Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Programa de Pós-Graduação em Sistemas e Computação
Mestrado Acadêmico em Sistemas e Computação

Estudo Empírico de Análise da Compatibilidade de Aplicações Android com Diferentes Versões da API da Plataforma

Adorilson Bezerra de Araújo

Natal-RN
Fevereiro de 2017

Adorilson Bezerra de Araújo

Estudo Empírico de Análise da Compatibilidade de Aplicações Android com Diferentes Versões da API da Plataforma

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Sistemas e Computação, do Centro de Ciências Exatas e da Terra, da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Mestre em Sistemas e Computação.

Linha de pesquisa:
Engenharia de Software

Orientador

Prof. Dr. Uirá Kulesza

PPgSC – Programa de Pós-Graduação em Sistemas e Computação
CCET – Centro de Ciências Exatas e da Terra
UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN

Fevereiro de 2017

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Araújo, Adorilson Bezerra de.
Estudo empírico de análise da compatibilidade de aplicações Android com diferentes versões da API da plataforma / Adorilson Bezerra de Araújo. – Natal, RN, 2017.
86f. : il.

Orientador: Prof. Dr. Uirá Kulesza.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Engenharia de software – Dissertação. 2. Gerência de variabilidades – Dissertação. 3. Implementação de variabilidades – Dissertação. 4. Aplicações Android – Dissertação. 5. Plataforma de computação móvel – Dissertação. 6. Estudos empíricos – Dissertação. 7. Suporte multi-versão de API – Dissertação. I. Kulesza, Uirá. II. Título.

RN/UF/BSE-CCET CDU 004.41

ADORILSON BEZERRA DE ARAÚJO

Estudo Empírico de Análise da Compatibilidade de Aplicações Android com Diferentes Versões da API da Plataforma

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Dr. Uirá Kulesza – UFRN
(Presidente)

Dr. Bruno Motta de Carvalho – UFRN
(Coordenador do Programa)

Banca Examinadora

Dr. Eduardo Henrique da Silva Aranha – UFRN
(Examinador)

Dr. Rodrigo Bonifácio de Almeida – UnB
(Examinador)

Fevereiro, 2017

Resumo

A plataforma Android é atualmente a mais popular para o desenvolvimento de aplicações móveis, ocupando mais de 80% do mercado de sistemas operacionais para dispositivos móveis, criando uma demanda por customizações de aplicações para lidar com diferentes dispositivos, tais como, tamanho de tela, poder de processamento e memória disponível, idiomas e necessidades específicas dos usuários. Já foram disponibilizadas 23 novas versões da plataforma Android desde o seu lançamento. De forma a permitir a execução com sucesso das aplicações em diferentes dispositivos, é fundamental oferecer suporte às múltiplas versões da API (Application Programming Interface). Esta dissertação de mestrado tem como objetivo: analisar, caracterizar e comparar técnicas utilizadas por aplicações Android para oferecer suporte a múltiplas versões da API. Em especial, o trabalho busca: (i) identificar na literatura quais as técnicas indicadas para suporte às múltiplas versões da API Android; (ii) analisar aplicações reais para quantificar o uso dessas técnicas; e (iii) comparar as características e consequências do uso de tais técnicas. Um estudo empírico foi conduzido para atingir tal objetivo, no qual foram analisadas 25 aplicações Android populares. Os resultados do estudo mostram que existem três técnicas para prover suporte àss múltiplas versões da API: i) pacote de compatibilidade, variabilidades de granularidade grossa da API que envolvam um conjunto de classes; ii) re-implementação de recurso, para situações pontuais e granularidade grossa em nível de classe ou quando o recurso não está disponível em pacote de compatibilidade; e iii) uso explícito da nova API, variabilidades de granularidade fina da API que envolva a chamada de métodos específicos. Através da análise de 25 aplicações identificamos que pacote de compatibilidade foi utilizada por 23 aplicações, re-implementação de recurso por 14 e uso explícito da nova API por 22. A API de fragmentos contêm os elementos mais comuns dentre os lançados em versões superiores da plataforma que são usados pelas aplicações durante sua evolução, sendo referenciados por 68% delas. No geral, as aplicações poderiam aumentar o seu mercado em potencial com adaptações de, em média, 15 trechos de código, por outro lado, os desenvolvedores das aplicações têm se preocupado em evitar código-morto em função da API da plataforma. Na análise de 7 aplicações, 4 delas continham código-morto, mas os quais em geral não representam mais do que 0,1% do seu código total.

Palavras-chave: gerência de variabilidades, implementação de variabilidades, aplicações Android, plataforma de computação móvel, estudos empíricos, suporte multi-versão de API

Abstract

Android is currently the most popular platform for the development of mobile applications, representing more than 80% of the operating systems market for mobile devices. This causes demands for application customizations to handle different devices such as screen size, processing power and available memory, languages, and specific user needs. Twenty-three new versions of Android platform have been released since its first release. In order to enable the successful execution of applications on different devices, it is essential to support multiple versions of the Application Programming Interface (API). This dissertation aims to analyze, characterize and compare techniques used by Android applications to support multiple versions of the API. In particular, the work seeks: (i) to identify the used techniques to support multiple versions of the Android API in the literature; (ii) to analyze real applications to quantify the usage of these techniques; and (iii) to compare the characteristics and consequences of using such techniques. An empirical study, in which 25 popular Android apps were analyzed, was conducted to achieve this goal. The results of the study show that there are three techniques to support multiple versions of the API: i) compatibility package, that adrresses API coarse granularity variabilities involving a set of classes; ii) re-implementation of resource used for specific situations and coarse granularity at class level or when resource is not available in compatibility package; and iii)explicit use of the new API that allows implementing fine grained variabilities of the API that involves calling of specific methods. Through the analysis of 25 applications, we have identified that compatibility package was used by 23 applications, re-implementation of resource was used by 14 applications and the explicit usage of the new API was used by 22 applications. The API fragments contains the most common elements among those released in higher versions of the platform that are used by applications during their evolution, and it is referenced by 68% of them. In general, applications could increase their potential market with adaptations of, on average, 15 code snippets. On the other hand, application developers have been worried about how avoiding dead code based on platform API. In the analysis of 7 applications, 4 of them contained dead code, but it did not represent more than 0.1% of total code.

Keywords: Variability management, implementation of variabilities, Android applications, mobile computing platform, empirical studies, multi-version API support.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

API – *Application Programming Interface*

FIC – *Fragmentation-Induced Compatibility*

DVM – *Dalvik Virtual Machine*

JVM – *Java Virtual Machine*

APK – *Android Package*

SDK – *Android Software Development*

XML – *Extensible Markup Language*

JAR – *Java ARchive*

EC – *Execução Condicional*

OO – *Orientação a Objetos*

FIC – *Fragmentation-Induced Compatibility*

Sumário

1 Introdução

Dispositivos móveis estão ficando cada vez mais populares e acessíveis para pessoas de diferentes poder aquisitivo ao redor do mundo [1]. A plataforma Android é atualmente a mais popular para o desenvolvimento de aplicações móveis, ocupando mais de 80% do mercado de sistemas operacionais para dispositivos móveis [2]. Tal realidade cria uma demanda por customizações de aplicações para lidar com diferentes dispositivos, tais como, tamanho de tela, bibliotecas de classes disponíveis (API - *Application Programming Interface*), disponibilidade de poder de processamento e memória, idiomas e necessidades específicas dos usuários. O Facebook, por exemplo, disponibiliza duas versões do seu aplicativo. A versão padrão é direcionada para os dispositivos mais modernos e a alternativa, chamada de Facebook Lite[1], para dispositivos mais antigos (que utilizam a partir da versão 2.2 do Android), que ocupa bem menos espaço de armazenamento, pode ser carregada na memória do dispositivo mais rapidamente e funciona com conexões de Internet instáveis ou lentas.

A existência de diferentes dispositivos para os quais a plataforma Android oferece suporte é conhecido como fragmentação, e isso se apresenta como um grande desafio em relação à outras plataformas. Há um número considerável de dispositivos com versões antigas da API [3]. Já em 2011, 86% dos desenvolvedores consideravam a fragmentação da plataforma um sério problema [4]. Diferentes versões da API é uma das variabilidades mais comuns na plataforma Android. Esse cenário leva os desenvolvedores de aplicações Android a buscarem técnicas e metodologias que otimizem o desenvolvimento de versões da aplicação que melhor atendam às restrições e características de cada dispositivo. A documentação oficial do Android [5] é rica em propor soluções para tais variabilidades, no entanto, a implementação dessas soluções pode sofrer variações. Cada equipe de desenvolvimento pode tomar decisões de projeto diferentes. Em especial, existe atualmente uma carência de estudos sobre a forma como aplicações Android atuais oferecem suporte às múltiplas versões da API. Neste con-

[1] Disponível em: https://play.google.com/store/apps/details?id=com.facebook.lite

texto, este trabalho busca entender as técnicas utilizadas para oferecer tal suporte e qual o impacto do seu uso.

1.1 Problema Abordado

Apesar da ampla adoção da plataforma Android e o problema da fragmentação [6], há uma carência de estudos que indiquem de que forma as aplicações de tal plataforma lidam e se adaptam às diversas versões lançadas da API. Enquanto atender a usuários com uma antiga versão da API significa ter um amplo mercado potencial, utilizar os recursos das versões mais novas da API é um dos três fatores mais importantes para as aplicações com alta avaliação na Google Play [7]. Oferecer suporte a múltiplas versões da API da plataforma, de forma a atender clientes com versões antigas ao mesmo tempo usar os recursos mais modernos quando disponíveis, é primordial para o sucesso das aplicações.

Uma alternativa possível é disponibilizar uma versão da aplicação para cada versão da API. Essa solução é a comumente utilizada para APIs locais - APIs de sistemas operacionais de computadores de mesa, por exemplo -, onde o número de versões de APIs é reduzido e a evolução é mais lenta em comparação à API do Android, na qual existem cerca de 10 versões da API com uma margem relevante do mercado. Manter diversas versões simultâneas de uma aplicação causa duplicação de código e prejudica a manutenibilidade. Assim, a plataforma oferece mecanismos para que uma única base de código atenda da melhor forma possível às diversas versões da API, usando os melhores recursos disponíveis nos aparelhos.

No entanto, no melhor do nosso conhecimento, não existem estudos sistemáticos que identifiquem e analisem técnicas para implementação de suporte às múltiplas versões da plataforma Android. Tais estudos podem trazer luz sobre as práticas adotadas pela comunidade de desenvolvedores e auxiliá-los a decidir quando e quais técnicas melhor atendem a determinadas situações.

1.2 Limitações de Trabalhos Existentes

Diversos trabalhos têm sido realizados sobre a API da plataforma Android. Sobretudo, no sentido de identificar os elementos mais comuns e padrões de uso. Lamba et al.[8] apresentaram resultados de um estudo em larga escala de análise do uso da API

em aplicações Android. O estudo envolveu 1.120 aplicações *open-source* e 17,4 milhões de linha de código. Foram identificados os métodos mais frequentemente invocados, os pacotes da API, classes, e padrões de chamadas mais populares.Os métodos mais comuns são `getString()`, `get()` e `toString()`. Os pacotes mais comuns são `java.util` e `android.content`. As classes mais comuns são `Context` e `View`. O padrão de chamada mais popular está relacionado à dimensão, espaçamento e margens de uma *view* na interface com o usuário. No entanto, na análise de popularidade de métodos não levaram em consideração suas respectivas classes. Por exemplo, o método `getString()` da classe `Activity` foi contabilizado juntamente com o método `getString()` da classe `Cursor`, que não possuem nenhuma relação entre si.

McDonnell et al. [9] conduziram um estudo sobre a co-evolução da API Android e suas aplicações usando o histórico de versões encontradas no Github. O estudo confirmou que a plataforma Android evolui rapidamente em uma taxa de 115 mudanças da API por mês na média. Contudo, a adoção pelas aplicações clientes não segue na mesma velocidade. Cerca de 28% das referências à API nas aplicações clientes estão desatualizadas com uma mediana de tempo de 16 meses. 22% das referências desatualizadas são eventualmente atualizadas para versões mais novas da API, mas com um tempo de propagação de cerca de 14 meses, que é mais lento que o tempo médio entre novas versões da API (3 meses). APIs de rápida evolução são mais usadas por clientes que APIs de evolução lenta, mas o tempo médio necessário para adoção de novas versões é mais longo para APIs de rápida evolução. Além disso, código adaptado para uso de novas APIs são mais sujeitos a erros que aqueles sem adaptação para novas APIs. Segundo os autores do trabalho, os resultados sugerem que os desenvolvedores não adotam novas APIs com rapidez, mantendo referências para APIs desatualizadas. Assim evitam a instabilidade de novas APIs e o trabalho de atualização propriamente dito. Porém, outras possibilidades devem ser consideradas: os desenvolvedores atualizam para as mais recentes APIs, mas mantém tais códigos para garantir a compatibilidade com APIs mais antigas; ou atualizam para APIs recentes e as referências às APIs antigas representam código-morto.

Aplicações Android declaram uma versão alvo da API. Quando estão em execução em um dispositivo cuja versão da API é inferior à versão alvo, as aplicações são executadas em um modo de compatibilidade, que busca reproduzir o comportamento da versão alvo indicada pelo aplicação. Esse cenário desabilita as melhorias da nova versão, incluindo as correções de problemas de segurança. Mutchler et al. [10] chamam essa característica de problema da fragmentação do alvo e analisam um con-

junto de 1.232.696 aplicações Android para mostrar que isso traz sérias consequências para todo o ecossistema de aplicações e não mudou consideravelmente com o passar dos anos. No total, 93% das aplicações atualmente definem como alvo uma versão desatualizada e possuem uma média de desatualização de 686 dias; 79% das aplicações já estão desatualizadas no dia que são carregadas para o Google Play. Eles também analisaram 5 mudanças na plataforma relacionadas a segurança e que estão desabilitadas nessas aplicações desatualizadas.

Wei [11] realizou um estudo sobre problemas de compatibilidade induzidos pela fragmentação (FIC *issues* - *Fragmentation-Induced Compatibility issues*) da plataforma Android. De forma manual, analisaram 191 *issues* nos sistemas de reporte de erros de 5 projetos de código aberto e os *commits* relacionadas a esses *issues*. Nesse estudo, identificaram que existem 5 maiores causas de FIC *issues*, entre elas a evolução da API. FIC *issues* podem causar tanto consequências funcionais como não-funcionais, como travamento da aplicação, aplicação não funcionar, degradação na performance e na experiência de uso. Os sintomas podem ser específicos da aplicação, o que dificulta a criação de testes de compatibilidade. Localizar a causa dos FIC *issues* é difícil na prática. Por outro lado, as correções são usualmente simples e apresentam padrões comuns: checar informações do dispositivo e disponibilidade de componentes de software/hardware antes de chamar APIs/métodos causadores de problemas. Este é o padrão mais comum, presente em 137 das 191 correções analisadas. Uma forma típica de evitar os erros é pular a chamada da API ou substituí-la por uma implementação alternativa. Também desenvolveram uma ferramenta, FicFinder, que realiza uma análise estática nas aplicações e informa se estas contém FIC *issues*. Em um experimento com 27 aplicações do mundo real, o FicFinder efetivamente detectou FIC *issues* desconhecidos com uma alta precisão na análise. Nas 27 aplicações foram emitidos 51 alertas, sendo 46 de casos confirmados (precisão de 90,2%). FicFinder também pode fornecer informações úteis para ajudar desenvolvedores a diagnosticar e corrigir FIC *issues*. Os alertas emitidos pelo FicFinder foram encaminhados para os desenvolvedores das aplicações. Alguns já haviam corrigido o problema e outros informaram que iriam corrigir.

McDonnell et al. [9] e Mutchler et al. [10] não abordam o tema de suporte a múltiplas versões da API por aplicações Android existentes. Todas as análises dos estudos foram realizadas sem considerar questões como a adaptação e organização do código para lidar com as múltiplas APIs disponíveis.

No experimento realizado por Wei et al. [11], o FicFinder foi alterado para reportar situações em que os FIC *issues* já estavam protegidos de execuções indevidas, tais casos foram identificados como "boas práticas". Porém, o trabalho não apresentou essas boas práticas, tampouco fez qualquer tipo de comparação, avaliação ou indicação de uso.

Nesse contexto, baseado em trabalhos anteriores acerca da evolução da API Android e suas limitações, são necessárias pesquisas que:

- Identifiquem o suporte a múltiplas versão da API provida pela plataforma;
- Analisem como tais técnicas são implementadas em aplicações do mundo real, de forma qualitativa e quantitativa;
- Analisem o impacto do uso de tais técnicas no código-fonte das aplicações cliente.

1.3 Objetivos

O objetivo geral desta dissertação de mestrado é analisar, caracterizar e comparar técnicas de implementação de suporte a múltiplas versões da API da plataforma Android por aplicações existentes. Os objetivos específicos do trabalho são:

- Identificar na literatura quais as técnicas indicadas para oferecer suporte a múltiplas versões da API Android;
- Analisar aplicações reais para quantificar o uso dessas técnicas indicadas;
- Comparar as características e consequências do uso de tais técnicas; e
- Identificar as mudanças mais comuns na API que afetam a evolução das aplicações para rodar em aparelhos com versões diferentes da API.

1.4 Metodologia

Visando atingir os objetivos desta dissertação de mestrado, foi definido uma metodologia de trabalho composta das seguintes etapas:

- Revisão da literatura: essa etapa buscou identificar quais as técnicas indicadas pela literatura para oferecer suporte a múltiplas versões da API, além de entender melhor os trabalhos relacionados atuais;
- Estudo empírico: a partir dos resultados da fase anterior, foi realizado um estudo empírico com o objetivo de identificar e caracterizar a adoção dessas técnicas por aplicações reais. Ele foi composto das seguintes atividades principais: seleção das aplicações, análise das aplicações e análise dos resultados.

1.5 Organização do Documento

O restante deste documento está organizado da seguinte forma:

- O capítulo 2 apresenta os principais referenciais teóricos que serviram de arcabouço para o desenvolvimento deste trabalho;
- O capítulo 3 traz o estudo sobre a compatibilidade de aplicações Android com as diferentes versões da API da plataforma. São apresentados resultados do estudo e discussão de tais resultados;
- No capítulo 4 são discutidos trabalhos relacionados;
- Por fim, o capítulo 5 apresenta as considerações finais, principais contribuições e limitações do trabalho, bem como trabalhos futuros de pesquisa que podem ser desenvolvidos como continuidade desta dissertação.

2 Fundamentação Teórica

Este capítulo apresenta uma visão geral das bases teóricas que fundamentam as proposições e discussões contidas nesta dissertação, tais como Plataforma Android (Seção 2.1), com ênfase no suporte a múltiplas versões da API; Padrões de Projeto (Seção 2.2), destacando os que foram identificados nesta pesquisa; e, por fim, uma Taxonomia dos Componentes da API Android (Seção 2.3).

2.1 Plataforma Android

Android é um sistema operacional projetado para dispositivos móveis, incluindo celulares, *tablets*, relógios inteligentes, televisões e até carros. O sistema operacional é baseado no núcleo Linux, interagindo com o *hardware* em baixo nível e fornecendo conjuntos de API para acesso a serviços e demais funções[12], apresentando uma arquitetura em camadas, conforme figura 1. É sobre essa API que os desenvolvedores irão criar suas aplicações, portanto, mudanças nela afetarão diretamente as aplicações.

Aplicações Android são desenvolvidas em cima da camada *Java API Framework*, utilizando essa linguagem de programação, e são executadas em uma máquina virtual chamada DVM (*Dalvik Virtual Machine*). Essa máquina virtual é uma tecnologia *open-source* e diferencia-se da JVM (*Java Virtual Machine*), que é baseada em pilhas, por possuir uma arquitetura baseada em registros. Essa arquitetura permite a DVM ser executada com pouca memória, além de possuir seu próprio *bytecode*.

2.1.1 Loja de Aplicativos

Usuários podem instalar novos aplicativos diretamente a partir de arquivos APK (*Android Package*) ou obtê-los de lojas de aplicativos. A principal delas é a Google

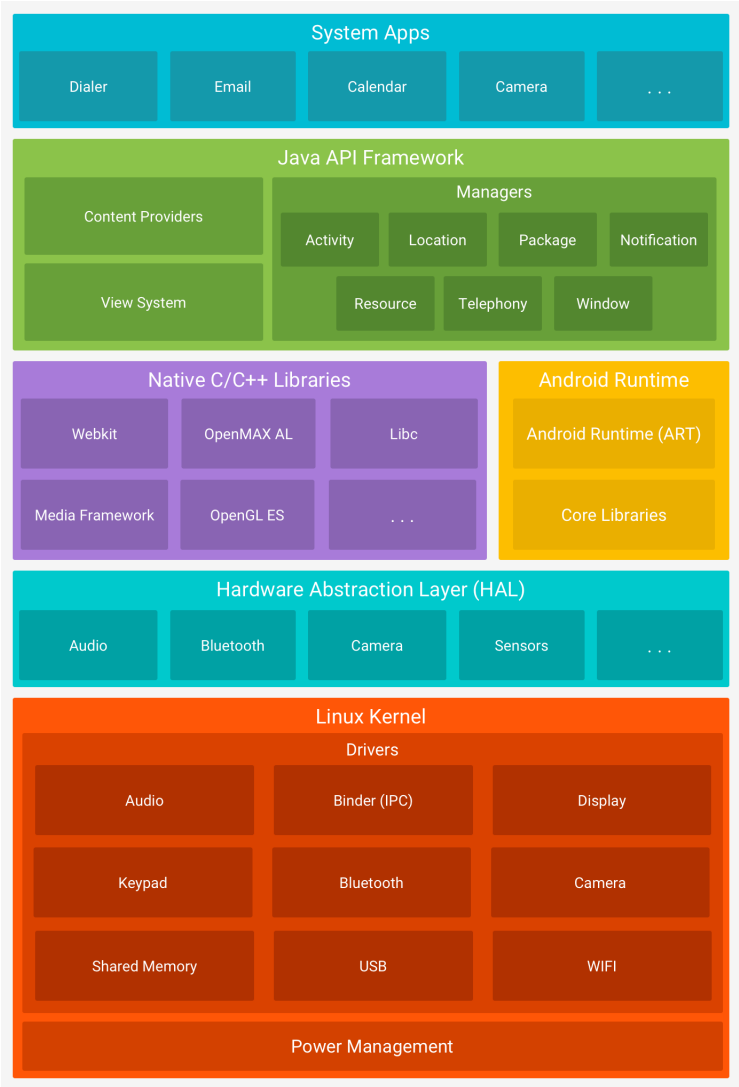

Figura 1: Arquitetura em Camadas da Plataforma Android.
Fonte: Guia Android [13].

Play[1], mas outras também estão disponíveis, como Amazon Appstore[2]. Na Google Play, estão disponíveis mais de 2,2 milhões de aplicativos [14].

2.1.2 Desenvolvimento de Aplicativos

Para criar uma aplicação Android são utilizados Android Software Development Kit (SDK), a linguagem de programação Java e a linguagem de marcação XML (*Extensible Markup Language*). O ambiente de desenvolvimento oficial é o Android Studio[15], que oferece de forma integrada diversas ferramentas para a criação de aplicações e recursos como edição de código, depuração, sistema flexível de compilação, emula-

[1]https://play.google.com
[2]https://www.amazon.com/b/ref=nav_shopall_adr_banjo?ie=UTF8&node=11350978011

dores, ferramentas de desempenho e análise de código, como o Android Lint [16], que foi utilizado neste trabalho e será mais detalhado na seção 2.1.4.

Uma aplicação Android consiste dos seguintes componentes [17]:

- Atividade: uma atividade é a classe onde a inferface com o usuário é implementada. Atividades também podem ser consideradas as tarefas que um usuário pode fazer em uma tela. Por exemplo, enviar um email pode ser feito em um atividade, enquanto escrever o email pode ser feito em outra atividade. Cada aplicação tem uma atividade especial chamada "atividade principal", que é o ponto de entrada da aplicação [18]. Uma aplicação contém múltiplas atividades que podem iniciar umas às outras para diferentes ações. Quando uma atividade inicia ela é colocada no topo de uma pilha chamada pilha de retorno. Quando o usuário pressiona o botão de voltar, a atividade atual é removida da pilha e a atividade anterior é exibida para o usuário. Normalmente, atividades removidas da pilha são destruídas e coletadas pelo coletor de lixo. Gerenciamento do ciclo de vida de uma aplicação é uma importante tarefa do desenvolvimento. Uma instância de uma atividade tem seu estado alterado entre diferentes possibilidades, que são:
  - Ativa: atividade está na frente da tela;
  - Pausada: atividade perdeu o foco, mas ainda é visível;
  - Parada: atividade está completamente oculta por outra atividade;
  - Destruída: quando parada ou pausada, o sistema encerra a atividade.

  Para cada estado, o sistema chama uma série de métodos do ciclo de vida [19], apresentada na figura 2. O método `onCreate()` é chamado quando a atividade é iniciada pela primeira vez, método `onStart()` é chamado quando a atividade torna-se visível para o usuário, `onRestart()` é chamado quando a atividade pausada mas não destruída é novamente iniciada, `onResume()` é chamado quando a atividade inicia a interação com o usuário, `onPause()` é chamado quando o sistema inicia a atividade anterior, `onStop()` é chamado quando a atividade não é mais visível, `onDestroy()` é o método final chamado antes da atividade ser destruída pelo sistema.

- Serviço: um serviço é um componente da aplicação que persiste por um longo tempo em *background*, o que significa dizer que não tem uma interface com o

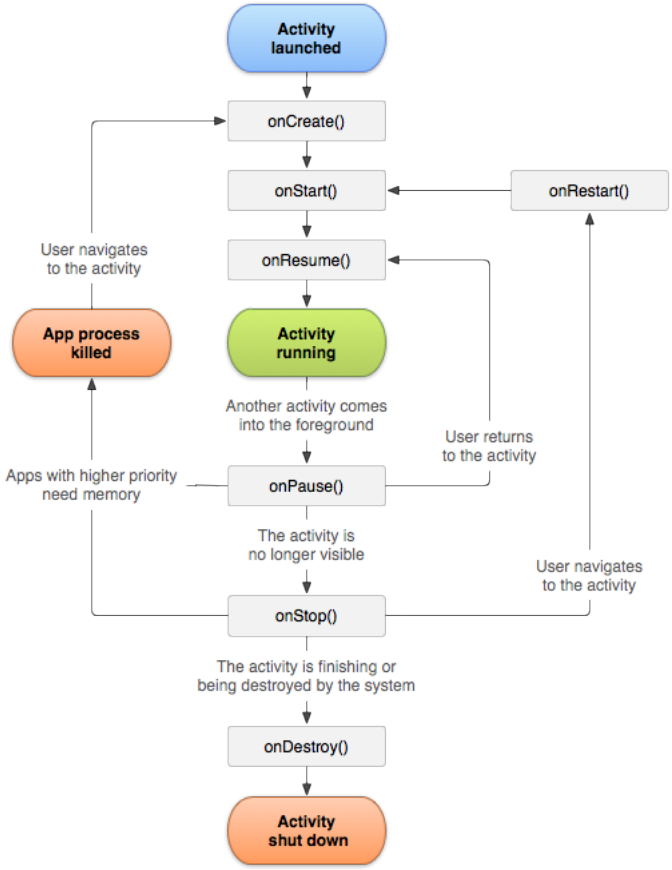

Figura 2: Ciclo de vida da atividade.
Fonte: Guia Android [19].

usuário. Uma das mais importantes propriedades dos serviços é que eles podem executar em *background* mesmo que o usuário alterne entre aplicações [20];

- Provedor de conteúdo: provedores de conteúdo podem ser considerados como gerenciadores de acesso a servidores de dados para um conjunto estruturado de dados. Eles são a padronização para transmissão de dados entre processos [21].

- Receptor de mensagem: um receptor de mensagem é usado para executar uma aplicação para responder a eventos enviados para todo o dispositivo, tais como o recebimento de uma mensagem de texto ou notificação de baixa carga de bateria [22].

2.1.3 Suporte a múltiplas versões da API

Um aspecto chave no desenvolvimento de aplicações Android são as versões da API. Google disponibilizou a primeira versão do Android (Android 1.0, API 1) em se-

tembro de 2008, desde então uma nova versão da API é disponibilizada aproximadamente a cada 3 meses. Nesse momento, a última versão disponibilizada é a 24. Para este trabalho foram consideradas, por questões de padronização das análises, configuração do ambiente de desenvolvimento e mercado, aplicações que usaram até a versão 23.

Apesar da rápida evolução do Android, a adoção das novas APIs é lenta e o mercado consumidor é fragmentado em relação ao número de versões da API. O que obriga os desenvolvedores de aplicações a tomarem decisões de projeto como usar as versões mais recentes da API e deixar de lado um mercado consumidor que não usa determinada API ou atender a esse mercado consumidor mas não usar as novidades da API mais recente.

Para auxiliar nessa decisão, o Google disponibiliza estatísticas de uso das versões da API[23], conforme mostra a figura 3, e mecanismos para o suporte a múltiplas versões. Ou seja, é possível atender um usuário com baixa versão de API ao mesmo tempo que é possível usar recursos mais modernos da plataforma. Tais mecanismos são apresentadas nas seções seguintes.

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.6%
4.1.x	Jelly Bean	16	6.0%
4.2.x		17	8.3%
4.3		18	2.4%
4.4	KitKat	19	29.2%
5.0	Lollipop	21	14.1%
5.1		22	21.4%
6.0	Marshmallow	23	15.2%

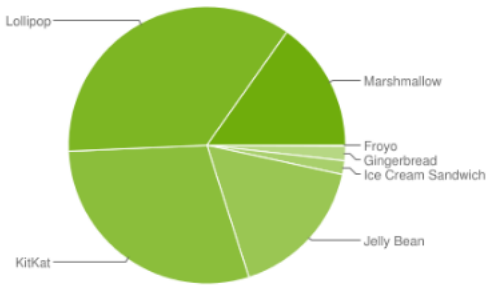

Figura 3: Distribuição das versões da API do Android.
Fonte: Android [24].

Embora 23 versões da API tenha sido lançadas, na figura 3 são apresentadas somente 10 versões. Isso ocorre porque versões com uma distribuição inferior a 0,1% são omitidas.

2.1.3.1 Pacote de Compatibilidade

Pacotes de compatibilidade [25], ou bibliotecas de suporte, permitem que aplicações em execução sob versões antigas da plataforma utilizem recursos que foram disponibilizadas em versões mais novas. Por exemplo, uma aplicação instalada em um aparelho com API de nível 8 pode usar a API de fragmentos, disponibilizada apenas no nível 11.

Esses pacotes são tradicionais arquivos JAR (*Java ARchive*) com classes, interfaces e outros artefatos de recursos que poderão ser adicionados na aplicação. Assim, quando se deseja usar a classe `Fragment`, por exemplo, em vez da importação ser da API padrão do Android, como mostra a Figura 4, deverá ser feita desse pacote que está junto da aplicação, como mostrado na Figura 5.

```
import android.app.Fragment;
import android.app.FragmentTransaction;
import android.app.ListFragment;
```

Figura 4: Importando classes relacionadas a fragmentos da API padrão

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.support.v4.app.ListFragment;
```

Figura 5: Importando classes relacionadas a fragmentos do pacote de compatibilidade

Como resultado, a aplicação será distribuída com uma experiência de uso mais consistente através de um grande número de versões da plataforma.

2.1.3.2 Re-implementação de recursos

Re-implementação de recursos ocorre quando se deseja um comportamento já provido por versões mais recentes ou outra biblioteca, mas opta-se por re-implementar tal recurso. Tal implementação pode ocorrer de três formas:

- Re-implementação própria [26];
- Cópia do código-fonte da API padrão [27];
- Cópia do código-fonte de biblioteca de terceiro que implementa o recurso [28].

Verificamos que 14 de 25 aplicações analisadas optaram por essa abordagem. Entre elas, o aplicativo de mensagens Telegram ao re-implementar algumas classes relacionadas a animações gráficas, quando poderia ter utilizado um pacote de compatibilidade com o recurso pronto. A figura 6 apresenta a estrutura de pacotes do Telegram, com destaque para as classes que re-implementam o pacote `android.animation` da plataforma, provento funcionalidades de animação para a aplicação, mesmo em dispositivos com nível da API inferior a 11, quando esse pacote foi disponibilizado.

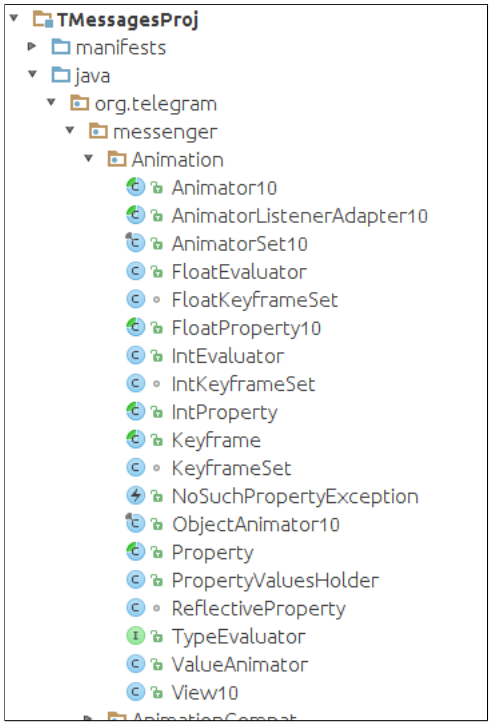

Figura 6: Estrutura parcial de pacote do Telegram

2.1.3.3 Uso explícito da nova API

Uso explícito da nova API ocorre quando é feita uma chamada a um novo recurso da API, disponibilizado apenas em uma versão da API superior à mínima exigida para execução da aplicação [29]. A figura 7 apresenta um alerta do Android Studio indicando uma chamada para um método da API nível 11, em uma aplicação que poderá ser instalada em qualquer aparelho com API igual ou superior a nível 4.

Caso a chamada seja realmente feita durante a execução da aplicação, será lançada uma exceção e a aplicação será interrompida, causado uma má experiência de uso. Dessa forma, é necessário proteger tal chamada, fazendo com que ela só seja realmente executada em ambientes seguros, no caso, em versões da API igual ou supe-

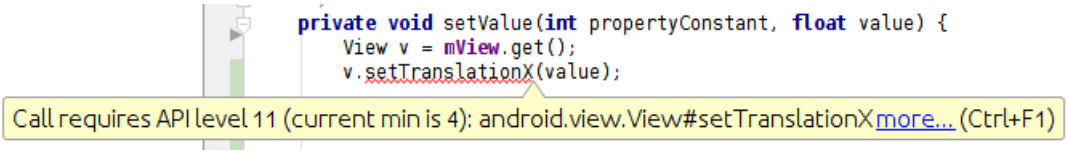

Figura 7: Alerta do Android Studio sobre uso explícito da nova API

rior a 11. As seções seguintes apresentam soluções de projeto que podem ser utilizadas para proteger usos explícitos de novas APIs.

2.1.3.4 Execução condicional

Execução condicional (EC) é um padrão para tratar variabilidades em granularidade fina em linhas de produto de software [30]. O padrão é composto por 4 elementos principais, ilustrados na figura 8:

- Classes alvo: componentes do sistema onde ocorre a variabilidade;
- Implementação da variabilidade: o código que representa a variabilidade, podendo ser um trecho diretamente na classe alvo ou uma chamada para um componente que implementa a variabilidade;
- Gerenciador da execução: é um elemento central do padrão, responsável por decidir qual implementação da variabilidade será executada;
- Repositório de parâmetros: responsável por armazenar os valores dos parâmetros que será utilizado durante a execução condicional. É fundamental que a recuperação desses parâmetros não prejudique o desempenho das aplicações.

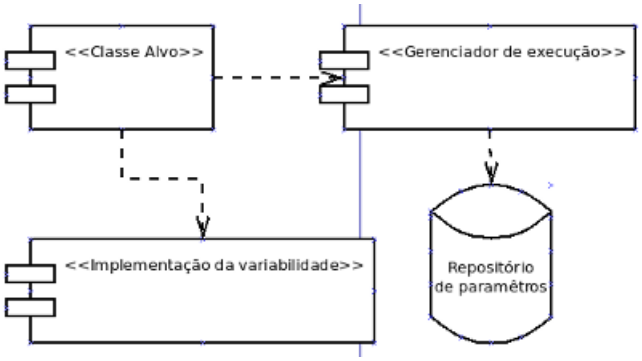

Figura 8: Componentes principais do padrão execução condicional
Adaptado de [30].

Em aplicações Android, as classes alvo podem ser qualquer classe da aplicação. Gerenciador de execução é uma instrução `if` que compara a versão da API no dispositivo e uma versão cujo valor tem alguma importância para a variabilidade em questão. Por exemplo, a versão em que um recurso surgiu ou mudou de comportamento. Implementação da variabilidade costuma ser uma sequência de código para quando o resultado dessa comparação é verdadeiro e outra sequência para quando ele é falso. O repositório de parâmetros que armazenar os valores utilizados nesse contexto é a classe `android.os.Build`, que contém diversas informações sobre o sistema instalado no aparelho, e suas classes aninhadas `VERSION` e `VERSION_CODES`. Em particular, a versão da API é obtida no atributo `Build.VERSION_CODE.SDK_INT`, já o outro valor da comparação é normalmente definida diretamente no código ou obtido de `VERSION_CODES`.

A figura 9 apresenta um exemplo do uso de EC na aplicação C:geo. A classe `AbstractDialogFragment` é a classe alvo no padrão, do repositório de parâmetros são obtidos os valores de `Build.VERSION.SDK_INT` e `Build.VERSION_CODES.HONEYCOMB`, para serem utilizados na condição do `if...else`, que faz o papel de gerenciador de execução. Também vemos que existem duas implementação de variabilidade: i) uma composta apenas por uma linha de código (`view.showContextMenu()`); e ii) a outra composta por várias linhas e encapsulada no método `showPopupHoneycomb(View view)`.

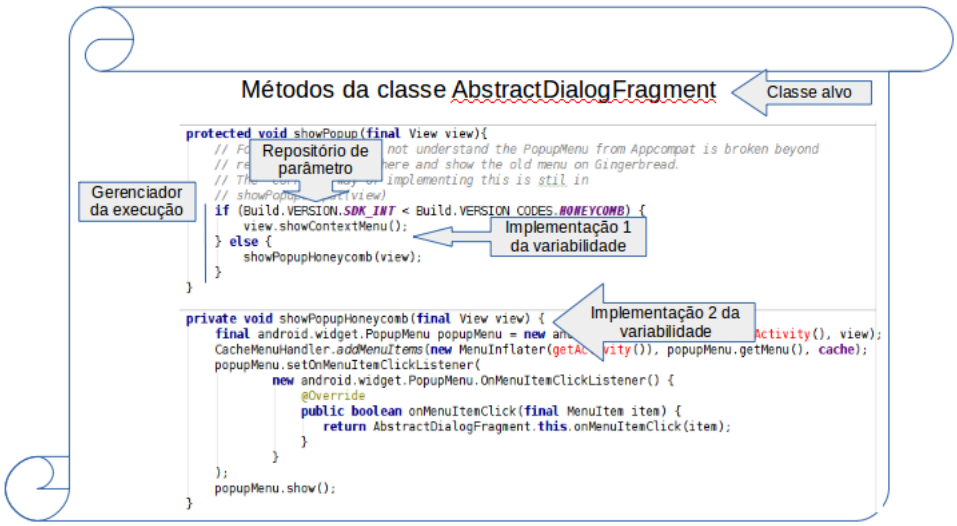

Figura 9: Exemplo do padrão EC no C:geo

Execução condicional é a forma indicada na documentação oficial [29] para o suporte a múltiplas versões. Pode ser utilizada em sua forma mais simples, como no

exemplo acima, ou combinado com padrões de projeto, como apresentado na seção 3.5.1.

2.1.4 Android Lint

Android Lint [16] é uma ferramenta do ambiente oficial de desenvolvimento de aplicações Android capaz de analisar o código-fonte de uma aplicação para identificar problemas estruturais, sem a necessidade de executá-la ou mesmo escrever testes.

Usos explícitos da nova API, portanto chamadas a elementos que não estão presentes na versão mínima de API exigida pela aplicação, são exemplos de problemas estruturais, podendo levar a falhas na execução, seja algum comportamento inesperado ou até mesmo interrupção da aplicação. Existem duas centenas de verificações, também chamadas de regras, feitas pelo Lint [31]. Duas delas verificam por acesso a elementos adicionados à API em versões superiores à versão mínima exigida pela aplicação: InlinedApi e NewApi.

InlinedApi verifica por acessos a constantes. Durante o *build* da aplicação os valores dessas constantes serão copiados para os arquivos das classes que as referenciam, o que significa dizer que os valores sempre estarão disponíveis, mesmo executando em dispositivos com API antigas. Em alguns casos não haverá problemas, em outros isso pode resultar em interrupção da execução ou comportamento incorreto. Dependerá do contexto, então os desenvolvedores deverão avaliar com cuidado se o código poderá ser executado em qualquer situação ou não.

NewApi verifica por chamadas a métodos e referências a classes. Como chamadas a métodos só podem ser resolvidas em tempo de execução, será lançada uma exceção do tipo `java.lang.NoSuchMethodError`, ou similar, levando ao travamento da aplicação, sempre que a chamada for feita em dispositivos com API antigas.

Dessa forma, nessa dissertação de mestrado focaremos apenas nas ocorrências de NewApi.

2.2 Padrões de Projeto

Padrões de projeto [32] e técnicas orientação a objetos (OO) – polimorfismo, herança, composição, agregação/delegação – são abordagens utilizadas para implemen-

tação de variabilidades de uma forma geral. Padrões de projeto identificam aspectos do sistema que podem variar e fornecem soluções para gerenciar a variação. Técnicas de OO são utilizadas para a implementação dos padrões de projeto [33]. Com efeito, há relatos do uso de tal abordagem em linhas de produto de software Android [34]. Essa seção apresenta alguns padrões de projeto encontrados durante a análise das aplicações.

2.2.1 Strategy

O padrão *Strategy* tem como objetivo "definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. *Strategy* permite que o algoritmo varie independentemente dos clientes que o utilizam" [32]. A figura 10 apresenta uma visão geral do padrão. `Context` é o cliente que usa os serviços de classes (`Strategy 1` e `Strategy 2`) que encapsulam os algoritmos de implementação da interface `Strategy`.

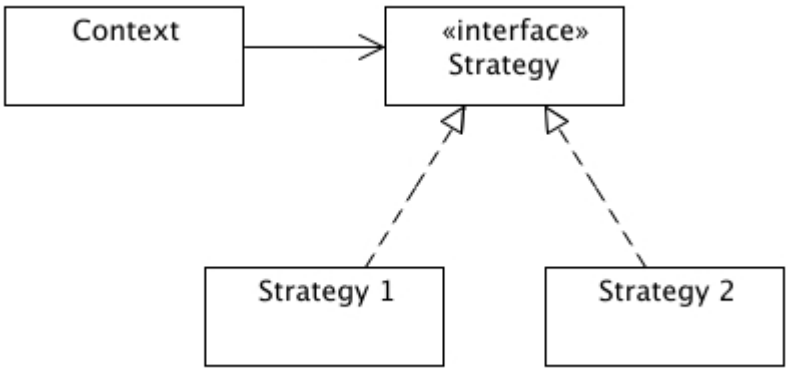

Figura 10: Visão geral do padrão Strategy
Fonte: Best Practice Software Engineering (BPSE) [35]

No contexto de versões da API, a interface `Strategy` pode conter um conjunto de métodos que encapsulam serviços que mudam de uma versão para outra, e cada implementação desse interface - `Strategy 1`, `Strategy 2` e assim por diante - encapsula o comportamento desse serviço na versão respectiva da API.

Implementação de padrões de projeto podem variar entre si. Por exemplo, em um dos aplicativos analisados, o AnkiDroid, foi definida uma relação semântica de dependência entre as implementações concretas da interface `Strategy`, que no caso foi chamado de `Compat`. A interface `Strategy` contém todos os métodos que encapsulam diferenças entre as diversas versões da API. A implementação concreta para a API de

menor nível que a aplicação dará suporte, API de nível 10 neste caso, deve implementar a interface e escrever o código para os métodos referentes à essa versão da API, deixando os métodos referentes às versões seguintes com corpo vazio. A implementação concreta referente a próxima versão da API, nível 11 no caso, deverá estender dessa classe e implementar o corpo dos métodos relevante. E assim sucessivamente com as demais versões.

A figura 11 apresenta um diagrama de classes simplificada dessa estrutura. Nessa figura, também notamos a presença de um outro elemento: a classe `CompatHelper`. Essa classe é responsável pela seleção e criação das estratégias concretas. A figura 12 mostra o código do método `CompatHelper`, onde a estratégia concreta é instanciada de acordo com a versão do API do aparelho. Em outras variações do padrão, essas tarefas podem ficar a cargo do próprio cliente ou em um método estático na interface `Compat`, que passará a ser uma classe abstrata.

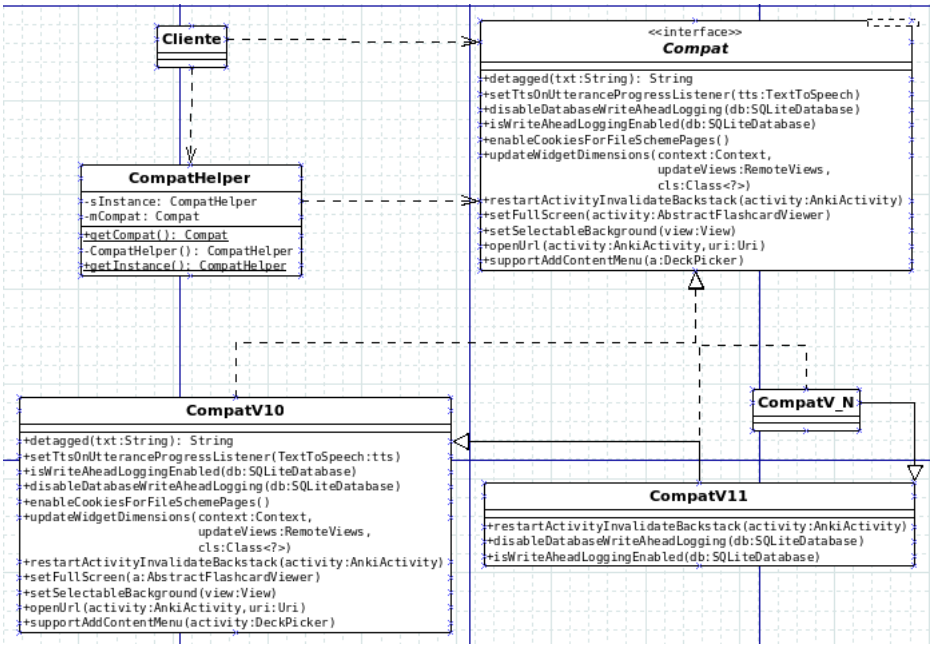

Figura 11: Diagrama de classe da implementação do padrão Strategy na aplicação AnkiDroid

2.2.2 Template Method

O padrão *Template Method* tem como objetivo "definir o esqueleto de um algoritmo em um método, deixando alguns passos para subclasses" [32]. A figura 13 apresenta uma visão geral do padrão. `AbstractClass` é a classe que contém o esqueleto do algoritmo, no método `TemplateMethod()`. No esqueleto são feitas chamadas para métodos (`PrimitiveOperation1` e `PrimitiveOperation2`) que são os passos que deverão

```
private CompatHelper() {

    if (isNookHdOrHdPlus() && getSdkVersion() == 15) {
        mCompat = new CompatV15NookHdOrHdPlus();
    } else if (getSdkVersion() >= 21) {
        mCompat = new CompatV21();
    } else if (getSdkVersion() >= 19) {
        mCompat = new CompatV19();
    } else if (getSdkVersion() >= 16) {
        mCompat = new CompatV16();
    } else if (getSdkVersion() >= 15) {
        mCompat = new CompatV15();
    } else if (getSdkVersion() >= 11) {
        mCompat = new CompatV11();
    } else if (getSdkVersion() >= 12) {
        mCompat = new CompatV12();
    } else {
        mCompat = new CompatV10();
    }
}
```

Figura 12: Trecho de código onde as implementações concretas são criadas de acordo com o nível da API no aparelho

ser implementados pelas subclasses, representada por `ConcreteClass`.

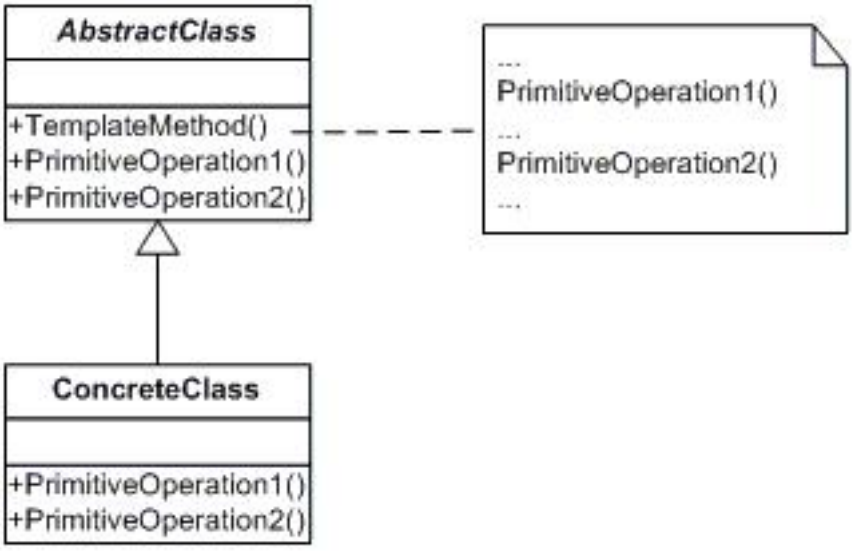

Figura 13: Estrutura geral do padrão de projeto Template Method
Fonte: DoFactory [36]

No contexto de versões da API, os métodos abstratos são comportamentos que mudam de uma versão para outra, já as classes concretas contém as implementações específicas necessárias para cada versão.

A aplicação que utilizou esse padrão de projeto, Wifiﬁxer, definiu como algoritmo para tratar as diferenças entre as versões da API os seguintes passos:

- Determinar o nível da API no aparelho;

- Executar o código específico do nível da API.

Assim, o algoritmo executa o passo **a** de forma fixa, implementado no *"template method"*, enquanto o passo **b** é delegado para a classe concreta. A figura 14 mostra o código do *template method* `getFile()`, onde a parte fixa do algoritmo faz a seleção da classe concreta de acordo com a versão da API e logo depois o método abstrato `vgetLogFile()` é chamado.

```
public static File getFile(Context context, String filename) {
   if (selector == null) {
        if (Build.VERSION.SDK_INT < 8) {
            selector = new LegacyFile();
        } else
            selector = new API8File();
    }
    return selector.vgetLogFile(context, filename);
}
```

Figura 14: Exemplo de template method no aplicativo Wififixer

2.2.3 Null Object

O padrão de projeto *Null Object* "encapsula a ausência de um objeto disponibilizando uma alternativa que nada faz como comportamento padrão" [37]. A figura 15 apresenta uma visão geral do padrão. `Client` é o cliente que usa os serviços de uma classe abstrata ou interface (`AbstractObject`), que terá implementações reais (`RealObject`) e ao menos uma implementação de um objeto nulo (`NullObject`) que faz nada.

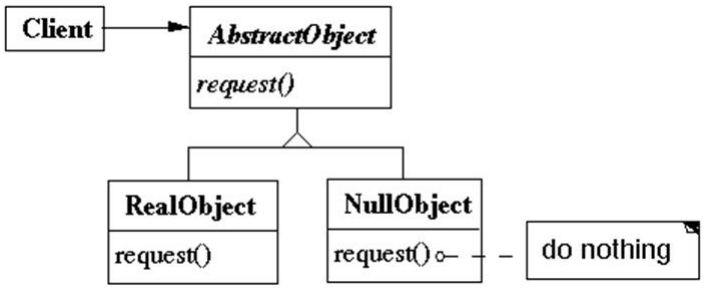

Figura 15: Estrutura geral do padrão de projeto Null Object
Fonte: SourceCodeMania [38]

No contexto de versões da API, os objetos nulos serão instanciados quando uma versão da API necessária para usar algum recurso não estiver disponível no aparelho, já as objetos reais contém as implementações específicas necessárias para cada versão.

A aplicação que utilizou esse padrão de projeto, C:geo, definiu uma instância do padrão para cada versão relevante, níveis 11, 13 e 19. Os serviços dessas instâncias são disponibilizadas para os clientes através de uma fachada (`Compatibility`). A figura 16 mostra um diagrama de classes dessa estrutura.

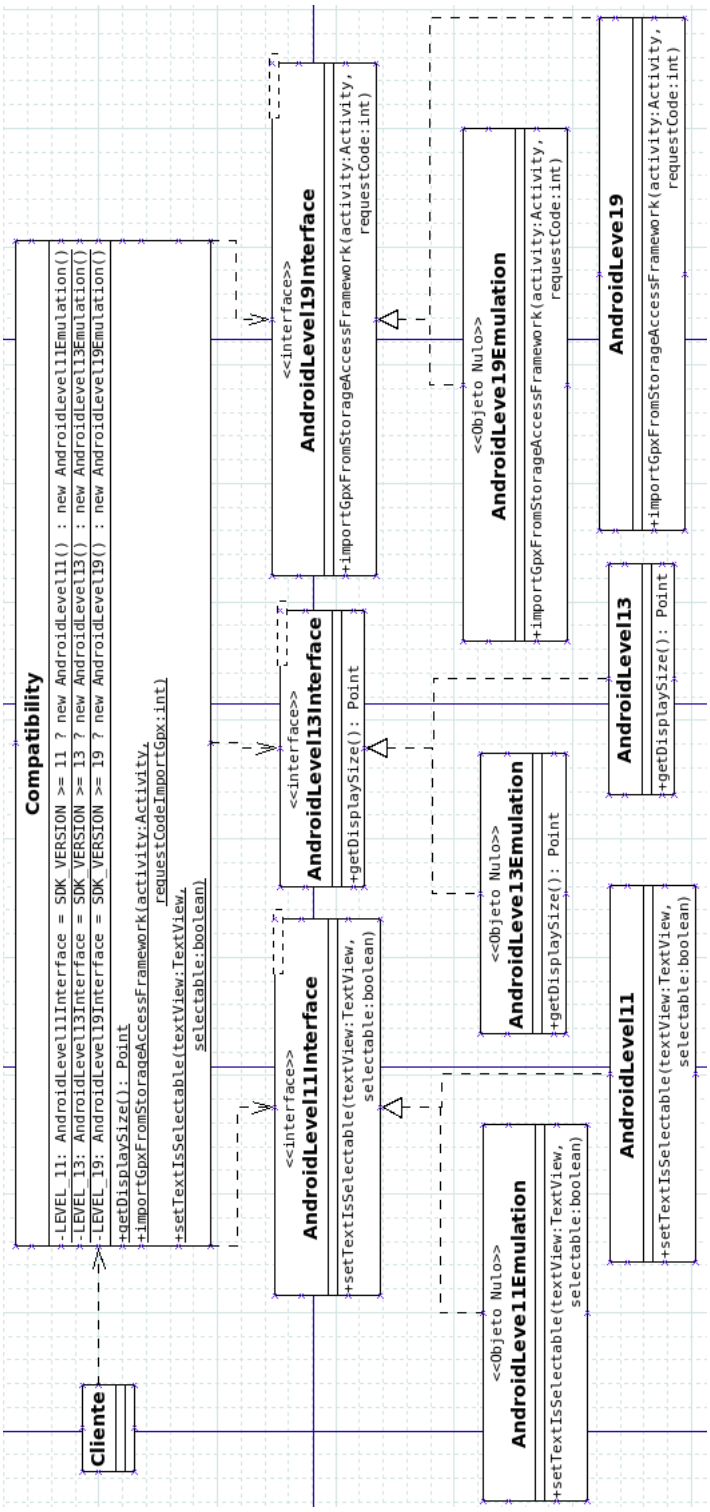

Figura 16: Diagrama de classes do padrão de projeto Null Object aplicado no C:geo

2.2.4 Proxy

O padrão de projeto *Proxy* "fornece um substituto de outro objeto para controlar acesso a este" [32]. A figura 17 apresenta uma visão geral do padrão. `Client` é o cliente que usa os serviços de uma classe abstrata ou interface (`Subject`), que terá duas implementações: o proxy (`Proxy`) e o objeto real (`RealSubject`). O proxy irá receber do cliente as chamadas de métodos e, mediante alguma condição, delegar a chamada para o objeto real.

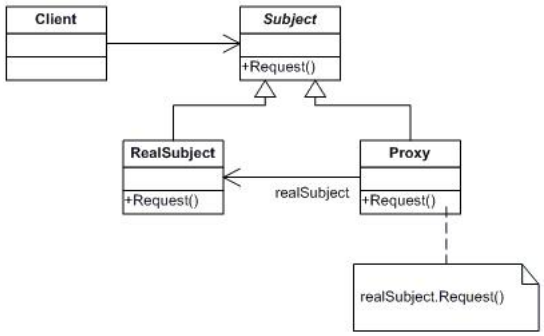

Figura 17: Estrutura geral do padrão de projeto Proxy
Fonte: DoFactory [39]

Existem três tipos de proxies:

- Proxies remotos: responsável por codificar uma requisição e enviar a requisição codificada quando o cliente e o objeto real estão em espaços de endereços diferentes;
- Proxies virtuais: podem fazer cache de informações adicionais sobre o objeto real, para adiar o acesso a eles;
- E, proxies de proteção: checa se o cliente tem as permissões de acesso necessárias para realizar a requisição.

No contexto de versões de API, os proxies são de proteção. Sendo a existência da versão esperada a "permissão" necessária para acesso ao objeto real.

As aplicações que utilizaram esse padrão de projeto definiram proxies para acessar objetos do tipo `View`. Esses objetos são instanciados pela própria plataforma e, por isso, não é possível resolver essa questão da versão na criação, somente no momento da chamada.

2.3 Taxonomia dos Componentes da API

McDonnell et al. [9] definiu algumas palavras chaves para categorizar componentes da API. Por exemplo, o termo `text` engloba todos os pacotes relevantes para renderizar ou manipular texto em dispositivos Android: `android.text.format`, `android.text.method`, `android.text.style` e `android.text.util`.

No entanto, identificamos alguns problemas de concordância com essa taxonomia. Por exemplo, são apresentados os termos `sip` e `rtp`, que referem-se aos pacotes `android.net.sip` e `android.net.rtp`, respectivamente. Mas estes em si não deveriam ser termos da taxonomia. Deveriam fazer parte do termo `net`. Então, entramos em contatos com os autores que nos confirmaram que realmente deveriam estar em `net`, e não se lembravam o porquê de estarem separados. A possibilidade, segundo os autores, era que teriam considerados esses casos como categorias muitos diferentes e por isso ficaram separados.

Assim, neste trabalho, consolidamos a taxonomia da forma original e sem exceções ou casos especiais: `android.TAXONOMIA.PACOTE`. Os pacotes de compatibilidade também seguem esse padrão: `android.support.TAXONOMIA.PACOTE` ou `android.support.vX.TAXONOMIA.PACOTE`, onde `X` é um número que compõe o nome do pacote. Além dos elementos específicos da plataforma Android, existem elementos do SDK padrão do Java, por exemplo `java.net` ou `java.xml`. Para diferenciar estes casos, consideramos como termos da taxonomia o primeiro e segundo nível dos nomes dos pacotes. A tabela 1 apresenta os termos relevantes para este trabalho, seu significado e a versão da API em que a taxonomia surgiu.

Tabela 1: Taxonomia dos componentes da API relevantes para este trabalho

Taxonomia	**Significado**	**API**
animation	Funcionalidades para o sistema de animação de propriedades	11
app	Classes de alto nível que encapsulam todo o modelo de aplicações Android	1
appwidget	Componentes necessários para criação de "app widgets"	3
content	Permite acessar e publicar dados em um dispositivo	1
database	Manipulação de dados retornados através de um provedor de conteúdo	1
graphics	Ferramentas gráficas de baixo nível, como canvas, filtros de cor, pontos e retângulos que permitem desenhar a tela diretamente	1
hardware	Acesso a funções de hardware, como câmera e outros sensores	1
media	Gerência de diversas interfaces de mídia em áudio e vídeo	1
net	Funcionalidades para acesso a rede	1
nfc	Funcionalidades para acesso a NFC (Near Field Communication)	9
opengl	Utilitários e interface para OpenGL ES	1
os	Serviços básicos do sistema operacional, passagem de mensagens, e comunicação inter-processo no dispositivo	1
preference	Gerência de preferências da aplicação	1
print	Funcionalidades para impressão	19
provider	Acesso aos provedores de conteúdo fornecidos pela plataforma	1
service	Serviços diversos, sem nenhuma relação entre eles	
speech	Funcionalidade de reconhecimento de voz	3
telephony	Monitoramento de informações básica do telefone, como tipo de rede e estado da conexão, além de utilitários para manipulação do número do telefone	1
text	Renderização e manipulação de textos na tela	1
transition	Permite funcionalidade de "cenas e transições" para hierarquias de view	19
util	Utilidades diversas, como manipulação de data/hora, codificadores e decodificadores de base64, conversão de string e números, e utilitários para XML	1
view	Manipulação básica de layout de tela e interação com o usuário	1
webkit	Ferramentas para navegação na web	1
widget	Elementos de interface com usuário (maioria visuais) para tela da aplicação	1
java.io	Funcionalidades de entrada e saída de fluxo de dados, serialização e sistemas de arquivos	1
java.lang	Fundamentos para o projeto da linguagem Java	1
java.net	Funcionalidades para o acesso a rede	1
java.util	Utilitários diversos e variados	1

3 Estudo de Análise da Compatibilidade a Múltiplas Versões da API Android

Este capítulo apresenta o estudo realizado para análise do suporte de aplicações existentes à múltiplas versões da API da plataforma Android. Foram analisados o código-fonte de 41 aplicações únicas com o objetivo de identificar e caracterizar as técnicas de implementação de variabilidades.

O restante do capítulo está organizado da seguinte maneira: a seção 3.1 destaca os objetivos do estudo e as questões de pesquisa. Proposições e hipóteses são formalizadas na seção 3.2. A seção 3.3 apresenta os critérios de seleção das aplicações alvo do estudo, bem como as aplicações selecionadas. A seção 3.4 descreve os procedimentos adotados para responder cada questão de pesquisa. A seção 3.5 apresenta os resultados do estudo, que são discutidos na seção 3.6. Por fim, as ameaças à validade são reportadas na seção 3.7.

3.1 Objetivos do Estudo e Questões de Pesquisa

Nosso estudo foi realizado com o objetivo de caracterizar, entender e analisar como as aplicações Android lidam com múltiplas versões da API da plataforma, além do impacto deste suporte no código-fonte das aplicações. Para atingir este objetivo, o estudo foi guiado pelas seguintes questões de pesquisa (QPs):

- **QP1**: Quais são as técnicas que aplicações atuais usam para manter compatibilidades com as diversas versões da API do Android? O objetivo dessa questão é indicar quais as técnicas vem sendo utilizadas e suas características, auxiliando desenvolvedores a decidirem o que e quando utilizar em suas aplicações.
- **QP2**: Qual o subconjunto de novas funcionalidades da plataforma que são mais

utilizadas por aplicações compatíveis com versões antigas da API? Esse levantamento auxilia desenvolvedores de aplicações Android a identificar os elementos que eles, provavelmente, precisarão utilizar em suas aplicações, permitindo a concentração de esforços nesses componentes.

- **QP3**: Qual o esforço necessário para aumentar a compatibilidade da aplicação com um maior número de APIs de versões anteriores da plataforma? Estabelecer uma versão elevada de API para execução do aplicativo pode significar deixar de atender um grande mercado potencial. Assim, a resposta para essa questão dará aos desenvolvedores um indicativo estimado do trabalho que terão para aumentar seu mercado potencial.

- **QP4**: Qual a incidência de código morto em função da versão da API do Android? A evolução da API mínima exigida pelas aplicações, aqui no sentido de edição da configuração no arquivo de manifesto, associada à execução condicional pode resultar em código-morto. Analisamos as aplicações para determinar se isso tem ocorrido com frequência e em qual volume.

3.2 Proposições e Hipóteses

A partir das questões de pesquisas apresentadas anteriormente, algumas proposições e hipóteses foram elaboradas para validação de que as respostas a tais questões podem de fato serem úteis para academia e indústria de desenvolvimento de aplicações Android. A estrutura de proposições e hipóteses baseia-se nas orientações do guia proposto por Runeson et al. [40] e seguem listadas abaixo.

3.2.1 Relacionada à QP1

- **Proposição P1**: Se a plataforma oferece suporte para múltiplas versões da API e tal suporte tem impacto no mercado em potencial então os desenvolvedores utilizam tal recurso.

- **Hipótese H1**: As aplicações utilizam pelo menos uma das técnicas disponíveis para oferecer suporte a múltiplas versões da API.

3.2.2 Relacionada à QP2

- **Proposição P2**: Se a cada nova versão da plataforma são disponibilizados diversos novos recursos então deve existir um subconjunto que é mais utilizado no contexto de suporte a múltiplas versões da API.
- **Hipótese H2**: Existe um subconjunto dos novos recursos da API que é mais utilizado no contexto de suporte a múltiplas versões da API.

3.2.3 Relacionada à QP3

- **Proposição P3**: Se as aplicações utilizam apenas APIs mais recentes, não possuindo um histórico de uso de APIs antigas, então elas foram desenvolvidas sem a preocupação de oferecer suporte às versões antigas da API o que, possivelmente, a leva a perder uma fatia do mercado de aplicações.
- **Hipótese H3**: As aplicações podem aumentar o seu mercado potencial com baixo esforço em termos de desenvolvimento (alterações em seu código-fonte).

3.2.4 Relacionada à QP4

- **Proposição P4**: Se as aplicações utilizam execução condicional para execução de determinados trechos de códigos e a sua API mínima exigida é superior à necessária para execução do trecho de código, então tal trecho de código é código-morto.
- **Hipótese H4**: As aplicações possuem código-morto, que só seriam executados na presença de APIs antigas.

3.3 Aplicações Alvo do Estudo

As aplicações utilizadas no estudo são aplicações *open-source* populares, com um mínimo de 6 mil linhas de código, contabilizados arquivos Java e XML, e de categorias diversas. Uma das aplicações possui 10 mil *downloads* na Google Play, outra possui 50 mil *downloads* e as demais mais de 100 mil.

A natureza das nossas questões de pesquisa exigiu a análise de grupos diferentes de aplicações para determinadas questões, com critérios de seleção adicionais aos citados anteriormente. Assim, definimos 3 grupos de aplicações, apresentados a seguir.

3.3.1 Grupo 1 - Aplicações usando API da plataforma com grande variação de versões

Aplicações do grupo 1 possuem uma diferença de pelo menos 7 versões entre a API mínima e a API alvo. Tais aplicações foram utilizadas para responder às questões QP1 e QP2 e são listadas na tabela 2. São 25 aplicações nesse grupo.

3.3.2 Grupo 2 - Aplicações usando apenas APIs mais recentes

Aplicações do grupo 2 possuem um alto valor de API mínima mas não tem um histórico de suporte às versões anteriores. Para esse estudo, as versões iguais ou superiores à 16 foram consideradas como alta. Versões inferiores a esta representam 3.4% do mercado mundial, conforme pode ser visto na figura 3. E para caracterizar ausência de um histórico de suporte às versões anteriores, consideramos aquelas cuja API mínima inicial seja igual ou superior a 11. Tais aplicações foram utilizadas para responder à questão QP3 e estão listadas na tabela 3. São 10 aplicações nesse grupo.

3.3.3 Grupo 3 - Aplicações usando APIs recentes com histórico de uso de APIs antigas

Aplicações do grupo 3 possuem um alto valor de API mínima e também um histórico de suporte às versões anteriores. Para esse grupo de aplicações, consideramos aquelas cuja versão da API seja igual ou superior a 15 e a API mínima inicial menor ou igual a 8. Tais aplicações foram utilizadas para responder à questão QP4 e estão listadas na tabela 4. São 7 aplicações nesse grupo.

Tabela 2: Aplicações usando API da plataforma com grande variação de versões

Aplicação	Categoria	Downloads	API Mínima	API Alvo	KLoc
AnkiDroid	Educação	1.000.000	10	22	89
AntennaPod	Mídia e Video	100.000	10	23	65
AnySoftKeyBoard	Ferramentas	1.000.000	7	23	180
BatteryBot Battery Indicator	Ferramentas	5.000.000	7	22	6
C:geo	Entretenimento	1.000.000	9	21	123
ConnectBot	Comunicação	1.000.000	4	22	24
Document Viewer	Produtividade	500.000	8	22	77
DuckDuckGo Search & Stories	Livros e referências	1.000.000	8	23	23
Firefox	Comunicação	100.000.000	15	22	233
Free Mobile Netstat	Ferramentas	100.000	8	23	6
Google I/O	Livros e Referências	500.000	14	22	52
K-9 Mail	Comunicação	5.000.000	15	22	117
MyExpenses	Finanças	100.000	8	23	80
OpenTasks	Produtividade	100.000	8	22	23
Persian Calendar	Produtividade	100.000	8	23	10
Quick Dice Roller	Ferramentas	100.000	4	21	35
Shattered Pixel Dungeon	RPG	500.000	8	23	58
Simon Tatham's Puzzles	Quebra-cabeças	100.000	7	23	8
Simple Last.fm Scrobbler	Música e áudio	500.000	7	22	10
Telegram	Comunicação	100.000.000	9	23	169
Terminal Emulator for Android	Ferramentas	10.000.000	4	22	18
VLC for Android	Reproduzir e editar vídeos	50.000.000	8	23	65
Vuze Remote	Ferramentas	100.000	7	23	20
Wifiﬁxer	Ferramentas	1.000.000	7	23	9
Zmanim	Estilo de Vida	10.000	10	22	25

Tabela 3: Aplicações usando apenas APIs mais recentes

Aplicação	**Categoria**	**Downloads**	**API Mínima Inicial**	**API Mínima Atual**	**KLoc**
AcDisplay	Personalização	1.000.000	19	16	40
Dash Clock	Personalização	1.000.000	17	17	15
Focal (Beta)	Fotografia	500.000	16	16	16
Hangar - Smart app shortcuts	Ferramentas	50.000	16	16	12
Indic Keyboard	Ferramentas	100.000	14	16	81
Numix Circle icon pack	Personalização	1.000.000	11	16	8
SnoopSnitch	Ferramentas	100.000	14	16	17
Termux	Ferramentas	100.000	21	21	10
WiFiAnalyzer (open-source)	Ferramentas	100.000	22	21	12
Yaac - IRC Client	Comunicação	50.000	16	16	12

Tabela 4: Aplicações usando APIs recentes com histórico de uso de APIs antigas

Aplicação	**Categoria**	**Downloads**	**API Mínima Inicial**	**API Mínima Atual**	**KLoc**
AFWall+	Ferramentas	500.000	7	15	32
aMetro	Mapas e Navegação	100.000	3	15	10
K-9 Mail	Comunicação	5.000.000	3	15	117
Orbot Proxy com Tor	Comunicação	5.000.000	4	16	33
Ringdroid	Reproduzir e Editar Vídeos	50.000.000	4	16	6
Transdrone	Ferramentas	100.000	4	15	36
Vanilla Music	Música e Áudio	500.000	3	15	25

3.4 Procedimentos

Uma vez selecionadas as aplicações alvos do estudo, foi realizada a atividade de análise das aplicações para buscar respostas às questões de pesquisa. A seguir, descrevemos os procedimentos adotados.

3.4.1 Questões QP1 e QP2

Para responder às questões de pesquisa QP1 e QP2, os procedimentos adotados foram os seguintes:

1. Remoção das anotações `@TargetAPI`, `@SuppressWarnings`, `@SuppressLint` do código-fonte da aplicação. Essas remoções foram necessárias porque tais anotações servem para silenciar o Lint, omitindo ocorrências do relatório;
2. Execução do Android Lint na aplicação;
3. Para cada ocorrência de NewApi, o contexto era analisado de forma manual para determinar se:
   - Trata-se da implementação do padrão Execução Condicional [30];
   - Implementação é baseada em padrões de projeto [32];
   - Ou algum outro mecanismo que evitasse a chamada em uma versão da API cujo recurso está ausente;
4. Para cada ocorrência de NewApi, de forma semiautomática, o elemento da API era classificado de acordo com a taxonomia apresentada na seção 2.3;
5. Nos aplicativos que utilizam pacote de compatibilidade, os elementos foram classificados na taxonomia com uma busca pelos `import`'s presentes nas classes das aplicações.

3.4.2 Questão QP3

Para responder à questão de pesquisa QP3, os procedimentos adotados foram os seguintes:

1. Remoção das anotações `@TargetAPI`, `@SuppressWarnings`, `@SuppressLint` do código-fonte da aplicação;
2. Execução do Android Lint na aplicação;
    - Esse resultado era salvo para uso futuro;
3. Edição da API mínima da aplicação para a versão imediatamente inferior à atual e presente na tabela 3;
    - APIs ausentes dessa tabela foram descartadas por não representarem um mercado minimamente significativo;
4. Nova execução do Android Lint na aplicação;
    - Esse resultado era comparado com o obtido em 2;
    - Se a quantidade de NewApi fosse a mesma, voltávamos para 3, se não o resultado era analisado em maiores detalhes, como a taxonomia, a dependência do recurso pela aplicação e como a ocorrência poderia ser tratada.

3.4.3 Questão QP4

Para responder à questão de pesquisa QP4 os procedimentos adotados foram os seguintes:

1. Pesquisa textual por `VERSION.SDK_INT` em todos os arquivos da aplicação;
2. Quando `VERSION.SDK_INT` fazia parte de um execução condicional, ou equivalente, relacionada a uma versão da API inferior à API mínima da aplicação, o código da aplicação era refatorado de forma a remover todo o código-morto.
3. Contabilização e sumarização das linhas de código e arquivos removidos.

O apêndice A apresenta informações adicionais sobre as aplicações analisadas e como acessar dados brutos obtidos durante o estudo.

3.5 Resultados do Estudo

Essa seção apresenta os resultados do estudo. As subseções estão organizadas em termos das questões de pesquisa (Seção 3.1).

3.5.1 Quais são as técnicas que aplicações atuais usam para manter compatibilidades com as diversas versões da API do Android?

Para permitir o uso de novos recursos da API em aplicações que ainda desejam oferecer suporte a dispositivos com versões mais antigas da API, foram identificadas 3 estratégias de implementação: (i) pacote de compatibilidade; (ii) re-implementação de recurso da API e (iii) uso explícito da nova API. Tais mecanismos podem ser utilizados em conjunto.

A figura 18 apresenta a quantidade total de aplicações que utilizam cada estratégia. Pacote de compatibilidade foi utilizada por 23 aplicações, re-implementação de recurso por 14, e uso explícito da nova API por 22.

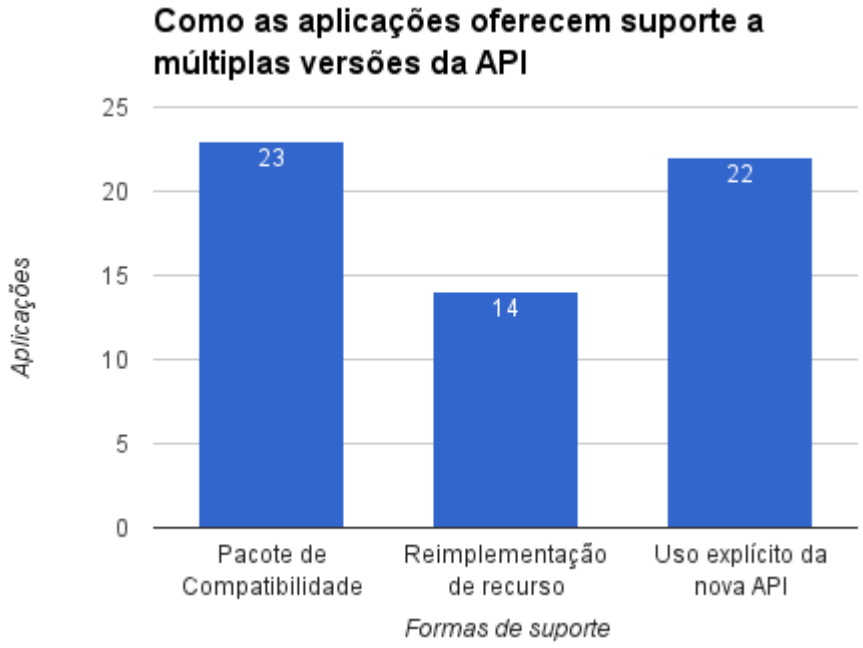

Figura 18: Como as aplicações oferecem suporte a múltiplas versões da API

A técnica de uso explícito da nova API pode causar um erro na aplicação caso seja executado em um dispositivo com versão da API inferior àquela em que o componente utilizado foi adicionado. Assim, é necessário algum mecanismo para evitar que isso aconteça. O mecanismo primário padrão é um comando condicional. No entanto, existem diversas maneiras de implementação de tal comando condicional, desde uma instrução `if` logo acima da linha de código, até a utilização de padrões de projeto, em que o `if` determina a criação de objetos baseado na versão da API Android do dispositivo, por exemplo.

O gráfico apresentado na figura 19 mostra as diferentes alternativas de projeto encontradas nas aplicações analisadas para implementação de execução condicional em torno das linhas com ocorrências de Lint NewApi. Tais resultados foram obtidos a partir da análise do resultado da execução do Lint NewApi nas aplicações. Execução condicional [30] indireta e uso de padrões de projeto foram as técnicas mais comum, estando presente em 16 aplicações analisadas. Execução condicional direta e suporte implícito foram utilizadas em 12 aplicações cada. 6 aplicações deixaram de fazer o tratamento de algumas ocorrências, 5 aplicações possuem ocorrências dentro de métodos que nunca serão executados e, por fim, uma aplicação utilizou tratamento de exceção para evitar erro durante a execução.

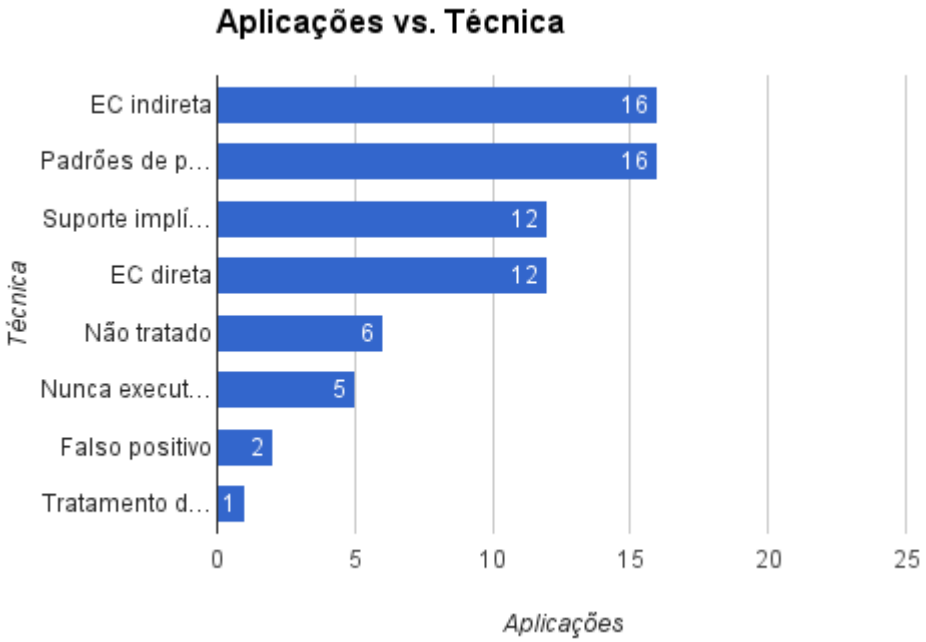

Figura 19: Quantidade de aplicações que utilizaram cada técnica para tratamento de ocorrência de NewApi

Também foram detectadas ocorrências de falsos positivos em duas aplicações. Falso positivo acontece quando um determinado método `X()` qualquer foi adicionado em uma classe da API em versão superior da aplicação, posteriormente uma classe estende essa classe da API e implementa o método `X()`, mas o Lint NewApi não identifica essa implementação, reportando o falso problema.

A figura 20 exemplifica tal situação com um dos casos reais encontrados. O método `invalidadeOptionsMenu()` foi adicionado à classe `Activity` na versão 11 da API, portanto ele não pode ser chamado em versões anteriores. Assim, foi disponibilizado através da re-implementação na classe `AppCompatActivity` do pacote de com-

patibilidade e que herda de `Activity`. Na aplicação Telegram, cuja API mínima é 10, existe uma classe que estende de `AppCompatActivity`, `CastEnableActivity`, que é então estendida por `MediaPlayerActivity`. No método `onOptionsItemSelected()` é feita uma chamada para `invalidadeOptionsMenu()`. Tal chamada é indevidamente indicada pelo Lint.

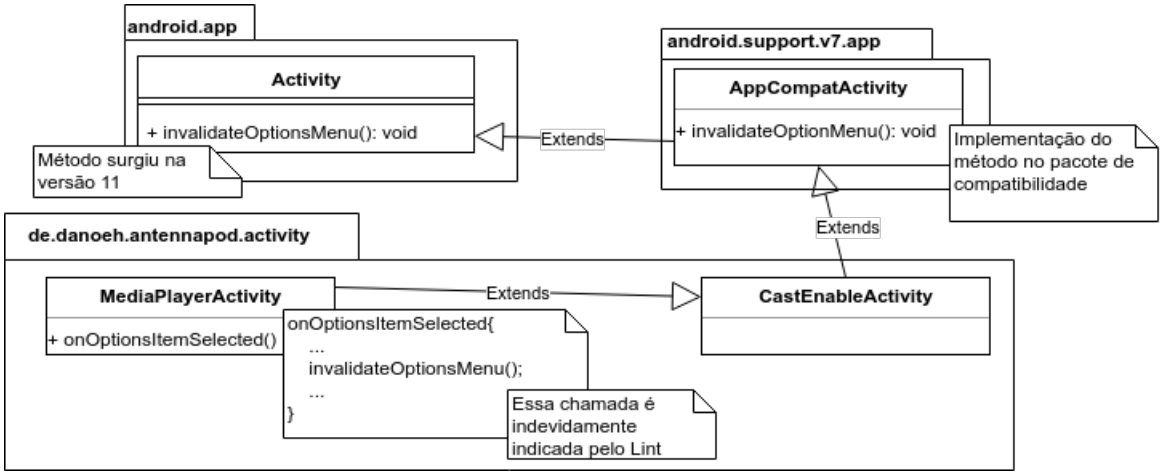

Figura 20: Diagrama de classe exemplificando situação de falso positivo do Lint

A figura 21 apresenta os percentuais das formas de tratamento para as ocorrências de NewApi. Foram analisadas um total de 1556 ocorrências de NewApi. Desse total, 40,6% foram tratados com EC indireta, 22,8% com algum padrão de projeto, 18,9% através de EC direta, 10,7% utilizando suporte implícito da plataforma, 5,3% das ocorrências não foram devidamente tratadas. Continuando a análise, 1,2% das ocorrências estão dentro de métodos que nunca serão executados e, por fim, em duas pequenas frações de 0,3% das ocorrências foram utilizados tratamento de exceção ou são falsos positivos. Essas ocorrências de NewApi estão distribuídas em 22 aplicações (3 aplicações não apresentaram ocorrências de NewApi).

O figura 22 apresenta os padrões de projeto utilizados por quantidade de aplicações. O padrão *Strategy* é o que mais foi utilizado em um total de 12 aplicações. *Template Method* e *Null Object* foram utilizados por uma aplicação cada um. *Proxy* foi utilizado por duas aplicações. Apenas uma aplicação utilizou dois padrões de projetos simultaneamente, *Strategy* e *Proxy*.

O figura 23 apresenta os percentuais dos padrões de projeto utilizados como forma de tratamento para as ocorrências de NewApi. Do total de 1554 ocorrências de NewApi analisadas, 355 (22,8%) foram tratadas com algum padrão de projeto. Sendo o mais comum o padrão *Strategy*, responsável por 77,7% (276) desse total. Os padrões *Proxy* e

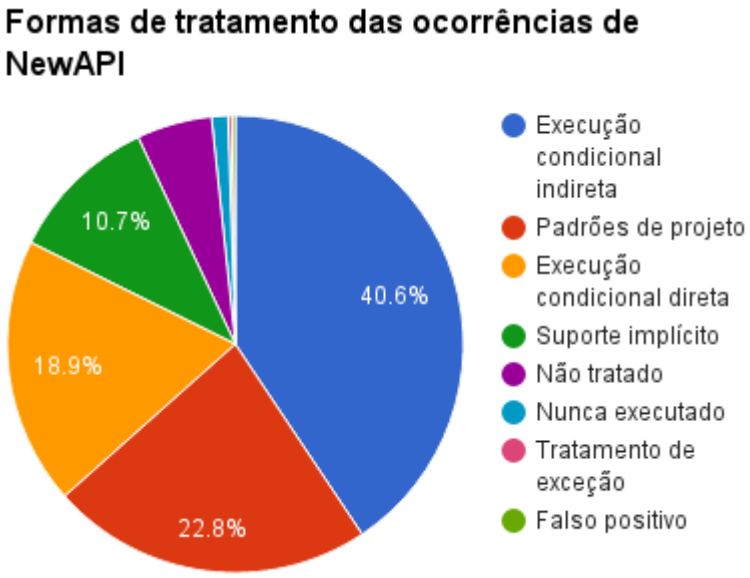

Figura 21: Percentuais de formas de tratamento das ocorrências de NewApi

Template Method foram responsáveis por 19,4% e 2,3% dos casos, respectivamente. *Null Object* foi utilizado por uma aplicação para tratar 2 (0,6%) ocorrências.

3.5.2 Qual o subconjunto de novas funcionalidades da plataforma que são mais utilizados por aplicações compatíveis com versões antigas da API?

Analisamos as chamadas às novas APIs para determinar o subconjunto das novas funcionalidades que são utilizadas pelas aplicações com suporte à versões antigas da API. Os resultados foram contabilizados em função da taxonomia da API. E, então, para as taxonomias mais comuns, apresentamos as classes mais recorrentes.

3.5.2.1 Pacote de Compatibilidade

Para computar os dados relativos à pacote de compatibilidade, realizamos uma análise estática sobre os projetos, identificando nos trechos de importação das classes Java os elementos advindos do pacote `android.support`, excetuando os subpacotes `annotations` e `tests`, pois estes além de não influenciarem na execução da aplicação, não são analisadas pelo Lint NewApi.

A tabela 5 apresenta os valores obtidos. Para cada taxonomia, temos o número de aplicações que fizeram `import`'s do pacote de compatibilidade, o total de `import`'s feitos, as importações únicas e a média de `import`'s por aplicação. Importações únicas

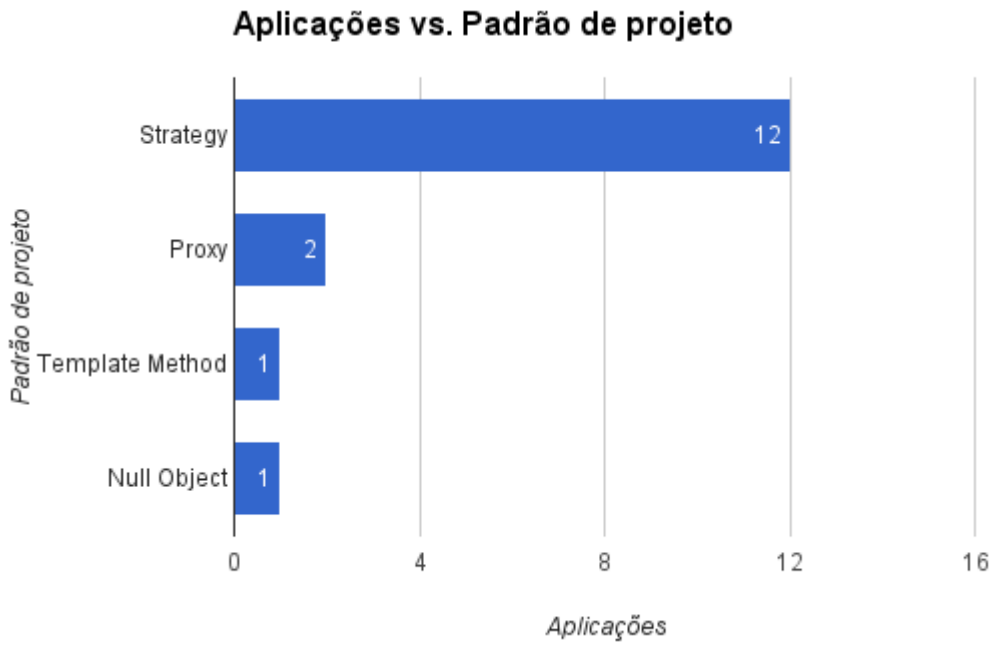

Figura 22: Quantidade de aplicações que utilizaram padrões de projeto para tratar ocorrências de NewApi

significa que se uma classe é importada mais de uma vez, independente de aplicação, ela é contabilizado apenas uma vez. Em destaque os maiores valores de cada coluna, além das duas taxonomias mais significativas, que possuem uma maior média de `import`'s por aplicação.

Tabela 5: Elementos mais utilizados do pacote de compatibilidade

Taxonomia	**Aplicações (A)**	**Total de import's (B)**	**Importações únicas**	**Média por app (A/B)**
animation	1	5	4	5,00
app	**22**	**849**	49	**38,59**
content	17	236	11	13,88
graphics	4	9	5	2,25
hardware	1	2	1	2,00
media	3	15	7	5,00
net	1	3	1	3,00
os	3	6	4	2,00
preference	5	99	17	19,80
provider	1	12	1	12,00
text	1	1	1	1,00
util	9	46	9	5,11
view	18	233	37	12,94
widget	18	410	**65**	22,78

As taxonomias *app* e *widget* são as mais comuns. A primeira contém classes de

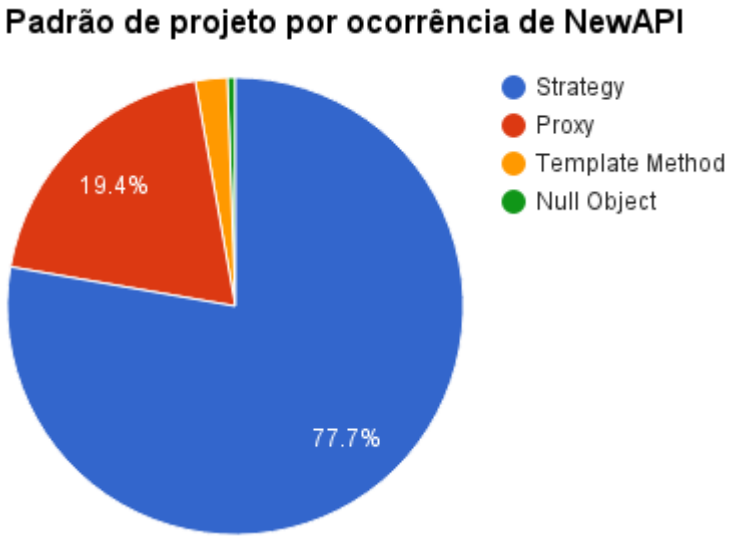

Figura 23: Percentuais de uso de padrões de projeto por ocorrências de NewApi

alto nível que encapsulam o modelo geral de aplicações Android. A última fornece elementos de interface com o usuário para serem utilizados na tela da aplicação. A figura 24 apresenta diagramas de boxplot para as distribuições de importações nas aplicações para essas duas taxonomias.

App possui o maior número importações, 849, e também maior média por aplicação, 39, distribuídos em 22 das 25 aplicações analisadas. Já *widget* possui um número bem menor de importações, 410, embora bem mais alto que a terceira posição. Estando distribuídos em 18 aplicações. Além de possuir o maior número de chamadas únicas, 65. Isso demonstra que os elementos de *app* são mais restritos, enquanto *widget* há uma granularidade maior, havendo um maior acoplamento da aplicação a esses elementos em relação aos elementos de *app*.

Da taxonomia *app*, a classe mais comum é `Fragment`, conforme apresenta a tabela 6. Além disso, outras três classes diretamente relacionadas a fragmentos estão entre as 10 mais utilizadas: `DialogFragment`, `FragmentManager` e `FragmentActivity`.

Se na taxonomia *app* as importações tendem a elementos relacionados a fragmentos, na taxonomia *widget* houve uma maior dispersão. As classes mais comuns são `RecyclerView` e `Toolbar`. A primeira possui o maior número de importações no total, enquanto a segunda, além de ter o segundo maior valor de importações, é a que está presente em mais aplicações, 12 contra 9 para *RecyclerView*. Os demais elementos aparecem em no máximo 9 aplicações, com não mais que 26 importações no total. A tabela 7 apresenta as 10 classes da taxonomia *widget* mais utilizadas.

Tabela 6: 10 classes da taxonomia *app* mais importadas do pacote de compatibilidade

Classes	Total de importações (A)	Aplicações (B)	Média por app (A/B)
v4.app.Fragment	144	15	9,60
v4.app.NotificationCompat	80	16	5,00
v4.app.DialogFragment	76	12	6,33
v7.app.AlertDialog	72	11	6,55
v7.app.AppCompatActivity	67	14	4,79
v4.app.FragmentManager	65	16	4,06
v4.app.FragmentActivity	58	12	4,83
v4.app.LoaderManager	50	7	7,14
v7.app.ActionBar	37	13	2,85
v4.app.ActivityCompat	25	14	1,79

Tabela 7: 10 classes da taxonomia *widget* mais importadas do pacote de compatibilidade

Classes	Total de importações (A)	Aplicações (B)	Média por app (A/B)
v7.widget.RecyclerView	81	9	9,00
v7.widget.Toolbar	52	12	4,33
v7.widget.LinearLayoutManager	26	9	2,89
design.widget.Snackbar	23	6	3,83
v7.widget.SearchView	21	8	2,63
v4.widget.DrawerLayout	15	9	1,67
design.widget.FloatingActionButton	13	4	3,25
v7.widget.PopupMenu	12	5	2,40
design.widget.TabLayout	10	6	1,67
v4.widget.SwipeRefreshLayout	10	5	2,00

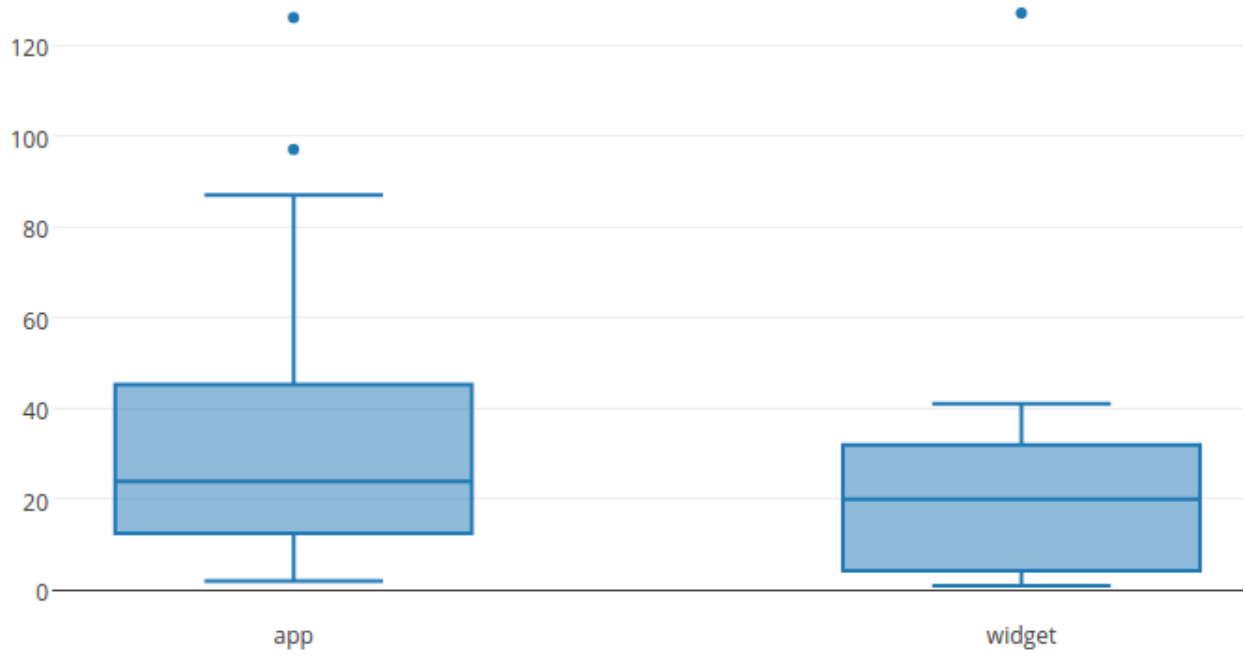

Figura 24: Distribuição de importações para as taxonomias *app* e *widget*

3.5.2.2 Re-implementação de Recursos

Re-implementação de um recurso existente na API padrão pode ser uma re-implementação completamente própria, quando não existe cópia de código de outros projetos, ou pode ser através da cópia do código-fonte do recurso na API padrão. Para este trabalho, consideramos apenas a re-implementação através de cópia do código-fonte da API padrão. O código-fonte do projeto Android é distribuído, preferencialmente, sob a licença Apache Software License, Versão 2.0 ("Apache 2.0") [41], e os direitos autorais pertencem a *Android Open Source Project*. Para indicar isso, todos os arquivos do projeto possuem o seguinte preâmbulo:

```
/*
* Copyright (C) <YEAR> The Android Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
```

```
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
```

Assim, para verificarmos se as aplicações re-implementaram algum recurso através da cópia de código da API padrão, realizamos uma busca textual nos arquivos Java das aplicações por "Android Open Source Project". Para cada ocorrência, foi feita uma pesquisa na documentação da API pelo nome da classe. Se não fosse encontrado algo, era feita uma busca no Google, pela string "android <nome da classe>". Analisamos as ocorrências para determinar se o código havia sido copiado realmente da API, de aplicações nativas do Android ou de projeto de exemplos oficiais, desenvolvidas pela mesma equipe da plataforma. Em algumas situações, os códigos tinham como origem o projeto Android, mas não faziam parte da API. Um exemplo disso é a classe `NetworkActivity` que, embora desenvolvida pelo projeto Android e possuir o preâmbulo citado acima, não faz parte da API. Ela é apresentada na documentação [42] e o código-fonte está no subdiretório `samples` do repositório [43]. Ocorrências como essa eram então descartadas. Embora um número considerável de aplicações (14 de 25) tenha utilizado tal técnica, foram usos bem pontuais e localizados.

O conjunto de classes mais significativo diz respeito à implementação do pacote `android.animation` da API e ocorreu em 3 aplicações. O código presente nas aplicações é proveniente da biblioteca NineOldAndroids[1], que disponibiliza seus arquivos para *download* através do código-fonte e não em arquivos JAR ou algum outro mecanismo para adição de dependência aos projetos, mas que isole o código da dependência do código principal do projeto. Por esse motivo, tal código foi considerado nessa análise. As classe `PreferenceFragment` e `PreferenceManager` foram também re-implementadas por 3 aplicações.

A classe `android.util.Base64`, utilizada para codificação e decodificação da representação de dados binários em Base64 [44], foi a classe mais recorrente, detectada em 5 aplicações. Outras mais de 30 classes com propósitos diversos também foram re-implementadas, no entanto, em não mais de uma aplicação.

[1] http://nineoldandroids.com/

3.5.2.3 Uso explícito da nova API

Para determinar o subconjunto das novas funcionalidades da API que são utilizadas pelas aplicações com suporte a versões antigas da API através de uso explícito da nova API, processamos os resultados da saída do Lint NewApi.

A tabela 8 apresenta os valores obtidos. Para cada taxonomia, temos a quantidade total de ocorrências de NewApi, o número de ocorrências únicas, a quantidade de aplicações em que as ocorrências apareceram e a média de ocorrências por aplicação.

Tabela 8: Ocorrências de NewApi por taxonomia

Taxonomia	Ocorrências de NewApi (A)	Ocorrências Únicas	Aplicações (B)	Média por app (A/B)
animation	149	29	7	21,29
app	**360**	57	16	22,50
appwidget	5	3	2	2,50
content	44	24	11	4,00
database	10	8	4	2,50
graphics	26	14	6	4,33
hardware	3	3	1	3,00
media	190	54	3	**63,33**
net	31	12	2	15,50
nfc	14	7	2	7,00
opengl	25	17	1	25,00
os	82	20	9	9,11
preference	56	4	5	11,20
print	10	7	1	10,00
provider	27	15	7	3,86
service	4	2	1	4,00
speech	3	2	1	3,00
telephony	29	25	2	14,50
text	6	5	5	1,20
transition	4	1	1	4,00
util	8	3	2	4,00
view	353	**153**	**20**	17,65
webkit	11	7	4	2,75
widget	92	41	10	9,20
java.io	3	2	2	1,50
java.lang	3	2	2	1,50
java.net	5	4	1	5,00
java.util	1	1	1	1,00

As taxonomias *app* e *view* foram as mais comuns nas aplicações analisadas, com 360 e 353, respectivamente, ocorrências de NewApi. Ocorrências em arquivos XML

também foram contabilizados, tendo um total de 50 ocorrências. No entanto, a maior média por aplicação pertence a taxonomia *media*, no valor de 63,33, distribuídos em 3 aplicações: VLC, Firefox e Telegram. Enquanto *app* e *view* estão distribuídas em 16 e 20 aplicações, respectivamente.

A figura 25 apresenta diagramas de boxplot para as distribuições de ocorrências de NewApi nas aplicações para essas duas taxonomias.

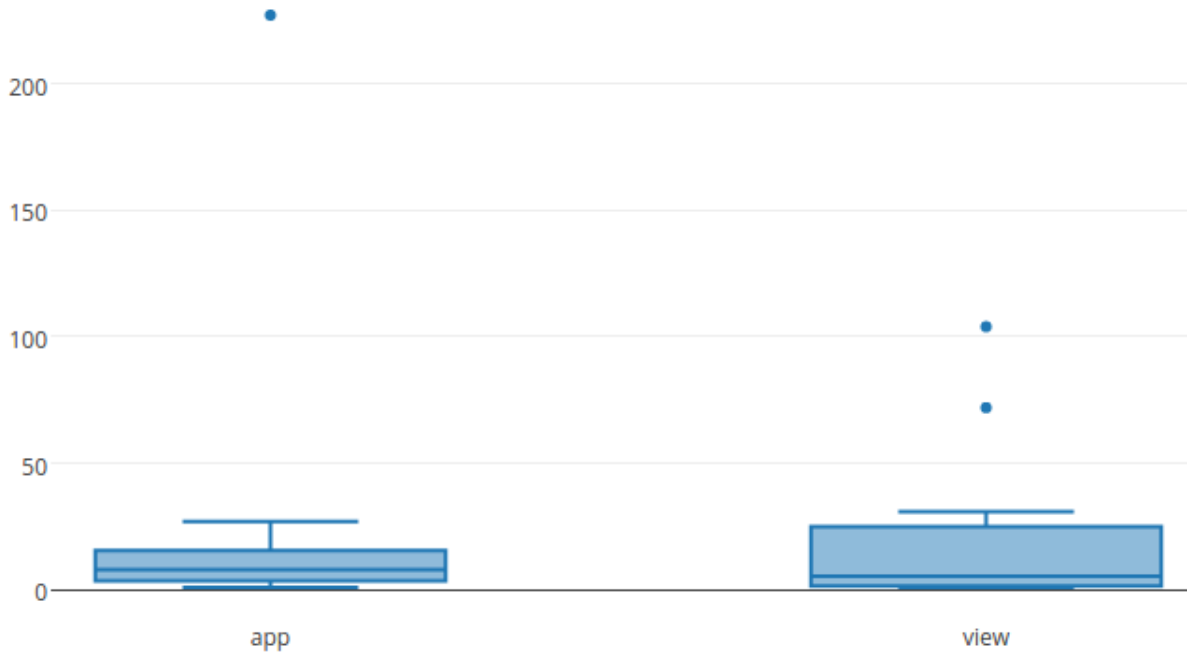

Figura 25: Distribuição de ocorrências de NewApi para as taxonomias *app* e *view*

Tais valores refletem o objetivo das taxonomias e a categoria das aplicações. *App* e *view* contém classes utilizadas em qualquer aplicação, enquanto que a taxonomia *media* mantém classes relacionadas a multimídia. Mais úteis para um reprodutor e editor de vídeos (VLC), um navegador web (Firefox) e um aplicativo de comunicação (Telegram) com recursos de reprodução de multimídia (imagem, som e vídeo) embutido.

A tabela 9 apresenta os elementos da taxonomia *app* com maior números de ocorrência de NewApi agrupados pelo elemento de maior granularidade que surgiu na versão da API indicada por NewApi. Assim, todas as classes do pacote `android.app.backup` foram agrupadas pelo pacote porque o próprio pacote surgiu apenas na versão 5. A classe `android.app.Fragment` surgiu na versão 11, então agrupamos aqui todos os seus métodos. Já a classe `android.app.Activity` existe desde a versão 1 e foram diversos métodos novos apontados por NewApi. Na classe `android.app.Notification`,

que existe desde a versão 1, apenas o método `bigContentView` é apontado por NewApi.

Tabela 9: Elementos da taxonomia *app* com ocorrências de NewApi

Descrição	Ocorrências de NewApi	Aplicações
Classe android.app.Fragment	188	2
Pacote android.app.backup	36	4
Métodos de android.app.Activity	30	8
Classe android.app.Presentation	28	2
Método android.app.Notification#bigContentView	17	1
Classe android.app.ActionBar	12	3
Classe android.app.Notification.Builder	10	1
Classe android.app.FragmentTransaction	9	2
Classe android.app.DownloadManager	9	1
Classe android.app.FragmentManager	7	2

Da taxonomia *app*, a classe mais comum é `Fragment`, conforme apresentado na tabela 9. Além disso, alguns métodos da classe `Activity` nessa relação são relacionadas aos fragmentos, assim como as classes `FragmentTransaction` e `FragmentManager`. Além dos elementos apresentados na tabela 9, outros 7 elementos possuem um valor máximo de 4 ocorrências relacionadas.

Se as ocorrências para a taxonomia *app* ficaram concentradas em elementos relacionados a fragmentos, na taxonomia *view* houve uma maior dispersão. A classe mais comum, tanto em aplicações (12) quanto em ocorrências de NewApi (108), é a classe `View`, classe base para todos os componentes de construção de interface gráfica. Atributos ou tags XML correspondem a 50 ocorrências, distribuídas em 7 aplicações. Os demais elementos, aparecem em no máximo 4 aplicações, o que demonstra serem específico do tipo de aplicação. Por exemplo, as classes `InputMethodSubtype` e `InputMethodManager` juntas somam 12 ocorrências de NewApi, mas todas elas em uma única aplicação, a AnySoftKeyboard, uma aplicação de teclado virtual.

A tabela 10 apresenta os 10 elementos da taxonomia *view* com mais ocorrência de NewApi. Além desses, outros 20 elementos apresentam no máximo 6 ocorrências.

Tanto os elementos relacionados a fragmentos, quanto a grande maioria dos métodos da classe `View` citados anteriormente surgiram na versão 11 da API, e foram uma importante melhoria na plataforma, o que reforça o alto índice de ocorrências de NewApi relacionado a esta versão, conforme apresentado no figura 26. Tais ocorrências estão distribuídas em 18 aplicações.

Tabela 10: 10 elementos da taxonomia *view* com mais ocorrências de NewApi

Descrição	Ocorrências de NewApi	Aplicações
Métodos de android.view.View	108	12
Atributos ou tags XML	50	7
Classe android.view.ViewPropertyAnimator	43	2
Classe android.view.WindowInsets	23	1
Classe android.view.TextureView	21	2
Classe android.view.ScaleGestureDetector	12	3
Classe android.view.ActionMode	11	3
Métodos de android.view.MotionEvent	10	1
Construtor de android.view.ViewOutlineProvider	10	1
Classe android.view.accessibility.AccessibilityNodeInfo	9	1

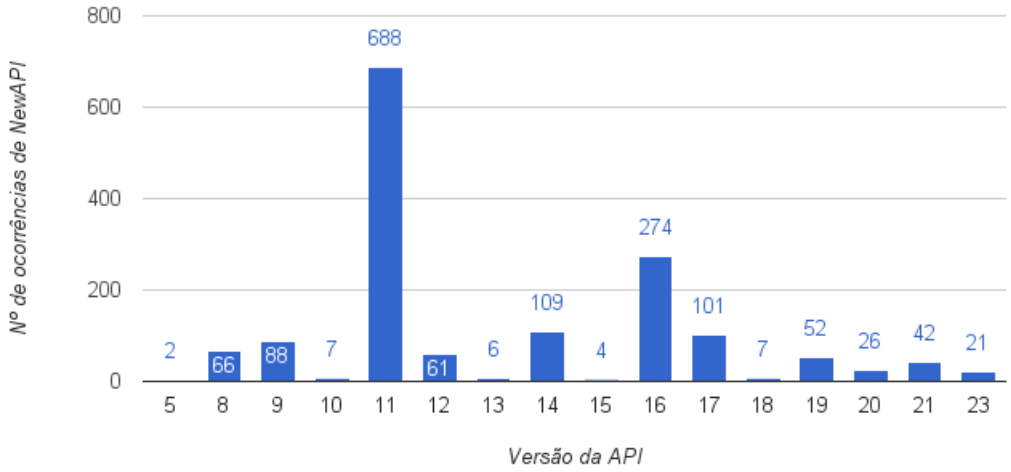

Figura 26: Nº de ocorrências de NewApi por versão da API

3.5.3 Qual o esforço necessário para aumentar a compatibilidade com um maior número de APIs de versões anteriores da plataforma?

Definir um valor de API mínima para execução da aplicação significa deixar de atender todos os aparelhos com API inferior. Em algumas situações, isso pode representar uma perda do mercado potencial. Assim, analisamos qual seria o esforço para aumentar a compatibilidade com um maior número de APIs. Para tal, editamos o código-fonte da aplicação (arquivos `build.gradle` ou `AndroidManifest.xml`, conforme o caso) e re-executamos o Lint NewApi. Então comparamos o resultado aqui obtido com o resultado obtido na versão original.

Inicialmente, identificamos a quantidade de novas ocorrências de NewApi que surgiam com a diminuição da versão da API. Isso fornece um valor quantitativo do

trabalho necessário para aumento da compatibilidade, quanto maior este valor mais adequações seriam necessárias no código-fonte. A tabela 11 apresenta tais resultados. Para cada aplicação, indicamos situação atual, com a API mínima atual e a quantidade de ocorrências de NewApi, a situação após nova edição, com a nova API e a quantidade de ocorrências de NewApi, e a diferença de ocorrência de NewApi, com a diferença total de novas ocorrências e dessas a quantidade de ocorrências únicas. Essas duas últimas colunas mensuram quantitativamente o esforço necessário para aumentar a compatibilidade.

Tabela 11: Esforço quantitativo para aumentar compatibilidade com mais versões da API

Aplicação	**Situação atual**		**Situação após edição**		**Diferenças de ocorrências de NewApi**	
	API Mínima	**Ocorrências de NewApi**	**API Mínima**	**Ocorrências de NewApi**	**Diferença total**	**Ocorrências únicas**
Numix Circle icon pack	16	0	15	1	1	1
SnoopSnitch	16	19	15	15	2	1
Yaaic - IRC Client	21	0	15	1	1	1
Hangar - Smart app shortcuts	16	14	15	21	7	5
WiFi Analyzer	22	0	21	1	1	1
AcDisplay	16	167	15	193	26	10
Indic Keyboard	16	7	15	16	9	1
Focal	16	3	15	14	11	5
DashClock Widget	17	5	16	46	41	9
Termux	21	2	19	59	57	9

Posteriormente, analisamos cada uma dessas ocorrências para identificar como ela poderia ser tratada e qual o impacto para a aplicação. Das 10 aplicações analisadas, 5 delas podem aumentar a compatibilidade com poucos e simples ajustes, fazendo uso de pacote de compatibilidade e/ou execução condicional, sem perda de funcionalidade. Quatro dessas aplicações, poderiam passar a exigir API 15, no lugar da atual 16. Mas uma delas, Yaaic, poderia substituir a API 21 pela API 15, com apenas 1 ocorrência de NewApi a mais. Tal ocorrência existe devido ao uso do tema Material[45], que pode ser tratado através do mecanismos de recursos da plataforma, bastando definir estilos gerais, para versões inferiores a 21, e um estilo específico (style-v21) para

versão 21 ou superior. Nesta aplicação, também é possível ir além, e prover compatibilidade com a versão 10 da API. Em que 8 novas ocorrências de NewApi surgiriam, mas poderiam ser resolvidas com pacote de compatibilidade.

Outras 2 aplicações - Indic Keyboard e AcDisplay - precisam tratar, respectivamente, 9 e 26 novas ocorrências de NewApi para alterar a API de 16 para 15. A primeira teria que apenas fazer uma re-implementação da classe `SentenceSuggestionsInfo`, surgida na API 16. Todas as 9 ocorrências de NewApi são referências para essa classe. A segunda teria que tomar providências diversas, como pacote de compatibilidade, execução condicional e re-implementação de recursos. Mas todas elas possíveis de resolução sem perda de funcionalidade na aplicação.

As três aplicações restantes usam funcionalidades mais críticas. Focal, uma aplicação de fotografia, apresenta um total de 11 ocorrências, sendo que uma delas parece não haver solução possível, pois depende de suporte do *hardware* do aparelho para auto focus da câmera. DashClock Widget apresentou 41 novas ocorrências ao reduzir a versão da API de 17 para 16. A maioria pode ser resolvida pelos métodos já citados, 17 delas são referências a classe `DreamService`, uma classe específica para realidade virtual. Portanto, a aplicação poderia fornecer as demais funções mas omitir esta específica. Por fim, a aplicação Termux, um emulador de terminal, é a que possui a maior quantidade de ocorrências de NewApi, ao reduzir a versão da API de 21 para 19: 57 ocorrências. A maioria facilmente tratada pelos métodos já citados. Porém, 9 delas fizerem referência a classe `android.system.Os`, as quais são funcionalidades fundamentais para a aplicação, de acesso ao sistema. Distribuir a aplicação sem tais recursos seria uma perda considerável. Essa classe expõe para a API pública do Android uma classe pertencente ao pacote `libcore`, o qual não pode ser importado pelas classes das aplicações da mesma forma que classes do pacote `android`. No entanto, é possível através de uma re-implementação de recurso mais elaborada que as citadas anteriormente, usando reflexão, acessar tal pacote e classes internas [46], e então fazer uma implementação própria da classe `Os`, ainda que fazer uso de APIs internas seja uma má prática de programação e existam recomendações contrárias à sua adoção [47] [48].

A tabela 12 resume o resultado qualitativo, agrupando as aplicações em três níveis de dificuldade e possíveis soluções para as novas ocorrências de NewApi. Os níveis de dificuldade foram definidos da seguinte forma:

1. Baixo: necessário apenas uso de pacote de compatibilidade, execução condici-

onal ou uso de estilo especializado;

2. Médio: necessário re-implementação simples de recurso. Pode-se, por exemplo, copiar classes do código-fonte da API;

3. Alto: necessário alguma implementação de recurso mais elaborada ou ausência do recurso em aparelhos com API mais antiga. Deve-se, por exemplo, usar reflexão para ter acesso à API interna.

Os resultados encontrados, associado à tabela de distribuição das versões (figura 3), apresenta evidências de que as aplicações poderiam, com um esforço de baixo a médio, aumentar a compatibilidade com um maior número de versões da API.

Tabela 12: Nível de dificuldade para aumentar a compatibilidade e possíveis soluções para as novas ocorrências de NewApi

Aplicação	**Nível de Dificuldade**	**Possíveis Soluções**
Numix Circle icon pack	Baixo	- Execução condicional - Pacote de compatibilidade - Uso de estilo especializado
SnoopSnitch		
Yaaic - IRC Client		
Hangar - Smart app shortcuts		
WiFi Analyzer		
AcDisplay	Médio	- Algumas ocorrências são ignoradas (atributos XML) - Pacote de Compatibilidade - Reimplementação de recurso
Indic Keyboard		
Focal (Beta)	Alto	- Pacote de compatibilidade - Reimplementação de recurso - Execução condicional - Versão sem recurso
DashClock Widget		
Termux		

3.5.4 Qual a incidência de código morto em função da versão da API do Android?

A técnica básica de prover suporte a multi-versões da API é o uso de execução condicional avaliando a versão da API do aparelho. Também faz parte da evolução natural das aplicações o fim de suporte a versões mais antigas da API. Dessa forma, é possível que os desenvolvedores alterem a configuração de API mínima no `AndroidManifest.xml`,

mas deixem ao longo do código-fonte da aplicação trechos de código que seriam executados apenas em versões de API inferiores a esta. Nessa situação, tais trechos de códigos são caracterizados como código-morto.

Para esta questão de pesquisa, foram analisadas 7 aplicações e foi encontrado código-morto em 4 delas. No entanto, em um valor proporcional muito pequeno em relação ao total de linhas de código da aplicação. Mesmo considerando apenas o código Java, identificamos uma média de apenas 0,09% de linhas de código-morto. A tabela 13 apresenta detalhes dessa análise. São exibidos as métricas de número de arquivos e total de linhas de código-fonte tanto com código-morto quanto com a remoção do código morto, assim como a diferença em valores absolutos e percentuais.

Tabela 13: Ocorrências de código morto (CM) por aplicação

Aplicação	**Métricas COM CM**		**Ocorrências de CM**		**Métricas SEM CM**		**Diferenças**		**Percentuais**	
	Arquivos	**LoC**	**#**	**Arquivos**	**Arquivos**	**LoC**	**Arquivos**	**LoC**	**Arquivos**	**LoC**
RingDroid	12	4189								
Orbot Proxy com Tor	86	11147	3	3	86	11132	0	15	0,00%	0,13%
Transdrone	243	23464								
Vanilla Music	90	17404	3	1	90	17398	0	6	0,00%	0,03%
AFWall+	84	14560	4	4	84	14540	0	20	0,00%	0,14%
aMetro	133	8167								
K-9 Mail	484	78984	8	8	484	78934	0	50	0,00%	0,06%
								Média	**0,00%**	**0,09%**

Tais resultados apresentam indícios que é possível encontrar código-morto em aplicações Android existentes. No entanto, em comparação com a quantidade de linhas de código total da aplicação, a quantidade de linhas de código morto é irrelevante, conforme mostra a tabela 13.

3.6 Discussão

Os dados coletados no estudo de caso e as análises realizadas nas seções anteriores são fonte de evidências para a constatação das hipóteses. Cada questão de pesquisa será abordada separadamente a seguir, juntamente com sua hipótese relacionada.

3.6.1 Quais são as técnicas que aplicações atuais usam para manter compatibilidades com as diversas versões da API do Android?

- **Proposição P1**: Se a plataforma oferece suporte para múltiplas versões da API e tal suporte tem impacto no mercado em potencial então os desenvolvedores utilizam tal recurso.
- **Hipótese H1**: As aplicações utilizam pelo menos uma das técnicas disponíveis para oferecer suporte a múltiplas versões da API.

No nosso estudo, das 25 aplicações analisadas: (i) apenas 2 utilizam uma única estratégia, (ii) 9 aplicações utilizam 2 estratégias; e (iii) 13 fazem uso das 3 estratégias diferentes. Finalmente, apenas a aplicação *Shattered Pixel Dungeon*, um jogo de RPG, não utilizou nenhuma dessas soluções. Portanto, é a única que exige uma baixa API para instalação (versão 8) sem se preocupar com o uso de recursos mais modernos. Seja através de uso explícito da nova API, quando estiverem disponíveis, seja adicionando a dependência ao pacote de compatibilidade ou da re-implementações de recursos, quando seria possível usar novos recursos mesmo em versões mais antigas da API, 96% das aplicações analisadas utilizam pelo menos uma técnica para oferecer suporte a múltiplas versões da API. Portanto, tais resultado oferecem elementos para validar a hipótese **H1**.

3.6.1.1 Quando as Técnicas Foram Utilizadas

Foram detectados alguns padrões de uso dos mecanismos para manter compatibilidades com as diversas versões da API do Android. Pacote de compatibilidade é utilizado quando se deseja um conjunto de novos recursos, cujo implementação está distribuída em diversos arquivos, como as APIs de fragmentos e barra de ação.

Re-implementação de recursos é utilizado para situações pontuais. Quando o código do recurso desejado está localizado em uma classe apenas, como a `Base64`, ou várias mas organizadas em um único pacote da API, como `android.animation`. Também é utilizada para situações em que o recurso não está disponível nos pacotes de compatibilidade.

Pacote de compatibilidade e re-implementação de recursos são utilizados para uso de recursos de granularidade grossa, a partir de classes. Para casos de granulari-

dade fina, quando o recurso desejado são métodos de classes já existentes em versões anteriores, usa-se execução condicional direta ou indireta.

Com o objetivo de concentrar as ocorrências às chamadas da nova API, evitando o espalhamento de instruções condicionais ao longo do código, pode-se utilizar algum padrão de projeto. No entanto, com exceção do padrão de projeto *Proxy*, não foram determinados os motivos de uso de um ou outro padrão, ficando a cargo da escolha individual de cada desenvolvedor. O padrão de projeto *Proxy* foi utilizado em situações em que a instanciação do objeto é feito pela própria plataforma, como objetos da classe `View`.

Também é importante destacar que os próprios pacotes de compatibilidade exigem uma versão mínima de API. Os nomes dos pacotes possuem em sua composição um trecho v#, que representa a versão mínima exigida por ele [25]. Por exemplo, os pacotes support-v4 e appcompat-v7 exigem, respectivamente, API 4 e API 7[2]. Este será um pré-requisito a ser considerado pelos desenvolvedores no uso dos mecanismos. Neste caso, deverão avaliar entre aumentar a versão da aplicação, e deixar de atender uma parcela do mercado, utilizar execução condicional, ou re-implementar os recursos. Nessa última opção, possivelmente será necessária também a re-implementação do recurso da API utilizada pelo pacote, o que pode impactar negativamente no esforço necessário.

3.6.2 Qual o subconjunto de novas funcionalidades da plataforma que são mais utilizados por aplicações compatíveis com versões antigas da API?

- **Proposição P2**: Se a cada nova versão da plataforma são disponibilizados diversos novos recursos então deve existir um subconjunto que é mais utilizado no contexto de suporte a múltiplas versões da API.
- **Hipótese H2**: Existe um subconjunto dos novos recursos da API que é mais utilizado no contexto de suporte a múltiplas versões da API.

Para determinar o subconjunto de novas funcionalidades da plataforma que são mais utilizados por aplicações compatíveis com versões antigas da API fizemos a aná-

[2]Recentemente (agosto de 2016), houve uma mudança nesse política e as versões dos pacotes superiores a 24.2.0 passaram a exigir API nível 9 para todos os pacotes. Por essa razão, os pacotes support-v4 e support-v7 exigem API mínima nível 9 para versões do pacote iguals ou superiores a 24.2.0

lise a partir da técnica usada para facilitar portabilidade para novas plataformas: pacote de compatibilidade, re-implementação de recursos e uso explícito da nova API.

Unificando os três resultados, os elementos mais utilizados são do pacote `android.animation` e classes relacionadas a API de fragmentos, ambas da versão 11 da API. Isso comprova a importância dessa versão da API para a plataforma.

O pacote `android.animation` fornece mecanismo para animação de propriedades de qualquer objeto. Com isso, é possível criar efeitos visuais, úteis para uma boa experiência do usuário, assim como fundamentais em aplicações de jogos. A API de fragmentos permite o reaproveitamento de pedaços de telas na composição das atividades, o que é muito útil considerando a diversidade de aparelhos que oferecem suporte para Android, motivo pelo qual acreditamos ser este um dos elementos mais utilizados.

A forma principal de uso do pacote `android.animation` foi através de re-implementação de recurso, fornecida pelo projeto NineOldAndroids[3], ou execução condicional. Já a API de fragmentos por pacote de compatibilidade. Essa distinção se dá pelo fato de que a API de fragmentos fornece classes bases, em particular `Fragment`, que deverão ser utilizadas por herança, o que não permite a composição em tempo de execução. Enquanto os elementos de animação pode ser instanciados dinamicamente, o que permite utilizar execução condicional. Atualmente, os elementos de animação são disponibilizados no pacote de compatibilidade, permitindo que a re-implementação de recurso seja substituída por esse mecanismo. Tais resultados apresentam evidências suficientes para validarmos a hipótese **H2**.

3.6.3 Qual o esforço necessário para aumentar a compatibilidade com um maior número de APIs de versões anteriores da plataforma?

- **Proposição P3**: Se as aplicações utilizam apenas APIs mais recentes, não possuindo um histórico de uso de APIs antigas, então elas foram desenvolvidas sem a preocupação de oferecer suporte às versões antigas da API o que, possivelmente, a leva a perder uma fatia do mercado de aplicações.

- **Hipótese H3**: As aplicações podem aumentar o seu mercado potencial com baixo esforço em termos de desenvolvimento (alterações em seu código-fonte).

[3] http://nineoldandroids.com/

A falta de compatibilidade com múltiplas versões da API acarreta uma perda de uma parcela do mercado potencial. Para aumentar esse mercado é necessário editar a API mínima exigida para a aplicação e proteger eventuais ocorrências de NewApi que surjam.

Em termos quantitativos, 5 aplicações (50%) apresentaram menos de 10 novas ocorrências de NewApi ao reduzir a versão mínima da API. No geral, surgem, em média, 15,60 novas ocorrências, considerando a redução para a API imediatamente inferior à atual. O caso mais extremo, que mais se beneficiaria com um menor esforço, é o aplicativo Yaac. Este aplicativo poderia reduzir sua API mínima de 21 para 15 com o ajuste de apenas uma ocorrência de NewApi. Indo mais além, poderia reduzir para API 10, com outras 8 ocorrências de NewApi a serem tratadas.

Em termos qualitativos, mesmo nas aplicações mais dependentes de novas APIs, o suporte poderia ser ampliado usando uma das técnicas discutidas neste trabalho. Tais resultados apresentam evidências suficientes para validarmos a hipótese **H3**.

3.6.4 Qual a incidência de código morto em função da versão da API do Android?

- **Proposição P4**: Se as aplicações utilizam execução condicional para execução de determinados trechos de códigos e a sua API mínima exigida é superior à necessária para execução do trecho de código, então tal trecho de código é código-morto.
- **Hipótese H4**: As aplicações possuem código-morto, que só seriam executados na presença de APIs antigas.

A hipótese de uma alta incidência de código-morto devido ao suporte a múltiplas versão da API (**H4**) não foi confirmada. Embora tenhamos encontrado código-morto em 4 de 7 aplicações analisadas, a quantidade percentual de linhas de código é muito baixa, menos de 0,1%. Estudos mais detalhados seriam necessários para determinar a causa disso. No entanto, acreditamos que o uso de padrões de projeto, que permitem uma melhor coesão do código, favoreçam tal cenário, fazendo com que o desenvolvedor tenha uma melhor percepção do código legado que ele deve remover já que não pretende mais atender versões antigas ou mesmo depender da ajuda de analisadores estáticos como o Lint. Enquanto que execução condicional pode dificultar a remoção

desses códigos durante a evolução da aplicação, dado existir um maior espalhamento de código.

3.7 Ameaças à Validade

Nesta subseção, destacamos algumas ameaças à validade do estudo. Identificá-las previamente pode minimizar seus efeitos. Em alguns casos, tais ameaças geram limitações do estudo que devem ser observadas e corrigidas em trabalhos futuros.

Análise manual. Embora tenhamos utilizado o Lint e desenvolvido um simples analisador estático, a maior parte das análises foi feita de forma manual. O que dificultou a determinação de padrões de projeto e ocorrências de execuções condicionais indiretas em algumas situações.

Comportamento do Lint com alguns formatos de EC direta. O Lint NewApi procura identificar se a linha que faz uso explícito da nova API está protegida com uma execução condicional direta. Quando ele consegue, omite esse resultado do relatório. Isso é bom para o desenvolvedor que não vai precisar se preocupar com algo que já está protegido, mas pode ter influência nos resultados desse estudo. Os números de execução condicional podem ser maiores que os aqui apresentados.

Desempenho das aplicações em aparelhos antigos. Na análise da QP3 consideramos apenas a quantidade de ocorrências de NewApi e possíveis soluções destas. Não avaliamos o desempenho das aplicações em aparelhos antigos. É possível que alguma aplicação deixe de prover suporte para alguma versão devido ao baixo desempenho de execução nesses aparelhos, assim teria um maior grau de dificuldade para aumentar a compatibilidade.

Versão da API mínima das aplicações. Os resultados das análises podem ser influenciados pelas versões da API mínima das aplicações, por esse motivo definimos 3 grupos de aplicações. No entanto, para as questões QP1 e QP2 um conjunto de aplicações exige API menor que ou igual a 8, enquanto outro conjunto contém aplicações com API igual ou superior a 9. Esses conjuntos não estão distribuídos de forma uniforme.

Quantidade de aplicações analisadas. Analisamos no total 41 aplicações únicas, sendo que foram 25 para a QP1 e QP2, 10 para QP3 e 7 para QP4. O que representa uma quantidade muito pequena dentro do universo de aplicações Android. Pretendemos

ampliar o estudo com a análise de outras aplicações.

Perfil das aplicações analisadas. O perfil das aplicações analisadas também pode ter influencia nos resultados obtidos. Por exemplo, aplicações com recursos de multimídia utilizam elementos da API que uma aplicação sem tais recursos não utiliza. Tal como aplicações com milhares de linhas de código estão mais propensas a falhas que aplicações com menos linhas de código.

4 Trabalhos Relacionados

Trabalhos de pesquisa relacionados à evolução de API e suporte a múltiplas versões de API, sejam da plataforma Android ou outras tecnologias, e popularidade dos elementos da API Android são relevantes para este trabalho. Em especial, verificamos que não existem muitos trabalhos relacionados ao suporte à múltiplas versões da API do Android. Assim, a principal contribuição deste trabalho é de caracterização das técnicas de implementação de suporte a múltiplas versões da API Android em aplicações reais. As seções seguintes detalham trabalhos de pesquisa relacionados, organizados por tópicos no qual oferecem contribuição relevante.

4.1 Evolução da API Android

McDonnell et al. [9] conduziram um estudo sobre a co-evolução da API Android e suas aplicações usando o histórico de versões encontradas no Github, comparando a velocidade de evolução da API Android com a adoção pelas aplicações. No entanto, não considerou o suporte multi-versão da API. O nosso estudo foi focado nesse aspecto. Embora as aplicações permaneçam oferecendo suporte para API antigas, elas buscam usar recursos das mais novas quando estão disponíveis ou através do pacote de compatibilidade ou re-implementação de recurso.

Diferentemente deste trabalho, que considerou a API como um todo, alguns trabalhos focaram em componentes específicos, como Wei et al.[49] que abordaram a evolução do sistema de permissões para acessar recursos de *hardware* e *software* da plataforma. No início de 2009, a API 3 possuía 103 permissões, enquanto na versão 15 esse número já chegava as 165. As aplicações também precisam adaptar-se a essa evolução.

Neste trabalho, não abordamos elementos de API que foram depreciados, mas alguns trabalhos focam seus estudos nesta perspectiva. Zhou e Walker [50] propuse-

ram a ferramenta *Deprecation Watcher* para identificar uso de API depreciadas em código-fonte de exemplos na web, assim desenvolvedores podem ser informados antes de investir tempo e esforço estudando-os. A ferramenta foi avaliada na detecção de APIs Android depreciadas utilizadas em código-fonte de exemplo no Stack Overflow. Eles obtiveram uma precisão de 100% e uma cobertura de 86% em um conjunto aleatório de 200 questões.

4.2 Fragmentação da API Android

Mutchler et al. [10] abordaram o problema da fragmentação da API alvo. Esse problema ocorre quando as aplicações não atualizam sua versão de API alvo para a mais recente da plataforma. Essa falta de atualização desabilita melhorias da plataforma mesmo quando as aplicações são executadas em dispositivos com a versão mais recente da API. Um exemplo é a vulnerabilidade que permite injeção de fragmentos. Aplicações maliciosas podem enviar mensagens para classes que herdam de `PreferenceActivity`, existente desde a primeira versão do Android. As mensagens são interpretadas como instâncias de fragmentos e carregadas dinamicamente, executando código arbitrário da aplicação mal-intencionada. A API nível 19 adicionou o método `isValidFragment` para resolver esse problema. Aplicativos que têm como alvo os níveis de API 19 ou superiores herdam uma implementação padrão de `isValidFragment` que sempre lança uma exceção e, portanto, são seguros por padrão, mas aqueles que têm como alvo os níveis 18 ou inferior herdam uma implementação que sempre retorna `true`, não oferecendo proteção contra esse ataque, ainda que em execução em um dispositivo com versão superior a 18. Como API 19 é uma API alta, provavelmente a versão mínima das aplicações será bem inferior, exigindo que seja utilizado uma execução condicional na chamada a `isValidFragment`. O nosso trabalho complementa este apresentando soluções para oferecer suporte a múltiplas versões da plataforma e indicando os usos mais comuns. Assim, as aplicações podem usar os melhores recursos de cada versão da plataforma.

Wei et al. [11] identificaram tipos e causas para o problema de compatibilidade induzidos pela fragmentação (FIC *issues* - *Fragmentation-Induced Compatibility issues*). Um dos tipos é não-específico de dispositivos, que pode ser causado devido a evolução da API da plataforma, impactando no suporte a múltiplas versões da API. Também apresentaram uma ferramenta, FicFinder, para identificar tais problemas nas aplicações. O nosso trabalho utilizou uma ferramenta nativa do ambiente de de-

senvolvimento oficial, o Android Lint (regra NewApi). Assim como o nosso trabalho, eles também identificaram os usos de novas APIs em métodos executados pela plataforma. Nós chamamos de suporte implícito (Seção 3.5.1), enquanto eles reportaram como falsos positivos. Na verdade, tanto o FicFinder quanto o Android Lint poderiam ter um catálogo desses métodos e não reportarem esses resultados para os desenvolvedores. Além disso, quantificamos as boas práticas citadas por eles para corrigir ou prevenir os FIC *issues* causadas pela evolução da API da plataforma.

4.3 Popularidade dos elementos da API Android

O estudo feito por Lamba et al. [8] considerou a popularidade dos elementos da API de forma absoluta, sem levar em consideração a versão da API exigida pela aplicação e a versão da API em que o elemento foi disponibilizado. Além disso, a quantificação foi feita por níveis diferentes de granularidade, do mais fino para o mais grosso: método, classe e pacote. Nosso trabalho considerou apenas os elementos que surgiram após a versão da aplicação e os resultados foram agrupados, primariamente, por uma taxonomia, e só após apresentamos métodos, classes e pacotes das taxonomias mais comuns. Os métodos mais comuns encontrados por eles são `getString()`, `get()` e `toString()`. O nosso estudo apresentou métodos da classe `Activity` relacionadas a API de fragmentos e o método `bigContentView()` da classe `Notification` como os mais comuns (contabilizado por uso explícito de nova API - Seção 3.5.2). No nível de classe, eles apontaram `Context` e `View` como as mais comuns. Enquanto que encontramos `Fragment`, `NotificationCompat` e `RecyclerView` (um subtipo de `View`) como as mais comuns (contabilizado por número de `import`'s do pacote de compatibilidade - Seção 3.5.2). Eles apontaram os pacotes `java.util` e `android.content` como os mais comuns. Não fizemos muitas sumarizações por pacotes, apenas o pacote `android.app.backup` apareceu durante a análise por uso explícito da nova API de aplicações que exigiam API nível 4. O pacote surgiu na API nível 5. Por outro lado, os pacotes `java.util` e `android.content` correspondem às nossas taxonomias `java.util` e `content`. A `java.util` apareceu uma única vez em nossa análise. Os resultados combinados nos levam a conclusão que esse pacote é bastante utilizado e muito estável. Já a taxonomia `content` encontra-se em um posição intermediária em nossos resultados.

4.4 Compatibilidade ou suporte a múltiplas versões de API

A plataforma Android permite compatibilidade simultânea com múltiplas versões da API e uma política conservadora em relação à depreciação, de forma que os métodos e classes permanecem nas novas versões mesmo depois de anotados como depreciados. Alguns trabalhos buscaram entender a política de depreciação e suporte a múltiplas versões em outras plataformas. Espinha et al. [51] analisaram a política de evolução de quatro APIs web: Google Maps, Twitter, Netflix e Facebook. Os três primeiros possuem um sistema de versionamento explícito, em que um identificador da versão é incluída na URL de acesso à API (ex.: `domain/<version>/method`), o que permite aos clientes escolherem qual versão da API irão utilizar. Essa abordagem facilita a migração dos clientes, bastando alterar o trecho da URL referente à versão. Porém, uma evolução de API pode ir além que somente adição ou remoção de métodos. Por exemplo, a evolução da Twitter API analisada no trabalho trata da versão 1 para 1.0. Além da mudança na URL (por exemplo, de `twitter.com/1/method` para `twitter.com/1.1/method`) outras elementos também foram modificados. Por exemplo, os clientes foram forçados a autenticar, o suporte para XML foi descontinuado em favor de JSON e mudanças foram feitas para limitar o acesso (que pode penalizar clientes com alta frequência de acesso). Para esses casos, é necessário uma intervenção mais profunda no código das aplicações clientes. O estudo foi complementado com um conjunto de recomendações para os provedores de API para facilitar a evolução dos clientes, entre elas: i) não mudar tão frequentemente, para que os clientes não se vejam também obrigados a alterar suas aplicações com frequência; e ii) não deixar as versões antigas por muito tempo, pois quando isso ocorre os desenvolvedores das aplicações clientes relaxam e atrasam muito a atualização. Ficando muito distante da versão mais recente, a atualização pode tornar-se difícil. A disponibilidade da versão da API no Android, através de `Build.VERSION`, permitindo execução condicional, é um mecanismo para versionamento. Já os pacotes de compatibilidade e a re-implementação de recursos são alternativas a ele.

Sohan et al. [52] realizaram um estudo de caso de evolução da API web de 9 serviços e identificaram dois modos de atualização: versão única e múltiplas versões. No modo de versão única, as APIs web forçam a migração dos clientes para a nova versão da API removendo a versão anterior após um período de tempo, que pode ser conhecido, como nos casos da Facebook REST API (90 dias) e Google Calendar (6 meses),

ou desconhecido, como nos casos da Twitter REST API, Github API, OpenStreetMap API e Google Maps. O modo de múltiplas versões mantém as APIs antigas por um longo período de tempo. Wordpress REST API, Salesforce API e Stripe API utilizam essa estratégia. Tal estratégia permite uma maior flexibilidade para os clientes, mas exige maior esforço dos fornecedores da API, como resultado nem todas as APIs são mantidas. Quando os clientes atrasam muito a atualização, ficando muito distante da versão mais recente, a evolução pode tornar-se difícil. O mesmo pode acontecer com aplicações Android. Foram também analisados diferentes abordagens para identificar a versão da API: i) identificador numérico: números inteiros contínuos que comunicam claramente uma sequência entre as versões; ii) identificador com data: datas também pode ser usadas para identificar versões; e iii) identificadores com números de versões principal e secundário (ex: 2.3): a maioria das APIs web analisadas utilizam essa abordagem. Nenhuma dessas abordagem, por si só, garantem compatibilidade entre as versões. Inclusive, frequentemente as APIs são atualizadas com quebra de compatibilidade sem receber um novo identificador de versão. A abordagem de identificador número é a adotada pela plataforma Android, sendo a compatibilidade garantida por contrato [53], ou seja, as novas versões são projetadas de forma a manter a compatibilidade com suas versões anteriores. Importante destacar que o acesso à APIs web é remoto, o que permite seus clientes escolherem qual versão utilizarão, enquanto o acesso à API Android é local, exigindo outras estratégias para acesso à API. Nosso trabalho identificou, e quantificou, pacote de compatibilidade, execução condicional e re-implementação de recurso como estratégias de suporte a múltiplas versões da API Android.

5 Conclusão e Trabalhos Futuros

Este capítulo apresenta as conclusões desta dissertação de mestrado. A Seção 5.1 apresenta os resultados alcançados para cada questão de pesquisa. A Seção 5.2 discute as principais limitações do trabalho. A Seção 5.3 ressalta suas contribuições. Finalmente, a Seção 5.4 apresenta proposições de trabalhos futuros que estendam e avancem os resultados desta pesquisa.

5.1 Análise dos Resultados da Dissertação

O primeiro objetivo específico era identificar na literatura quais as técnicas indicadas para suporte a múltiplas versões da API Android. Tal objetivo foi contemplado ao identificarmos que são três as técnicas primárias indicadas, sobretudo pela documentação oficial para desenvolvedores de aplicações Android: (i) pacote de compatibilidade; (ii) re-implementação de recurso, e (iii) uso explícito da nova API.

O uso dos pacotes de compatibilidade e re-implementação de recurso possuem formas mais uniformes de adoção. Para o primeiro, basta adicionar a dependência na aplicação, fazer a importação dos arquivos necessários e usar as classes conforme documentação. Para o segundo, é realizado um *"copy-and-paste"* do código-fonte do recurso desejado. Já para fazer uso explícito da nova API é necessário garantir que a chamada à nova API somente será executada em dispositivos em que a nova API esteja presente, sob pena de travamentos da aplicação em dispositivos antigos. Tal proteção é feita com execução condicional. No entanto, a forma dessa execução condicional pode variar bastante. Desde uma simples execução condicional direta, o que pode acarretar em espalhamento do código, até o uso de padrões de projeto, que permite uma maior modularização.

Pacote de compatibilidade é utilizado quando se deseja um conjunto de novos recursos cuja implementação está distribuída em diversos arquivos. No entanto, deve-

se analisar a relação entre a quantidade de recursos utilizados e o aumento no tamanho do instalador da aplicação. Se for desfavorável, pode-se fazer re-implementação de recurso do item específico. Re-implementação de recurso também pode ser utilizado quando o recurso desejado não está disponível por meio de pacote de compatibilidade. Porém, os desenvolvedores ficarão responsável por manter mais um elemento, juntamente com o restante da aplicação. Essas duas técnicas permitem o uso dos recursos em todos os dispositivos que a aplicação for instalada, independente da versão da API, e são utilizadas para uso de recursos em granularidade grossa, a partir de classes. Para os casos de granularidade fina, chamadas a métodos de classes pré-existentes, e quando o recurso pode deixar de ser utilizado em dispositivos com versões antigas da API sem prejuízos para o funcionamento da aplicação, são feitos acessos diretos a nova API.

No nosso estudo analisamos 25 aplicações para quantificar o uso desses mecanismos. Pacote de compatibilidade é utilizado por 92% (23) das aplicações, re-implementação de recurso por 56% (14) e 88% (22) fazem uso explícito da nova API. 88% (22) das aplicações utilizam mais de um mecanismo simultaneamente.

Nas 22 aplicações que fizeram uso explícito da nova API, totalizaram 1556 ocorrências de NewApi. Desse total, 40,6% (632 ocorrências) foi tratado com EC indireta; 22,8% (355) com algum padrão de projeto; 18,9% (294) através de EC direta; 10,7% (166) utilizando suporte implícito da plataforma; 5,3% (82) das ocorrências não foram devidamente tratadas; 1,2% (19) das ocorrências estão dentro de métodos que nunca serão executados; e por fim, duas pequenas frações de 0,3% (5) das ocorrências foi utilizada tratamento de exceção ou são falsos positivos. Os padrões de projeto utilizados para tratar as 355 ocorrências foram *Strategy*, com 77,7% (276) desse total, *Proxy*, com 19,4% (69), *Template Method*, com 2,3% (8), e *Null Object*, com 0,6% (2) das ocorrências.

Também identificamos quais os elementos mais comuns lançados em versões superiores à versão das aplicações mas que são utilizados por elas. Os elementos mais comuns estão relacionados à classe `Fragment`, que surgiu no Android 3.0 API Level 11, quando a plataforma foi otimizada para dispositivos de telas grandes, como os *tablets*. Essa classe foi referenciada por 68% (17) das aplicações, sendo 15 por pacote de compatibilidade e 2 por execução condicional.

Se por um lado detectamos que, no geral, as aplicações poderiam aumentar o seu mercado em potencial com adaptações de, em média, 15 novas ocorrências de NewApi, por outro lado, identificamos que os desenvolvedores têm se preocupado

em evitar código-morto em função da API da plataforma. Na análise de 7 aplicações, 4 delas contém código-morto. No entanto, em termos de linha de código, não passam de 0,1% das linhas de código Java.

5.2 Limitações do Trabalho

Os resultados deste estudo forneceram evidências importantes para discutir e responder às questões de pesquisa desta dissertação. Contudo, existem limitações na pesquisa realizada que precisam ser destacadas para serem exploradas em trabalhos futuros. São elas:

- Foi realizada uma revisão do estado da arte para investigar a existência de outros trabalhos que lidem com suporte a múltiplas versões de APIs, seja restrita a Android ou não, mas é importante a condução de uma revisão sistemática da literatura que busque ampliar o escopo dos trabalhos identificados nesta dissertação;
- A análise foi feita de forma semi-automática. Em especial, a categorização das formas de implementação do acesso explícito às novas APIs foi manual, o que pode trazer inconsistências nos resultados;
- A quantificação dos elementos mais comuns foi feita baseada na importação dos pacotes ou a partir dos resultados do Android Lint. Assim, não foi considerado heranças em classes bases que seriam estendidas por outra ou encapsulamentos de chamadas em classes das aplicações, o que reduz o número de referência diretas aos elementos. Portanto, os elementos podem ser mais utilizados do que os resultados apresentados neste trabalho;
- Analisamos apenas 41 aplicações únicas. O que é um número muito baixo em relação ao universo de aplicações Android existentes.

5.3 Principais Contribuições

A seguir destacamos as principais contribuições desta dissertação:

- Levantamento e identificação de técnicas para suporte a múltiplas versões da API Android;

- Condução de um estudo empírico para analisar como as aplicações atuais adotam e implementam tais técnicas;
- Identificação de quais elementos de novas API são mais utilizados por aplicações que também oferecem suporte para os aparelhos com versões antigas da API;
- Identificação das situações mais comuns em que as técnicas de suporte foram utilizadas;
- Embora inicialmente citada por McDonnell et al. [9], a taxonomia da API apresentava algumas inconsistências em sua definição. Neste trabalho, consolidamos tais conceitos, que podem vir a ser utilizados por trabalhos futuros.

5.4 Trabalhos Futuros

Como trabalhos futuros, planejamos estender os resultados desta pesquisa de modo a lidar com limitações existentes no trabalho, assim como ampliar nossas contribuições. Os seguintes trabalhos de pesquisa podem ser desenvolvidos como desdobramento desta dissertação:

- Realizar uma revisão sistemática da literatura para ampliar nosso escopo de trabalho sobre suporte a múltiplas versões de API, restritos ou não ao Android. Com isso, podemos realizar análises comparativas com outros trabalhos e também refinar nossa metodologia de análise;
- Ampliar a automação do processo de análise das aplicações. Com isso, poderemos analisar mais aplicações de uma forma mais precisa;
- Ampliar o número de aplicações analisadas. Tal trabalho pode ser facilitado pela automação do processo, além disso, daremos mais confiabilidade para os dados obtidos;
- Análise do histórico do repositório de código-fonte das aplicações para responder a outras questões, como "quando os padrões de projetos foram implementados? O uso explícito às novas APIs foram protegidos sempre com padrões de projeto ou inicialmente foi utilizada execução condicional simples?";

- Estudo qualitativo com desenvolvedores de aplicações Android com o objetivo de responder, entre outras questões, os motivos pelos quais escolheram uma ou outra técnica;
- Análise comparativa entre FicFinder e Android Lint (regra NewApi) e, possivelmente, outros analisadores sintáticos.

Referências

[1] LHAMAS, R.; CHAU, M.; SHIRER, M. *Worldwide Smartphone Market Grows 28.6% Year Over Year in the First Quarter of 2014, According to IDC*. http://www.idc.com/getdoc.jsp?containerId=prUS24823414: [s.n.], 2014. Acesso em: 04 aug. 2014.

[2] EDWARDS, J. *The iPhone 6 Had Better Be Amazing And Cheap, Because Apple Is Losing The War To Android*. http://www.businessinsider.com/iphone-v-android-market-share-2014-5: [s.n.], 2014. Acesso em: 04 jun. 2016.

[3] GRONLI, T. M. et al. Mobile application platform heterogeneity: Android vs windows phone vs ios vs firefox os. In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. [S.l.: s.n.], 2014. p. 635–641. ISSN 1550-445X.

[4] ELMER-DEWITT, P. *Android is a mess, say developers*. http://fortune.com/2011/04/04/android-is-a-mess-say-developers/: [s.n.], 2011. Acesso em: 21 jan. 2016.

[5] GOOGLE. *Android Developers*. https://developer.android.com/index.html: [s.n.], 2016. Acesso em: 04 jun. 2016.

[6] PARK, J.-H.; PARK, Y. B.; HAM, H. K. Fragmentation Problem in Android. In: *2013 International Conference on Information Science and Applications (ICISA)*. IEEE, 2013. p. 1–2. ISBN 978-1-4799-0604-8. Disponível em: <http://ieeexplore.ieee.org/document/6579465/>.

[7] TIAN, Y. et al. What are the characteristics of high-rated apps? A case study on free Android Applications. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015. p. 301–310. ISBN 978-1-4673-7532-0. Disponível em: <http://ieeexplore.ieee.org/document/7332476/>.

[8] LAMBA, Y.; KHATTAR, M.; SUREKA, A. Pravaaha: Mining Android Applications for Discovering API Call Usage Patterns and Trends. In: *Proceedings of the 8th India Software Engineering Conference on XXX - ISEC '15*. New York, New York, USA: ACM Press, 2015. p. 10–19. ISBN 9781450334327. Disponível em: <http://dl.acm.org/citation.cfm?doid=2723742.2723743>.

[9] MCDONNELL, T.; RAY, B.; KIM, M. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013. p. 70–79. ISBN 978-0-7695-4981-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2550526.2550557>.

[10] MUTCHLER, P. et al. Target Fragmentation in Android Apps. In: *Security and Privacy Workshops (SPW), 2016 IEEE*. [S.l.: s.n.], 2016. p. 204–213.

[11] WEI, L.; LIU, Y.; CHEUNG, S.-C. Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016. p. 226–237. ISBN 9781450338455. Disponível em: <http://dl.acm.org/citation.cfm?doid=2970276.2970312>.

[12] LECHETA, R. *Google Android*. [S.l.]: Novatec, 2015.

[13] GOOGLE. *Platform Architecture*. https://developer.android.com/guide/platform/index.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[14] STATISTA. *Number of apps available in leading app stores as of June 2016*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/: [s.n.], 2016. Acesso em: 22 jan. 2017.

[15] GOOGLE. *Android Studio*. https://developer.android.com/studio/index.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[16] GOOGLE. *Improve Your Code with Lint*. https://developer.android.com/studio/write/lint.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[17] GOOGLE. *Application Fundamentals*. https://developer.android.com/guide/components/fundamentals.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[18] GOOGLE. *Introduction to Activities*. https://developer.android.com/guide/components/activities/intro-activities.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[19] GOOGLE. *The Activity Lifecycle*. https://developer.android.com/guide/components/activities/activity-lifecycle.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[20] GOOGLE. *Services*. https://developer.android.com/guide/components/services.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[21] GOOGLE. *Content Providers*. https://developer.android.com/guide/topics/providers/content-providers.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[22] GOOGLE. *BroadcastReceiver*. https://developer.android.com/reference/android/content/BroadcastReceiver.html: [s.n.], 2016. Acesso em: 22 jan. 2017.

[23] GOOGLE. *Dashboards*. https://developer.android.com/about/dashboards/index.html: [s.n.], 2017. Acesso em: 22 jan. 2017.

[24] GOOGLE. *Dashboard*. https://developer.android.com/about/dashboards/index.html: [s.n.], 2016. Acesso em: 04 ago. 2016.

[25] GOOGLE. *Support Library*. https://developer.android.com/topic/libraries/support-library/index.html: [s.n.], 2017. Acesso em: 22 jan. 2017.

[26] ZIMMERS, G. *How to base64 encode decode Android*. http://androidcodemonkey.blogspot.com.br/2010/03/how-to-base64-encode-decode-android.html: [s.n.], 2010. Acesso em: 22 jan. 2017.

[27] GOOGLE. *How come I can not use android.util.Base64?* http://stackoverflow.com/questions/9304927/how-come-i-can-not-use-android-util-base64/9305274#9305274: [s.n.], 2016. Acesso em: 22 jan. 2017.

[28] WHARTON, J. *NineOldAndroids*. http://nineoldandroids.com/: [s.n.], 2012. Acesso em: 22 jan. 2017.

[29] GOOGLE. *Supporting Different Platform Versions*. https://developer.android.com/training/basics/supporting-devices/platforms.html: [s.n.], 2012. Acesso em: 22 jan. 2017.

[30] SANTOS, J. et al. Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines. In: *SugarLoafPLoP '12 Proceedings of the 9th Latin-American Conference on Pattern Languages of Programming Article No. 1*. Natal: [s.n.], 2012. p. 1–17. Disponível em: <http://dl.acm.org/citation.cfm?id=2600810>.

[31] GOOGLE. *Android Lint Checks*. http://tools.android.com/tips/lint-checks: [s.n.], 2016. Acesso em: 22 jan. 2017.

[32] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994. ISBN 9780321700698. Disponível em: <https://books.google.com.br/books?id=6oHuKQe3TjQC>.

[33] GACEK, C.; ANASTASOPOULES, M. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 26, n. 3, p. 109–117, maio 2001. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/379377.375269>.

[34] PAVLIC, L. The mobile product line. In: *Second Workshop on Software Quality Analysis, Monitoring, Improvement and Applications*. [s.n.], 2013. v. 1053, p. 63–69. ISSN 16130073. Disponível em: <http://ceur-ws.org/Vol-1053/sqamia2013paper8.pdf>.

[35] Best Practice Software Engineering (BPSE). *Strategy Pattern*. http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/strategy.html: [s.n.], 2013. Acesso em: 22 jan. 2017.

[36] DoFactory. *Template Method*. http://www.dofactory.com/net/template-method-design-pattern: [s.n.], 2017. Acesso em: 22 jan. 2017.

[37] HENNEY, K. Null Object Something for Nothing. In: *European Conference on Pattern Languages of Programs*. [s.n.], 2002. Disponível em: <http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf>.

[38] SourceCodeMania. *NULL Object Design Pattern* . http://sourcecodemania.com/null-object-design-pattern/: [s.n.], 2017. Acesso em: 22 jan. 2017.

[39] DoFactory. *Proxy*. http://www.dofactory.com/net/proxy-design-pattern: [s.n.], 2017. Acesso em: 22 jan. 2017.

[40] RUNESON, P. et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. ed. [S.l.]: Wiley Publishing, 2012. ISBN 1118104358, 9781118104354.

[41] Google. *License*. https://source.android.com/source/licenses.html: [s.n.], 2017. Acesso em: 22 jan. 2017.

[42] Google. *Managing Network Usage*. https://developer.android.com/training/basics/network-ops/managing.html: [s.n.], 2017. Acesso em: 22 jan. 2017.

[43] Google. *Google Git*. http://bit.ly/2rROjC5: [s.n.], 2017. Acesso em: 22 jan. 2017.

[44] Network Working Group. *The Base16, Base32, and Base64 Data Encodings*. https://tools.ietf.org/html/rfc4648: [s.n.], 2006. Acesso em: 22 jan. 2017.

[45] Google. *Material Design para Android*. https://developer.android.com/design/material/index.html: [s.n.], 2017. Acesso em: 22 jan. 2017.

[46] StackOverFlow. *How to use Libcore.os to set an enviroment variable*. http://stackoverflow.com/questions/35193646/how-to-use-libcore-os-to-set-an-enviroment-variable: [s.n.], 2016. Acesso em: 22 jan. 2017.

[47] BUSINGE, J.; SEREBRENIK, A.; BRAND, M. G. J. van den. Eclipse api usage: the good and the bad. *Software Quality Journal*, v. 23, n. 1, p. 107–141, 2015. ISSN 1573-1367. Disponível em: <http://dx.doi.org/10.1007/s11219-013-9221-3>.

[48] MASTRANGELO, L. et al. Use at your own risk: The java unsafe api in the wild. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2015. (OOPSLA 2015), p. 695–710. ISBN 978-1-4503-3689-5. Disponível em: <http://doi.acm.org/10.1145/2814270.2814313>.

[49] WEI, X. et al. Permission evolution in the android ecosystem. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2012. (ACSAC '12), p. 31–40. ISBN 978-1-4503-1312-4. Disponível em: <http://doi.acm.org/10.1145/2420950.2420956>.

[50] ZHOU, J.; WALKER, R. J. Api deprecation: A retrospective analysis and detection method for code examples on the web. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY,

USA: ACM, 2016. (FSE 2016), p. 266–277. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950298>.

[51] ESPINHA, T.; ZAIDMAN, A.; GROSS, H. G. Web api growing pains: Stories from client developers and their code. In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. [S.l.: s.n.], 2014. p. 84–93.

[52] SOHAN, S. M.; ANSLOW, C.; MAURER, F. A case study of web api evolution. In: *2015 IEEE World Congress on Services*. [S.l.: s.n.], 2015. p. 245–252. ISSN 2378-3818.

[53] Google. *O que é um nível de API?* https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels: [s.n.], 2016. Acesso em: 22 jan. 2017.

APÊNDICE A – Informações adicionais sobre as aplicações analisadas

A.1 Repositórios de código-fonte das aplicações

Aplicação	Repositorio	Commit
AcDisplay	https://github.com/AChep/AcDisplay	2ae3c79
AFWall+	https://github.com/ukanth/afwall	8c1f2d2
aMetro	https://github.com/RomanGolovanov/ametro	a3f717a
AnkiDroid	https://github.com/ankidroid/Anki-Android	1926fcc
AntennaPod	https://github.com/danieloeh/AntennaPod	d5e63cb
AnySoftKeyBoard	https://github.com/AnySoftKeyboard/AnySoftKeyboard	7066bfe
BatteryBot Battery Indicator	https://github.com/darshan-/Battery-Indicator-Free	b20bb76
C:geo	https://github.com/cgeo/cgeo	8fb79d3
ConnectBot	https://github.com/connectbot/connectbot	cfafd40
Dash Clock	https://github.com/romannurik/dashclock/	8d9581d
Document Viewer	https://github.com/SufficientlySecure/document-viewer	9fdef3d
DuckDuckGo Search & Stories	https://github.com/duckduckgo/android	87a605b
Firefox	http://hg.mozilla.org/mozilla-central	a66bf0a800f3
Focal (Beta)	https://github.com/xplodwild/android_packages_apps_Focal	d9cc873
Free Mobile Netstat	https://github.com/vdavy/freemobilenetstat	118173d
Google I/O	https://github.com/google/iosched	2531cbd
Hangar - Smart app shortcuts	https://github.com/corcoran/Hangar	90c44705
Indic Keyboard	https://github.com/smc/Indic-Keyboard	f52b974
K-9 Mail	https://github.com/k9mail/k-9	f794cc1
MyExpenses	https://github.com/mtotschnig/MyExpenses	ac08238
Numix Circle icon pack	https://github.com/numixproject/com.numix.icons_circle	bdcf570

Continua na página seguinte

Continuação da página anterior

Aplicação	**Repositorio**	**Commit**
OpenTasks	https://github.com/dmfs/opentasks	b9e5d52
Orbot Proxy com Tor	https://gitweb.torproject.org/orbot.git	34079c7
Persian Calendar	https://github.com/ebraminio/DroidPersianCalendar	a4272cd
Quick Dice Roller	https://github.com/Ohmnibus/quick-dice-roller	32b2c1c
Ringdroid	https://github.com/google/ringdroid/	81bdf25
Shattered Pixel Dungeon	https://github.com/00-Evan/shattered-pixel-dungeon	aaadf4a
Simon Tatham's Puzzles	https://github.com/chrisboyle/sgtpuzzles	6f170b5
Simple Last.fm Scrobbler	https://github.com/tgwizard/sls	f7a3eca
SnoopSnitch	https://opensource.srlabs.de/projects/snoopsnitch	4332a66
Telegram	https://github.com/DrKLO/Telegram	2114024
Terminal Emulator for Android	https://github.com/jackpal/Android-Terminal-Emulator	f5ecc5a
Termux	https://github.com/termux/termux-app	746dc75
Transdrone	https://github.com/erickok/transdroid	8b96074
Vanilla Music	https://github.com/vanilla-music/vanilla	94d6995
VLC for Android	https://code.videolan.org/videolan/vlc-android	fad7ab3
Vuze Remote	https://github.com/vuze/vuze-remote-for-android	118a52a
WiFiAnalyzer (open-source)	https://github.com/VREMSoftwareDevelopment/WifiAnalyzer	b9475a1
Wifiﬁxer	https://github.com/Zanshinmu/Wifi-Fixer	112da88
Yaac - IRC Client	https://github.com/pocmo/Yaaic	5da4949
Zmanim	https://bitbucket.org/jgindin/zmanim	832:4ec02fb6be56

A.2 IDs das aplicações na Google Play

Aplicação	**ID da aplicaçõe na Google Play**
AcDisplay	com.achep.acdisplay
AFWall+	dev.ukanth.ufirewall
aMetro	org.ametro
AnkiDroid	com.ichi2.anki
AntennaPod	de.danoeh.antennapod
AnySoftKeyBoard	com.menny.android.anysoftkeyboard
BatteryBot Battery Indicator	com.darshancomputing.BatteryIndicator

Continua na página seguinte

Continuação da página anterior

Aplicação	**ID da aplicaçõe na Google Play**
C:geo	cgeo.geocaching
ConnectBot	org.connectbot
Dash Clock	net.nurik.roman.dashclock
Document Viewer	org.sufficientlysecure.viewer
DuckDuckGo Search & Stories	com.duckduckgo.mobile.android
Firefox	org.mozilla.firefox
Focal (Beta)	fr.xplod.focal
Free Mobile Netstat	org.pixmob.freemobile.netstat
Google I/O	com.google.samples.apps.iosched
Hangar - Smart app shortcuts	ca.mimic.apphangar
Indic Keyboard	org.smc.inputmethod.indic
K-9 Mail	com.fsck.k9
MyExpenses	org.totschnig.myexpenses
Numix Circle icon pack	com.numix.icons_circle
OpenTasks	org.dmfs.tasks
Orbot Proxy com Tor	org.torproject.android
Persian Calendar	com.byagowi.persiancalendar
Quick Dice Roller	ohm.quickdice
Ringdroid	com.ringdroid
Shattered Pixel Dungeon	com.shatteredpixel.shatteredpixeldungeon
Simon Tatham's Puzzles	name.boyle.chris.sgtpuzzles
Simple Last.fm Scrobbler	com.adam.aslfms
SnoopSnitch	de.srlabs.snoopsnitch
Telegram	org.telegram.messenger
Terminal Emulator for Android	jackpal.androidterm
Termux	com.termux
Transdrone	org.transdroid.lite
Vanilla Music	ch.blinkenlights.android.vanilla
VLC for Android	org.videolan.vlc
Vuze Remote	com.vuze.android.remote

Continua na página seguinte

Continuação da página anterior

Aplicação	**ID da aplicaçõe na Google Play**
WiFiAnalyzer (open-source)	com.vrem.wifianalyzer
Wifiﬁxer	org.wahtod.wifiﬁxer
Yaac - IRC Client	org.yaaic
Zmanim	com.gindin.zmanim.android

A.3 Dados brutos

Dados brutos obtidos durante o estudo podem ser acessados através da URL: http://bit.ly/multversion-android