborou?
6. Com qual frequência você estuda/lê sobre boas práticas de desenvolvimento de aplicativos Android nativo? (1-5)

Sobre as boas e más práticas.

1. Activities

- Quais **BOAS** práticas você costuma utilizar ao lidar com Activities?
- O que você considera **MÁS** práticas ao lidar com Activities?

2. Fragments

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com Fragments?
- O que você considera ****MÁS**** práticas ao lidar com Fragments?

3. Adapters

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com Adapters?
- O que você considera ****MÁS**** práticas ao lidar com Adapters?

4. Listeners

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com Listeners?
- O que você considera ****MÁS**** práticas ao lidar com Listeners?

5. Recursos de Layouts

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com recursos de Layouts?
- O que você considera ****MÁS**** práticas ao lidar com recursos de Layouts?

6. Recursos de Menu

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com recursos de Menus?
- O que você considera ****MÁS**** práticas ao lidar com recursos de Menus?

7. Recursos Gráficos

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com recursos Gráficos?
- O que você considera ****MÁS**** práticas ao lidar com recursos Gráficos?

8. Recursos de Animação

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com recursos de Animação?
- O que você considera ****MÁS**** práticas ao lidar com recursos de Animação?

9. Recursos de Estilos

- Quais ****BOAS**** práticas você costuma utilizar ao lidar com recursos de Estilos?

- O que você considera **MÁS** práticas ao lidar com recursos de Estilos?

10. Recursos de Cor

- Quais **BOAS** práticas você costuma utilizar ao lidar com recursos de Cor?
- O que você considera **MÁS** práticas ao lidar com recursos de Cor?

11. Recursos de Textos

- Quais **BOAS** práticas você costuma utilizar ao lidar com recursos de Textos?
- O que você considera **MÁS** práticas ao lidar com recursos de Textos?

12. Recursos de Inteiros, Booleanos, Dimensões ou Arrays

- Quais **BOAS** práticas você costuma utilizar ao lidar com recursos de Inteiros, Booleanos, Dimensões ou Arrays?
- O que você considera **MÁS** práticas ao lidar com recursos de Inteiros, Booleanos, Dimensões ou Arrays?

Últimos Pensamentos

1. Existe alguma outra **BOA** prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?
2. Existe alguma outra **MÁ** prática na camada de apresentação que nós não perguntamos a você ou que você ainda não mencionou?
3. Podemos entrar em contato para outras etapas desta pesquisa se for necessário?
4. Deixe seu email para participar de futuras pesquisas necessárias ainda para a conclusão do trabalho ou mesmo para receber os resultados desta pesquisa.

Referências Bibliográficas

- [1] Activities. <https://developer.android.com/guide/components/activities.html>. Last accessed at 29/08/2016. 2
- [2] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 1
- [3] Google android software spreading to cars, watches, tv. <http://phys.org/news/2014-06-google-android-software-cars-tv.html>, June 2014. Last accessed at 26/07/2016. 1
- [4] Android fragmentation visualized. <http://opensignal.com/reports/2015/08/android-fragmentation/>, August 2015. Last accessed at 12/09/2016. 12
- [5] Ford terá apple carplay e android auto em todos os modelos nos eua. <http://g1.globo.com/carros/noticia/2016/07/ford-tera-apple-carplay-e-android-auto-em-todos-os-modelos-nos-eua.html>, 2016. Last accessed at 26/07/2016. 1
- [6] Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016. <http://www.gartner.com/newsroom/id/3323017>, May 2016. Last accessed at 23/07/2016. 1
- [7] Number of available applications in the google play store from december 2009 to february 2016. <http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>, 2016. Last accessed at 24/07/2016. 1
- [8] Worldwide smartphone growth forecast to slow to 3.1% in 2016 as focus shifts to device lifecycles, according to idc. <http://www.idc.com/getdoc.jsp?containerId=prUS41425416>, June 2016. Last accessed at 23/07/2016. 1

- [9] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Core Series. Prentice Hall PTR, 2003. 1
- [10] Maurício Aniche, Bavota G., Treude C., Van Deursen A., and Gerosa M. A validated set of smells in model-view-controller architectures. 2016. 3
- [11] Maurício Aniche, Marco Gerosa, São Paulo, Bullet INPE, Bullet Sant, and Anna ufba. Architectural roles in code metric assessment and code smell detection. 2016. Regras Arquiteturais na Avaliação de Matrizes de Código e Detecção de Maus Cheiros Regras Arquiteturais na Avaliação de Matrizes de Código e Detecção de Maus Cheiros. 1, 2, 4
- [12] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. 6, 19
- [13] Martin Fowler. Code smell. <http://martinfowler.com/bliki/CodeSmell.html>, February 2006. 6
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994. 1
- [15] Geoffrey Hecht. An approach to detect android antipatterns. page 766–768, 2015. 1, 19
- [16] Mario Linares-Vásquez, Sam Klock, Collin Mcmillan, Aminata Saban, Denys Poshyvanyk, and Yann-Guillaume Guéhéneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 19
- [17] Umme Mannan, Danny Dig, Iftexhar Ahmed, Carlos Jensen, Rana Abdullah, and M Al-murshed. Understanding code smells in android applications. 2, 19
- [18] Adriana B. Manzato, Antonio J. e Santos. A elaboração de questionários na pesquisa quantitativa. http://www.inf.ufsc.br/~vera.carmo/Ensino_2012_1/ELABORACAO_QUESTIONARIOS_PESQUISA_QUANTITATIVA.pdf. Last accessed at 12/09/2016. 21
- [19] Roberto Minelli and Michele Lanza. Software analytics for mobile applications, insights & lessons learned. *In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013. 19

- [20] Eric Steven Raymond. *The Art of Unix Usability*. 2004. Last accessed at 26/10/2016. 17
- [21] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2013. 2, 20
- [22] A.J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley Publishing Company, 1996. 2
- [23] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012. 1
- [24] Android Developer Site. Android fundamentals. <https://developer.android.com/guide/components/fundamentals.html>. Last accessed at 04/09/2016. 9, 12
- [25] Android Developer Site. Platform architecture. <https://developer.android.com/guide/platform/index.html>. Last accessed at 04/09/2016. 7
- [26] Android Developer Site. Layouts. <https://developer.android.com/guide/topics/ui/declaring-layout.html>, 2016. Last accessed at 23/10/2016. 15
- [27] Android Developer Site. Resource type. <https://developer.android.com/guide/topics/resources/available-resources.html>, 2016. Last accessed at 12/09/2016. 13
- [28] Android Developer Site. Ui overview. <https://developer.android.com/guide/topics/ui/overview.html>, 2016. Last accessed at 23/10/2016. 13
- [29] TecMundo. A história da interface gráfica. <http://www.tecmundo.com.br/historia/9528-a-historia-da-interface-grafica.htm>. Last accessed at 26/10/2016. 17
- [30] Nikolaos Tsantalis. *Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings*. PhD thesis, University of Macedonia, August 2010. 6
- [31] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 2, 6, 19
- [32] Wikipedia. Interface do utilizador. https://pt.wikipedia.org/wiki/Interface_do_utilizador. Last accessed at 26/10/2016. 17