Um Estudo Exploratório da Arquitetura e Projeto de Aplicativos Android

EDMILSON CAMPOS, Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte
UIRÁ KULESZA, Universidade Federal do Rio Grande do Norte
ROBERTA COELHO, Universidade Federal do Rio Grande do Norte

Este artigo apresenta um estudo exploratório de análise de arquiteturas de aplicações desenvolvidas para a plataforma Android. Ao todo foram analisados código de seis aplicativos *opensource*, selecionados por amostragem entre os mais populares da loja oficial de aplicativos Android, a Google Play. A metodologia do trabalho consistiu em extrair o código dos aplicativos diretamente dos seus repositórios e aplicar técnicas de engenharia de reversa para investigar a arquitetura de cada um deles. O objetivo principal do estudo foi investigar as decisões comuns adotadas por desenvolvedores para a modularização da arquitetura de aplicações Android. O artigo apresenta análises qualitativas realizadas sobre as arquiteturas extraídas, com base em três critérios predefinidos no estudo: (i) análise arquitetural; (ii) utilização de padrões de projeto gerais ou específicos da plataforma Android; e (iii) política de tratamento de exceções adotada.

Category and subject descriptors: **E.2.2 [Software Arquitetures]**: User Interfaces—*Evaluation/methodology*.

General Terms: Design Pattern, Android.

Additional Key Words and Phrases: Architecture, language.

ACM Reference Format:

Campos, E.; Kulesza, U. e Coelho, R. 2014. Um Estudo Exploratório da Arquitetura e Projeto de Aplicativos Android.

1. INTRODUÇÃO

Os recentes avanços na área de telecomunicações têm possibilitado um aumento exponencial na venda de *smartphones* e *tablets* nos últimos anos (Carneiro, Roman, & Fagundez, 2014). Segundo dados recentes da *International Data Corporation* (IDC), empresa global especializada em análise do mercado tecnológico, publicados pela Revista Exame (Caputo, 2014), apenas em 2013, foram vendidos mais de 1 bilhão de *smartphones* em todo o mundo. A pesquisa aponta ainda que 78,6% desse aparelhos comercializados utilizavam o sistema operacional (SO) Android, o que representa 793,6 milhões de unidades vendidas em um ano, um aumento de 58,7%, se comparado as vendas do mesmo SO no ano de 2012.

Em consequência disso, é crescente também a demanda por consumo e desenvolvimento de aplicações para dispositivos móveis. A popularidade é tanta que o número de aplicativos disponíveis somente na *Google Play*, loja de aplicativos do sistema operacional com maior participação no mercado, já ultrapassa a marca de 1 milhão de aplicativos em 2014 (AppBrain, 2014). Contudo, apesar do crescente número de aplicativos desenvolvidos diariamente, há uma carência de estudos sobre a arquitetura de implementação de tais sistemas. Alguns trabalhos recentes investigam aspectos de implementação de aplicações Android, contudo ainda são incipientes pesquisas relacionadas a arquitetura e projeto dessas aplicações. Os trabalhos de Ruiz et. al (2012) e Mojica et al (2013) são alguns dos poucos que analisaram um aspecto arquitetural dessas aplicações, contudo focando-se apenas na questão do reuso de código. Eles analisaram aplicações diretamente da loja de aplicativos, a partir da extração de modelos do código fonte já compilado, e então quantificaram e dividiram as aplicações em duas dimensões: a do reuso por herança e por classe.

Neste contexto, este trabalho propõe a realização de um estudo exploratório com o objetivo de investigar e analisar arquiteturas de aplicações Android e de que forma vem sendo implementadas. O processo de seleção de tais aplicações, buscou selecionar por amostragem aplicações populares disponíveis na *Google Play*. Foram investigadas ao todo seis aplicações populares que possuem código-fonte aberto. Um procedimento de engenharia reversa foi aplicado em cada aplicativo selecionado para investigar a sua estrutura arquitetural. O código de tais aplicativos foi então analisado para extrair seus principais componentes e mecanismos de comunicação, com objetivo de identificar padrões de projetos e políticas de tratamentos de exceções comuns à plataforma Android.

O restante deste artigo está organizado da seguinte forma. A Seção 2 expõe detalhes sobre a metodologia do estudo exploratório realizado para análise das arquiteturas das aplicações Android. A Seção 3 apresenta os principais resultados do estudo realizado e alguns discussões. Por fim, a Seção 4 traz algumas considerações finais e perspectivas de trabalhos futuros.

2. ESTUDO EXPLORATÓRIO DE ANÁLISE DE ARQUITETURAS DE APLICAÇÕES ANDROID

Esta seção apresenta os detalhes da metodologia adotada no estudo exploratório realizado. Nosso estudo envolveu a análise de aplicativos de código aberto desenvolvidos para a plataforma Android, para fins de analisar padrões arquiteturais e de projeto que vêm sendo adotados pelo desenvolvedores dessas aplicações.

2.1 Objetivos do Estudo

O objetivo deste estudo exploratório foi identificar as soluções arquiteturais e padrões que vêm sendo utilizados pelos principais desenvolvedores da plataforma Android, a partir da análise de repositórios de código aberto. Em especial, o estudo busca responder as seguintes questões de pesquisa:

QP1: Qual a arquitetura que os aplicativos Android vêm sendo estruturados atualmente?
QP2: Que padrões de projeto vêm sendo adotados nessas arquiteturas?
QP2.1: Têm se adotado algum padrão específico para a plataforma Android?
QP3: Que política de tratamento de exceções vêm sendo adotadas em aplicações Android?

Para responder tais questões, foram definidos alguns critérios de análise (Seção 2.4) e aplicados os procedimentos de análise (Seção 2.3) em aplicações de domínio real selecionadas (Seção 2.2).

2.2 Aplicações selecionadas

Um dos requisitos durante a seleção de aplicações para o nosso estudo é que elas precisavam, necessariamente, estar disponível em um repositório de código aberto para que possibilitasse a sua análise. A seleção de tais de aplicações foi realizada de forma manual e por amostragem das principais aplicações disponíveis da loja que, além de (i) possuir código aberto, (ii) fosse uma aplicação popular, com um mínimo de 1 milhão de downloads na loja oficial de aplicativos; e/ou (iii) tivesse caráter profissional, ou seja, aplicações oficiais desenvolvidas como soluções empresariais.

A Tabela 1 apresenta uma relação com todas as aplicações que foram selecionadas para este estudo, apresentando o nome de cada aplicação, uma curta descrição, a categoria pertencente, o número de instalações realizadas e o desenvolvedor de cada uma delas. Todos os dados foram extraídos diretamente da loja oficial de aplicativos da plataforma Android.

Tabela 1 Listagem de aplicações selecionadas

#	NOME DA APLICAÇÃO	DESCRIÇÃO	CATEGORIA	Nº. DE DOWNLOADS	EMPRESA
01	Wikipedia móvel	App official da enciclopédia virtual livre Wikipedia, permite acesso online e off-line aos diversos artigos da enciclopédia	Livros e Referências	entre 10 e 50 milhões	Wikimedia Foundation
02	Wordpress App	Cliente oficial do Wordpress para Android, permite gerenciar páginas hospedadas no site, além de escrever, editar e publicar posts	Social	entre 1 e 5 milhões	Automattic, Inc
03	iFixit: Repair Manual	Aplicação com diversos guias e tutorais embutidos com orientações para consertos de variados modelos e especificações de aparelhos	Livros e Referências	entre 500 mil e 1 milhão	Dozuki
04	c:geo	Um cliente não oficial do site geocaching, que permite acessar caches de mapas ao vivo e se comunicar com aps externas, tais como Radar, Google Maps, etc.	Entretenimento	entre 1 e 5 milhões	c:geo team
05	ZapZap	Versão brasileira e gratuita do famoso comunicador de mensagens instantâneas WhatsApp	Comunicação	entre 500 mil e 1 milhão	Private host
06	Barcode Scanner	Popular leitor de códigos de barras para Android, permite também escanear Data Matrix e QR Codes contendo URLs, informação de contato, etc.	Compras	entre 100 e 500 milhões	ZXing Team

Os dados coletados foram retirados diretamente da loja de aplicativos oficiais da plataforma Android, a Google Play.

2.3 Procedimento de análise

Para realizar a análise do código e arquitetura das aplicações Android, os procedimentos a seguir foram realizados:

(i) *Extração de código dos repositórios*: Após a definição dos critérios de seleção e a escolha das aplicações, os respectivos repositórios de tais aplicações foram acessados para realizar uma cópia para a máquina local dos códigos de cada projeto;

(ii) *Extração da Arquitetura*: Por fim, após a análise qualitativa manual dos códigos, utilizamos o *plugin* Eclipse AmaterasUML (Project Amateras, s.d.) para realizar a engenharia reversa e extrair automaticamente diagramas de classes que pudessem representar as arquiteturas de cada aplicação. Em seguida, tais diagramas foram refinados manualmente, de maneira a sintetizar e representar simplificadamente a arquitetura dessas aplicações, possibilitando análises mais precisas sobre as mesmas;

(iii) *Análise das aplicações*: A análise da arquitetura e projeto das aplicações selecionadas baseou-se em critérios objetivos (Seção 2.4) definidos para sistematizar a resolução das questões de pesquisa (Seção 2.1) do estudo. Esta etapa consistiu numa análise qualitativa da arquitetura das aplicações, baseada nos critérios definidos e realizada de forma manualmente, assim como a etapa anterior.

2.4 Critérios de Análise

Alguns critérios foram definidos para sistematizar a análise da arquitetura e projeto das aplicações Android. Tais critérios serviram também de base para análise de aspectos específicos a serem considerados durante a análise individual de cada aplicação selecionada. A seguir listamos estes critérios com seus respectivos pontos de análise.

(i) Critério 1: Arquitetura das Aplicações

- Representar graficamente a arquitetura das aplicações analisadas;
- Identificar a adoção de algum padrão arquitetural (MVC, Camadas, etc.);
- Identificar na arquitetura de que forma estão estruturadas e implementadas os principais componentes da arquitetura (GUI, *Controller*, DAOs, etc.).

(ii) Critério 2: Padrões de Projeto

- Identificar se as aplicações investigadas utilizaram alguma solução ou padrão de projeto específico da plataforma Android (quais?);
- Identificar se as aplicações utilizaram algum padrão de projeto tradicional (quais e com que finalidade?).

(iii) Critério 3: Tratamento de Exceções

- Identificar qual(is) política(s) de tratamento de exceções vem sendo adotada por tais aplicações (subsistema de lançamento e tratamento, tratamento local ou criação de classes específicas).

3. RESULTADOS DO ESTUDO

Esta seção apresenta os principais resultados do estudo, com ênfase nos artefatos arquiteturais extraídos de cada aplicação investigada e nas análises obtidas sobre o uso de padrões de projeto e tratamentos de exceções na plataforma Android.

3.1 Arquitetura das aplicações

A seguir são apresentados resumos descritivos das arquiteturas de cada uma das aplicações investigadas nesse artigo, enfatizando, em cada caso, a solução adotada e razão para tal.

a) Wikipédia Móvel

O aplicativo Wikipédia Móvel apresenta características para funcionar em várias plataformas a partir de um mesmo projeto. Seu código está organizado de tal modo a integrar implementações de diferentes versões da aplicação, para diferentes plataformas de sistema operacional móveis, num mesmo projeto. Cada

versão, incluindo a versão para a plataforma Android, que investigamos neste estudo, implementa apenas uma interface visual que aponta para os respectivos artefatos web (desenvolvidos em HTML) da versão em questão. O projeto utiliza um framework multi-plataforma conhecido como Cordova (Apache Cordova, 2012), que permite ao desenvolvedor implementar o aplicativo utilizando linguagens para web (HTML, CSS, JavaScript e similares) e o framework gera automaticamente os códigos paras as plataformas, apenas para redirecionar para o código web implementado. Devido a esta característica específica do projeto Wikipédia Móvel, observou-se poucas atualizações recentes nos arquivos relacionados diretamente a implementação Android. A Fig. 1 exemplifica um fragmento dessa arquitetura, com ênfase na subdivisão hierárquica dos pacotes do projeto. As atualizações se concentram com maior ênfase nos arquivos HTML e similares que estão concentrados no módulo/pasta "`assets`" de sua arquitetura. O componente *controller* (classe `ActivityMain`) principal da aplicação Android é responsável por carregar (método `loadUrl`) o "index.html" atualizado, conforme versão da plataforma executada.

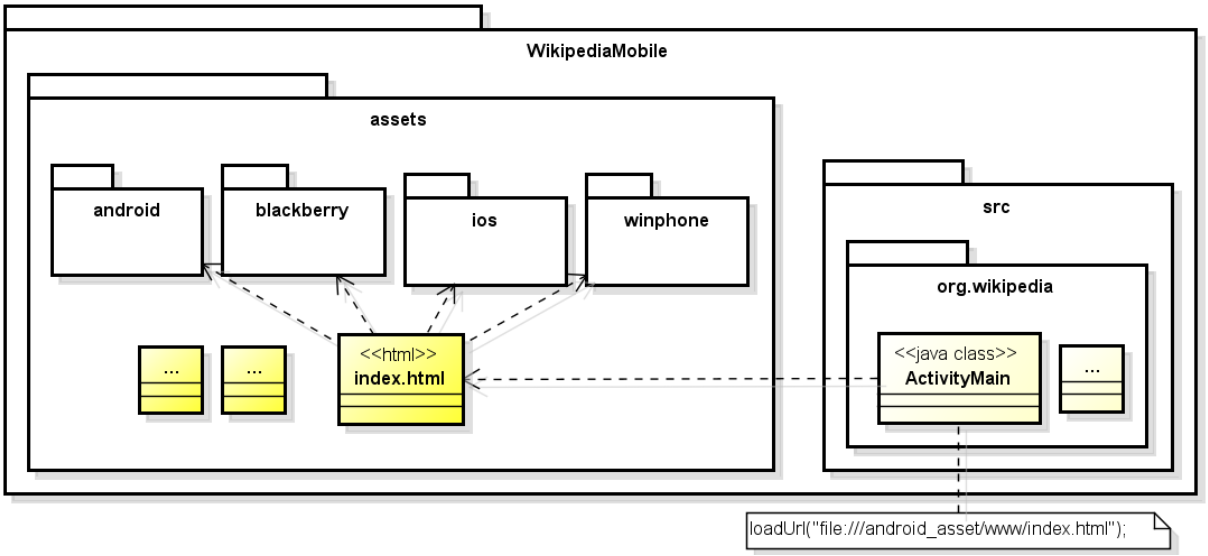

Fig. 1. Arquitetura da aplicação Wikipédia Móvel.

b) WordPress for Android

O aplicativo WordPress, por sua vez, já apresenta uma arquitetura mais conhecida (Fig. 2), organizada de acordo com o padrão arquitetural tradicional *Model-View-Controller* (MVC) (Fowler, 2003), com variações para se adequar à arquitetura da API Android. As classes de *Model* estão implementadas no pacote "`org.wordpress.android.model`", já as classes de *Controller*, no Android implementadas pelas classes do tipo *Activity*, no projeto WordPress estão agrupadas no pacote "`org.wordpress.android.ui`". Por fim, a camada de interface do usuário, ao invés de usar classes Java, são utilizados arquivos XML criados como *resources* e referenciados pelas respectivas *activities* que as controlarão. Por esta razão, tal camada não está representada no pacote "`src`", que contém apenas classes Java, e sim no diretório "`res`", específico para alocação de *resources* padrão em aplicações Android. Há ainda um pacote (`org.wordpress.android.datasets`) para os objetos de acesso a dados (*data access objects* – DAO), que permite separar as classes de persistência das demais classes do projeto, e uma camada para serialização dos objetos, por meio do protocolo XML-RPC, para chamada de procedimento remoto, com classes específicas para tratamento de exceções deste protocolo. Um pacote específico para classes utilitárias (`org.wordpress.android.util`) foi criado.

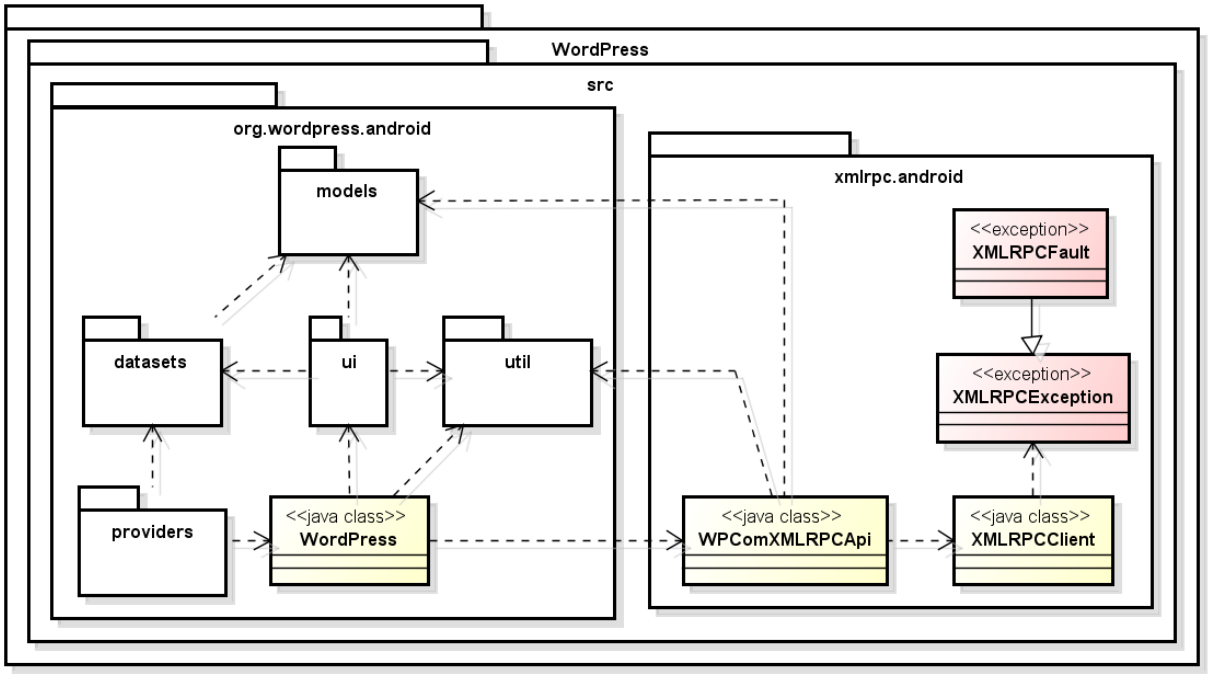

Fig. 2. Arquitetura da aplicação WordPress

c) iFixit: Repair Manual

A aplicação iFixit também adota uma arquitetura simular ao modelo MVC, com pacotes de classes separados para classes de modelos (`com.dozuki.ifixit.model`), classes de controladores (*Activity*, *Fragment*) e classes de interface com usuário (`com.dozuki.ifixit.ui`). De forma similar ao Wordpress, há um pacote só para classes com métodos utilitários (`com.dozuki.ifixit.util`), responsável, entre outras funcionalidades, por realizar o registro e *login* dos usuários do aplicativo. Observou-se neste pacote também algumas classes base de controladores de interface, que eram herdadas por outras classes do pacote de interface gráfica para controlar diferentes tipos de interfaces. A Fig. 3 apresenta um fragmento das principais partes desta arquitetura.

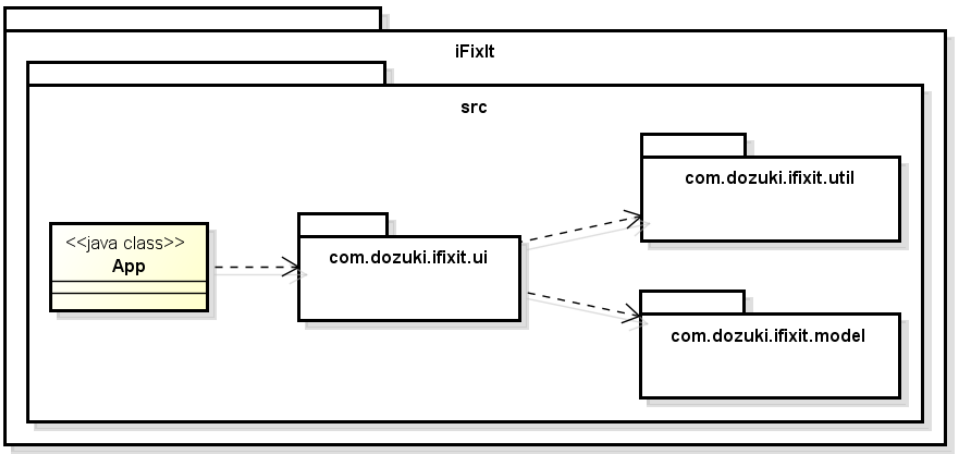

Fig. 3. Arquitetura da aplicação iFixIt

d) c:geo

O aplicativo c:geo possui uma das mais complexas arquiteturas estudadas, sobretudo pela própria complexidade da aplicação, que envolve processamento de mapas e integração com outras APIs externas. A Fig. 4 apresenta apenas um fragmento desta arquitetura, destacando algumas de suas principais camadas. A arquitetura em si está toda organizada em camadas, bem específicas do domínio e da aplicação. Por mais

que haja um pacote específico para interface gráfica (`ui`) e outro para classes do tipo *Activity*, as classes de *Controller* (*Activity* e *Fragment*) não estão concentradas em uma única camada e sim divididas conforme afinidade da regra de negócio. Nesse projeto, os pacotes de classes específicos dos componentes "`activity`" e "`ui`" servem apenas para implementar as classes abstratas que serão implementadas pelas demais *activities* do projeto. A camada "`connector`" é responsável por realizar a comunicação com serviço, através de um *login* do usuário e está diretamente relacionada à camada de configuração da aplicação, denominada "*settings*". Esta camada contém as classes que controlam a configuração base da aplicação, sobretudo utilizando a classe "`MapProviderFactory`", presente no pacote "`maps`", para integrar-se à diferentes APIs externas (*GoogleMaps*, *MapsForge*, etc.) que fornecem os mapas utilizados pela aplicação. Como nas demais aplicações analisadas, um pacote específico para classes com métodos utilitários está presente em sua arquitetura, para entre outras coisas, recuperar a versão da aplicação em execução. Outros pacotes, tais como integração com APIs sociais (`twitter`) e recuperação de dados dos sensores do aparelho (`sensor`), compõem a arquitetura desta aplicação. Não observou-se nesta arquitetura, pacote específicos para classes de acesso a banco de dados.

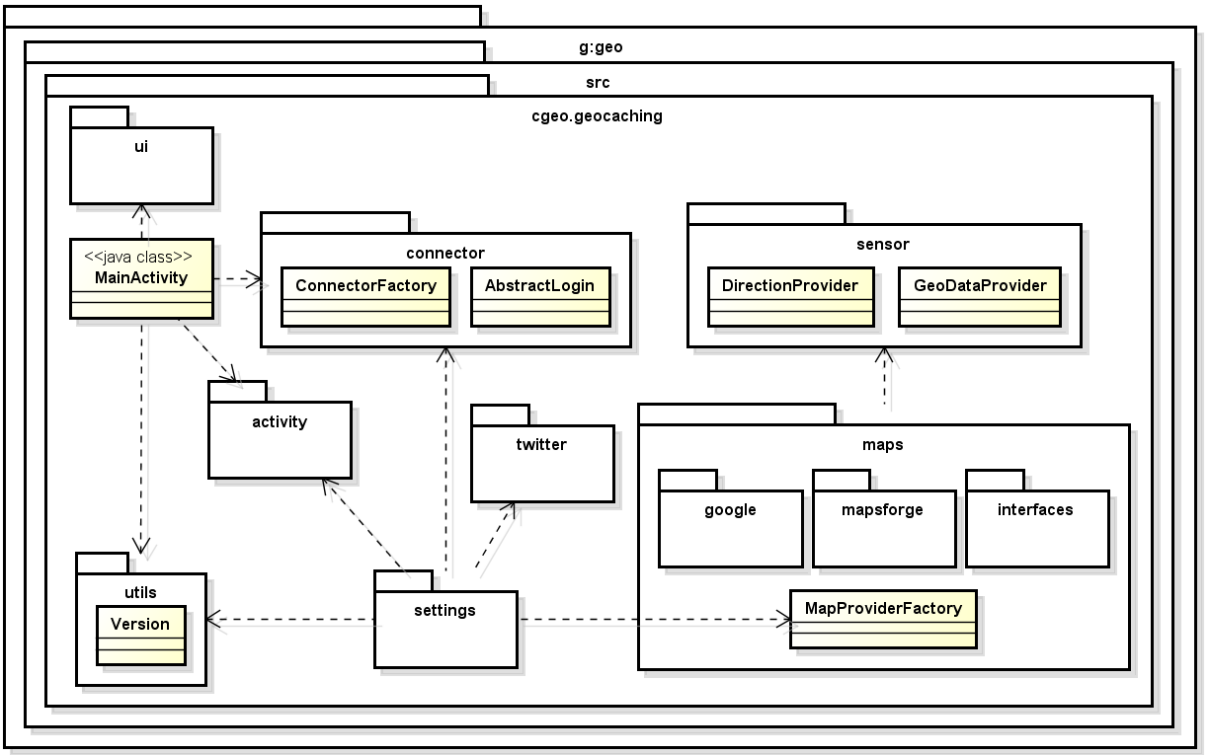

Fig. 4. Arquitetura da aplicação c:geo

e) ZapZap

A arquitetura da aplicação ZapZap utiliza a base de um outro projeto *opensource*, o *telegram*. Sua arquitetura manteve elementos originais desta arquitetura base com adaptações para implementação de alguns padrões de projetos (Fig. 5). De forma similar as demais aplicações, a arquitetura é organizada em camadas. Há uma camada para objetos de negócio (`objects`) e uma para classes controladoras de interface (`ui`). Uma camada equivalente ao pacote de classes utilitárias das demais aplicações analisadas, foi encontrada nesta arquitetura com um nome específico para a sua função, no caso o pacote "`phoneformat`", que possui classes utilitárias para auxiliar na formatação dos diferentes formatos de números de telefone. Há também um pacote específico para o protocolo de conexão, que é o pacote "`messenger`", e um pacote para as classes de acesso ao banco de dados (`SQLite`), com classes para tratamento das exceções específicas.

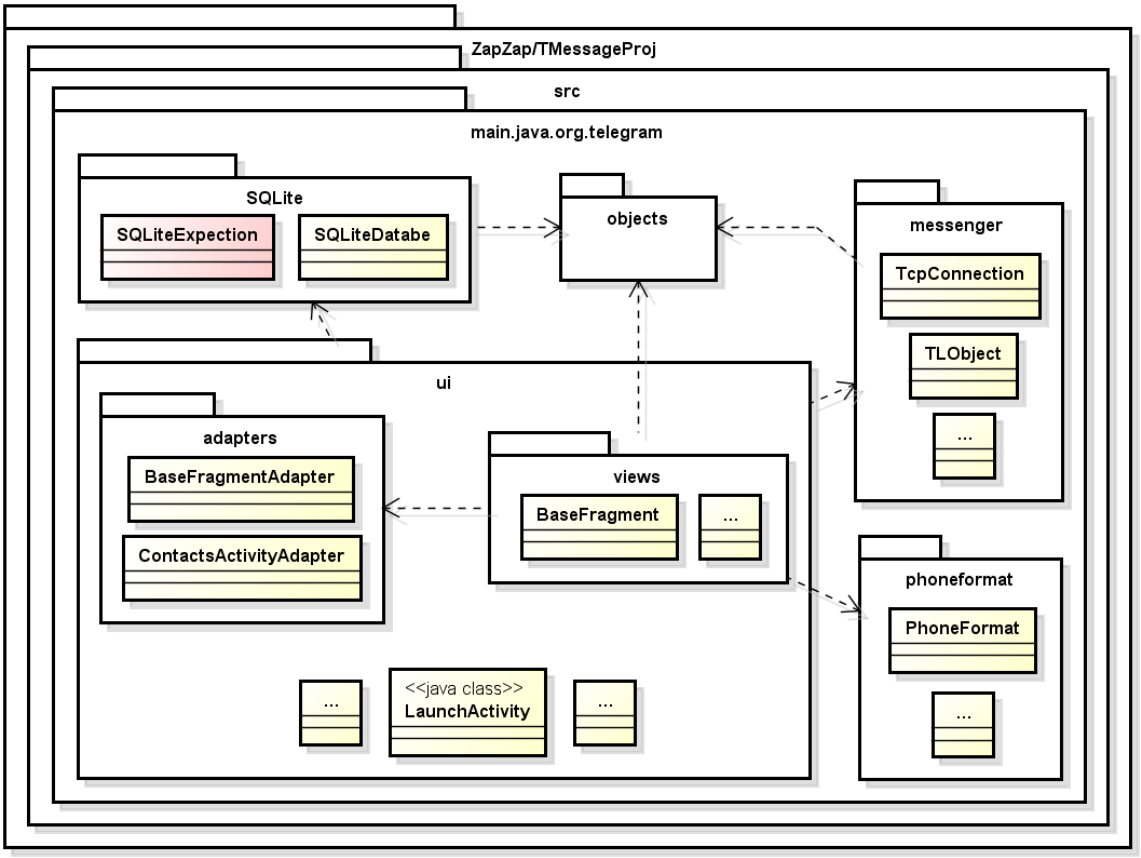

Fig. 5. Arquitetura da aplicação ZapZap

f) Barcode Scanner

A arquitetura da aplicação Barcode Scanner foi a que menos adotou padrões estruturais conhecidos dentre aquelas analisadas. Não foi possível identificar na arquitetura de tal aplicação, por exemplo, camadas específicas para objetos de modelo, de interface ou de acesso a dados. Classes de controladoras de interfaces Android (*activity* e *fragment*) não estão concentradas em uma camada e sim distribuídas em diferentes pacotes (`camera`, `book`, `wifi`, etc.), conforme afinidade de negócio. A Fig. 6 apresenta uma visão fragmentada dos pacotes que compõem a arquitetura desta aplicação.

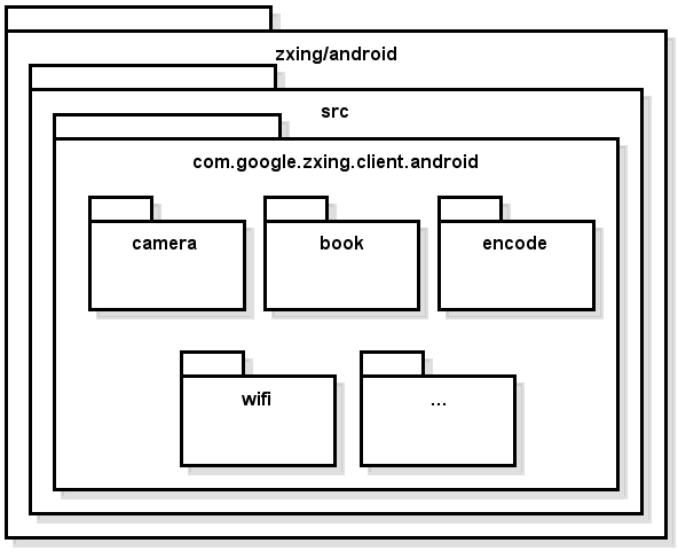

Fig. 6. Arquitura da aplicação Barcode Scanner

3.2 Padrões de projeto adotados

Nosso estudo também analisou o uso de padrões de projeto específicos da plataforma Android, além de outros padrões de projeto tradicionais, tais como, *Provider Model, Factory Method* e *Adapter* (Gamma, Helm, Johnson, & Vlissides, 1994). Para isso, dividimos e apresentamos a seguir os resultados desta seção em dois grupos: a) utilização de soluções específicas para a plataforma Android; e b) utilização de padrões de projetos tradicionais.

a) Padrões de implementação na plataforma Android

O Guia Oficial do Desenvolvedor Android (Android, s.d.) estabelece alguns padrões de implementação para gerenciar as principais variabilidades existentes na plataforma, tais como variações de: (i) idiomas, de (ii) telas (resolução e tamanho) e de (iii) versões da plataforma (Android, s.d.). Tais padrões de implementação permitem tratar um problema comum no desenvolvimento Android e propõem soluções cuja implementação permite adaptações conforme o cenário no qual é aplicado. Nosso estudo observou que a maioria dessas soluções foram de fato implementadas, mesmo que parcialmente, pelos desenvolvedores das aplicações analisadas. A seguir, são apresentados os resultados do nosso estudo em relação a adoção desses padrões em aspectos específicos.

(i) Gerenciamento de Idiomas: As recomendações oficiais da plataforma prevê o suporte a diferentes idiomas em aplicações Android, de acordo com o idioma padrão do aparelho cliente onde a aplicação for instalada. Para esse típico caso de variabilidade em aplicações Android, o guia do desenvolvedor prevê um padrão de estruturação de pastas no diretório "`res`" da aplicação, com diferentes arquivos XML em cada uma delas, com uma nomenclatura distinta e padronizada para que o sistema Android possa reconhecer automaticamente qual arquivo XML usar, conforme idioma padrão selecionado no aparelho cliente. Estes arquivos XML contém tags "`strings`" comuns a todos os arquivos, que são referenciadas nas telas da aplicação ou no código Java e substituídas em tempo de execução pelos valores alocados em seu conteúdo, conforme arquivo/idioma selecionado. Esta é uma solução típica da plataforma Android e de bastante aceitação pela comunidade, de forma que observou-se, neste estudo, a utilização desta solução em todas as aplicações analisadas. A Fig. 7 ilustra um exemplo da implementação deste padrão na aplicação iFixit, desde a estruturação e criação de pastas no diretórios "`res`" (Fig. 7.a e Fig. 7.b), em duas diferentes línguas (espanhol e francês) até a utilização dos *strings* criados para compor as telas da aplicação (Fig. 7.c).

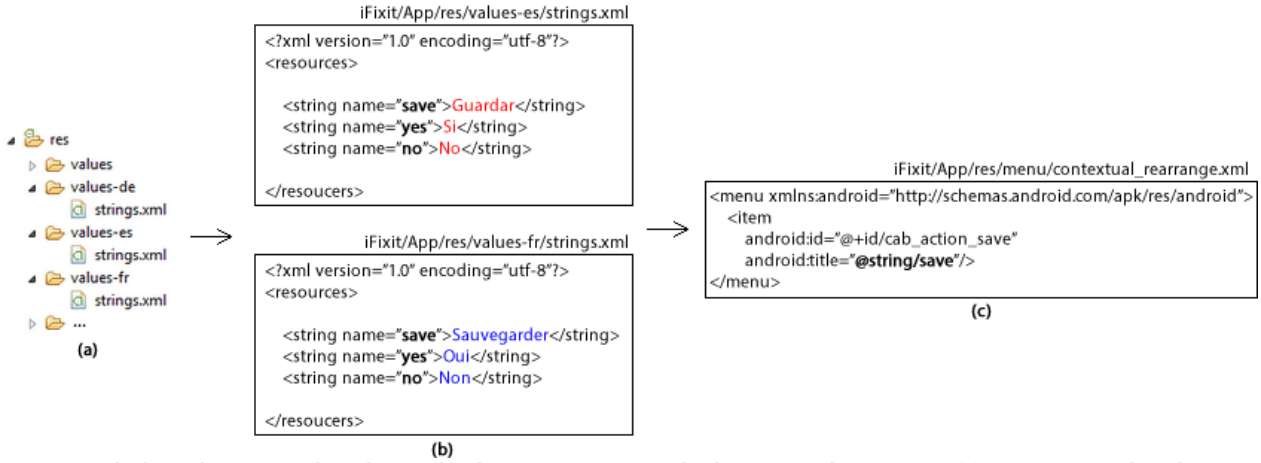

Fig. 7. Exemplo de implementação do padrão Android para gerenciamento de idiomas na aplicação iFixit: (a) organização padrão de pastas e arquivos XML no diretório res; (b) fragmento de dois arquivos XML com *strings*; e (c) exemplo de referência nas telas Android aos *strings* declarados nos arquivos de *resources*.

(ii) Gerenciamento de Diferentes Telas: O tratamento de diferentes telas em aplicações Android deve considerar basicamente duas características do aparelho: tamanho e densidade/resolução. Para estes cenários o guia oficial apresenta soluções de projeto baseadas, respectivamente, no uso de *resources* (similar a solução anterior para gerenciamento de idiomas) e no uso de arquivos XML fragmentados para compor as telas dos aplicativos (conhecidos como *fragments*). Esta última solução é conhecida também pela

comunidade Android como padrão *Multi-Pane Layouts*[1] e surgiu a partir da versão 3.0 do Android. Este padrão prevê a implementação das telas de um aplicativo em partes (fragmentos) de modo que possam ser reusados e combinados para montar uma visão composta, ajustada da melhor forma possível aos espaços disponíveis para visualização, conforme tamanho e orientação (horizontal ou vertical) do aparelho cliente. A Fig. 8 ilustra um exemplo com diferentes visões compostas a partir da implementação deste padrão no aplicativo *WordPress*.

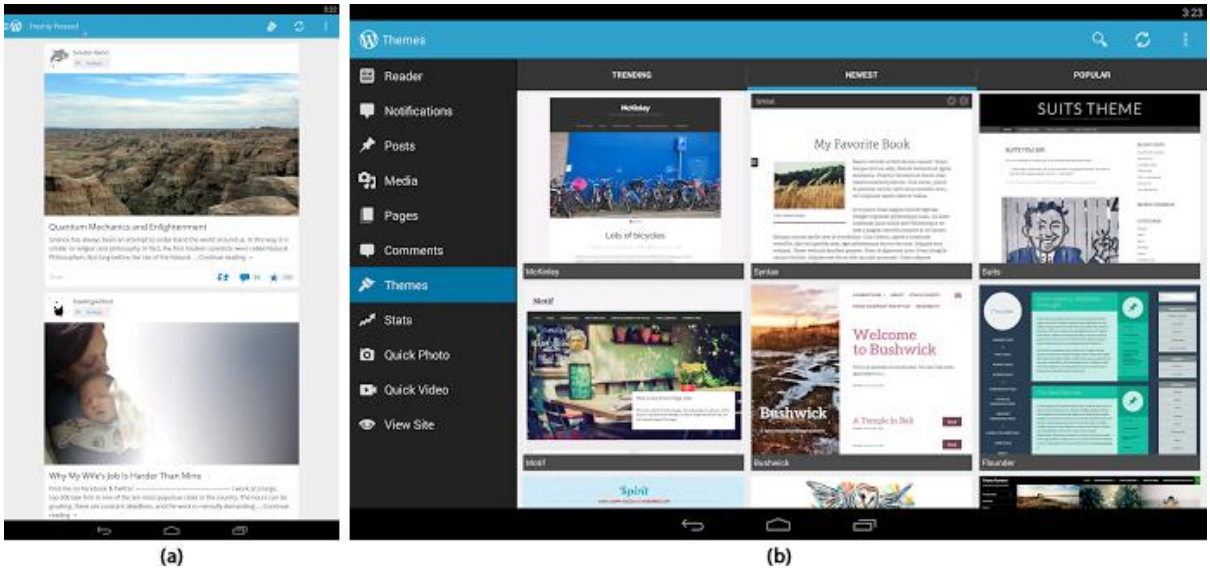

Fig. 8. Diferentes visões do aplicativo Wordpress: (a) orientação vertical; e (b) orientação horizontal.

Os principais papéis previstos pelo padrão *Multi-Panel Layouts* são os seguintes:

- Fragmento (*ArticleFragment*): Fragmento de uma atividade responsável por representar uma parte da tela a ser composta. Com clico de vida próprio, recebe os seus próprios eventos de entrada, e que você pode adicionar ou remover enquanto a atividade está em execução. Ele deve estender a classe *Fragment* (da API) ou similares (quando utilizada API externas ou de compatibilidade) para poder utilizar o método `onCreateView()` para definir o layout.
- Gerenciador (*FragmentManager*): Responsável por realizar uma operação como adicionar ou remover um fragmento. É necessário instanciar um `FragmentTransaction`, que fornece APIs para adicionar, remover, substituir e executar outras operações de fragmento.

A adoção deste padrão foi identificado em quase todas as aplicações analisadas, com exceção apenas para a aplicação da *Wikipédia*, pela característica peculiar da arquitetura desta aplicação, que utiliza a implementação Android apenas como um navegador para os artefatos web que de fato constituem a aplicação. Com exceção desta, todas as demais implementaram de algum modo o padrão *Multi-Pane Layouts*. Contudo, observou-se variações em sua implementação. A Fig. 9 ilustra um exemplo da implementação deste padrão realizada pelo aplicativo ZapZap. Neste caso, foi criada uma classe específica, chamada `BaseFragment`, para herdar de `Fragment` e ser usada como base para criação de quaisquer outros fragmentos do aplicativo. No exemplo, está ilustrada a implementação de uma classe `ContactAddFragment`, que herda de `BaseFragment` para a exercer a função de um *ArticleFragment* especificado pelo padrão, responsável, no caso, pela tela que compõem a funcionalidade de adição de contatos no aplicativo. Além dessas, foi implementada uma classe chamada `LaunchActivity` para funcionar como o gerenciador de fragmentos (*FragmentManager*) na aplicação. Ela contém um método específico (`presentFragment`) que recebe como parâmetro um fragmento-base e instanciando um objeto `FragmentTransaction`, controla a apresentação/rotação do fragmento na tela.

[1] Especificação do padrão *Multi-Panel Layout*: https://developer.android.com/design/patterns/multi-pane-layouts.html.

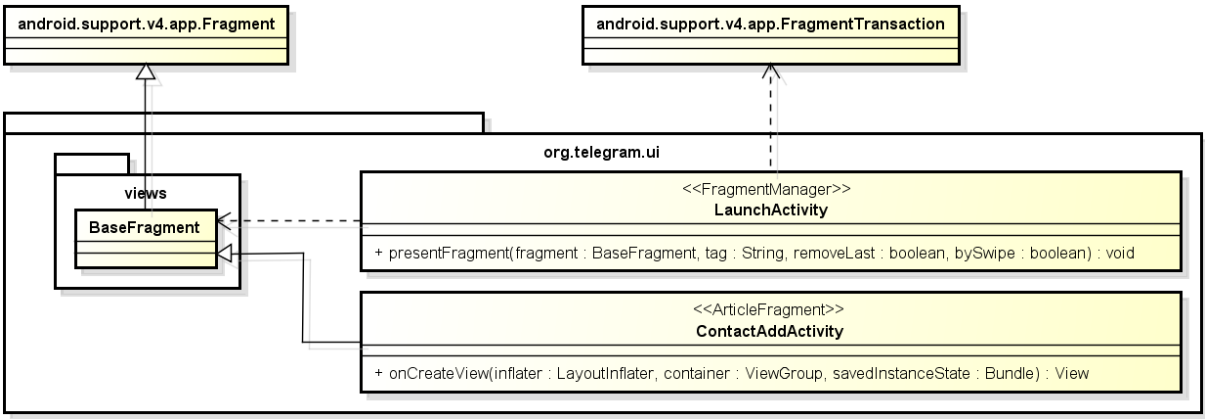

Fig. 9. Implementação do padrão *Multi-Panel Layouts* no aplicativo ZapZap

Algumas aplicações, como é o caso da própria *WordPress* e da *iFixit*, utilizaram uma API externa conhecida como *SherLock API*[2] para criação de fragmentos. Esta API possibilitou, entre outras coisas, a execução de fragmentos em aparelhos com versões anteriores a 3.0 do Android, uma vez que API nativa do Android fornece suporte apenas para aparelhos a partir desta versão. Contudo, há uma API de compatibilidade conhecida como *AppCompv7*, também oficial do Android, que fora disponibilizada depois para atender aos desenvolvedores que queriam desenvolver aplicações também para versões anteriores a 3.0. Inclusive, as demais aplicações, a exemplo da *c:geo*, já utilizaram tal pacote de compatibilidade para implementar este padrão, quando não implementaram apenas utilizando as classes padrões do API nativa do Android. Em termos de implementação do padrão, a principal mudança perceptível em cada um desses casos ocorre apenas nas classes herdadas para criação dos fragmentos. Por exemplo, na Fig. 9, se ao invés de utilizar a API nativa do Android para implementar o padrão tivesse sido utilizada a API *SherLock*, a classe `BaseFragment` herdaria de `SherLockFragment`, a classe específica dessa API que substitui a nativa.

(iii) Gerenciamento de Diferentes Versões da Plataforma: A cada nova versão do Android e, consequentemente, nova versão do kit de desenvolvimento Android (SDK), novos recursos são incluídos e outros depreciados, o que requer uma gerencia das funcionalidades que utilizam os antigos e novos recursos para que a aplicação se adapte a versão do Android instalado no aparelho cliente. Para lidar com esse problema, o Android criou o padrão *Compatibility*[3], que prever o uso de *tags* específicas no arquivo "AndroidManifest.xml" para especificar informações sobre a versão da aplicação desenvolvida de forma que elas podem ser recuperadas no código e confrontadas, com estruturas condicionais, em tempo de execução, com a versão do Android do aparelho, para escolher por essa ou aquela implementação. O AndroidManifest.xml é um arquivo base e obrigatório de toda aplicação Android. Nele é possível ainda especificar diferentes versões de estilos e temas para a aplicação, que estejam disponíveis como *resources* do projeto. A Fig. 10 exemplifica uma possível implementação deste padrão, por meio de especificação de *tags* no arquivo "AndroidManifest.xml" para definir a versão mínima e máxima suportada pelo aplicativo (Fig. 10.a). As versões especificadas no arquivo *xml* são então recuperadas no código Java por meio da classe `Build` e comparadas ao da versão do SDK em execução (Fig. 10.b).

Fig. 10. Exemplificação do uso do padrão *Compatibility*: (a) tags no arquivo AndroidManifest.xml e (b) exemplo de uso dos B

No geral, o que observou-se foi que esta solução foi implementada por todas as APIs utilizadas, porém complementada em alguns casos, quando questões de compatibilidade excedia a escolha pelo uso deste ou

[2] *Site* oficial do API *Sherlock*: http://actionbarsherlock.com.

[3] Especificação do padrão *Compatibility*: https://developer.android.com/design/patterns/compatibility.html.

daquele recurso da plataforma. Esse foi o caso da aplicação *c:geo*. Para esta aplicação, a compatibilidade dos aparelhos dependia de outras características do mesmo, tais como, tamanho da tela, presença de sensores GPS e câmeras, etc. Para resolver estes problemas, foi implementado em sua arquitetura uma hierarquia de classes específica para emular diferentes cenários, conforme ilustra o fragmento de classes apresentado na Fig. 11.

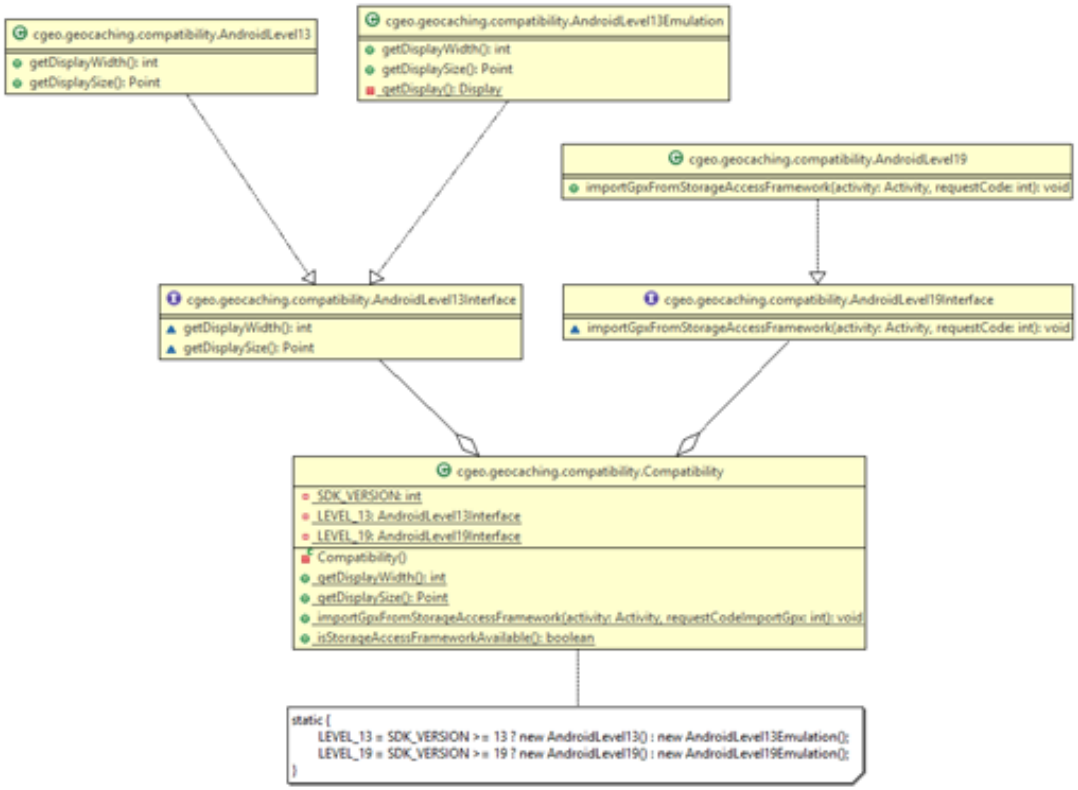

Fig. 11. Fragmento das classes do pacote *Compatibility* da aplicação c:geo

b) Utilização de outros padrões de projetos tradicionais

Nosso estudo também buscou identificar a implementação de outros padrões de projetos tradicionais que foram adotados no desenvolvimento das aplicações Android analisadas. Foi observada a utilização de alguns padrões de projetos conhecidos na maioria das aplicações com finalidades recorrentes, foram esses os padrões: *Provider Model* (Howard, 2004), *Factory Method* e *Adapter* (Gamma, Helm, Johnson, & Vlissides, 1994). A Tabela 2 apresenta um resumo esquemático da adoção desses padrões nas aplicações que analisamos, com ênfase na finalidade da utilização em cada uma delas, que quase sempre são congruentes.

Tabela 2. Quadro esquemático da utilização de padrões de projeto nas aplicações Android investigadas

PADRÃO DE PROJETO	APLICAÇÕES QUE ADOTAM	FINALIDADE
Provider Model	Wordpress for Android	Prover uma interface comum para registros na tabela de estatística da aplicação
	iFixit: Repair Manual	Prover uma interface comum para configuração da funcionalidade autossugestão durante uma busca
	c:geo	Prover interfaces comuns para configuração dos plug-ins utilizados para captura de mapas na aplicação
Factory	iFixit: Repair Manual	Implementar uma fábrica de conexões para aplicação
	WordPress for Android	Implementar uma fábrica de conexões para aplicação
	c:geo	Implementar diferentes interfaces dos mapas da aplicação
	ZapZap	Implementar uma fábrica de conexões para aplicação

Adapter	iFixit: Repair Manual	Estender o pacote "Android.widget" através das interfaces BaseAdapter e ArrayAdapter para customizar os componentes visuais da sua aplicação, tais como seus próprios ListView e Widget
	c:geo	
	ZapZap	
	Barcode Scanner	

O *provider model* é um padrão de projeto criado pela Microsoft para permitir ao aplicativo escolher uma das várias implementações ou configuração da aplicação, a partir da implementação de uma classe abstrata `ProviderBase` (Howard, 2004), que pode ser implementada por um método de fábrica, por exemplo, para fornecer acesso a diferentes armazenamentos de dados, para obter informações de *login*, ou à utilização de metodologias de armazenamento diferentes, como um banco de dados, binário para o disco, XML, etc. No estudo, identificamos a adoção deste padrão em três aplicações: *WordPress*, *iFixit* e *c:geo*. Na primeira aplicação, uma classe `ProviderBase` foi implementada para configurar as estatísticas de um *blog* na aplicação. Já no *iFixit*, o padrão foi utilizado para permitir optar por uma outra implementação da funcionalidade de autossugestão durante uma busca de guias e manuais no aplicativo. Por fim, no aplicativo *c:geo*, uma implementação do padrão foi realizada para prover uma base comum de configuração dos *plug-ins* utilizados para captura de mapas na aplicação.

O padrão de projeto *factory method* é um padrão de criação de objetos que utiliza métodos de fábrica para lidar com o problema da criação de objetos sem especificar a classe exata do objeto que será criado (Gamma, Helm, Johnson, & Vlissides, 1994). Isto é feito através da criação de objetos através de um método de fábrica, o que é ou especificados em uma interface (classe abstrata) e implementado na implementação de classes (classes concretas); ou implementado em uma classe base, que pode ser substituído quando herdou em classes derivadas; e não por um construtor. É comum a atualização deste padrão para criar uma fábrica de conexões com o banco de dados, tanto que neste estudo, observamos a utilização deste padrão com este mesmo fim em duas aplicações: *iFixit* e *ZaZap*. Além delas, na aplicação *c:geo* o padrão foi utilizado para implementar a classe *ProviderBase* que controla os *plug-ins* dos mapas da aplicação.

Por fim, a implementação do padrão *Adapter* foi identificado em quatro das seis aplicações analisadas: *iFixit*, *c:geo*, *ZapZap* e *Barcode Scanner*. Em todas elas o objetivo da adoção do padrão foi de implementar classes específicas do pacote "`android.widget`", através das interfaces `BaseAdapter` e `ArrayAdapter`, para customizar alguns componentes visuais na aplicação, tais como, seus próprios *ListView* e outros *Widget*. O padrão *Adapter* é muito utilizado quando precisamos adaptar uma nova biblioteca de classes (*frameworks* ou APIs), adquirida de um fornecedor, a um projeto do nosso interesse (Gamma, Helm, Johnson, & Vlissides, 1994). Ele é adotado quando não temos o código do novo fornecedor e também não podemos alterá-la, o que requer que seja feito uma classe que faça essa adaptação, ou seja, uma classe que fique responsável por adaptar a interface do novo fornecedor ao formato que o sistema espera. Portanto, o adaptador é um intermediador que recebe solicitações do cliente e converte essas solicitações num formato que o fornecedor entenda. Muito utilizado, como observamos nesse estudo, para estender, a partir de uma interface adaptativa, os componentes nativos de uma API, no caso, componentes nativos da API Android.

3.3 Tratamento de exceções

Não foi identificado a utilização de nenhum padrão específico para tratamento de exceção em nenhuma das aplicações analisadas. Em algumas delas até observou-se a criação de classes de exceções específicas, como nas aplicações WordPress e ZapZap. Em ambas tais classes foram criadas para definir novos tipos de exceções camada que implementava as classes do protocolo de comunicação RPC. Nas demais aplicações analisadas, o tratamento de exceções, de modo geral, eram repassadas para serem tratadas nas camadas de *controller* e de classes utilitárias, porém esse comportamento não se mostrou tão padrão em todas as aplicações analisadas.

A Tabela 3 apresenta um quadro esquemático com as exceções que foram explicitamente capturadas e tratadas nas aplicações analisadas. Esses dados foram extraídos das aplicações que possuam códigos hospedados no repositório Github, por meio de buscar automatizadas utilizando palavras reservadas da linguagem Java para captura e tratamento de exceções. O projeto WordPress foi o único a não ser analisado nesse critério, por está hospedado em um repositório diferente do Github que não possibilita a execução de buscar automatizadas no código.

Tabela 3. Principais exceções capturadas e tratadas pelas aplicativos

APLICAÇÃO	QUANTIDADE DE CLASSES COM OCORRÊNCIA DE TRATAMENTO		PACOTE/ CAMADA	PRINCIPAIS EXCEÇÕES TRATADAS
Wikipédia	7 (sete)		wikipedia	NameNotFoundException, JSONException, ClassCastException, Exception
iFixit	22	11	util	
		9	ui	ArrayIndexOutOfBoundsException, NumberFormatException, ClassNotFoundException, UnsupportedEncodingException, IOException, JSONException, Exception
		1	model	JSONException
c:geo	78	9	utils	NameNotFoundException, IOException, StackTraceDebug, NoSuchAlgorithmException, UnsupportedEncodingException, GeneralSecurityException
		7	ui	IndexOutOfBoundsException, NoSuchFieldException, IllegalAccessException, RuntimeException, ParseException, Exception
		15	connector	NumberFormatException, SAXException, JsonProcessingException, NullPointerException, IOException, ClassCastException, ParserException, ParseException, IOException, RuntimeException , Exception
		2	outros	---
ZapZap	59	31	ui	Exception
		1	phone format	Exception
		3	sqlite	SQLiteException, Exception
		1	objects	Exception
		23	messanger	ClassCastException, Exception
Barcode Scanner	29		zxing	RejectedExecutionException, RuntimeException, IOException, NullPointerException, URISyntaxException, IllegalStateException, FileNotFoundException, WriterException, JSONException, UnsupportedEncodingException, ActivityNotFoundException, InterruptedException, SQLiteException, ReaderException, InterruptedException, ActivityNotFoundException, ParseException, ActivityNotFoundException

Os dados coletados foram colhidos a partir de buscar automatizadas nos respectivos repositórios "Github" utilizando palavras-chaves da sintaxe java para capturar e tratamento de exceções.

4. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou os resultados de um estudo exploratório realizado com aplicações desenvolvidas para a plataforma Android com o objetivo de identificar características arquiteturais comuns a essas aplicações e analisa-las. Nesse sentindo, nossa principal contribuição com este estudo foi a realização de uma análise qualitativa sobre as arquiteturas extraídas. Os resultados da análise foram divididos em três grupos principais de discussões, que considerou como base três critérios predefinidos para análise: (i) análise arquitetural; (ii) utilização de padrões de projeto específicos da plataforma Android ou gerais; e (iii) política de tratamento de exceções adotada.

Como trabalhos futuros, pretendemos estender os estudos sobre aplicativos Android, refinando os mecanismos de busca e seleção de aplicativos e automatizando a análise dos mesmos. Além disso, pretendemos aprofundar nosso estudo sobre as soluções adotadas por desenvolvedores da plataforma Android para lidar com mais tipos de variabilidades neste cenário de aplicativos.

REFERENCES

Android. (s.d.). *Design Patterns*. Fonte: Android Developers: https://developer.android.com/design/patterns/

Android. (s.d.). *The Developer's Guide*. Fonte: Android Developers: https://developer.android.com/guide/

Apache Cordova. (2012). *Apache Cordova, learn more about the project"*. Fonte: http://cordova.apache.org/

AppBrain. (1º de Agosto de 2014). *Number of Android applications*. Fonte: AppBrain Stats: http://www.appbrain.com/stats/number-of-android-apps

Caputo, V. (12 de Fevereiro de 2014). *Android e iPhone foram 93,8% dos aparelhos vendidos em 2013*. Acesso em 29 de Julho de 2014, disponível em EXAME.com: http://exame.abril.com.br/tecnologia/noticias/android-e-iphone-foram-93-8-dos-aparelhos-vendidos-em-2013

Carneiro, M., Roman, C., & Fagundez, I. (01 de Janeiro de 2014). *Vendas de smartphones e tablets crescem mais de 100% em 2013*. Acesso em 27 de Julho de 2014, disponível em Coluna sobre Mercado do Jornal Folha de São Paulo: http://www1.folha.uol.com.br/mercado/2014/01/1391973-vendas-de-smartphones-e-tablets-cresceram-mais-que-100-em-2013.shtml

Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* New York: Addison-Wesley Professional.

Howard, R. (02 de Março de 2004). *Provider Model Design Pattern and Specification, Part 1*. (Microsoft Corporation) Acesso em 29 de Julho de 2014, disponível em Microsoft Developer Network (MSDN): http://msdn.microsoft.com/en-us/library/ms972319.aspx

Mojica, I., Adam, B., Nagappan, M., Dienst, S., & Berger, T. (2013). A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEE Software, 31*(2), 78-86. doi:10.1109/MS.2013.142

Project Amateras. (s.d.). *AmaterasUML*. Fonte: http://amateras.sourceforge.jp/

Ruiz, I. J., Nagappan, M., Adams, B., & Hassan, A. E. (2012). Understanding reuse in the Android Market. *Proceeding of 20th International Conference on Program Comprehension (ICPC)* (pp. 113-122). Passau: IEEE. doi:10.1109/ICPC.2012.6240477

Received August 2014; revised XXXX/YYYY; accepted XXXX/YYYY.