AlienDroid – Abstração do Acesso a Dados em Android

Marlon S. Carvalho, Rodrigo Hjort

Serpro - Serviço Federal de Processamento de Dados
Salvador – BA – Brazil

{marlon.carvalho, rodrigo.hjort}@serpro.gov.br

Abstract. *The purpose of this paper is to propose a framework which main focus is to improve the development of Android Database-Enabled applications inside Brazilian Federal Government. AlienDroid provides a simplified architecture, with only few negative impacts in the architecture and performance of the applications that extends it. The main focus is to facilitate the development of Android applications that uses relational databases as storage.*

Resumo. *O objetivo deste trabalho é apresentar uma solução que visa a facilitar o desenvolvimento de aplicativos em Android no âmbito do Governo Federal. O AlienDroid provê uma arquitetura simplificada, com poucos impactos negativos na arquitetura e desempenho dos aplicativos, cujo foco é facilitar a criação de aplicações em Android que necessitam realizar a persistência de dados em bancos de dados relacionais.*

1. Introdução

Embora o uso das metodologias de desenvolvimento e linguagens de programação orientadas a objetos tenha crescido notavelmente, a forma de armazenamento de dados mais comumente utilizada continua sendo através de bancos de dados relacionais [Date 2000]. De fato, intensas pesquisas e melhorias de desempenho consolidaram esta tecnologia como a forma mais eficiente e confiável no armazenamento de dados. Este fato, explana [Silberschatz et al. 1999], tornou o modelo relacional como o principal modelo de dados para o desenvolvimento de sistemas.

Este uso do modelo relacional é visível também no desenvolvimento de aplicativos para dispositivos móveis. A grande maioria dos sistemas operacionais voltados para este ambiente adotam soluções mais reduzidas e leves de sistemas gerenciadores de bancos de dados relacionais. O modelo orientado a objetos também tem se destacado neste cenário, no qual as linguagens que adotam este paradigma estão presentes em quase todos os ambientes de desenvolvimento para dispositivos móveis.

Contudo, a semântica do modelo relacional é diferente e incompatível com a semântica do modelo orientado a objetos, causando o problema comumente designado como diferença de impedância [Ambler 2003]. Diferença de impedância refere-se às incompatibilidades semânticas existentes entre estes dois modelos, o que dificulta a adoção conjunta destas duas tecnologias amplamente adotadas na construção de *softwares* para dispositivos móveis. De um lado, um banco de dados relacional consiste, basicamente, de tabelas e relacionamentos entre estas tabelas; e, de outro lado, objetos consistem em dados e comportamentos sobre estes dados, o que dificulta a integração entre ambos [Yoder et al. 1998].

2. Apresentação do Problema

Existe um esforço, na área de desenvolvimento de aplicações, direcionado para solucionar, ou minimizar, o impacto decorrente do uso do banco de dados relacional. Este impacto reflete-se, principalmente, no quesito produtividade. Criar e incorporar o

código de acesso ao banco de dados diretamente nas classes de negócio aumenta o tempo de desenvolvimento, além de impor futuras dificuldades para a manutenção [Ambler 2003].

No âmbito das aplicações para o sistema operacional Android [Android 2013], este problema apresenta-se ainda mais agudo, uma vez que os *smartphones* e *tablets* ainda carecem de um poder de processamento maior e estão fortemente restringidos ao tempo de vida de suas baterias. Neste contexto, as propostas que visam a solucionar este problema devem cercar-se de cuidados para que não causem um impacto negativo, de tal forma, que possa implicar aplicativos lentos ou que consomem desnecessariamente a bateria.

Considerando a crescente demanda por aplicativos para dispositivos móveis e a forte necessidade de desenvolvê-los em tempo hábil e com boa qualidade, diversos *frameworks* foram propostos como, por exemplo, o *db4o* [DB4a 2012] ou, ainda, a solução ActiveAndroid [Activeandroid 2012]. Entretanto, estas propostas não fornecem uma arquitetura mais simplificada e condizente com as características comuns aos aplicativos para estas plataformas.

3. Objetivo e Relevância deste Trabalho

O principal objetivo deste trabalho é construir um arcabouço para o desenvolver sistemas para Android, denominado AlienDroid, que propiciará aos desenvolvedores maior transparência no acesso a banco de dados nativo deste sistema operacional: SQLite [SQLite 2013]. O AlienDroid fornece uma interface simples e leve, cujo objetivo é não afetar negativamente o desempenho em geral do *software* que o utiliza.

O objetivo do AlienDroid é facilitar a persistência de dados em dispositivos móveis, viabilizando uma transição simples entre o modelo orientado a objetos utilizado no código da aplicação e o modelo relacional do banco de dados embarcado.

Sua característica principal é ser de fácil aplicação.O AlienDroid distingue-se também pelo desempenho, fator preponderante para aplicativos desta natureza. Sua arquitetura é simples e utiliza-se de mecanismos disponíveis no ecossistema do Android para alcançar seus objetivos. Este *framework* também adota as melhores práticas e padrões de projeto para este ambiente, favorecendo a criação de aplicações sem códigos desnecessários ou redundantes.

Este trabalho contribui significativamente para o desenvolvimento de qualquer aplicativo que utilize-se de bancos de dados embarcados em dispositivos móveis. Sua eficácia foi testada no desenvolvimento de dois aplicativos, que estão disponíveis em produção há mais de seis meses, sem problemas de desempenho ou incompatibilidade de dados reportados.

4. Caracterização de Aplicativos para Android

O paradigma de desenvolvimento para dispositivos móveis é relativamente novo e, por este motivo, alguns programadores e analistas ainda não se encontram adaptados a esta nova forma de desenvolver sistemas. Os padrões de projeto e técnicas amplamente adotados em outros paradigmas podem não ser viáveis neste cenário.

Neste momento de transição de um ambiente para outro, aplicativos podem ser escritos com padrões notadamente provenientes de arquiteturas para aplicativos *Web* ou *Desktop*. Como resultado, têm-se sistemas com arquitetura desnecessariamente complexa e que não trazem ganhos em produtividade e facilidade de manutenção futura. Neste caso, aplicativos podem degradar o tempo de vida da bateria do dispositivo, recurso vital para estes aparelhos.

O tempo de vida da bateria, é um desafio a ser superado nesta área [Dantas 2009]. Mesmo com o constante avanço tecnológico, que permitiu a criação de

dispositivos com capacidade de processamento e armazenamento cada vez maiores, o tempo de vida das baterias continua bastante limitado. Um maior processamento permite realizar uma quantidade maior de cálculos em um espaço de tempo menor, contudo, ainda hoje, significa um consumo de energia também maior.

A grande maioria dos dispositivos móveis são dotados de bateria e uma arquitetura desnecessariamente complexa implica em um consumo maior de processamento e, consequentemente, um consumo maior da bateria, diminuindo o tempo de utilização do aparelho [PCWorld 2012].

Sistemas operacionais, como o Android, foram projetados visando a minimizar, e, em alguns casos até evitar, alguns descuidos dos desenvolvedores que possam comprometer o processamento e utilização de memória por parte do aplicativo. Entretanto, nem todos deslizes podem ser previstos pelo sistema operacional, cabendo ao desenvolvedor conhecê-los e evitá-los.

4.1. Aplicativos para Dispositivos Móveis

Devido à natureza dos dispositivos móveis, os aplicativos escritos para estas plataformas tendem a ter um escopo bastante reduzido. Alguns tipos de aplicativos muito comuns em outras plataformas não são viáveis para *smartphones* e *tablets*. Aplicativos que agregam muitas funcionalidades tendem a ser grandes e complexos de usar, indo de encontro com as características inerentes aos dispositivos móveis: entrada de dados textuais limitada, tamanho físico reduzido, tempo de vida da bateria bastante limitado.

Neste contexto, é mais comum encontrar aplicativos que possuem funcionalidades limitadas e com foco concentrado em um único objetivo. De fato, estes aplicativos são, muitas vezes, estendidos com mais funcionalidades por versões específicas para *desktops* ou *Web*. Outra característica destes aplicativos é a utilização extensiva de servidores em Nuvens. Embora guardem informações localmente, para que possam utilizá-las quando não há conexão com a internet, estes dados são mantidos e sincronizados, de tempos em tempos, em um servidor externo.

Trata-se, inclusive, de uma característica desejável, uma vez que é bastante comum que as pessoas tenham mais de um dispositivo móvel. Neste caso, torna-se importante que os mesmos dados inseridos ou atualizados em um dispositivo estejam disponíveis nos demais dispositivos que possuem o mesmo aplicativo.

5. AlienDroid

Existe um esforço, na área de desenvolvimento de aplicações para Android, direcionado para minimizar o impacto decorrente do uso do banco de dados relacional denominado SQLite. Os bancos de dados relacionais são considerados, hoje, a forma de armazenamento mais eficiente e confiável, entretanto, o modelo orientado a objetos possui uma semântica diferente e incompatível com a semântica do modelo relacional.

Com o intuito de concretizar o mapeamento entre estas duas tecnologias díspares, várias ideias foram sugeridas, convergindo para o conceito de *frameworks* para persistência de objetos. Existe atualmente no mercado um pequeno número de soluções que realizam o mapeamento objeto-relacional, entretanto, nota-se que o desenvolvimento de aplicações, através destas propostas, torna-se demasiadamente complexo.

Isto ocorre, principalmente, devido a estas soluções adotarem padrões de projeto que não são condizentes com as características das aplicações para dispositivos móveis. De fato, em alguns casos, estas características são totalmente desprezadas, criando-se um *framework* que acarreta em mais código para ser produzido, além de ter um impacto negativo no desempenho das aplicações.

5.1. Trabalhos Correlatos

Em [DB4o 2012] é apresentada a solução denominada DB4o. A empresa responsável por este *framework* destaca-se pela criação de soluções de persistência para outras plataformas, como o .NET da Microsoft e o Java da Oracle. O DB4a apresenta uma solução que se assemelha ao padrão de projeto *Persistence Broker* [Suurjak 2001], na qual uma interface de alto nível é provida para interação com o mecanismo de persistência.

O principal ponto negativo encontrado nesta solução refere-se ao processo de utilização da ferramenta. Observou-se uma complexidade de média para alta em sua adoção. Este fato decorre do fato de esta ferramenta ser direcionada não apenas para o desenvolvimento em Android, mas para aplicações na plataforma Java, de forma geral. Neste contexto, não foram identificadas características que facilitassem o desenvolvimento especificamente para o ambiente do Android.

Em [Activeandroid 2012] é apresentada a solução ActiveAndroid, um *framework* baseado no padrão de projeto ActiveRecord. O ActiveAndroid caracteriza-se por ser uma solução aberta e que implementa o padrão ActiveRecord, embora possua pequenas diferenças com relação a este padrão. De forma contrária ao proposto por este padrão, o ActiveAndroid desacopla o mecanismo de consulta em um módulo separado.

A ferramenta é de simples uso, contém boa documentação e está disponível em repositórios públicos. Como ponto negativo, destaca-se a necessidade de se utilizar o mecanismo de anotações para determinar informações como o nome da tabela e o nome dos campos para os quais uma classe será mapeada. Trata-se de informações que podem ser tratadas de forma automática, sem necessidade de interferência do desenvolvedor.

5.2. ActiveRecord

Neste padrão para mapeamento objeto-relacional, a classe que contém as regras de negócio, também denominada de classe de domínio, é também responsável pela lógica necessária para realizar sua própria persistência no banco de dados [Fowler 2003].

Este padrão tornou-se bastante difundido a partir do *framework* Ruby on Rails [Rails 2012], escrito na linguagem Ruby [Ruby 2012]. O padrão caracteriza-se principalmente por sua simplicidade em solucionar o problema de mapeamento objeto-relacional, contudo, é fortemente indicado apenas para sistemas de pequeno porte. Neste padrão, o próprio objeto a ser persistido conterá os métodos que permitirão gravar, recuperar, remover e consultar suas informações no banco de dados.

5.3. Arquitetura Interna

O funcionamento interno do AlienDroid está baseado no padrão arquitetural ActiveRecord, que sugere a incorporação das regras para persistência dos objetos diretamente nas classes de domínio da aplicação. Esta estratégia, embora possua opositores quanto a sua aplicação em sistemas de grande porte, apresenta-se extremamente útil no âmbito dos aplicativos para Android.

Aplicativos para dispositivos móveis, incluindo o Android, tendem a ser pequenos e com escoporeduzido. Neste contexto, o padrão ActiveRecord destaca-se por não tornar o aplicativo desnecessariamente complexo, incorporando regras e características indesejáveis.

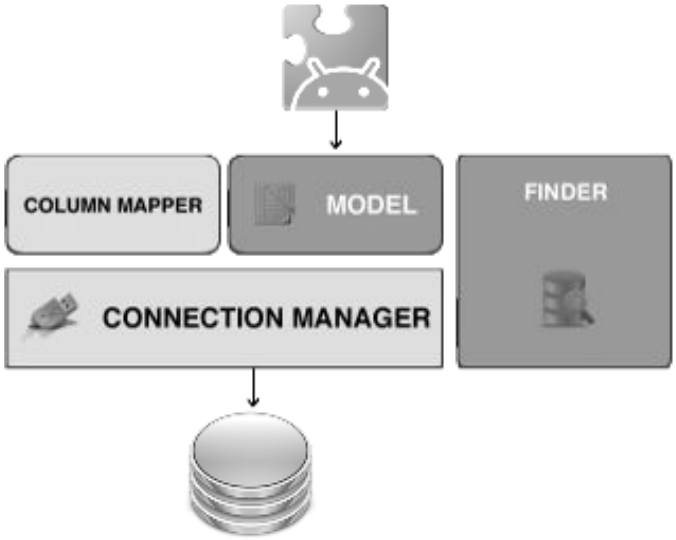

Figura 1. Arquitetura Interna do AliendDroid.

Através da Figura 1 percebe-se que toda a comunicação entre o AlienDroid e as aplicações que o instanciarão será realizada pelo componente denominado *Model*. Os desenvolvedores não necessitarão ter conhecimentos avançados sobre a arquitetura interna, a não ser a interface pública fornecida por este componente que encapsula todas as informações necessárias para realizar o mapeamento de objetos para uma forma equivalente no SQLite.

No decorrer deste trabalho, os componentes desta arquitetura serão elucidados com mais detalhes como, por exemplo, os componentes *ColumnMapper*, *ConnectionManager* e *Finder*. Para tornar mais clara a arquitetura apresentada, pode-se afirmar, neste momento, que todos os objetos de negócio que serão persistidos através do AlienDroid deverão herdar da classe *Model*.

Distingue-se nesta arquitetura, principalmente, sua simplicidade. Grande parte do código responsável pela interação com o SQLite encontra-se na classe *Model*. Esta arquitetura simplificada é proposital, visando uma menor sobrecarga para os aplicativos que utilizarão o AlienDroid. Cabe destacar, também, que o *framework* utiliza-se do recurso de Reflexão em poucas ocasiões, reduzido apenas para a tarefa de definir os valores dos atributos dos objetos. A reflexão, embora de grande utilidade para a grande maioria dos *frameworks* em geral, degrada o desempenho de aplicativos escritos nesta linguagem [Attard 2008].

O recurso de anotações também não foi adotado no AlienDroid. Sua utilização, embora facilite a codificação, traz uma sobrecarga extra para o *framework* devido à Reflexão: para obter os dados provenientes das anotações, faz-se necessário recorrer a esta característica da linguagem.

O diagrama de classes da Figura 2 demonstra as partes que compõem este *framework*. Pode-se ver as classes *Model*, *ConnectionManager*, *Finder*, *Where* e *Query*. A classe *Model* representa o próprio componente de mesmo nome, assim como a classe *ConnectionManager*. As classes *Finder, Where* e *Query*, entretanto, representam o componente *Finder*. Cada classe desta será melhor detalhada nas seções subsquentes.

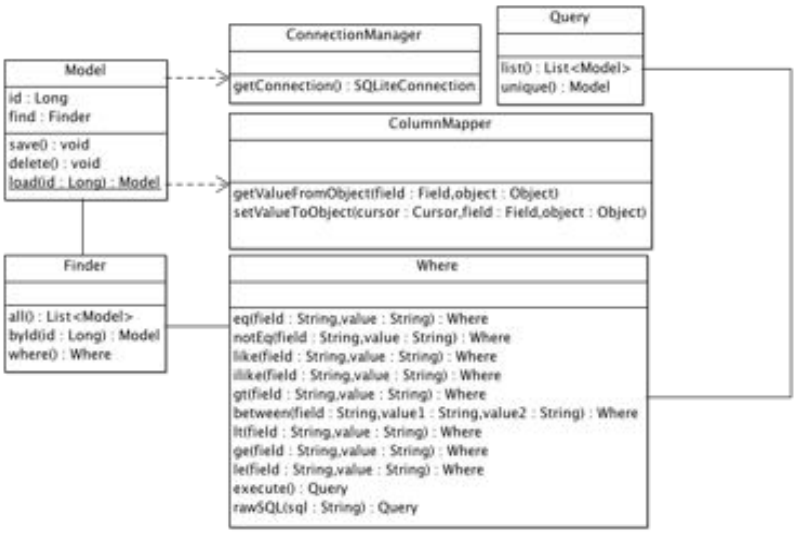

Figura 2. Diagrama de Classes

5.4. Mapeamento de Classes

O mapeamento de uma classe de domínio do projeto para o banco de dados não requer grande esforço. Para que isto ocorra, basta que toda classe de domínio estenda da classe *Model,* própria do *framework*. Não há a necessidade de se utilizar anotações para informar quais atributos serão persistidos ou para indicar informações extras sobre a classe ou a tabela.

De fato, todo atributo de uma classe torna-se automaticamente elegível para ser mapeado para uma coluna em uma tabela. Caso não seja este o objetivo, deve-se informar explicitamente esta intenção adotando a palavra-chave *transient*, nativa da linguagem Java, em conjunto com o atributo.

O AlienDroid criará uma tabela para cada classe de domínio, onde cada atributo desta classe será representado por uma coluna. Conforme destacado, aplicativos para Android não possuem a característica de serem grandes e complexos. De fato, são aplicativos de foco bastante reduzido e contendo poucas, senão nenhuma, regra de negócio embutida. Deste modo, considerou-se desnecessário fornecer no AlienDroid mecanismos que permitam a modificação do nome das tabelas e atributos criados. Esta decisão também vai ao encontro do objetivo de tornar esta solução simples, requerendo pouca intervenção do seu usuário para utilizá-la.

5.5. Mapeamento de Heranças

De acordo com a arquitetura proposta para o AlienDroid, o padrão *One Inheritance Path One Table*, sugerido por [Keller 1997], foi escolhido para tratar do mapeamento de hierarquia de classes, conforme os seguintes critérios:

- Requer uma quantidade menor de operações de junções entre tabelas para obter um único objeto, favorecendo um melhor desempenho;
- Requer uma quantidade menor de operações de inserção para incluir um u nico objeto. Não ocorre muito desperdício de espaço com colunas vazias, similar ao

que ocorre com o padrão *One Inheritance One Table* [Keller 1997];

- Possui uma estrutura mais simples de ser implementada em comparação com o padrão *One Class One Table* [Keller 1997].

Seguindo este padrão, cada classe concreta e persistente do sistema será mapeada para uma tabela própria no banco de dados.

5.6. Mapeamento de Relacionamentos

O AlienDroid não provê qualquer forma de se realizar o mapeamento de relacionamentos entre objetos. Esta decisão deve-se, principalmente, devido à necessidade preponderante de se manter um grau de desempenho satisfatório do AlienDroid.

Concluiu-se, também, que o tratamento de relacionamentos é bastante imprevisível, podendo alcançar um nível hierárquico que pode comprometer sensivelmente o desempenho geral da aplicação. Diante destes fatos, considerou-se que este mapeamento não será idealizado pelo *framework*, sendo necessário que o próprio desenvolvedor do aplicativo o resolva de forma manual.

5.7. Componente ColumnMapper

O objetivo deste componente é traduzir um tipo nativo da linguagem Java para um tipo nativo do banco de dados. Embora esta tarefa pudesse ser realizada diretamente pelo componente *Model*, decidiu-se por separar esta tarefa em um componente específico, visando a uma maior modularidade e facilidade de manutenção futura.

Conforme exibido no diagrama de classes da Figura 2, este componente é representado pela classe *ColumnMapper*, que contém apenas dois métodos, responsáveis por definir um valor oriundo de uma tabela em um objeto e, também, obter o valor de um objeto para inserção na tabela.

5.8. Componente Finder

Este componente tem como objetivo prover um mecanismo de consulta que facilite a recuperação de objetos do banco de dados. O Finder provê uma interface com comportamentos que permitem obter listas de objetos ou objetos simples.

Através deste componente é possível, para o desenvolvedor, obter uma lista completa de todos os objetos de um determinado tipo que estão persistidos e realizar consultas específicas através de filtros usando determinados atributos.

O componente é exibido no diagrama de classe da Figura 2 através das classes *Query*, *Finder* e *Where*. Do ponto de vista do desenvolvedor, o componente será utilizado, inicialmente, através da classe *Finder*. Através desta classe, pode-se solicitar uma listagem completa de todos os objetos persistidos ou um único objeto, usando, para isto, seu identificador.

Caso haja necessidade de uma consulta mais elaborada deve-se utilizar o método *where()*, que provê meios para realizar filtros usando todos os atributos do objeto. Finalizando, há também a possibilidade de se utilizar consultas nativas, usando, para isto, o método *rawSQL()*.

5.9. Geração de Identificadores

Embora existam diversos padrões para a geração de identificadores de objetos, conforme discutido anteriormente, decidiu-se adotar o mecanismo padrão do próprio SQLite. Esta decisão deve-se à principal característica desejada ao AlienDroid: desempenho.

5.10. Componente de Conexão

O tratamento necessário para se realizar e manter uma conexão ativa com o banco de dados SQLite é fornecido através do componente denominado Connection Manager. Trata-se de um componente que utiliza-se dos meios já fornecidos pelo Android para estabelecer conexões com o SQLite.

Neste contexto, este componente contém uma instância de *SQLiteOpenHelper*, um gerenciador nativo de conexões com o SQLite. Contudo, visando não tornar o AlienDroid totalmente acoplado com o SQLite, o componente ConnectionManager abstrai das demais partes do AlienDroid esta instância de *SQLiteOpenHelper*.

6. Exemplo de Uso

Esta seção tem como principal objetivo descrever e exemplificar todo o processo de utilização do AlienDroid, onde uma pequeno exemplo será desenvolvido seguindo todas as etapas necessárias para transforma-la em uma instância deste *framework*. Cabe destacar que este exemplo não possui nenhuma utilidade prática, tendo como objetivo unicamente ilustrar a utilização do AlienDroid.

6.1. Primeiro Passo

A primeira etapa no desenvolvimento de uma aplicação que adota o AlienDroid é garantir que todas as classes que deverão ter seu estado persistido herdem, obrigatoriamente, de *Model*, conforme destacado no trecho de código da Listagem 1.

```
01. public class Avaliacao extends Model {
02.     public String email;
03.     public String comentario;
04. }
```

Listagem 1. Criando uma Classe de Modelo.

Ao estender *Model* a classe herdará, consequentemente, o atributo *id*, responsável por manter o identificador do objeto. Também terá a sua disposição um conjunto de métodos que permitirão realizar a inserção, atualização e remoção deste objeto da tabela correspondente ao tipo do objeto.

6.2. Segundo Passo

O segundo passo é opcional e consiste em definir o nome e a versão do banco de dados. Para isto, faz-se necessário editar o arquivo *AndroidManifest.xml*, adicionando duas novas linhas, conforme ilustra o trecho de código da Listagem 2.

```
01. <meta-data android:name="DATABASE_NAME"
android:value="mydatabase.sqlite"/>
02. <meta-data android:name="DATABASE_VERSION" android:value="1"/>
```

Listagem 2. Configurando o AndroidManifest.xml.

A primeira linha destaca o código necessário para informar o nome do arquivo que será criado para manter o banco de dados, enquanto a segunda linha demonstra como informar a versão do banco de dados.

6.3. Terceiro Passo

Para que o AlienDroid esteja ativo, deve-se criar uma classe que estende da classe *android.app.Application*, nativa do Android. Ao estender esta classe, deve-se implementar o método *onCreate()*, adicionando a linha *AlienDroid.init(this)*. Este passo está ilustrado no trecho de código da Listagem 4.

```
01. public class MinhaApplication extends android.app.Application {
02.
03.       @Override
04.       public void onCreate() {
05.                 super.onCreate();
06.                 AlienDroid.init(this);
07.       }
08. }
```

Listagem 3. Estendendo de android.app.Application

6.4. Consultas e Atualizações

O AlienDroid fornece mecanismos bastante simplificados para realizar a consulta e atualizações de dados no banco de dados. De fato, toda sua estrutura está totalmente baseada no projeto ActiveRecord, adotando todas as recomendações e boas práticas adotadas na implementação deste padrão por outros *frameworks*, como o Ruby on Rails. O trecho de código da Listagem 4 demonstra um exemplo de utilização do AlienDroid para realizar a inserção de um objeto no banco de dados.

```
01. public MyActivity extends Activity {
02.
03.   @Override
04.   public void onCreate(Bundle savedInstanceState) {
05.     super.onCreate(savedInstanceState);
06      setContentView(R.layout.main);
07.
08.     Contact contact = new Contact();
09.     contact.name = "My Name";
10.     contact.birth = new Date();
11.     contact.age = 21;
12.     contact.save();
13.    contact.age = 23;
14.    contact.save();
15.    contact.delete();
16.   }
17. }
```

Listagem 4. Inserindo e Removendo.

A oitava linha apresenta a criação de um objeto denominado *Contact*. Este objeto, posteriormente, nas linhas nove até onze, tem seus atributos definidos com dados fictícios. Na linha doze, tem-se a solicitação para que o AlienDroid persista estes dados.

Ainda é possível observar, na linha treze, a alteração do valor de um atributo e

consequente solicitação para que esta nova informação seja atualizada no banco de dados. Este trecho finaliza com o pedido para que os dados deste objeto sejam removidos do SQLite, através da linha quinze.

```
01. public MyActivity extends Activity {
02.
03.     @Override
04.     public void onCreate(Bundle savedInstanceState) {
05.         super.onCreate(savedInstanceState);
06.         setContentView(R.layout.main);
07.
08.         Contact contact = Contact.find.byId(1);
09.         contact.name = "Changed Name";
10.         contact.save();
11.
12.         List<Contact> contacts = Contact.find.all();
13.         List<Contact> contacts =
Contact.where().eq(“name”,”teste”).execute().list();
14.     }
15.   }
16. }
```

Listagem 5. Consultando Dados.

O trecho de código da Listagem 5 finaliza esta seção apresentando como o AlienDroid permite que consultas sejam realizadas. A linha oito exemplifica como se deve proceder para obter os dados de uma determinada classe, através do método byId*()* da classe *Finder*. Este método recebe o identificador do objeto que se quer obter.

A linha doze demonstra a obtenção de todos os objetos, de uma determinada classe, que estão armazenados em uma tabela. Finalizando, a linha treze apresenta uma consulta personalizada de objetos do tipo *Contact*, no qual utiliza-se do componente *Finder* para realizar a consulta.

7. Caso de Sucesso e Testes de Desempenho

O AlienDroid foi adotado em dois aplicativos desenvolvidos para Android. Estes aplicativos encontram-se disponíveis na loja Google Play. Os aplicativos, há seis meses em produção, até o momento da escrita deste artigo, não apresentaram problemas relacionados ao desempenho. O uso do AlienAndroid facilitou o desenvolvimento destas aplicações.

Houve uma considerável diminuição em linhas de código, uma vez que não houve mais necessidade de se escrever instruções SQL para a consulta de objetos no banco de dados. A manutenção do código também apresentou-se mais fácil, sem a necessidade de transferir o foco do código em Java para códigos em SQL.

Em testes de laboratório, detectou-se uma pequena perda de desempenho de um aplicativo adotando o AlienDroid, quando comparado a um que não o utiliza. Esta perda era aguardada, uma vez que o mecanismo de Reflexão foi utilizado. Contudo, como a utilização deste mecanismo foi criteriosa e, portanto, bastante reduzida, tornou-se possível reduzir seu impacto no desempenho geral do *framework*.

Os testes foram realizados em emuladores e verificando o tempo de execução entre as chamadas para inserir, atualizar, remover e consultas todos os objetos de uma tabela. Observou-se que a utilização de reflexão aumentou quase em cem vezes o tempo para realizar estas operações, enquanto, sem o seu uso, o tempo diminuiu consideravelmente.

De fato, as mesmas operações, quando realizadas diretamente em código SQL e sem o AlienDroid, são mais rápidas. Mas perdem em outros aspectos, como a capacidade de ter uma manutenção mais fácil e tempo de codificação reduzido.

8. Conclusão e Resultados Obtidos

O estudo apresentado neste trabalho propôs a pesquisa e implementação de uma solução para desenvolvimento de aplicações para Android que provesse serviços de persistência para bancos de dados relacionais, e que tivesse uma boa capacidade de personalização e desempenho satisfatório. Apresentou-se como maior desafio neste trabalho a necessidade de se garantir um bom desempenho. Considerando as fortes restrições tanto do Android, como dos dispositivos que rodam este sistema operacional, esta característica foi alcançada ao custo do descarte de algumas funcionalidades que trariam maiores facilidades para o desenvolvedor.

Frente às soluções existentes para mapeamento objeto-relacional em Android, o AlienDroid apresentou-se superior em determinados aspectos. Embora nenhuma técnica científica de mensuração tenha sido adotada, pôde-se notar este fato através da reduzida necessidade de se estender classes ou, também, de implementar interfaces próprias do *framework*. Leva-se em consideração, também, o fato de o AlienDroid não impor regras demasiadamente extensas no processo persistência de um objeto, principalmente em comparação ao db4o.

Em contrapartida com o exemplo apresentado, também foi desenvolvido um exemplo sem a utilização do AlienDroid. Verificou-se uma quantidade maior de código fonte produzido, com cerca de vinte até trinta por cento de linhas de código a mais, principalmente para operações de consulta. Houve também uma grande mescla de código fonte em Java e SQL, dificultando ainda mais a manutenção.

Comparando estes dois exemplos, foi identificado como ganho no uso do AlienDroid: quantidade menor de código para ser produzido, não há necessidade de uso de instruções em outra linguagem (SQL), manutenção de código mais simples. A principal desvantagem verificada refere-se à questão do desempenho, pois ainda há a necessidade do uso de reflexão. O uso do mecanismo nativo de consulta do banco de dados do Android possui desempenho melhor, pois trata-se de um acesso direto, sem uso de recursos extras, como a reflexão para criar objetos.

8.1. Trabalhos Futuros

Como propostas para continuidade a este trabalho, destacam-se: Pesquisa e implementação de um gerenciador de transações, implementação do mecanismo de Inicialização Lenta para coleções, melhorar ainda mais o desempenho, buscando formas para evitar totalmente o uso de reflexão.

Referências

Attard, Albert. (2013) “Reflection in Action”, http://today.java.net/pub/a/today/2008/02/12/reflection-in-action.html, Fevereiro.

Saylor, Michael. “The Mobile Wave”. Vanguard Press.

PCWorld. (2013) “Why Your Smartphone Battery Sucks” http://www.pcworld.com/article/228189/why_your_smartphone_battery_sucks.html, Agosto.

Android. (2013) **“**Sistema Operacional Android”. http://www.android.com/, Fevereiro.

SQLite. (2013) “SQL Database Engine”. http://www.sqlite.org/, Fevereiro.

Ambler, Scott W. (2003) “Agile database techniques” http://www.agiledata.org/essays/impedanceMismatch.html”, Agosto.

Ambler, Scott W. (2009) “Mapping objects to relational databases”. http://www-106.ibm.com/developerworks/webservices/library/ws-mapping-to-rdb/, Agosto.

Dantas, Valéria. (2009) “Requisitos para Testes de Aplicações Móveis”. Dissertação de Mestrado. Universidade Federal do Ceará.

DB4o. (2012) “Database for Android”. http://www.db4o.com, Fevereiro.

Activeandroid. (2012) “Active Record Style SQLite Persistence for Android” http://www.activeandroid.com/, Agosto.

Fowler, Martin. Patterns of Enterprise Application Architecture. Boston: Addison Wesley, 2003.

Rails. (2012) “Ruby on Rails”. http://www.rubyonrails.org/, Agosto.

Ruby. (2012) “The Ruby Programming Language”. http://www.ruby-lang.org/, Agosto.

Keller, Wolfgang. (1997) “Mapping objects to tables: a pattern language”. In: European Conference on Pattern Languages of Programming Conference (EuroPLOP97), Irsee, Germany, 1997. Proceedings... Irsee, 1997. http://citeseer.ist.psu.edu/rd/85059614%2C123497%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/2720/http:zSzzSzwww.sdm.dezSzgzSzarcuszSzpublicatzSzmapo2t.pdf/keller97mapping.pdf, Agosto.

Suurjak, Erki. (2001) “Javal baseeruv objektide püsivuse kiht Jakamar: Bakalaureusetöö”. 100 f. Tese (Masters of Philosophy) – Instituto de Informática, Tallinn University of Technology, Tallin, Estônia.

Date, C. (2000) “Introdução a sistemas de bancos de dados”. Tradução Vandenberg Dantas de Souza. 7. ed. Rio de Janeiro: Campus, 2000. 803 p. Tradução de: Plublicare Consultoria e Serviços.

Silberschatz, Abraham; Korth, Henry F.; Sudarshan, S. (1999) “Sistema de banco de dados”. 3. ed. São Paulo: Makron Books, 1999. 806 p.

Yoder, Joseph W.; Johnsson, Ralph E.; Wilson, Quince D. (1998) “Connecting business objects to relational databases”. Urbana, Illinois, 1998. http://www.joeyoder.com/Research/objectmappings/Persista.pdf, Agosto.