Gerência de Variabilidades na Plataforma Android

ADORILSON BEZERRA, Universidade Federal do Rio Grande do Norte
UIRÁ KULESZA, Universidade Federal do Rio Grande do Norte
RODRIGO BONIFÁCIO, Universidade de Brasília
ELDER CIRILO, Pontifícia Universidade Católica do Rio de Janeiro
EDMILSON CAMPOS NETO, Universidade Federal do Rio Grande do Norte

Over the past few years there has been rapid growth in the use and demand for mobile application development. Android platform being the most used worldwide. Devices running Android have different characteristics, different size, different sensors, different versions of the API. In addition, different users have different requirements of their applications. This paper presents techniques for management of variability that can be used on the Android platform to address these aspects. We use as case study the software product line MobileMedia, an application to manage multimedia files that were originally created for Java Micro Edition, and now migrated to Android. The JME version used techniques such as conditional compilation and aspect-oriented programming, while the Android version uses native platform resources, conditional execution and object-oriented programming techniques such as polymorphism.

Categories and Subject Descriptors:

General Terms:

Additional Key Words and Phrases: android, software product line, mobile

ACM Reference Format:

---

1. INTRODUÇÃO

Ao longo dos últimos anos, houve um crescimento vertiginoso na utilização e demanda de desenvolvimento de aplicações para dispositivos móveis (Lhamas et al., 2014). Diversas lojas *online* para aplicações - como Google Play Store, Apple Store, Blackberry App World e Windows Store - disponibilizam milhões de diferentes aplicações para usuários de *smartphones e tablets*. A perspectiva atual é que o mercado de aplicações móveis continuará crescendo ao longo dos próximos anos, seguindo a tendência de crescimento do número de dispositivos móveis vendidos e de aplicações que vem sendo distribuídas nas lojas *online*. Além disso, dispositivos móveis estão ficando cada vez mais populares e acessíveis para pessoas de diferentes poder aquisitivo ao redor do mundo.

O vasto e crescente mercado de aplicações móveis demanda por personalizações das aplicações para lidar com diferentes dispositivos, idiomas, e necessidades específicas dos usuários. A variedade de dispositivos demanda por variações na aplicação, influenciadas pelo tamanho de tela, bibliotecas de classes (APIs) para manipulação de imagens gráficas, e disponibilidade de poder de processamento e memória disponível. De fato, personalizações em larga escala de aplicações são um requisito fundamental que já vem sendo considerado por abordagens existentes de engenharia de linhas de produtos de software. Recentes pesquisas e trabalhos da indústria direcionam esforços para a gerência de variabilidade no contexto de aplicações móveis.

Neste contexto, este trabalho investiga e explora técnicas disponíveis para a gerência de variabilidades no desenvolvimento de aplicações móveis para a plataforma Android. Atualmente, Android é a plataforma mais popular para o desenvolvimento de aplicações móveis (Jim E.,2014). Ela define um sistema operacional e um *middleware* que fornecem diversas facilidades para o desenvolvimento de aplicações implementadas na linguagem Java. Este trabalho apresenta e discute como diferentes tipos de variabilidades de negócios e plataforma podem ser modularizadas usando a plataforma Android. O trabalho utiliza o MobileMedia, uma linha de produtos de software que provê suporte para gerenciar (criar, apagar, visualizar, reproduzir e enviar para contatos) diferentes mídias (fotos, música e vídeo) em dispositivos móveis, para ilustrar e discutir as técnicas de implementação de

variabilidades disponíveis na plataforma Android. Considerando que o MobileMedia (Figueiredo et al. 2008) foi originalmente desenvolvido na plataforma Java Micro Edition (JME), são realizadas comparações de técnicas e mecanismos fornecidos por essa plataforma para a modularização de variabilidades com os novos mecanismos fornecidos no Android. Finalmente, o trabalho também discute as diferenças no processo de derivação de produtos das aplicações para diferentes plataformas.

As seções restantes deste artigo estão organizadas da seguinte forma: a Seção 2 apresenta uma visão geral sobre as técnicas de gerência de variabilidades disponíveis na plataforma Android. A Seção 3 ilustra como a arquitetura e variabilidades da aplicação MobileMedia podem ser implementados utilizando as técnicas disponíveis na plataforma. Na seção 4 discutimos diferenças entre a derivação de produto para JME e para Android. Por fim, concluímos o trabalho na seção 5.

2. GERÊNCIA DE VARIABILIDADES NA PLATAFORMA ANDROID

Esta seção introduz técnicas de gerência de variabilidades disponíveis na plataforma Android. Uma variabilidade refere-se à habilidade das aplicações adaptarem seu comportamento para uso em um contexto particular. Considerando a produção em massa de aplicações móveis para Android, pode se identificar duas forças predominantes que orientam a necessidade por adaptações. A primeira é a imensa variedade de dispositivos existentes. Aplicações móveis para Android devem ser capaz de fornecer a mesma experiência de usuário em diferentes tipos de dispositivos. Por exemplo, toda aplicação móvel deve adaptar sua interface de usuário de acordo com as configurações de tela disponíveis. Além disso, elas devem também garantir que o *layout* correto são aplicados para as telas corretas. A segunda são as necessidades específicas dos usuários. Atualmente, o crescente número e diversidade de usuários demanda por aplicações personalizadas. Considerando uma aplicação para gerenciar mídias, usuários *avançados* gostariam de uma aplicação completa, capaz de gerenciar fotos, vídeos e música, enquanto outros usuários podem preferir uma aplicação mais simples e fácil de aprender, por exemplo, somente para gerenciar fotos.

Gerência de variabilidades está relacionado às atividades de suporte a variabilidades no ciclo de vida do *software*, permitindo o gerenciamento de dependências e interações entre variabilidades. Técnicas de gerência de variabilidades permitem sistematicamente desenvolver e evoluir artefatos comuns e variáveis pertencentes a uma linha de produto de *software*. As seções seguintes apresentam como variabilidades de negócio e técnicas podem ser atendidas em aplicações presentes na plataforma Android.

2.1 Variabilidades de Negócio

Variabilidades de negócio são adaptações em requisitos de negócio da aplicação requerida pelo usuário final. Por exemplo, visualização de fotos ou reprodução de músicas/vídeos podem ser vistos como variabilidades de negócio para uma aplicação de gerência de mídias. Em geral, gerentes de produto de aplicações são responsáveis por decidir quais variabilidades de negócio uma linha de produto de *software* pretende oferecer suporte. O usuário final é responsável por escolher quais variabilidades estarão presentes nos produtos desejados.

Existem diferentes soluções para implementar variabilidades de negócio. Variabilidades alternativas que podem funcionar com determinados produtos, por exemplo, podem ser implementadas através do uso dos mecanismos de herança e polimorfismo de orientação a objetos, permitindo que classes abstratas sejam estendidas para oferecer implementações específicas para um dado produto. Padrões de projeto orientados a objetos (Gamma et al. 1995) podem ser aplicados dentro deste contexto, oferecendo implementações modularizadas para cada uma das variantes de uma variabilidade específica. Em tais soluções, a escolha de quais variantes serão selecionadas é delegada para uma ferramenta de derivação de produto, a qual se responsabiliza pela inclusão das subclasses específicas responsáveis pela inclusão no produto final.

2.2 Variabilidades de Plataforma

Plataforma refere-se ao ambiente de *hardware* e *software* em que uma aplicação irá ser executada (Preuveneers et al. 2004). Assim, variabilidades de plataforma dizem respeito a variações de elementos desse conjunto. Ela está intrinsecamente ligada a aspectos técnicos relacionados ao dispositivo no qual a aplicação é instalada, tais como: implementações alternativas para versões que utilizam diferentes bibliotecas de classes (APIs) e ajuste de *layout* de telas para diferentes dispositivos. Atualmente, a plataforma Android oferece suporte para quatro diferentes tipos de variabilidades de plataforma:

- **Múltiplas telas**: a plataforma Android suporta uma variedade de tamanho de tela e faz a maioria do trabalho para adaptar o layout das aplicações para preencher cada tela adequadamente. A base para o suporte a múltiplas telas é a habilidade de renderizar o *layout* das aplicações redimensionando-os para se adequar ao tamanho da tela e redimensionando imagens para a densidade da tela. Para melhor adequação às diferentes telas, desenvolvedores devem também: (i) explicitamente declarar em arquivos de configuração quais tamanhos de tela suas aplicações suportam; (ii) fornecer diferentes layouts para diferentes telas através do nome do diretório onde os arquivos são armazenados (ex: layout-sw600dp – para um *layout* de 600dp); e (iii) fornecer imagens de tamanhos diferentes para telas de densidades diferentes. Outra opção para suportar variedades de telas diferentes é o reuso de fragmentos da interface do usuário na composição da configuração de layouts diferentes, geralmente otimizadas para ocupar todo o espaço disponível na tela. A ideia é compor o *layout* em tempo de execução. Por exemplo, em um dispositivo pequeno, o adequado é mostrar apenas um fragmento por vez em uma interface com um único painel. Por outro lado, considerando dispositivos maiores, pode ser mais apropriado alocar fragmentos lado-a-lado, exibindo mais informações para o usuário.

- **Localização**: a principal forma de localizar aplicativos é fornecendo textos alternativos para diferentes idiomas. Desenvolvedores podem também disponibilizar imagens, sons, vídeos e outros recursos alternativos. Para adaptar a localização de aplicativos, a plataforma Android busca por um diretório com os recursos específicos para a localização. Nesse caso, o nome do diretório deve ser padronizado, especificando um idioma ou uma combinação de idioma-região de acordo com seu conteúdo. Em tempo de execução, baseado na configuração de local do dispositivo, a plataforma Android usa o conjunto de recursos presente no diretório apropriado;

- **Compatibilidade de dispositivo**: é bem conhecido que nem todo dispositivo dispõe de todas funcionalidades, assim, em adição a prover uma interface de usuário flexível que se adapta a configuração de telas diferentes, aplicações Android também devem ser tolerantes a: (i) alguma variabilidade de *hardware*, como a presença ou não de um sensor; (ii) variabilidade de *software*, como um *widget* específico;

- **Versões da API**: cada versão da plataforma geralmente adiciona novas funcionalidades, não previstas em versões anteriores, ou muda a maneira de interação com o usuário. Nesse caso, a plataforma permite que sejam declarados a versão mínima da API com qual sua aplicação é compatível. Também é possível descobrir em tempo de execução qual a versão da API está disponível no dispositivo e, assim, fazer algum tratamento diferenciado.

Android provê uma plataforma de aplicação dinâmica que oferece suporte para recursos específicos por configuração, como declarar que um dispositivo de hardware é necessário ou qual a versão mínima da API. Quando um usuário localiza um aplicativo de seu interesse na Google Play Store, automaticamente é determinado quando o aplicativo é compatível o dispositivo, e quando o dispositivo não provê as funcionalidades necessárias, o usuário não poderá instalar a aplicação. Embora seja possível a instalação manual caso o usuário possua

o arquivo de instalação da aplicação. Também é possível descobrir em tempo de execução se algum recurso está ou não presente no dispositivo e, assim, fazer algum tratamento diferenciado.

3. MODULARIZANDO VARIABILIDAES NO MOBILEMEDIA

Esta seção apresenta e discute as técnicas de implementação de variabilidades disponíveis na plataforma Android, usando como base a LPS MobileMedia.

3.1 Variabilidades de Plataforma

A aplicação *MobileMedia* para Android, denominada MMApp, foi desenvolvida inicialmente para ser usada como auxílio no ensino de alunos de graduação em cursos de desenvolvimento Android. Tal aplicação oferece, além dos requisitos para gerenciamento (reprodução, visualização, seleção, compartilhamento via SMS e redes sociais) de mídia (fotos, imagens e vídeos), as seguintes funcionalidades adicionais: (i) definição de listas de mídias; (ii) compartilhamento de lista de mídias e histórico de execução através de redes sociais; e (iii) identificação de dados de localização depois da execução da mídia.

De forma a oferecer suporte para diferentes customizações, a aplicação MMApp vem sendo refatorada para expor novas variabilidades e assim ser caracterizada como uma linha de produto de software. Tanto variabilidades de negócio quanto de plataforma vêm sendo modularizadas na MMApp. A Figura 1 apresenta uma representação parcial do modelo de *features* da MMApp contemplando as variabilidades de negócio.

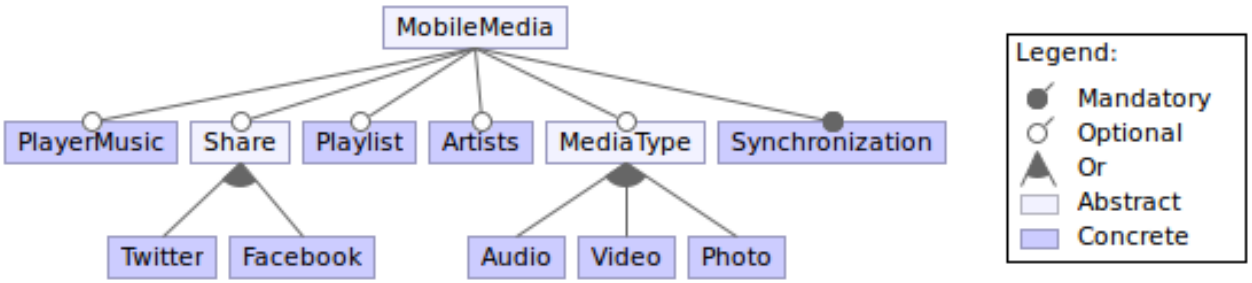

Figura 1: Modelo de features da LPS MMApp

A LPS MMApp possui somente uma feature obrigatória: Synchronization, responsável por fazer a leitura dos arquivos gravados no aparelho e atualizar o banco de dados da aplicação com informações desses arquivos (localização e metadados). Os tipos de arquivos que serão lidos são definidos pelas variantes da feature MediaType (Audio, Video e Photo). Dados acerca da aplicação podem ser compartilhados nas redes sociais, caso a feature Share esteja presente. É possível configurar a LPS para gerar produto que possam compartilhar para Twitter e/ou Facebook. Existem ainda três outras features opcionais: PlayerMusic (um tocador de músicas), Playlist (o usuário pode gerenciar - criando, editando e apagando - listas de músicas) e Artists (músicas e vídeos são exibidos ordenados por artistas e álbuns, de acordo com os metadados lidos na importação). No que diz respeito a variabilidades de plataforma, MMApp pode lidar com diferentes línguas (internacionalização), telas de tamanhos diferentes e diferentes versões do Android, conforme será detalhado adiante.

MMApp foi projetada com uma arquitetura em camadas (Busschman et al. 1996). A Figura 2 ilustra a arquitetura de tal aplicação. A camada mais básica (camada inferior na Figura 2) promove o reuso de diversas bibliotecas de classes externas e da plataforma Android, tal como SocialAuthAndroid para autenticação de redes sociais. A camada de integração oferece

serviços para persistência e extração de informações de metadados de diferentes formatos que representam conteúdo multimídia (áudio ou vídeo). A camada de aplicação contém classes que servem de Fachada (padrão Facade) para oferecer serviços de gerenciamento de conteúdo multimídia (criação de listas de mídias, relacionar conteúdo de mídia com listas de mídias, etc), executar conteúdo de mídia, e compartilhar dados através de redes sociais. Finalmente, a camada superior da arquitetura agrega as classes relacionadas com a interface do usuário. Todas as camadas específicas da MMApp depende do modelo de domínio, o qual implementa os conceitos fundamentais da aplicação, agregando, portanto, classes como Author, Audio, Video, MultimediaContent, e PlayList.

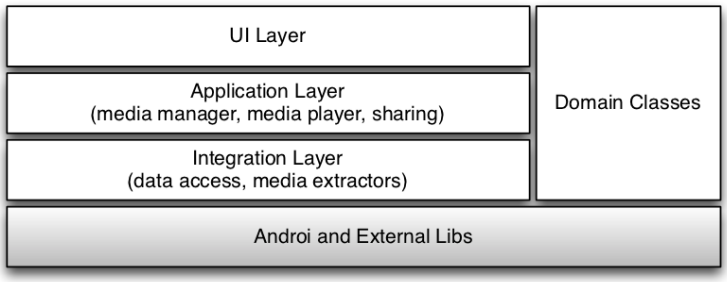

Figura 2: Arquitetura em camada da MMApp

A Figura 3 apresenta um diagrama de classes da LPS MMApp indicando as principais classes responsáveis pela implementação de *features* específicas. A classe `MMUnBActivity`, juntamente com classes do pacote `core.view`, fazem parte da camada UI (ver Figura 2). Os pacotes `core.manager`, `core.audioPlayer`, `core.videoPlayer` e `core.PhotoViewer` fazem parte da camada Application. A camada Domain é implementada pelas classes do pacotes `core.domain`. Por fim, `core.db` e `core.extractor` são responsáveis pela camada Integration. O primeiro faz integração com banco de dados, aplicando o padrão de projeto “data access object” (Nock, 2003), enquanto o segundo é responsável por ler os arquivos e seus metadados dos dispositivo. A classe `Manager` implementa o padrão de projeto Façade. O componente SocialAuthAndroid é uma biblioteca de terceiros responsável por fazer a comunicação com redes sociais. De acordo com a legenda na figura e indicação dos pontos coloridas nas classes, temos que a *feature* Artists é, de forma geral, implementada nas classes `MMUnBActivity` e `ArtistFragment`. Embora algumas classes sejam responsáveis por implementar determinadas features ou variabilidades, não foi necessário alterá-las diretamente, conforme marcações em preto indicam. Além disso, classes anotadas com mais de uma *feature*, tratam das mesmas através de uma granularidade fina, como as classes `MMUnBActitivity`, responsável pela tela principal da aplicação (Figura 1), e `ExtractorFactory`, que instância extratores específicos de acordo com as variantes da *feature* MediaType presentes no produto. Enquanto classes anotadas com somente uma *feature* possuem uma granularidade grossa, como as classes `DefaultAudioExtractor`, `DefaultVideoExtractor` e `DefaultPhotoExtractor`, que são os extratores para áudio, vídeo e fotos, respectivamente.

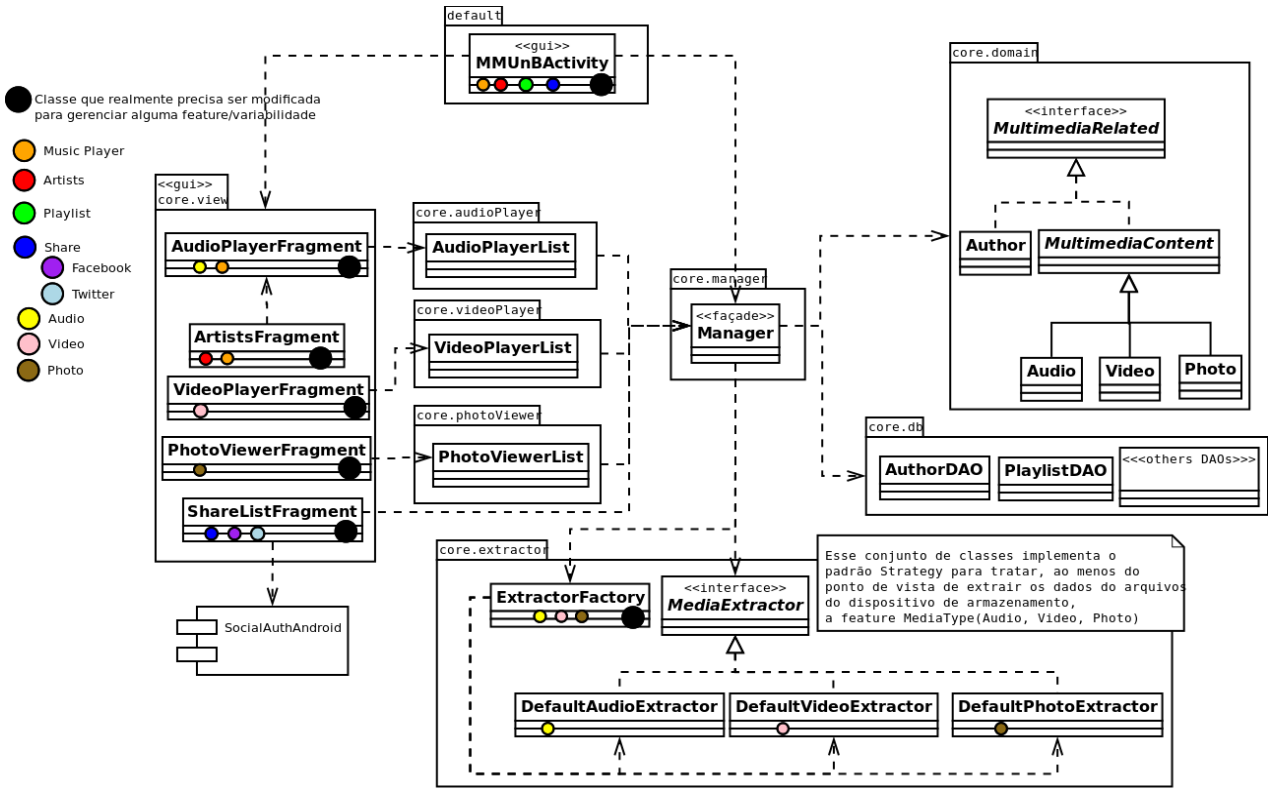

Figura 3: Diagrama de classes parcial de MMApp com marcações auxiliares indicando alguns classes que são responsáveis pela implementação de features

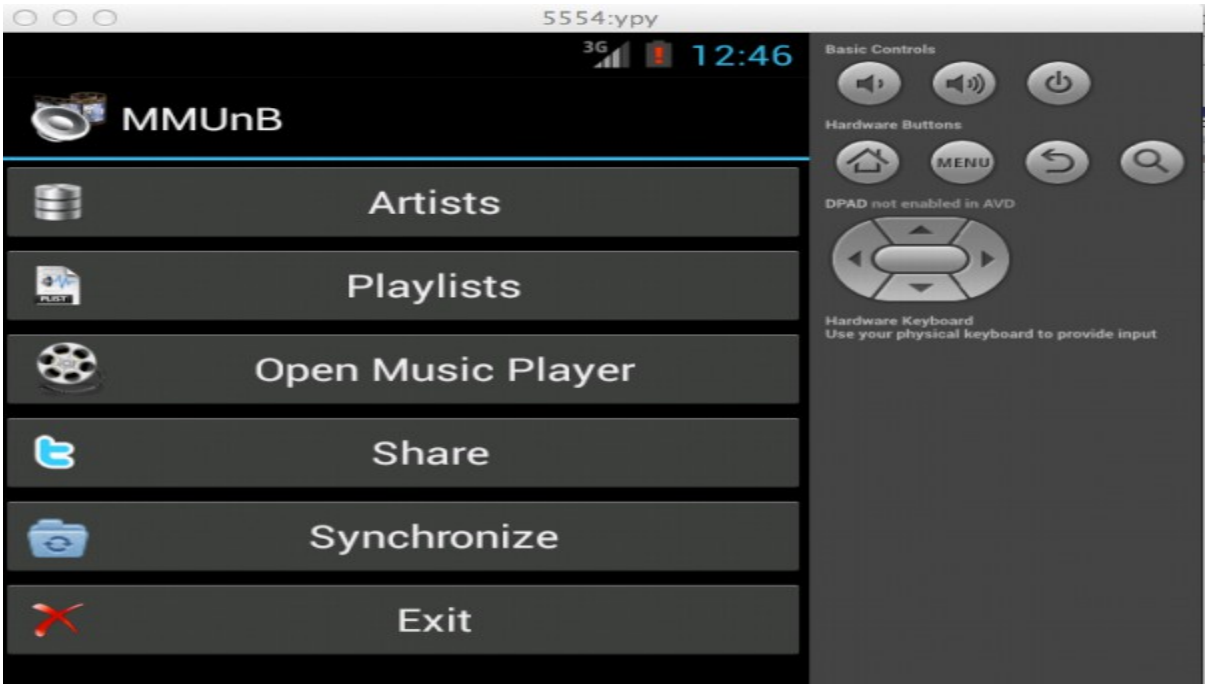

Figura 4: Janela principal da aplicação

3.2 Implementação de Variabilidades de Negócio

Dentre as diversas técnicas conhecidas para implementação de variabilidades de negócios em linhas de produtos de software, compilação condicional, execução condicional e programação orientada a objetos são algumas delas. Nesse trabalho, foram utilizadas execução condicional e técnicas de programação orientada a objetos, como herança e polimorfismo, associadas a aplicação de padrões de projeto. Nessa seção, discutimos como e onde essas metodologias foram aplicadas.

Compilação condicional e execução condicional são técnicas utilizadas para implementação de variabilidades. A primeira ocorre em tempo de compilação, através da definição de diretivas de pré-processamento no código fonte, indicando a inclusão ou não de fragmentos de códigos nos produtos da LPS. Contudo, somente podem ser definidos parâmetros booleanos (Santos et al. 2012). Além disso, compilação condicional não é uma técnica com suporte nativo na plataforma Android. Ao passo que a técnica de execução condicional (Santos et al. 2012) ocorre em tempo de execução, e os parâmetros podem assumir quaisquer tipos e valores.

Nesse trabalho, utilizamos somente execução condicional para auxiliar na gerência de variabilidades. A configuração dos produtos foi feita em arquivos XML's, onde definimos os parâmetros que serão avaliados em estruturas condições *if.* Mapeamos as features e variantes do modelo de *features* para *tags* XML. A Figura 5 mostra a configuração de um produto com a presença das *features* PlayerMusic e Artists, e a variante do tipo de mídia Audio, mas sem as features Playlist, Share e a variante do tipo de mídia Vídeo. As Figura 6, 7 e 8 mostram como esses valores são lidos na aplicação em tempo de execução.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer name="feature_playlist">2</integer>
    <integer name="feature_music_player">0</integer>
    <integer name="feature_artists">0</integer>

    <bool name="feature_share">false</bool>
    <bool name="feature_share_facebook">true</bool>
    <bool name="feature_share_twitter">true</bool>

    <bool name="feature_audio">true</bool>
    <bool name="feature_video">false</bool>
</resources>
```

Figura 5: Arquivo de configuração que define as features de um produto

Na implementação atual do MMApp, diversas *features*, tais como, compartilhamento em redes sociais, possuem uma implementação modular, mas que ainda assim possuem um grau de espalhamento dentro da aplicação. Em particular, o código de compartilhamento de dados em redes sociais, por exemplo, requer clicar no botão "Share" na janela principal do aplicativo (Figura 2). Assim, uma primeira questão é: como ocultar/exibir esse botão de acordo com a presença ou ausência da *feature* opcional Share? Na plataforma Android, a interface de usuário é definida em arquivos XML, mas a plataforma permite que seus elementos sejam manipulados via linguagem de programação Java. Além disso, a API disponibiliza um método (`setVisibility(int)`) para alterar a visibilidade de elementos que herdam da classe `View`, como é o caso da classe `Button`, utilizada para os botões do menu da aplicação. O uso desse método confirmou-se como uma estratégia simples e viável devido uma das opções possíveis para o seu parâmetro (a constante `View.GONE`), além de ocultar o elemento, faz com o que o espaço que estava reservado para o mesmo não seja preenchido, como se o botão não tivesse sido adicionado. Assim, não foi necessário fazer adaptações na interface devido à ausência ou presença de alguma *feature*. A Figura 6 mostra esse código.

```
button = ((Button) view.findViewById(R.id.btn_share));
button.setOnClickListener(new OnClickMenuItemListener());
if (!getActivity().getResources().getBoolean(R.bool.feature_share))
    button.setVisibility(View.GONE);
```

Figura 6: Trecho de código que ilustra como a ocultação de botões do menu foi feita

A *feature* Share é alternativa, possuindo algumas variabilidades possíveis: as redes sociais pelas quais o usuário poderá compartilhar dados do aplicativo. Para auxiliar nessa tarefa foi utilizada a biblioteca de terceiros SocialAuth[1], uma biblioteca Java para autenticação, recuperação de perfil e contatos e atualização de status em diversos serviços que permitem a conexão através de oAuth e/ou OpenID. A Figura 7 mostra como as variantes de Share são incluídas na aplicação.

```
private void configAdapter(SocialAuthAdapter adapter, String conteudo, Button btn){
    ResponseListener listener = new ResponseListener(conteudo, adapter);
    adapter.setListener(listener);

    if (getActivity().getResources().getBoolean(R.bool.feature_share_facebook))
        adapter.addProvider(Provider.FACEBOOK, br.unb.mobileMedia.R.drawable.facebook);

    if (getActivity().getResources().getBoolean(R.bool.feature_share_twitter))
        adapter.addProvider(Provider.TWITTER, br.unb.mobileMedia.R.drawable.twitter);

    adapter.enable(btn);
}
```

Figura 7: Adicionando ou não opções de redes sociais para o compartilhando de dados da aplicação

A MMApp é capaz de trabalhar com tipos de mídias diferentes: áudio, vídeo e foto (essa última ainda não totalmente implementada). Uma importante atividade da aplicação é a sincronização de sua base local com os arquivos que estão salvos no dispositivo, feito através do botão *Synchronize* do menu principal. Foi definida uma interface `MediaExtractor`, e para cada tipo de mídia uma classe que implementada essa interface, caracterizando assim aplicação do padrão de projeto Strategy, onde são utilizadas recursos de programação orientada a objetos como herança e polimorfismo, o que também proporcionou um desacoplamento (Larman, 2004) da classe `Manager` e implementações da interface `MediaExtractor` através de uma fábrica de extratores. As Figuras 8 e 9 ilustram isso.

[1]https://github.com/3pillarlabs/socialauth/

```
public List<MediaExtractor> createExtractor(){

    List<MediaExtractor> extrs = new ArrayList<~>();
    MediaExtractor extr = null;

    if (context.getApplicationContext().getResources().getBoolean(R.bool.feature_video)){
        extr = new DefaultVideoExtractor(context);
        extrs.add(extr);
    }

    if (context.getApplicationContext().getResources().getBoolean(R.bool.feature_audio)){
        extr = new DefaultAudioExtractor(context);
        extrs.add(extr);
    }

    return extrs;
}
```

Figura 8: Método fábrica de extratores de mídia

```
public void synchronizeMedia(Context context) throws DBException {

    ExtractorFactory fac = new ExtractorFactory(context);
    List<MediaExtractor> extrs = fac.createExtractor();

    for(MediaExtractor extractor: extrs){
        List<Author> authors = (List<Author>) extractor.processFiles();
        saveAuthor(context, authors);
    }
}
```

Figura 9: Utilização da fábrica de extratores

Na fábrica de extratores, poderia-se utilizar algum mecanismo para recuperação desses extratores de forma automática, por exemplo utilizando a API de localização de serviços do Java, com a classe `ServiceLoader`, no entanto, embora essa classe esteja presente no Android, ela não funciona adequadamente por depender do diretório `META-INF` nas aplicações, e na montagem do arquivo APK das aplicações Android, esse diretório é omitido, mesmo que esteja definido no código-fonte do projeto[2].

3.3 Implementação de Variabilidades de Plataforma

A implementação atual da MMApp oferece suporte para três diferentes variabilidades de plataforma: (i) diferentes línguas (internacionalização); (ii) diferentes tamanhos de tela; e (iii) diferentes versões do Android. Elas foram criadas com o propósito de analisar as técnicas disponíveis atualmente para implementar tais variabilidades, assim como analisar seu impacto para derivação de um produto específico da LPS (Seção 4). Tais variabilidades são detalhadas a seguir.

Múltiplas Telas. A implementação atual do Android oferece suporte para a implementação de múltiplas telas de uma aplicação para dispositivo móvel que precisa ser executada em diferentes dispositivos, tais como, celulares ou tablets. Versões anteriores do Android permitiam a implementação de layouts de telas para dispositivos com diferentes tamanhos, sendo a plataforma responsável pela seleção do mais apropriado layout para um dado dispositivo. Entretanto, tais *layouts* eram completamente independentes, e não haviam classes ou métodos reutilizados entre eles. Versões recentes do Android incorporam o

[2]Há um bug report sobre isso no bug tracker do Android: https://code.google.com/p/android/issues/detail?id=59658

mecanismo de fragmento que permite definir unidades de telas e reutilizá-las para implementar *layouts* customizados para a mesma aplicação.

As Figura 11 e 13 mostram, respectivamente, duas telas diferentes do MMApp definido para um telefone celular e um *tablet*. Como podemos ver, a primeira (Figura 11) apresenta a tela da aplicação composta de duas telas separadas, uma representando as opções do menu e a outra a funcionalidade selecionada (representada pela tela de artistas na tela ilustrada na figura). A outra (Figura 13) apresenta a tela para *tablet* que integra o menu e a funcionalidade selecionada na mesma tela. Para promover o reuso de telas entre essas duas configurações diferentes de telas para telefones celulares e *tablets*, usamos o mecanismo de fragmentos da plataforma Android para implementá-las.

A Figura 10 mostra como são definidos os *layouts* de telas em dispositivos com tela pequena e com tela grande. Na tela pequena, haverá apenas um espaço (FrameLayout) para alocar elementos gráficos, enquanto que em uma tela grande, haverão dois desses espaços.

```
<FrameLayout
    android:id="@+id/main"
    android:layout_width="0dip"
    android:layout_height="fill_parent"
    android:layout_weight="2"/>
```

```
<FrameLayout
    android:id="@+id/menu"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="2"/>

<FrameLayout
    android:id="@+id/content"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"/>
```

Figura 10: Trechos de códigos dos arquivos que definem o layout principal da aplicação

A plataforma Android determina de forma automática e transparente qual layout será utilizado, de acordo com as dimensões da tela do dispositivo em que o a aplicação está executando. No entanto, o conteúdo a ser alocado em cada espaço deve ser determinado programaticamente, conforme mostra a Figura 12. Baseado na existência do FrameLayout main, o layout terá um único painel, onde será inicialmente exibido o menu da aplicação ou dois painéis, um exibindo o menu da aplicação (FrameLayout menu) e o outro o conteúdo selecionado (FrameLayout content).

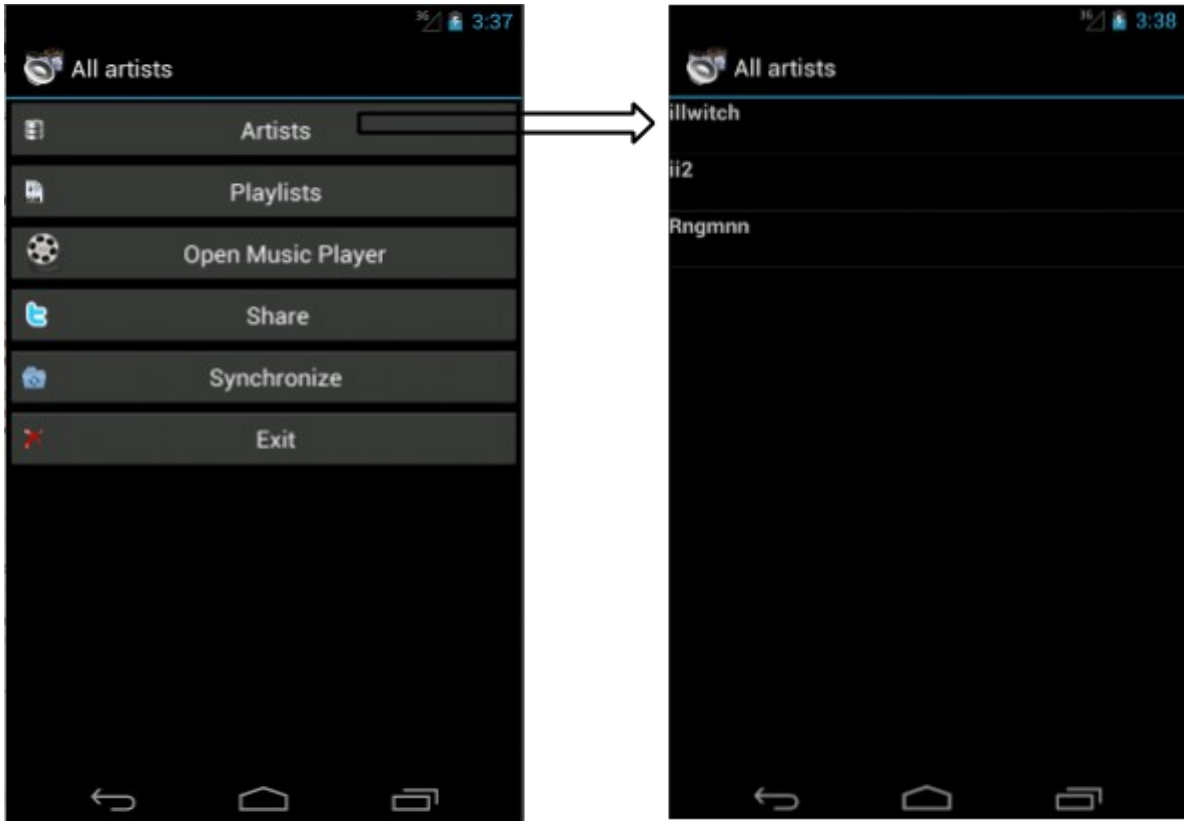

Figura 11: O fragmento com a lista de artistas é exibida em outra atividade, que sobrepõe o menu principal

```
MenuFragment menuFrag = new MenuFragment();

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
if (findViewById(R.id.main)!=null){
    transaction.add(R.id.main, menuFrag);
}else{
    transaction.add(R.id.menu, menuFrag);
    transaction.add(R.id.content, new ContentFragment());
}
transaction.commit();
```

Figura 12: Trecho de código que efetivamente monta o layout da aplicação

Exemplo 1: Uso de `Fragment` para suporte a múltiplas telas

Atualmente, a forma mais eficiente para prover suporte a telas de tamanho diferentes é com o uso de fragmentos, porções de interfaces de usuários que podem ser combinadas para montar um layout de uma tela. Em versões do Android anteriores a 3.0, esse suporte já existia, na medida em que podia-se definir layouts de telas diferentes, para dispositivos de telas de tamanhos diferentes e o Android se encarregava de selecionar o layout mais adequado para o dispositivo em uso. No entanto, esses layouts eram totalmente independentes, não havendo reaproveitamento de suas partes. Com o surgimento dos fragmentos, é possível definir pedaços de telas, e montar os layouts a partir deles.

Figura 13: O fragmento com a lista de artistas é exibida em lado-a-lado com o menu principal, na mesma atividade

Exemplo 2: Idioma

A plataforma Android provê suporte nativo para implementação de internacionalização para aplicações móveis. Para implementar essa variabilidade, são necessários apenas: (i) criar um arquivo diferente para armazenar as mensagens de diferentes idiomas e organizá-los em diretórios diferentes; e (ii) indicar no código da aplicação qual mensagem será utilizada. Em tempo de execução, o Android localizará a mensagem de acordo com o idioma configurado no dispositivo. A Figura 15 mostra, por exemplo, os arquivos *strings.xml* definidos para os idiomas inglês e português suportados pela linha de produto MMApp. O idioma padrão é o inglês e está definido pelos arquivos no diretório *values*. O idioma alternativo é o português brasileiro, cujos arquivos são definidos no diretório *values-pt-rBR*. Também é possível forçar um idioma para a aplicação, que irá sobrepor o idioma definido no dispositivo. Os arquivos com as mensagens da interface de uma aplicação Android devem ficar no diretório *res/values*. E, dentro da aplicação, devem ser referenciado como mostra a Figura 14.

```
Manager.instance().synchronizeMedia(getApplicationContext());
Toast.makeText(getApplicationContext(),
               R.string.message_synchronization_finished,
               Toast.LENGTH_LONG).show();
```

Figura 14: Trecho de código que faz referencia a string message_synchronization_finished

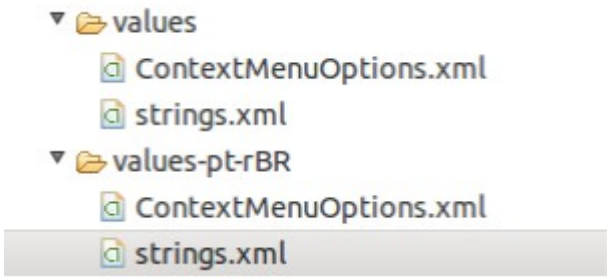

Figura 15: Arquivos com strings padrões da aplicação, e a tradução para português do Brasil

O idioma padrão da MMApp é o inglês. Para prover suporte para o português do Brasil, é necessário apenas criar o arquivo com as strings traduzidas e colocá-lo no diretório res/values-pt-rBR/, como mostra a Figura 16.

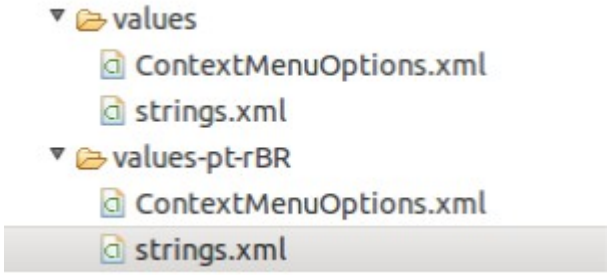

Figura 16: Arquivos com strings padrões da aplicação, e a tradução para português do Brasil

Na Figura 17, temos um trecho do arquivo *`values-pt-rBR/strings.xml`*.

```
<string name="title_activity_audio_player">AudioPlayerActivity</string>
<string name="message_synchronization_finished">O conteúdo do banco de dados foi sincronizado</string>
<string name="title_activity_playlist_editor">PlaylistEditorActivity</string>
```

Figura 17: Trecho do arquivo values-pt-rBR/strings.xml

Exemplo 3: Uso de uma dada versão da API

Alguns recursos da API estarão disponíveis somente em versões mais recentes, no entanto, eles poderão ser substituídos por outros de versões anteriores, sem prejuízo na execução das aplicações. Um exemplo disso é a possibilidade de definir uma cor do item ativo de uma lista, que só é possível em versões iguais ou superiores a 3.0 (chamada de Honeycomb), pois é necessário usar um *layout* não presente em versões anteriores. Assim, em tempo de execução, é verificado qual a versão da API e então define-se um ou outro *layout* para os itens da lista, como mostrado na Figura 18.

```
int layout = 0;
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB){
    layout = android.R.layout.simple_list_item_activated_1;
}else{
    layout = android.R.layout.simple_list_item_1;
}
ArrayAdapter<String> adapter = new ArrayAdapter<String>(getActivity(), layout, values);
setListAdapter(adapter);
```

Figura 18: Checando em tempo de execução a versão da API

Exemplo 4: Pacote de compatibilidade

Uma vez que alguns recursos só estão disponíveis em versões mais recentes, mas que por outro lado, o número de dispositivos com versões antigas é relevante, a Google disponibiliza pacotes de compatibilidade. Esses pacotes são tradicionais arquivos JAR com classes, interfaces e outros artefatos de recursos que poderão ser adicionados na sua aplicação. Assim, quando se desejar usar a classe Fragment, por exemplo, em vez da importação ser da API padrão do Android, como mostra a Figura 19, deverá ser desse pacote que está junto da aplicação, como mostrado na Figura 20.

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

Figura 19: Importando classes relacionadas a fragmento da API padrão do Android

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

Figura 20: Importando classes relacionadas a fragmentos do pacote de compatibilidade

3.4 Outros padrões Android

A documentação oficial da plataforma Android dispõe de uma seção específica sobre padrões[3] , contendo a descrição de 17 padrões. No entanto, somente 3 deles estão relacionados ao tratamento de algum tipo de variabilidade, resumidos a seguir:

- Multi-pane layouts: esse padrão descreve como aplicações podem manter um layout consistente através do ajuste de conteúdo quando em dispositivos com telas de tamanho diferentes. Esse balanceamento é alcançado conforme foi mostrado na seção 3.3.
- Compatibility: no contexto dessa seção da documentação, esse padrão trata apenas da compatibilidade com versões antigas do Android com um componente presente apenas em versões mais recentes, a “action bar”. Nesse trabalho, não usamos esse componente. Compatibilidade com versões anteriores em um contexto mais amplo foi exemplificado na seção anterior.
- Pure Android: esse padrão trata do desenvolvimento de aplicações multiplataformas, que não foi o objetivo desse trabalho.

[3]https://developer.android.com/design/patterns

Os demais padrões fogem ao escopo desse trabalho. Seja por não tratar de qualquer tipo de variabilidade, tratam de interações, por exemplo, seja por serem muito dependentes de um componente presente somente em versões mais recentes da plataforma, a “action bar”.

3.5 Derivação de produto

(Cirilo et al, 2012) apresenta duas diferentes estratégias de derivação para implementação de linhas de produto de software na plataforma Java Micro Edition (JME). Essas estratégias variam de acordo com a tecnologia de modularização escolhida para tratar as variabilidades da LPS: (i) compilação condicional; e (ii) programação orientada a aspectos. Ambas estratégias foram especificadas utilizando uma ferramenta para derivação de produtos baseada em *features* chamada GenArch (Cirilo et al, 2008) e ilustrada com a LPS MobileMedia.

A derivação de produto do MM no JME realizada com a técnica compilação condicional foi apoiada pela geração de arquivos de configuração que definem a inclusão (ou remoção) de diretivas `ifdef`. Cada diretiva CC define se a implementação de uma *feature* será ou não incluída no produto específico representado pelo arquivo de configuração. A criação de arquivos de configuração representando um produto foi especificado usando modelos específicos do GenArch - *features*, modelos de configuração e arquitetura e geração baseada em *template* - que permite a inclusão de trechos de texto (ex., texto de diretiva `ifdef`) associados uma *feature* em um arquivo de configuração. Por outro lado, a derivação de produto apoiada em programação orientada a aspectos facilmente suportada pelo mapeamento direto de todas variabilidades de *features* do *feature model* do GenArch para cada aspecto responsável por implementá-la.

A derivação de produtos de LPSs implementadas na plataforma Android requer uma combinação de estratégias combinando: (i) o mapeamento direto de variabilidades do *feature model* para classes que implementam uma variante específica - que é especialmente apropriada para ser usada em variabilidades de negócio que são implementadas usando mecanismos de polimorfismo e herança; (ii) a definição de arquivos de configuração que determinam presença ou ausência de uma variabilidade de negócio específica; e (iii) organização de arquivos diversos para tratar variabilidades de plataforma (idiomas, tamanho e densidade de tela etc.). As variabilidades de negócio são explicitamente verificadas em tempo de execução, enquanto que as variabilidades de plataforma são verificadas de forma transparente e a própria plataforma deduz qual arquivo deve ser carregado, embora também seja possível consultar esses valores caso necessário para algum tratamento específico.

4. TRABALHOS RELACIONADOS

Reuso de software é uma ideia central na engenharia de domínio (Frakes, 2005). (Mojica et al., 2014) fazem uma análise de como reuso de software, em termos de herança e uso de classes (orientação a objetos) *entre* aplicações Android diferentes, está presente nos aplicativos móveis gratuitos disponibilizados na Play Store, utilizando o conceito de Software Bertillonage (Davies et al, 2011) para rastrear o código das mesmas. O estudo aponta que algumas aplicações possuem um alto grau de similaridade, além de serem desenvolvidas pelo mesmo autor. Através da identificação de quais classes bases são mais frequentemente herdadas, os desenvolvedores podem alocar recursos para torná-las mais confiáveis, eficientes e modulares. Tais classes costumam formar o núcleo de uma LPS. Também indica a classe `Activity` como sendo a de maior percentual de herança. Com o a disponibilização do recurso de fragmento na plataforma recentemente, a tendência é que haja um aumento da herança de `Fragment`, ao invés de `Activity`, nos próximos anos.

(McDonnell et al, 2013) analisa a evolução da API Android (frequência de mudanças) e como os códigos clientes reagem a essa evolução. Eles analisaram 10 projetos de código-fonte aberto de domínios diferentes para determinar: (i) grau de dependência da API Android; (ii)

espaço de tempo entre o cliente da API e a última versão disponível; (iii) tempo de adoção de novas APIs; (iv) a relação entre instabilidade da API e adoção; e (v) a relação entre atualização da API e erros (*bugs*) nos códigos clientes. (Linares-Vasquez et al, 2013) também analisa o impacto de mudanças da API no sucesso das aplicações, apontando que as 50 aplicações de menor sucesso usam APIs que são 500% mais falhas que aquelas utilizadas pelas 50 aplicações de maior sucesso. Baseado nessa conclusão, (Linares-Vasquez, 2014) propõe uma abordagem para recomendar ações e soluções para problemas gerados pela evolução da API. Entretanto, o estudo de tais autores não contempla o aspecto de variabilidades de aplicações Android.

5. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho discutiu mecanismos para gerência de variabilidades na plataforma Android. A plataforma possui recursos para tratar de forma automática as diferenças existentes entre os diversos aparelhos, tanto diferenças de *hardware* (ex. tamanho diferentes de telas), quanto diferenças de *software* (ex. versões diferentes da API). Além de suportar a internacionalização das aplicações. Utilizamos como estudo de caso, a LPS MMApp, que possui layout otimizado para telas grandes e pequenas, funciona em versões antigas da API e também tem tradução para inglês e português, sendo a adição de um novo idioma uma tarefa trivial.

Além disso, é possível derivar produtos com funcionalidades diferentes. Para isso, mapeamos as *features* e variabilidades de seu *feature model* para arquivos de configuração no formato XML, que são lidos, usando um mecanismo nativo da plataforma, em tempo de execução, ativando ou não uma *feature* através de execução condicional. Algumas variabilidades foram implementadas com mecanismos de polimorfismo e herança, como as variantes de extratores e de mídia. Na versão Java Micro Edition do MobileMedia não era possível utilizar polimorfismo ou demais técnicas de programação orientada a objetos, devido a restrições de memória dos dispositivos da geração passada, o que não acontece com o Android.

A derivação de produtos atualmente usada para aplicações Android é manual, sendo necessário copiar o arquivo de configuração do produto desejado. No entanto, algumas iniciativas estão surgindo para tornar esse processo automatizado. FeatureIDE[4] é um plugin para Eclipse que provê suporte para o desenvolvimento de software orientadas a features. Recentemente, foi lançado uma atualização com suporte para Android. A Google está desenvolvendo uma nova IDE, Android Studio[5], cuja uma das novidades é o uso de Gradle como nova ferramenta de *build* automático que, entre outras características, contará com "*build variants*" e geração de múltiplos arquivos APK's. Um estudo de como essas ferramentas podem ser úteis para gerência de variabilidades e derivação de produtos será alvo de trabalho futuro.

O alvo de pesquisa desse trabalho, foi a geração de produtos em tempo de build. Depois de prontos, não terão suas features alteradas. No entanto, é comum no contexto de aplicações Android, os usuários poderem adquirir novas funcionalidades por demanda. Para esse tipo de interação, é disponibilizado um serviço do Google Play chamado Google Play In-App Billing[6]. Trabalhos futuros abordarão essa distribuição de features por demanda. Além disso, a estratégia apresentada gera produtos com todos os arquivos da LPS. Abordagens que permitam a criação de produtos somente com os arquivos estritamente necessários precisam sem exploradas.

REFERÊNCIAS

Buschmann, F. et al *Pattern-Oriented Software Architecture: a System of Patterns*. John Wiley & Sons, Inc. 1996.

[4]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[5]http://developer.android.com/sdk/installing/studio.html
[6]http://developer.android.com/google/play/billing/index.html

Cirilo, E., Kulesza, U., & Lucena, C. (2008). *A product derivation tool based on model-driven techniques and annotations*. Journal of Universal Computer Science , 1344–1367.

Cirilo, E., Kulesza, U., Torres, M., & Lucena, C. (2012). *Experience with Automatic Product Derivation of Mobile Applications Using Model-Driven Techniques*. Handbook of Research on Mobile Software Engineering, I, 113–123. doi:10.4018/978-1-61520-655-1

Davies, J., German, D. M., Godfrey, M. W., & Hindle, A. (2011). *Software bertillonage*. In Proceeding of the 8th working conference on Mining software repositories - MSR '11 (p. 183). New York, New York, USA: ACM Press. Doi:10.1145/1985441.1985468

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., … Dantas, F. (2008). *Evolving software product lines with aspects: an empirical study on design stability*. Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on , 261–270.

Frakes, W. B., & Kang, K. (2005). *Software reuse research: status and future*. IEEE Transactions on Software Engineering, 31(7), 529–536. doi:10.1109/TSE.2005.85

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jim E. (2014) *The IPhone 6 Had Better Be Amazing And Cheap, Because Apple Is Losing The War To Android*. Business Insider. http://www.businessinsider.com/iphone-v-android-market-share-2014-5. Acesso em 21 de julho de 2014.

Larman, C. *Applying UML and Patterns: an introduction to object-oriented analysis and design and interative development*. 3. ed. Nova Jersey: Prentice Hall PTR, 2004.

Linares-Vásquez, M. (2014). *Supporting evolution and maintenance of Android apps*. In Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014 (pp. 714–717). New York, New York, USA: ACM Press. doi:10.1145/2591062.2591092

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., & Poshyvanyk, D. (2013). *API change and fault proneness: a threat to the success of Android apps*. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013 (p. 477). New York, New York, USA: ACM Press. Doi:10.1145/2491411.2491428

Llamas, R., Chau, M., & Shirer, M. (2014). *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 Acesso em 04 de agosto de 2014.

McDonnell, T., Ray, B., & Kim, M. (2013). *An Empirical Study of API Stability and Adoption in the Android Ecosystem*. In 2013 IEEE International Conference on Software Maintenance (pp. 70–79). IEEE. doi:10.1109/ICSM.2013.18

Mojica, I. J., Adams, B., Nagappan, M., Dienst, S., Berger, T., & Hassan, A. E. (2014). *A Large Scale Empirical Study on Software Reuse in Mobile Apps*. Software, IEEE, 31(2), 78–86. doi:10.1109/MS.2013.142

Nock, C. (2003). *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Prentice Hall Professional Technical Reference.

Preuveneers D., Van den Bergh J., Wagelaar D, Georges A., Rigole P, Clerckx T., Berbers Y., Coninx K., Jonckers V. e De Bosschere K. (2004) *Towards an extensible context ontology for ambient intelligence*. In Proceedings of the Second European Symposium on Ambient Intelligence (EUSAI 2004), Eindhoven, The Netherlands (Markopoulos P, Eggen B, Aarts EHL and Crowley JL, Eds), volume 3295 of Lecture Notes in Computer Science, pp 148–159. Springer-Verlag.

Santos, J., Lima, G., Kulesza, U., Sena, D., Pinto, F., Lima, J., Vianna, A., Pereira, D., Fernandes, V. *Conditional Execution: A Pattern for the Implementation of Fine-Grained Variabilities in Software Product Lines*. SugarLoafPLop'2012