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 móvel mais usado atualmente, com 83% do mercado mundial e mais de 2 milhões de aplicativos disponíveis na loja oficial. Desenvolvedores de software frequentemente precisam 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 (*code smells*). Apesar de já existirem diversas anomalias 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 ou, 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 grandes distinções quando se comparado a projetos tradicionais. Em outros trabalhos sobre maus cheiros Android, foram identificados maus cheiros relacionados a segurança, consumo inteligente de recursos ou que de alguma maneira influênciam a experiência ou expectativa do usuário. Diferentemente deles, nossa proposta é catalogar maus cheiros Android que influênciem a qualidade do código. Com isso os desenvolvedores terão mais uma ferramenta 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 maintanence, code metrics.

Sumário

Lista de Abreviaturas

SDK *Software Development Kit*

IDE *Integrated Development Environment*

APK *Android Package*

ART *Android RunTime*

Lista de Símbolos

Σ Sistema de transição de estados

Lista de Figuras

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 [6]. Mais de 83,5% dos *devices* no mundo usam o sistema operacional móvel Android (plataforma Android), e esse percentual vem crescendo ano após ano [5, 7]. Atualmente é possível encontrá-lo também em outros dispositivos como *smart TVs*, *smartphones*, carros, dentre outros [3, 4].

Desenvolvedores de software constantemente escrevem código fácil de ser mantido e evoluído e detectam trechos de código problemáticos. Para a primeira, desenvolvedores comumente buscam se apoiar em boas práticas e *design patterns* já estabelecidos [8, 13, 20]. Para a segunda, é comum se utilizar de estratégias de detecção de maus cheiros de código (*code smells*) [10], 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 *code smells*, 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 a camada de apresentação, onde ele apresenta suas maiores distinções. Conforme relatado por [14] com relação a projetos Android, "*antipatterns* específicos ao Android são mais comuns e ocorrem mais frequentemente do que *antippaterns* OO" (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 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 estes arquivos costumam ser poucos e pequenos. Conforme o projeto evolui, a quan-

tidade e complexidade dos recursos tende a aumentar, trazendo problemas em encontrá-las, reaproveitá-las e entendê-las. 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 são classes que possuem muitas responsabilidades [24], sempre estão vinculadas a um LAYOUT e normalmente precisam de acesso a classes do modelo da aplicação. Analogamente ao padrão MVC, ACTIVITIES fazem os papéis de VIEW e CONTROLLER simultâneamente. Isto posto, é razoável considerar que o *code smell God Class* [19] é aplicável neste caso, no entanto, conforme bem pontuado por [10] "*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 componente específico da plataforma Android, responsáveis pela apresentação e interações do usuário com a tela [1].

Na prática, desenvolvedores Android percebem estes problemas diariamente. Muitos deles já se utilizam de práticas para solucioná-los, conforme relatado pelo [18] "o problema no desenvolvimento móvel é que desenvolvedores estão cientes sobre maus cheiros apenas indiretamente porque a estas definições [dos maus cheiros] são informais (boas práticas, relatórios de problemas, fóruns de discussões, etc) a recursos onde encontrá-los estão distribuídos pela internet" (tradução livre). Ou seja, não é encontrado atualmente um catálogo destas 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 [16]. Apesar do rápido crescimento da plataforma Android, pesquisas em torno dela não veem sendo realizadas na mesma proporção e a ausência de um catálogo de maus cheiros Android resulta em (i) uma carência de conhecimento sobre boas e más práticas a ser compatilhado 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 ímpacto destas más práticas na manutenabilidade do código de projetos Android. Por este motivos, boas e más práticas que são específicos a uma plataforma, no caso Android, tem emergido como tópicos de pesquisa sobre manutenção de código [9].

1.1 Questões de Pesquisa

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

Q1: O que representa a *Camada de Apresentação* em um projeto Android?

Esta é uma pegunta importante a ser respondida, visto que esta dissertação está gira em torno deste tema. No entanto, não foi encontrada na documentação oficial do Android (site developer.android.com) uma resposta direta. Desta forma, partimos por um estudo teórico da documentação oficial para extrair a definição a ser usada nesta dissertação. O estudo teórico e definição são detalhados na seção 4.1 do capítulo 4.

Q2: O que desenvolvedores consideram boas e más práticas com relação a *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 isso elaboramos um questionário para coletar boas e más práticas utilizadas ou percebidas por desenvolvedores Android com relação a camada de apresentação.

Estes dados serão analisados de forma a resultar numa lista de maus cheiros. Esta listagem será validada com desenvolvedores Android especialistas, dos quais, alguns compõem times que lidam com Android no Google. Pretende-se aplicar o questionário na comunidade de desenvolvedores Android do Brasil e exterior. Estas etapas são detalhadas na seção 4.2 do capítulo 4.

Q3: 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. 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 4.3 do capítulo 4.

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 a 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 *Code Smells*

  Neste capítulo, serão catalogados os *code smells* validados.

- **Capítulo 6** Conclusão

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

Capítulo 2

Trabalhos Relacionados

Diversas pesquisas em torno de maus cheiros de código veem sendo realizadas ao longo dos últimos anos. Já existem inclusive diversos maus cheiros mapeados, porém poucos deles são específicos da plataforma Android [16]. Segundo [14] estudos sobre maus cheiros de código sobre aplicações Android ainda estão em sua infância. Outro ponto que reafirma esta questão são os trabalhos [15] e [14] concluem que é mais comum em projetos Android maus cheiros específicos do que os tradicionais maus cheiros orientados a objetos.

O trabalho [24] avalia a presença de *code smells* definidos por [11] e [17] em projetos Android. Apesar das relevantes contribuições feitas, a conclusão sobre a incidência de tais *code smells*, não é plenamente conclusiva, visto que dos 6 *code smells* 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, [24] conclui com algumas recomendações de refatoração de forma a mitigar a presença do *code smell 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.

Esta dissertação se difere do trabalho [24] pois pretende-se identificar, validar e catalogar, com base na experiência de desenvolvedores, quais boas e más práticas, ao longo da última década, eles vem praticando ou evitando, respectivamente em projetos Android. Ou seja, não pretende-se a partir de um catálogo já estabelecido de *code smells* e sim catalogar *code*

smells específicos da camada de apresentação Android.

Outro trabalho muito relevante realizado neste tema é o [18] que, baseado na documentação do Android, documenta 30 *quality smells* específicos Android. No texto *quality smells* são definidos como *"A quality smell is a certain structure in a model, indicating that it negatively influences specific quality requirements, which can be resolved by particular model refactorings"*. 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, no uso de energia e meméria) e segurança.

Esta dissertação se difere do trabalho [18] pois pretende-se encontrar *code smells* em termos de qualidade de código, ou seja, que influênciem na legibilidade e manutenabilidade do código do projeto.

Capítulo 3

Fundamentação Teórica

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

3.1 Qualidade de Código

3.2 Code Smells

Mau cheiro de código é uma indicação superficial que usualmente corresponde a um problema mais profundo em um software. Por si só um *code smell*, seu termo em inglês, não é algo ruim, ocorre que frequentemente ele indica um problema mas não necessáriamente é o problema em si [12]. O termo em inglês *code smell* foi cunhado pela primeira vez por Kent Beck enquanto ajudava Martin Fowler com o seu livro Refactoring [11] [12].

Code Smells 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 erroneos. *Code Smells* apontam para áreas na aplicação que podem se beneficiar de refatorações. [24]. Refatoração é definido por "uma técnica para reestruturação de um código existente, alterando sua estrutura interna sem alterar seu comportamento externo" [11].

Escolher não resolver *code smells* pela refatoração não resultará na aplicação falhar mas irá aumentar a dificuldade de mantê-la. Logo, a refatoração ajuda a melhorar a manutenabilidade de uma aplicação [24]. Uma vez que os custos com manutenção são a maior parte dos custos envolvidos no ciclo de desenvolvimento de software [23], aumentar a manutenabilidade através de refatoração irá reduzir os custos de um software no longo prazo.

3.3 Android

3.3.1 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 3.1 [22].

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 multiplas 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, 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

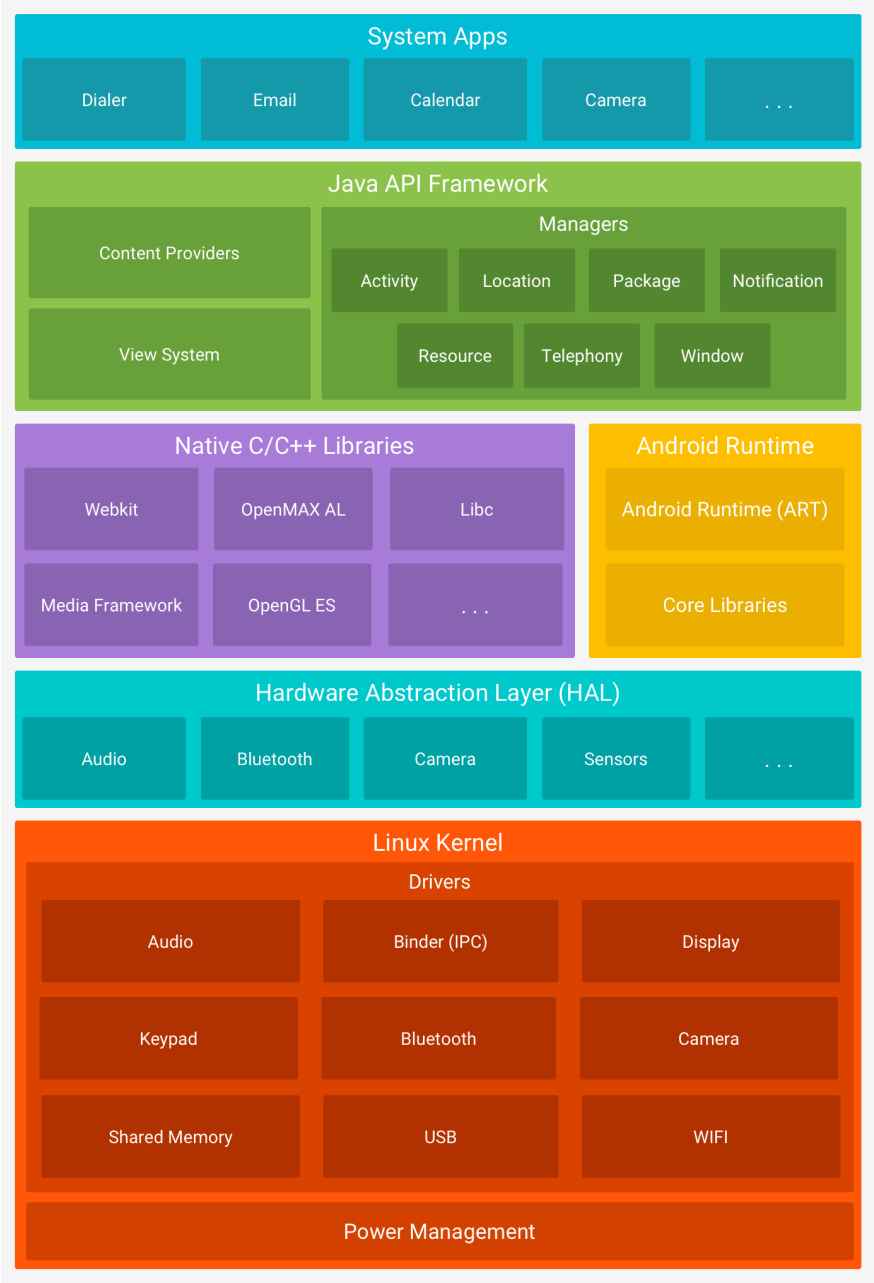

Figura 3.1: *Arquitetura do sistema operacional Android.*

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.

3.3.2 Aplicativos Android

Aplicativos Android são escritos na linguagem de programação Java. O Kit para Desenvolvimento de Software (SDK do inglês *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 [21].

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 componente 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 supremencionados, 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 rais 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* 3.1.

```
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 3.1: *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 da classe completamente qualificado de uma subclasse de `Activity` e o atributo `android:label` especifica um texto para ser usado como título da atividade.

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

- `<activity>` elemento para atividades.

- `<service>` elemento para serviços.
- `<receiver>` elemento para recebedor de mensagens.
- `<provider>` elemento para provedores de conteúdo.

3.3.3 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 [21]. 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. 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. Po 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

Default resources are those that should be used regardless of the device configuration or when there are no alternative resources that match the current configuration. Alternative resources are those that you've designed for use with a specific configuration. To specify that a group of resources are for a specific configuration, append an appropriate configuration qualifier to the directory name.

For example, while your default UI layout is saved in the res/layout/ directory, you might specify a different layout to be used when the screen is in landscape orientation, by saving it in the res/layout-land/ directory. Android automatically applies the appropriate resources by matching the device's current configuration to your resource directory names.

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 frânces) e a configuração de idioma do usuário, o Android aplica o conjunto de *strings* mais apropriado.

Capítulo 4

Pesquisa

4.1 Camada de Apresentação Android

4.2 Definição dos Maus Cheiros

4.2.1 Coleta de Dados

4.2.2 Análise dos Dados

4.2.3 Validação com Especialistas

4.3 Percepção dos Desenvolvedores

Capítulo 5

Catálogo de Maus Cheiros

5.1 Code Smell 1

A fazer.

Capítulo 6

Conclusão

A fazer.

6.1 Principais contribuições

A fazer.

6.2 Trabalhos futuros

A fazer.

Apêndice A

XYZ

A.1 Apêndice 1

A fazer.

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] 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

[5] 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

[6] 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

[7] 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

[8] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Core Series. Prentice Hall PTR, 2003. 1

[9] MaurÃŋcio Aniche, Bavota G., Treude C., Van Deursen A., and Gerosa M. A validated set of smells in model-view-controller architectures. 2016. 2

[10] MaurÃŋcio Aniche, Marco Gerosa, SÃčo Paulo, Bullet INPE, Bullet Sant, and Anna ufba. Architectural roles in code metric assessment and code smell detection. 2016. Regras Arquiteturais na AvaliaÃğÃčo de MÃľtricas de CÃşdigo e DetecÃğÃčo de Maus Cheiros Regras Arquiteturais na AvaliaÃğÃčo de MÃľtricas de CÃşdigo e DetecÃğÃčo de Maus Cheiros. 1, 2

[11] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. 5, 7

[12] Martin Fowler. Code smell. http://martinfowler.com/bliki/CodeSmell.html, February 2006. 7

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994. 1

[14] Geoffrey Hecht. An approach to detect android antipatterns. page 766âĂŞ768, 2015. 1, 5

[15] Mario Linares-VÃąsquez, Sam Klock, Collin Mcmillan, Aminata SabanÃľ, Denys Poshyvanyk, and Yann-GaÃńl GuÃľhÃľneuc. Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. 5

[16] Umme Mannan, Danny Dig, Iftekhar Ahmed, Carlos Jensen, Rana Abdullah, and M Almurshed. Understanding code smells in android applications. 2, 5

[17] 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. 5

[18] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. 2013. 2, 6

[19] A.J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley Publishing Company, 1996. 2

[20] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2012. 1

[21] Android Developer Site. Andriod fundamentals. https://developer.android.com/guide/components/fundamentals.html. Last accessed at 04/09/2016. 10, 13

[22] Android Developer Site. Plataform architecture. https://developer.android.com/guide/platform/index.html. Last accessed at 04/09/2016. 8

[23] 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. 7

[24] Daniël Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013. 2, 5, 7