Universidade Federal do Rio de Janeiro – UFRJ
Instituto de Matemática – IM
Departamento de Ciência da Computação – DCC

REDUÇÃO DO ACOPLAMENTO COM FRAMEWORKS ESPECÍFICOS DE PLATAFORMA NO MDARTE: ESTUDO DE CASO EM AMBIENTES MÓVEIS

Bianca Munaro Lima, Diogo Borges Lima e
Erich de Souza Oliveira

Orientador: Geraldo Zimbrão da Silva – D.Sc. COPPE/UFRJ
Co-Orientador: Rodrigo Salvador Monteiro – D.Sc. COPPE/UFRJ

2011

REDUÇÃO DO ACOPLAMENTO COM FRAMEWORKS ESPECÍFICOS DE PLATAFORMA NO MDARTE: ESTUDO DE CASO EM AMBIENTES MÓVEIS

Bianca Munaro Lima, Diogo Borges Lima e Erich de Souza Oliveira

Projeto Final de Curso submetido ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

APROVADA em _____, ___ de _____ de 2011.

_____
Geraldo Zimbrão da Silva - D.Sc. COPPE/UFRJ

_____
Rodrigo Salvador Monteiro - D.Sc. COPPE/UFRJ

_____
Geraldo Bonorino Xexéo - D.Sc. COPPE/UFRJ

_____
Leonardo Guerreiro Azevedo - D.Sc. Unirio

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2011

Resumo

REDUÇÃO DO ACOPLAMENTO COM FRAMEWORKS ESPECÍFICOS DE PLATAFORMA NO MDARTE: ESTUDO DE CASO EM AMBIENTES MÓVEIS

Bianca Munaro Lima, Diogo Borges Lima e Erich de Souza Oliveira

Orientador: Geraldo Zimbrão da Silva
Co-orientador: Rodrigo Salvador Monteiro

Cada vez mais a Arquitetura Dirigida a Modelos (MDA – Model-Driven Architecture) vem sendo utilizada em sistemas reais, e um dos principais fatores para sua adoção é o grande auxílio que os frameworks MDA têm proporcionado aos desenvolvedores durante a construção de sistemas robustos e de qualidade. Infelizmente, a geração completa de aplicações a partir das transformações de modelos ainda não é uma realidade. Ao invés disso, as ferramentas de MDA disponíveis permitem a geração de parte da aplicação e fornecem pontos de implementação para que os desenvolvedores introduzam código escrito diretamente na plataforma alvo. O uso dessa abordagem, conhecida como geração parcial de código, tende a produzir pontos de implementação com código fortemente acoplado aos frameworks específicos de plataforma em uso, diminuindo a portabilidade do sistema. Este trabalho tem como propósito definir uma interface padrão entre os pontos de implementação e os frameworks específicos de plataforma, visando a redução do acoplamento entre os mesmos. De forma a avaliar a solução proposta, apresentamos um estudo de caso onde uma pequena aplicação, desenvolvida originalmente para um servidor de aplicação, foi portada para um dispositivo móvel.

Abstract

COUPLING REDUCTION WITH PLATFORM SPECIFIC FRAMEWORKS IN THE MDARTE: A CASE STUDY IN MOBILE ENVIRONMENTS

Bianca Munaro Lima, Diogo Borges Lima e Erich de Souza Oliveira

Supervisor: Geraldo Zimbrão da Silva
Co-Supervisor: Rodrigo Salvador Monteiro

Increasingly, the Model Driven Architecture – MDA, has been used in real systems, and one of the main factors for its adoption is the great assistance that the MDA frameworks have provided for developers to build robust and quality systems. Unfortunately, the generation of complete applications from models is not yet a reality. Instead, MDA tools available allow for generating part of the application and provide implementation points for developers to introduce code written directly on the target platform. Using this approach, known as partial code generation, it tends to produce implementation points with code tightly coupled to the platform-specific frameworks in use, reducing the portability of the system. This work aims to define a standard interface between the implementation points and platform-specific frameworks, in order to reduce coupling between them. In order to evaluate the proposed solution, we present a case study where a small application originally developed for an application server has been ported to a mobile device.

Agradecimentos

Agradeço primeiramente aos meus pais Maria Lúcia Munaro Lima e Jorge Edmundo Lima por todo o apoio e compreensão, não só na graduação, mas em toda a minha vida. Ao meu irmão, Felipe Munaro Lima pelos momentos de diversão e alegria.

Aos amigos da graduação que sempre trocaram conhecimentos e organizaram grupos de estudos para que todos se ajudassem no aprendizado das matérias e aos que, apesar de não serem da graduação, sempre me deram forças, ouvindo meus problemas e me apoiando.

Ao meu namorado, Rogério Almada de Angelis que, sendo também estudante da computação, me ajudou nas questões técnicas sempre estando ao meu lado, mesmo nos momentos mais difíceis.

A todos que souberam entender uma resposta breve ao telefone, um convite não aceito e tantos outros senões...

Aos meus companheiros de projeto final, Erich de Souza Oliveira e Diogo Borges Lima que sempre estiveram presentes, trabalhando em conjunto e enfrentando as dificuldades desse trabalho e da graduação.

Ao meu orientador Geraldo Zimbrão da Silva e coorientador Rodrigo Salvador Monteiro pela dedicação, nos auxiliando no desenvolvimento do projeto, sanando nossas dúvidas e contribuindo com a evolução desse trabalho.

E a Deus, que colocou tantos anjos em meu caminho que me iluminaram e acompanharam em mais esta etapa da minha vida.

Bianca Munaro Lima

Agradecimentos

Agradeço, em primeiro lugar, a Deus pelas oportunidades que me foram dadas e pelas pessoas que conheci as quais me proporcionaram a evoluir meu aprendizado.

Meus agradecimentos também à minha mãe, Mônica Maria Borges Lima, e meus avós, Rita do Amaral Borges e Benedito de Aragão Borges por sempre terem me apoiado ao longo de minha vida, e por terem me educado para que hoje eu pudesse ser a pessoa que sou.

Não posso deixar de agradecer minha irmã, Aline Thaís Borges Lima, por ter me dado forças nessa jornada que é a faculdade, e por seus filhos, Davi Borges Lima Stallone e Jonatas Borges Lima Stallone, pelas diversas brincadeiras.

Aos amigos de Projeto Final, Bianca Munaro Lima e Erich de Souza Oliveira, pelas ajudas ao longo de toda graduação e pelos auxílios no término deste trabalho, além de proporcionar vários momentos bons.

Aos diversos amigos que conheci durante a graduação que apesar dos momentos difíceis, sempre me deram forças para seguir em frente. Dentre eles cabem lembrar: Daniel, Felipe Martinez, Francisco, Bruno Buss, Flávio França, Jonas Arêas, Rafael Oliveira, Marcos Serpa, Débora, Marcelo e Thiago. Destaco ainda Diego Xavier, Diego Cardoso, Vinícius Ribeiro, Patrícia Zudio, Davi Vercillo e Pedro Alberto por sempre se portarem como verdadeiros irmãos em todos os momentos.

Agradeço aos meus professores por me ajudar a crescer profissionalmente. Agradeço principalmente meu orientador Geraldo Zimbrão da Silva e meu co-orientador Rodrigo Salvador Monteiro pelas várias horas dedicadas ao meu auxílio no desenvolvimento deste Projeto.

E por fim, agradeço a todos os outros amigos que apesar de não fazerem parte da minha vida acadêmica me ajudaram de diversas formas. Agradeço à Lívia, Vitor Rodrigues, Vitor Cezário, Guilherme, Camila, Marcelle, Jillian, Fabiana, Suzana, Glaucia, Wallace, Ivone e Mariana Areas por tornar as 4 horas diárias de ida e vinda à faculdade menos dolorosas. Agradeço à Flávia Oliveira, Gabriel Tirre, Danielle de Sá, Paulo de Souza, Philipe, Marcelo Vargas, Denise, Bruno, Rodrigo, Isabella Abreu e Isabel Vilela por sempre estarem presentes em minha vida como verdadeiros irmãos.

Diogo Borges Lima

Agradecimentos

Não posso cair no erro de citar nomes, pois vou acabar sendo injusto, com pessoas que me ajudaram a ser quem sou hoje, seria injusto também com as pessoas que ainda irei conhecer e que farão parte da minha história.

Em primeiro lugar, agradeço a minha família, por toda educação, amor e carinho, que me foram dados, pelo apoio psicológico e financeiro que sempre me deram, desde o início da vida, por acreditarem em mim e nos meus sonhos, por confiarem nas minhas escolhas, por estar comigo em todos os momentos, mesmo naqueles em que o mundo ameaçava desabar a minha volta, nada disso seria possível sem a inestimável ajuda deles.

Em segundo lugar, agradeço aos meus orientadores neste projeto, por todo o tempo investido, por acreditarem na nossa capacidade de concluir o que foi proposto e por sanar minhas dúvidas.

Não posso deixar de agradecer aos meus amigos mais próximos que me acompanharam nesta trajetória, por todo apoio que me deram, pelos momentos que me fizeram entender que deveria dar um tempo a mim mesmo, por terem aceitado as minhas ausências, pelas festas e também pelos dias de estudo, por partilharem comigo de toda as alegrias e tristezas que encontrei na minha vida. Dentre estes, especialmente, e desta vez nominalmente, Diogo Borges e a Bianca Munaro, por acreditarem junto comigo neste projeto, e por termos juntos, concretizado este desafio.

Agradeço a todos os amigos de trabalho, por me apresentarem ao mundo do MDA e me ajudar nos meus primeiros passos, graças a eles, hoje sou um profissional melhor e capaz de grandes realizações na área em que escolhi trabalhar.

Erich de Souza Oliveira

Índice

Lista de Figura

1) Introdução

Hoje em dia, a forte competição no mercado de desenvolvimento de software exige que as empresas tenham diferenciais para ter sucesso. Dentre estes podemos destacar a velocidade de criação e qualidade do sistema. A Engenharia de Software é uma área da computação que tem por objetivo garantir organização, produtividade e qualidade no desenvolvimento de sistemas e, para tal, utiliza técnicas e padrões de especificação e manutenção de sistemas. Dentre essas técnicas, chama a atenção a Arquitetura Dirigida a Modelos (MDA[1] – Model-Driven Architecture), um padrão OMG[2] (Object Management Group), que consiste em criar softwares a partir de modelos que expressem ações e estados do sistema. Dessa forma existe a garantia de adequação entre a documentação, obtida a partir dos modelos que representam a aplicação, e o produto final.

A UML[3] (Unified Modeling Language) e o MOF[4] (Meta Object Facility), ambos padrões da OMG, são os pilares principais da abordagem MDA. A UML é usada para modelar a aplicação, enquanto que o MOF é uma linguagem de metamodelagem padronizada. O MOF tem como papel fundamental padronizar o acesso a todas as informações passíveis de representação pela UML sendo crucial para a automação de procedimentos baseados em informações contidas nos modelos.

A MDA tem por objetivo gerar todo o código fonte do sistema a partir de modelos UML. Entretanto, não existe atualmente uma ferramenta que alcance plenamente esse objetivo, com exceção de alguns casos dedicados a domínios específicos de aplicação. De uma forma geral, a abordagem MDA vem se popularizando através da abordagem de geração parcial de código, onde parte da aplicação é gerada a partir das transformações de modelos e pontos de implementação são fornecidos para que os desenvolvedores introduzam código escrito diretamente na plataforma alvo.

Segundo a abordagem MDA, numa primeira etapa do desenvolvimento, a aplicação deve ser representada através de um modelo com um alto nível de abstração que seja independente de qualquer tecnologia, ou seja, um Modelo Independente de Plataforma (PIM[5] - Platform Independent Model) . Posteriormente, o PIM deverá ser transformado em um ou mais Modelos Específicos de Plataforma (PSM[6] - Platform Specific Model) que finalmente serão transformados em código.

O modelo PIM facilita a portabilidade do sistema, fazendo com que ele possa migrar para outra plataforma com menor esforço. Entretanto, na abordagem de geração parcial os pontos de implementação continuam tendo que ser migrados manualmente. Um fator crítico que aumenta a complexidade da migração dos pontos de implementação é o seu forte acoplamento com frameworks específicos de plataforma.

Este trabalho tem por objetivo definir uma API[7] a ser usada pelos desenvolvedores nos pontos de implementação com a finalidade de reduzir o acoplamento dos mesmos com frameworks específicos de plataforma. Com esta abordagem, a tarefa de migração dos pontos de implementação fica segmentada em dois esforços: (1) fornecer uma nova implementação da API para a nova plataforma alvo e (2) migrar o código dos pontos de implementação usando a API. Esta segmentação, além de organizar melhor o trabalho de migração, também reduz o esforço total, principalmente quando o código a ser migrado já faz uso da API. Para melhor avaliação e compreensão da proposta, apresentamos um protótipo desenvolvido para comprovar sua aplicação em cenários reais. O protótipo se baseia no framework MDArte[8] utilizado na elaboração e manutenção de uma série de projetos reais que se encontram em plena produção.

No capítulo 2, é exposta a Fundamentação Teórica, onde será explicado todo material base para a compreensão do trabalho. No capítulo 3, é mostrada a Solução Proposta, onde se falará sobre a API desenvolvida e quais os benefícios do seu uso. É mostrado também como estendê-la e implementá-la em projetos de sistemas reais. No capítulo 4, um Estudo de Caso é apresentado, baseado em uma migração para uma plataforma móvel, onde atualmente são encontrados grandes desafios, devido a heterogeneidade dos dispositivos; aqui é implementada a API proposta para a plataforma Android[9]. Por fim, no capítulo 5 é apresentada a Conclusão e sugestões de trabalhos futuros.

2) Fundamentação teórica

O objetivo deste Capítulo é descrever conceitos que serão explorados ao longo do trabalho. Primeiramente, serão explicados a metodologia MDA e os frameworks AndroMDA[10] e MDArte[8] que são utilizados no projeto. Posteriormente, será apresentada uma introdução sobre ambientes móveis e a plataforma Android para que seja compreendido o estudo de caso desenvolvido.

2.1) MDA (Model-Driven Architecture)

O Object Management Group, ou OMG, é uma organização internacional sem fins lucrativos criada em 1989 que define padrões para aplicações orientadas a objetos. Em 2001 seus membros votaram na MDA como sua arquitetura base.

O objetivo da arquitetura dirigida a modelos(MDA – Model-Driven Architecture) é separar a especificação do sistema das capacidades da sua plataforma. Ela reconhece a importância dos modelos, tornando-os o ponto-chave no desenvolvimento, focando em portabilidade, interoperabilidade, reuso e documentação.

A metodologia da MDA define que inicialmente deve existir um Modelo Independente de Computação (CIM) baseado nos requisitos do sistema. O CIM também é chamado de modelo de domínio e omite os detalhes das estruturas e processamento.

A partir do CIM é gerado o Modelo Independente de Plataforma (PIM - Platform Independent Model). Na maioria das vezes, esse processo é manual. O PIM mostra a parte da especificação que não muda com a alteração da plataforma.

Posteriormente, uma transformação deve ser realizada no PIM para que seja gerado o Modelo Específico de Plataforma (PSM - Platform Specific Model) que contém as especificações já realizadas e os detalhes da plataforma escolhida.

A MDA permite que o desenvolvedor esteja sempre em contato com a documentação e abstraia os detalhes da plataforma, pois eles serão gerados automaticamente na transformação do PIM para PSM, permitindo que o foco esteja

voltado para a solução das regras de negócio. A *Figura 1* exemplifica as transformações realizadas pelo MDA.

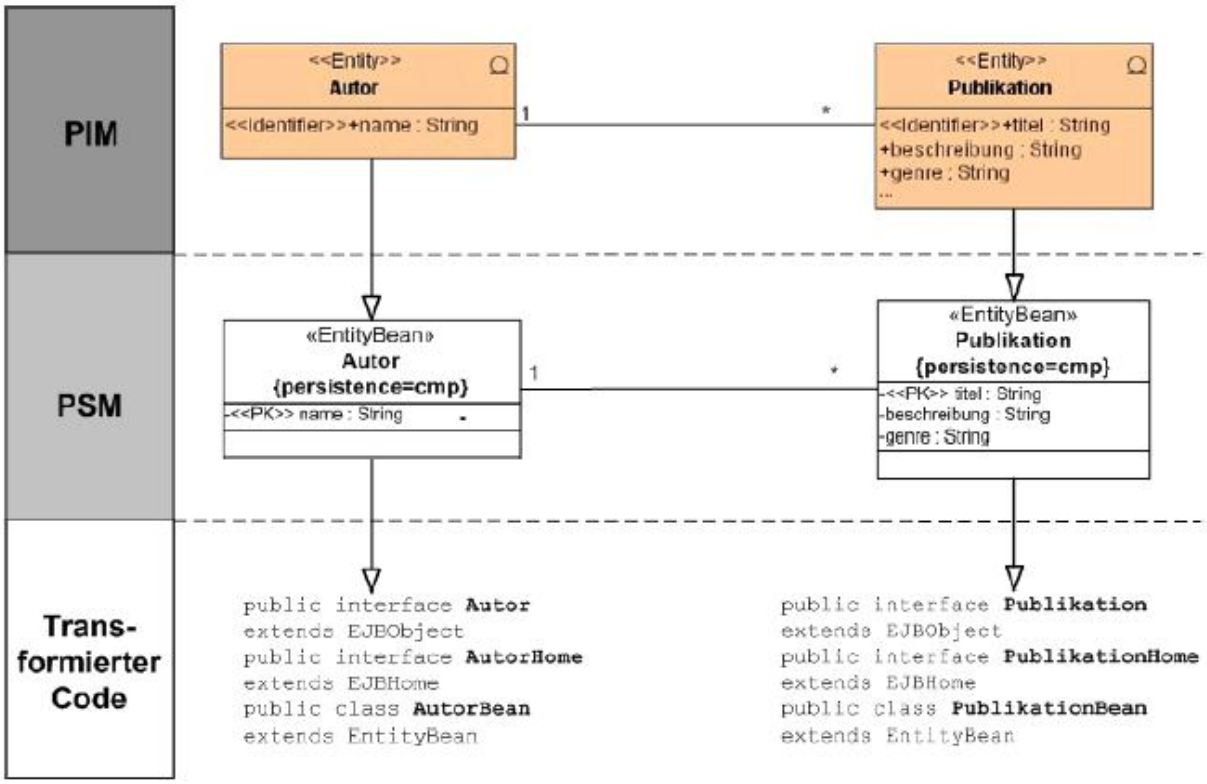

Figura 1: PIM e PSM[71]

Um modelo é uma abstração de algo que existe na realidade. Geralmente para uma aplicação existe mais de um modelo e podem existir diferentes modelos para cada camada da aplicação (por exemplo, domínio, visão e serviço).

A linguagem UML vem sendo largamente utilizada para a representação de modelos na metodologia MDA. Ela permite extensões em sua linguagem para que os desenvolvedores possam criar novos elementos de modelo para usos específicos.

Existem três tipos de extensões UML que facilitam as transformações MDA, são elas: estereótipos, marcas (*tagged-values*) e restrições (*constraints*). Elas tornam a linguagem UML flexível e adaptável.

Estereótipos são usados para alterar o comportamento da geração de código ou facilitar a compreensão do modelo. Existem alguns estereótipos pré-definidos como

entity, *actor* e *extends*. Na *Figura 2* pode-se observar o estereótipo <<*Entity*>>, o qual caracteriza a classe como uma Entidade.

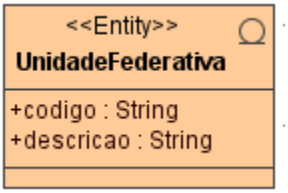

Figura 2: Exemplo de Estereótipo [72]

Marcas, ou *Tagged-values* (Valores Etiquetados), podem introduzir novas propriedades como nomes, atributos e operações nas classes. Com elas, também é possível alterar o valor das propriedades existentes. Na *Figura 3*, pode-se observar um exemplo de *tagged-value*. Nela, o valor etiquetado "*@andromda.presentation.view.table.columns*" fará com que na tela de apresentação ao usuário apareça uma tabela com duas colunas, uma com o nome "código" e outra com o nome "valor".

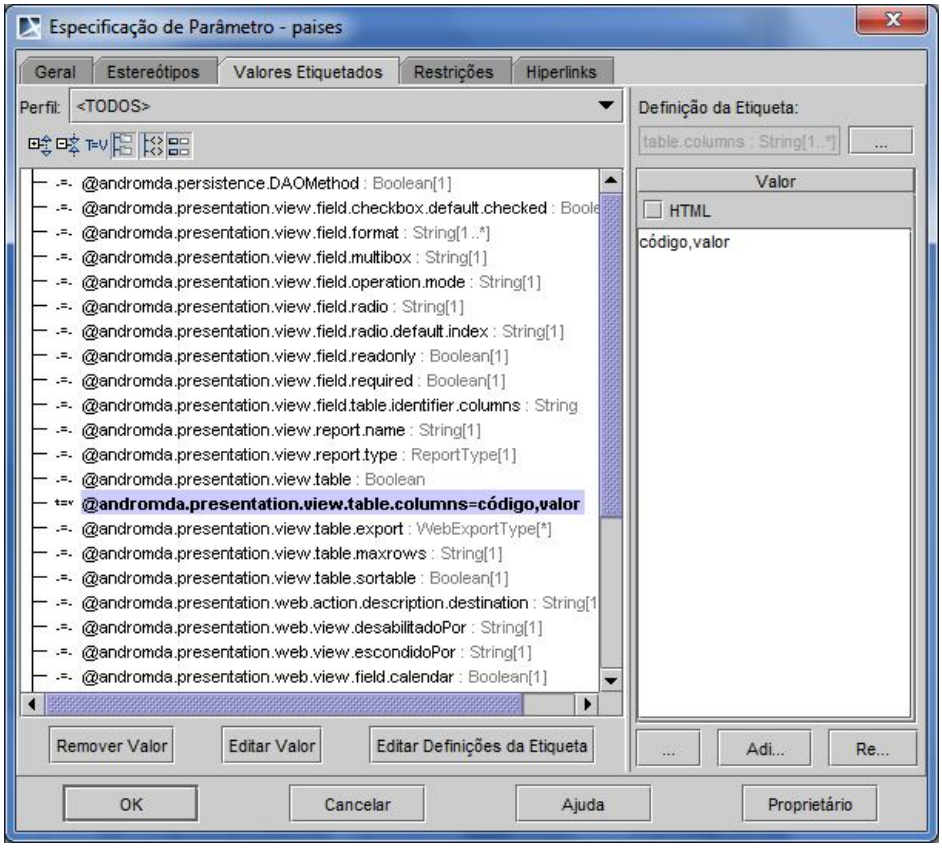

Figura 3: Exemplo de Tagged-values[73]

As restrições podem adicionar ou alterar a semântica. Uma restrição especifica condições que têm de ser validadas para que o modelo seja aceito, ou seja, “bem definido”. Para definir uma restrição, utiliza-se o Object Constraint Language, ou simplesmente, OCL[11].

Metamodelos são modelos que descrevem outros modelos. Eles definem a estrutura, semântica e as restrições para uma família de modelos. O metamodelo UML é expressado usando o MOF, criado pela OMG. Esse metamodelo descreve vários aspectos estruturais dos modelos UML, como a utilização do padrão XMI[12] (XML[13] Metadata Interchange).

O OMG padronizou linguagens para facilitar a integração entre os modelos da MDA. Essas linguagens são classificadas de acordo com seu nível de abstração. Elas são definidas em quatro níveis (M0,M1,M2 e M3). O MOF reside no nível M3 dessa arquitetura[67]. A *Figura 4* demonstra a relação entre os níveis de arquitetura; a seguir serão explicados cada nível:

- **M0** - Está o sistema real onde as instâncias atuais existem. Um exemplo seriam os dados de clientes cadastrados em um sistema.

- **M1** - Contém o modelo do sistema, isto é, a definição de seus conceitos.

- **M2** - Contém o metamodelo que captura a linguagem de modelagem. Na linguagem UML seriam elementos como Class, Attribute e operation.

- **M3** - É o metamodelo que descreve as propriedades que o metamodelo pode exibir.

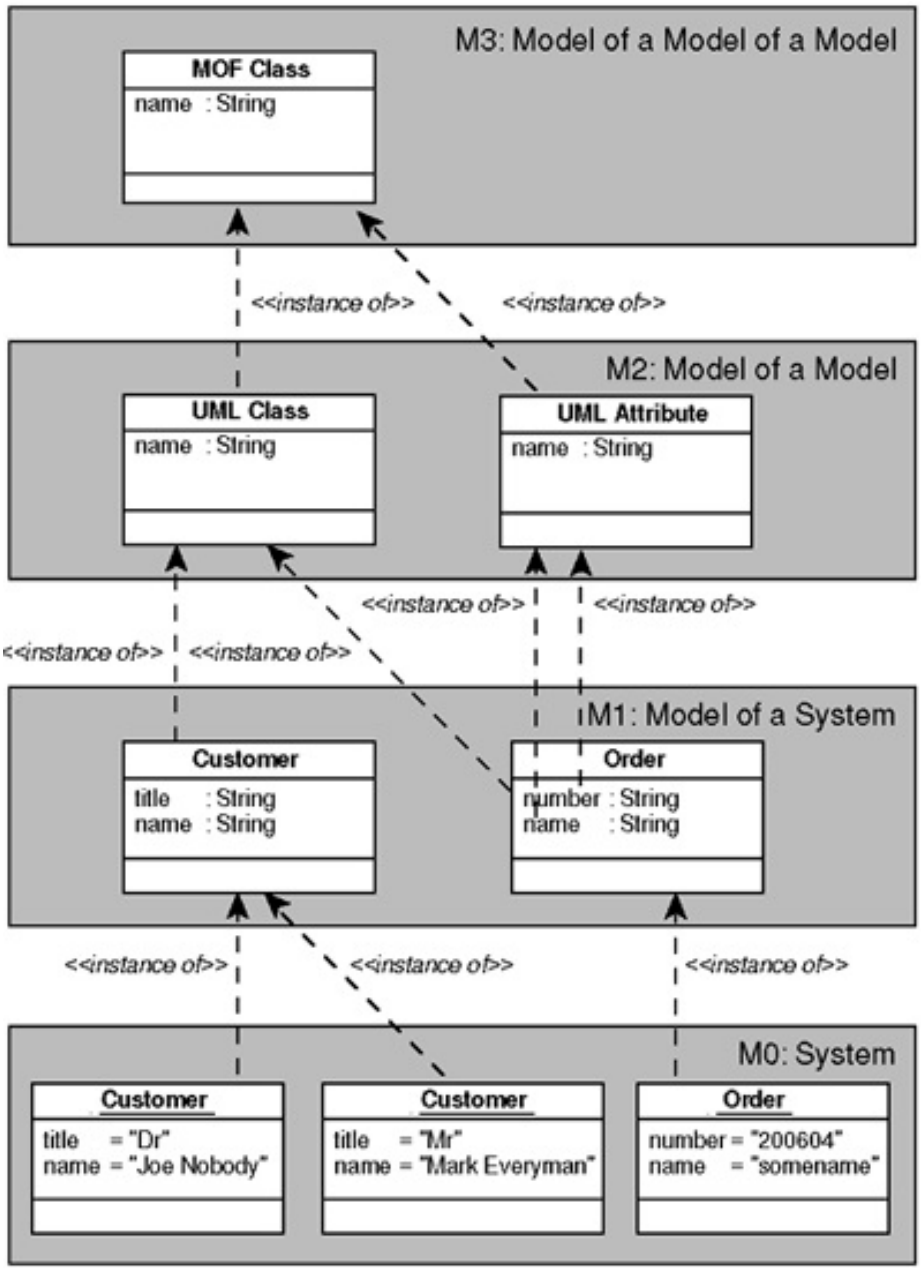

Figura 4: Relação entre os quatro níveis da arquitetura de Metamodelos[74]

No MDA ocorrem essencialmente duas transformações. A primeira é do PIM para PSM e a segunda do PSM para código. Elas são baseadas em mapeamentos que consistem em um conjunto de regras de transformação.

2.1.1) Transformações

Os Metamodelos têm um papel fundamental na construção das transformações. São esses componentes que permitirão as transformações entre os modelos. Assim, pode-se falar que um modelo específico é capturado por um Metamodelo próprio que contém as funções de mapeamento e regras que serão utilizadas. Pode-se dizer que, quando caminhamos por um modelo criado, estamos falando da linguagem de Metamodelos. A *Figura 5* exemplificará uma transformação:

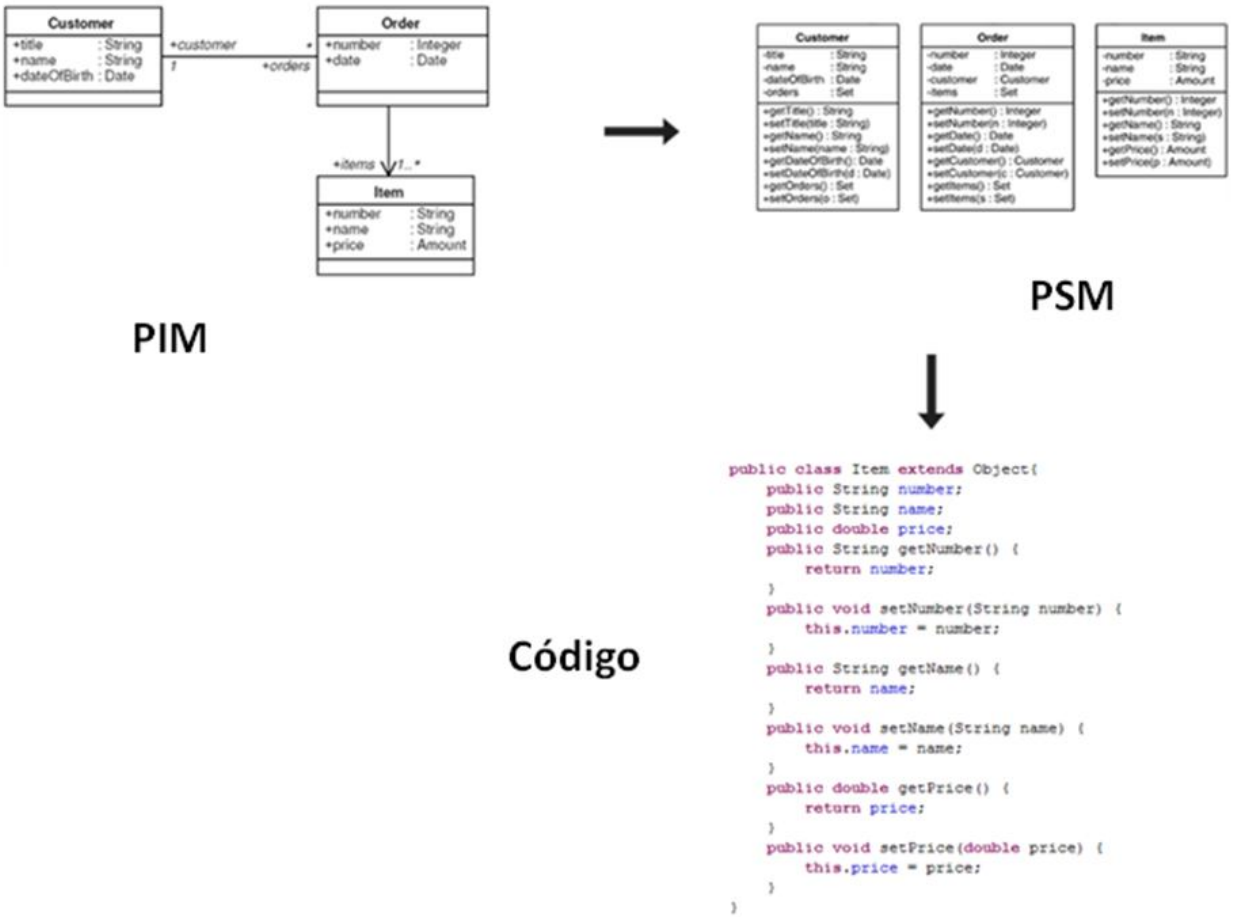

Figura 5: Transformação[75]

Nesta figura pode-se observar que no PIM há um modelo sem especificidade, ou seja, ele não foi desenvolvido para nenhuma plataforma específica. Ao contrário, no PSM, o modelo possui ‘getters’ e ‘setters’, os quais são métodos específicos de uma plataforma. E por último há o código gerado baseado no modelo PSM.

2.2) AndroMDA

O AndroMDA é um framework MDA de código aberto. Ele utiliza modelos UML(geralmente no formato XMI) e cartuchos próprios para produzir código. Esses cartuchos são plug-ins do AndroMDA que têm a habilidade de processar elementos de modelo utilizando Templates[14] que acessam Metafaçades[15].

O framework vem com vários cartuchos implementados e é possível gerar um projeto J2EE sem alterar nada nele. Porém, podemos adicionar Templates aos cartuchos, modificar os pré-existentes, definir novos estereótipos e criar novos cartuchos. Dessa maneira, é possível customizar o AndroMDA e gerar código para qualquer plataforma.

Cada cartucho deve ter obrigatoriamente um descritor (cartridge.xml), o arquivo namespaces.xml e os Templates que determinam como o AndroMDA vai formatar o código gerado. Opcionalmente, também podem existir os arquivos metafacades.xml, profile.xml e arquivos Metafaçades (ver seção 4.2.4).

Os Templates são escritos em Velocity[16], linguagem de script do Apache[17], e então submetidos a uma engine capaz de traduzi-los, gerando um arquivo texto. Os Metafaçades permitem o acesso aos modelos a partir dos Templates. Ao utilizá-los, os Templates tornam-se muito mais simples, pois a inteligência e responsabilidade pela transformação de código ficam centralizadas em objetos Java e não na linguagem script do Template.

O projeto do AndroMDA tem uma estrutura principal que é mostrada na *Figura 6*.

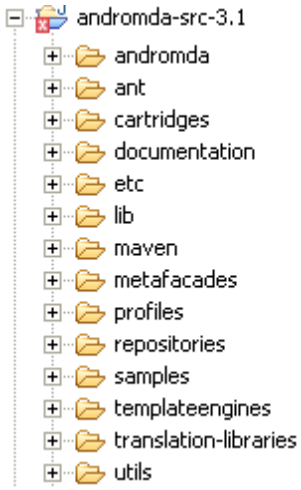

Figura 6: O AndroMDA

Os Elementos mais importantes contidos no projeto são descritos a seguir:

- **andromda**: contém o núcleo do Andromda. É ele o responsável por descobrir namespaces, processar os cartuchos existentes, criar os metafaçades e definir interfaces para o usuário.
- **cartridges**: contém todos os cartuchos declarados.
- **lib**: contém as bibliotecas utilizadas pelo AndroMDA.
- **maven**: é nele que estarão os plug-ins do AndroMDA para o Maven.
- **metafacades**: contém os metafaçades base.
- **profiles**: nele estão os arquivos xml.zip que contém o profile UML definido pelo framework.
- **repositories**: contém as classes que implementam os serviços do repositório.
- **templateengines**: aqui estão as classes que implementam o serviço de processamento do Template.
- **translation-libraries**: contém as classes para traduzir as restrições OCL em código Java.
- **utils**: aqui estão classes utilitárias que serão usadas dentro dos Templates e dos Metafaçades.

Estes arquivos são distribuídos em JAR's (Java Archives)[18] e para estarem disponíveis precisam estar no classpath do Projeto. Isto é feito adicionando uma dependência no arquivo project.xml do Maven. A *Figura 7* exemplifica como disponibilizar um cartucho.

```
<!-- Specify and configure our cartridges -->
<dependency>
    <groupId>andromda</groupId>
    <artifactId>andromda-android-cartridge</artifactId>
    <version>${cartridge.version}</version>
    <type>jar</type>
</dependency>
```

Figura 7: Dependência

Os elementos da dependência serão explicados a seguir:

- **groupId**: define qual a pasta, ou seja, o repositório em que o arquivo será gerado. Este arquivo poderá ser JAR (Java Archive), WAR (Web Application Archive)[19], EAR (Enterprise Archive)[20] etc.
- **artifactId**: nele é definido o nome do arquivo a ser gerado.
- **version**: designa a versão do arquivo a ser gerado.
- **type**: define o tipo do arquivo a ser gerado.

2.2.1) Cartucho

O Cartucho é um dos principais componentes dentro do framework do AndroMDA. Ele é definido quando um namespace tem um componente cartridge.

Um cartucho possui uma variedade de regras baseadas em texto as quais constituem um conjunto de Templates que acessam os Metafaçades e geram uma parte do código da aplicação.

O AndroMDA possui diversos cartuchos, e cada um deles é responsável por transformar o modelo em um PSM específico de uma tecnologia. O analista de sistema deverá escolher quais tecnologias irá utilizar e selecionar os cartuchos mais apropriados. O Hibernate[21], o EJB[22], o Spring[23] e o JSF[24] são exemplos de cartuchos já implementados no AndroMDA.

Existe um cartucho especial chamado “Meta”, o qual é responsável por processar modelos de Metafaçades. Logo, pode-se perceber que o AndroMDA gera seus próprios Metafaçades.

Um cartucho é composto por quatro arquivos básicos de configuração:

- **namespace.xml**: define o caminho para arquivos de configuração e nomes de variáveis utilizadas no cartucho.
- **cartridge.xml**: define principalmente os Templates e os arquivos que serão gerados por eles e a qual elemento de modelo eles estão relacionados.
- **profile.xml**: define os estereótipos e *tagged-values* (marcas) dependentes da plataforma que serão utilizados pelos metafaçades. Este componente não é obrigatório.

- **metafacade.xml**: especifica os metafaçades e seus estereótipos relacionados. Não é obrigatório existir metafaçades em um cartucho, podemos utilizar os metafaçades independentes de plataforma.

A *Figura 8* apresenta a parte principal do *cartrigde.xml*.

```
<template
  path="templates/hibernate/HibernateEntity.vsl"
  outputPattern="$generatedFile"
  outlet="entities"
  overwrite="true"
  outputToSingleFile="false"
  outputOnEmptyElements="false">
  <modelElements variable="entity">
    <modelElement>
      <type name="org.andromda.cartridges.hibernate.metafacades.CoppetecHibernsteEntity"/>
    </modelElement>
  </modelElements>
</template>
<template
  path="templates/hibernate/webServiceData.java.vsl"
  outputPattern="$generatedFile"
  outlet="entities"
  overwrite="true">
  <modelElements variable="webServiceData">
    <modelElement stereotype= "WebServiceData"/>
  </modelElements>
</template>
```

Figura 8: Trecho do cartridge.xml

A tag <*modelElements*> especifica os elementos do Metamodelo, ou seja, do Metafaçade a ser usado que podem ser identificados pelo tipo ou por um estereótipo do profile.

A tag <*template*> possui diversos atributos a serem descritos a seguir:

- **path**: caminho relativo do Template a ser utilizado.
- **outputPattern**: nome do artefato gerado. No caso do valor ser uma variável, o conteúdo será definido dentro do Template. Na *Figura 8* está sendo exemplificado a variável *$generatedFile*.
- **outlet**: propriedade do namespace que especifica o diretório do arquivo gerado. Quando não existir valor default ou no arquivo de configuração, este Template não será utilizado.
- **overwrite**: especifica se o arquivo será sobrescrito em processamentos subsequentes.

- **outputToSingleFile**: especifica se o Template será processado para cada Metafaçade encontrado ou se todos serão agrupados e o Template será processado uma única vez.
- **outputOnEmptyElements**: especifica se o Template será processado, mesmo que não seja encontrado nenhum Metafaçade no Metamodelo.

Os Templates serão processados pela template-engine, um serviço que o AndroMDA possui.

2.2.2) MDArte

O framework MDArte[8] compreende um conjunto de cartuchos AndroMDA com diversas soluções de projeto e arquitetura incorporadas nos procedimentos de transformação de modelos seguindo a abordagem MDA. Ele está baseado no padrão MVC (Model-View-Controller)[25], o qual é amplamente difundido na área de desenvolvimento de sistemas.

As soluções implementadas também se destinam a plataforma Java como no AndroMDA e agregam muitas funcionalidades a ele. Dentre elas, foram definidos inúmeros novos estereótipos e *tagged-values* e alguns cartuchos e arquivos foram alterados.

O MDArte é um software livre e com a ajuda da comunidade pode continuar crescendo, agregando funcionalidades e até novas linguagens ao framework. Com isso, o trabalho de desenvolvimento de software torna-se cada vez mais simples e padronizado.

2.3) MVC – Model-View-Controller

O Model-View-Controller, ou MVC, é um padrão de desenvolvimento que visa separar a lógica de negócio da lógica de apresentação. Isto se justifica pelo surgimento de sistemas bastante complexos, tornando relevante a separação dos dados e a apresentação das aplicações.

O MVC é composto por três camadas, apresentadas da *Figura 9*.

1- Model: Responsável por reunir as informações que mostram o estado de um componente, além de informar para seus observadores sobre as mudanças ocorridas nos dados. É no Model que se gerencia e definem-se as classes de domínio.

2- View: É a parte da aplicação que interage com o usuário. É na View que haverá a integração do "Model" e a especificação da maneira como os dados serão apresentados ao Usuário.

3- Controller: Responsável pelo tratamento de eventos, ou seja, é no Controller que as informações e/ou eventos do usuário, realizados na View, serão capturados e processados para que o "Model" seja modificado. Ele é responsável também por validar e filtrar a entrada de dados realizada pelo usuário.

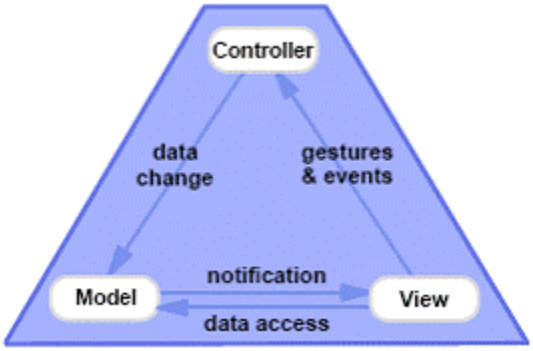

Figura 9: Interação entre os componentes do MVC[76]

2.4) Layers

Para melhor organizar e manter componentes, é crucial que eles sejam separados por algum critério. Isolando-os em grupos é possível diminuir o acoplamento entre os componentes, fazendo com que as mudanças em um grupo não impactem muito em outro grupo.

Entre as diversas formas possíveis de separar os componentes, está a técnica de Camadas. Ao separar componentes em grupos chamados Camadas (Layers[26] em inglês), o projetista agrupa componentes por responsabilidade em comum que possuam.

Uma aplicação Java EE tradicional, por exemplo, usa o modelo de três Camadas: Apresentação, Negócios e Persistência. Na Camada de Apresentação estão todas as classes e demais artefatos (como páginas JSP) relacionados com a interação entre o usuário e a aplicação. A Camada de Negócios contêm a modelagem do

domínio do problema em classes de negócio e a Camada de Persistência contém as classes com conhecimento sobre como persistir objetos no banco de dados (por exemplo, DAO's – *Data Access Objects*).

2.5) Velocity

O Apache Velocity, ou como ficou reconhecido no início, Jakarta Velocity, é um projeto open source criado pela Apache Software Foundation. O Velocity é uma template-engine de modelo baseado em Java[27]. Contém uma linguagem simples, porém quando referenciado a objetos definidos em código Java, torna-se extremamente poderoso.

As capacidades do Velocity vão muito além dos ambientes Web (por exemplo, xdoclet[28], middlegen[29], Intellij[30] etc.), permitindo que desenvolvedores se concentrem mais no código funcional, e por outro lado, faz com que os designers se concentrem mais nas transformações que podem realizar. Ele também é utilizado nos cartuchos AndroMDA para realizar as transformações dos Metamodelos e gerar o código desejado.

Os comandos do Velocity são muito parecidos com uma linguagem procedural, como por exemplo, a linguagem C. Na *Figura 10*, é apresentado um pequeno trecho de código, onde demonstra-se uma transformação utilizando Velocity para gerar código Java. Nesta transformação, será gerado um arquivo Java com o nome da Entidade modelada acrescentada da String "Impl". Será também declarado o pacote da entidade e o nome da Classe, assim como seu Construtor.

```
#set ($generatedFile = "${entity.name}Impl.java")
#if($stringUtils.isNotEmpty($customTypesPackage))
#set ($generatedFile = "${stringUtils.replace($customTypesPackage,'.','/')}/${modelPackageName}/${generatedFile}")
#end
#if ($entity.packageName)
package $entity.packageName;
#end

public class ${entity.name}Impl extends ${entity.name}{

    public ${entity.name}Impl(){

    }

}
```

Figura 10: Exemplo de Velocity

O Velocity é uma ótima linguagem para a programação de transformações que necessitem de uma linguagem robusta. Além disso, sua sintaxe é de fácil compreensão, integrando-se com objetos da linguagem Java e, o mais interessante, ela pode gerar qualquer arquivo texto.

2.6) Padrões de Projeto (Design Patterns)

Um padrão de projeto é uma forma de apresentar uma solução para um problema recorrente[77]. Atualmente o MDArte implementa diversos padrões de projeto, e para aumentar a legibilidade e a qualidade do sistema gerado, a API apresentada neste trabalho foi implementada utilizando os padrões de projeto que foram convenientes.

2.6.1) Fachada (Facade)

O nome deste padrão de projeto vem da analogia com a fachada de um prédio. Classes que adotam o padrão facade são responsáveis por prover uma interface simples de acesso a outros objetos, encapsulando grandes trechos de código necessários para fazê-lo. Este padrão de projeto é utilizado em uma das principais interfaces da camada de visão da API a ser proposta, chamada de PageFacade.

2.6.2) Singleton

Este padrão de projeto é utilizado quando se quer garantir que existirá uma e apenas uma instância de um dado objeto. Ele é utilizado na API para garantir que um usuário tenha apenas uma sessão durante toda a utilização do sistema.

2.6.3) Command

O command é empregado para encapsular toda a informação de um determinado método que será utilizado posteriormente. Este padrão de projeto é especialmente útil, em linguagens onde não é possível passar uma função como parâmetro para um método.

2.6.4) Objeto de Acesso aos Dados (Data Access Object - DAO)

Utiliza-se o DAO para persistência de dados e permite separar regras de negócio das de acesso ao banco de dados. No Estudo de Caso (apresentado no Capítulo 4) é demonstrado como este padrão foi utilizado no Projeto Piloto.

2.6.5) Objeto de Transferência de Dados (Data Transfer Object - DTO)

O Objeto de Transferência de Dados, ou Objeto de Valor (Value Object), como era conhecido antigamente, é um padrão utilizado para transferir dados entre os subsistemas da aplicação. Eles são frequentemente usados em conjunto com os DAOs a fim de recuperar as informações de um Banco de Dados. Ele será melhor demonstrado no Capítulo 4, onde será explicado o Estudo de Caso.

2.7) Ambientes Móveis

Com o avanço da tecnologia, os dispositivos móveis estão se tornando cada vez mais poderosos com relação as suas capacidades de armazenamento, processamento e cada vez mais acessíveis aos consumidores. Com o uso cada vez maior, o número de plataformas e ambientes de desenvolvimento cresceu proporcionalmente.

O processo de desenvolvimento de software para celular tende a ser bastante influenciado pelas tecnologias escolhidas, os dispositivos e seus sistemas operacionais. Atualmente existem muitas plataformas e muitas delas estão vinculadas ao sistema operacional do dispositivo. Isto é, caso seja escolhido um celular com sistema Symbian[31], deve ser usada a linguagem Symbian C++. Isso dificulta a portabilidade das aplicações móveis.

Os dispositivos móveis também oferecem dificuldades devido a sua capacidade de processamento, memória interna e tamanho das telas. Mesmo com o rápido aumento desses atributos eles não podem ser comparados com computadores em questão de memória e principalmente processamento.

2.7.1) Android

O Android é um ambiente operacional que roda sobre o núcleo Linux. Ele é um produto do Open Handset Alliance[32], um grupo de organizações que constrói aplicações visando o crescimento do uso de dispositivos móveis. O grupo, liderado pelo Google[33], inclui muitos colaboradores e seus sistemas são de código aberto.

As aplicações para o Android são escritas na linguagem de programação Java, utilizando bibliotecas desenvolvidas pela Google. Os desenvolvedores podem criar aplicações, utilizando o Android SDK que inicialmente contém a mesma API que é

usada pelas aplicações core da plataforma. A arquitetura das aplicações favorece o reuso de componentes, logo é possível utilizar as capacidades publicadas de qualquer aplicação. O SDK inclui emulador, ferramentas para debugging e análise de performance. Para melhor interação com o programador, a IDE Eclipse[34] pode ser utilizada através do plug-in Android Development Tools (ADT).

No Sistema Operacional Android existe uma Máquina Virtual chamada Dalvik[35] que tem uma arquitetura baseada em registradores, fazendo com que ela seja otimizada para máquinas com pouca memória. Essa Máquina Virtual converte os arquivos Java (.class) para o formato Dalvik Executable[36](.dex). Strings duplicadas e outras constantes são incluídos apenas uma vez para conservar espaço. E para armazenar dados é utilizado o SQLite, uma biblioteca em linguagem C que tem embutido um banco de dados SQL.

O Android disponibiliza um conjunto de serviços, incluindo:

- Um conjunto extensível de Views que podem ser usadas para construir uma aplicação, incluindo listas, grids, text boxes, botões e até um navegador web.
- *Content providers* que fazem com que as aplicações possam acessar dados de outras aplicações ou compartilhar seus próprios dados.
- *Resource Manager*, fornece acesso a recursos como gráficos e arquivos de layout.
- *Notification Manager*, permite as aplicações mostrar alertas customizados na barra de status.
- *Activity Manager* que controla a navegação entre as telas.

A *Figura 11* exemplifica a arquitetura do Sistema Android:

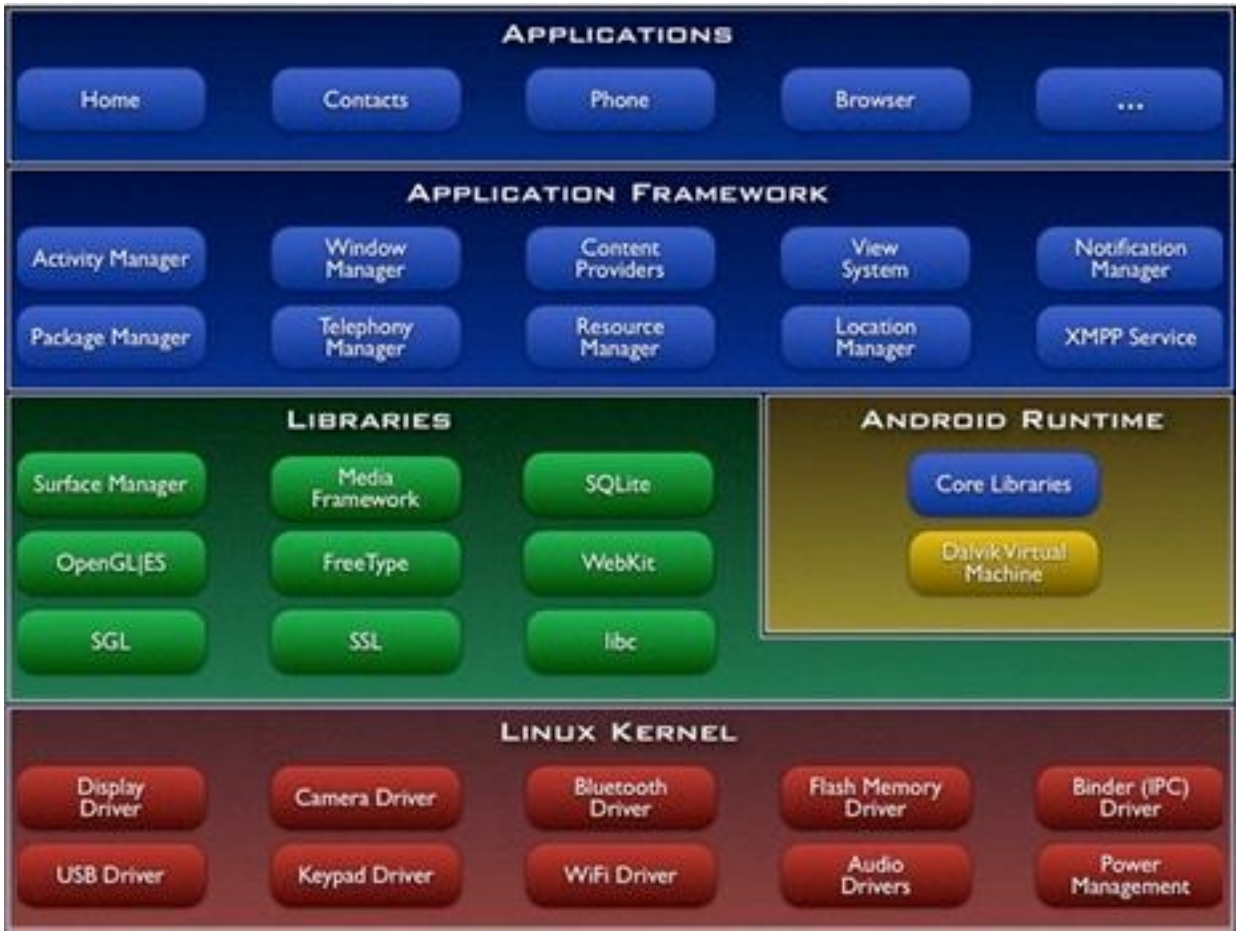

Figura 11: Arquitetura Android

Um arquivo é criado e instalado dentro do dispositivo de plataforma móvel, contendo os conjuntos previamente mencionados, assim como o AndroidManifest.xml (ver seção 4.3.2.6.1), o classes.dex e o resource.arsc, que são os arquivos de configuração, e também as classes de objeto Java convertidos. Este arquivo tem extensão apk (Android Package) e é uma variante do formato JAR (Java Archive).

Um fator importante que posiciona o Android distante da maioria dos sistemas operacionais móveis é que ele está baseado em uma plataforma de código aberto. Assim existe a possibilidade para que o sistema operacional possa funcionar em telefones feitos por diferentes fabricantes.

Recentemente, foi noticiado que o Android superou o Symbian, sistema operacional desenvolvido pela empresa finlandesa Nokia[37], em celulares, este último considerado o sistema operacional móvel mais popular, até então. A pesquisa foi

divulgada pela empresa de pesquisas Canalys[38], e mostra que 32,9 milhões de aparelhos celulares com Android instalado foram vendidos, contra 31 milhões de dispositivos da Nokia. A estratégia da Google é bastante simples: ao invés de investir em um smartphone específico, a empresa procura fazer acordos com diversos fabricantes, como a Motorola[39], Samsung[40] e HTC[41], que utilizam seu sistema operacional de forma gratuita. Mais detalhes podem ser vistos em [42].

3) Solução Proposta

Com o objetivo definido de reduzir o acoplamento de sistemas a frameworks específicos de plataforma, foi tomada a decisão de definir uma API que permitisse a maior independência possível dos mesmos. Esta API será consumida pelo sistema. Além disso, ela terá diversas implementações, utilizando os frameworks que forem julgados necessários. Com esse tipo de abordagem, o sistema deve consumir apenas os serviços que estiverem descritos nela, sendo assim, ainda que a implementação da API se altere, ou seja, sejam alterados os frameworks específicos de plataforma, a implementação do sistema não precisará ser alterada. Neste capítulo, abordaremos todos os aspectos que dizem respeito à forma como foi idealizada e implementada a API, além disso, mostraremos como ela pode ser expandida trazendo novas funcionalidades e comportamentos aos sistemas, sem que haja impacto nos que já a utilizam.

3.1) O Desenvolvimento da API

Para o desenvolvimento de um sistema de grande porte é necessário escolher a arquitetura do sistema. Quando falamos em sistemas produzidos utilizando o MDArte em seu estado atual, estamos falando de uma arquitetura híbrida entre os padrões MVC (ver seção 2.3) e *Layers* (ver seção 2.4) .

Desta forma, por estar sendo utilizado o framework MDArte, foi decidido que seria mantido este padrão arquitetural, onde as classes do sistema estão divididas de acordo com o padrão MVC e este dividido em camadas, onde a camada superior deve utilizar os serviços da camada inferior. Além de manter o padrão atual utilizado no MDArte, a adoção desta arquitetura facilita a manutenção e leitura do código criado. Dada esta organização, a API foi dividida, fundamentalmente em duas áreas: View e Controller.

3.1.1) View

No desenvolvimento da camada de visão, se encontram grandes desafios quando o objetivo é diminuir o acoplamento de um sistema a frameworks específicos de plataforma ou a determinadas plataformas. Cada plataforma tem uma forma específica de descrever as telas e interações com o usuário e para tal podem ser utilizadas diferentes linguagens como XML, HTML, Java entre outras. Neste ponto ficou decidido que a descrição das telas se manteria específica a plataforma onde o sistema está sendo inicialmente desenvolvido, quando houver a necessidade de portar um sistema, as telas devem ser refeitas para a plataforma específica. Porém, a interação da tela com as classes será feita através dos serviços disponibilizados pela API. Para que tal abordagem seja possível, é de suma importância que os componentes de tela tenham identificadores, o que torna possível ler e apresentar dados através deles.

Além disso, é importante ressaltar que a navegação do sistema deve ser feita através de páginas, independente da forma como estas são escritas. É imprescindível que o usuário dê uma entrada em uma página, um processamento seja feito e ele seja redirecionado a outra página. Esta escolha foi feita, por sua facilidade de ser expressa em diagramas UML e para facilitar a implementação.

A API conta com cinco interfaces na camada de visão, com suas descrições abaixo.

3.1.1.1) PageFacade

Esta é a principal interface da camada de visão da API. É ela a responsável por fornecer serviços de suma importância no gerenciamento dos componentes de tela, recuperando-os ou incluindo novos. Fornece os seguintes serviços:

- *public InputText getInputText(String id):* Método que recebe como parâmetro o identificador de uma caixa de entrada de texto, e retorna o elemento correspondente a ele. A partir deste objeto é possível recuperar o texto dado como entrada por um usuário.
- *public Button getButton(String id):* Método que recebe como parâmetro o identificador de um botão , e retorna o elemento correspondente a ele.

- *public void changePage(String id):* Recebe como parâmetro o identificador de uma tela, este método remove a tela atual e exibe a nova tela cujo identificador foi recebido como parâmetro
- *public void showErrorMessage(String message):* Este método deve ser invocado quando algum erro ocorrer, ele serve para orientar o usuário de que algo inesperado aconteceu, recebe como parâmetro a mensagem para o usuário. É sugerido que na implementação o erro seja exibido em forma de pop-up, pela familiaridade que os usuários têm com este tipo de elemento.
- *public void createTable(String tableId , Collection elements,List<Button[]> Buttons, String... fields):* Este método é responsável por criar uma tabela na página atual; ele recebe como parâmetro o identificador da tabela que será criada, um conjunto de *value objects* com os campos que se deseja exibir na tabela e uma lista de botões caso seja necessário colocar algum botão em cada linha da tabela.
- *public Button createButton(String label):* Método que cria um botão na página, recebe como parâmetro o nome do botão, e retorna o botão criado.
- *public void setLabel(String id, String valor):* Altera o valor de algum rótulo (*label*) na página.

3.1.1.2) Session

Esta interface deve ser utilizada para guardar objetos que sejam criados dentro de um caso de uso e que somente serão utilizados em outro. É de vital importância que uma vez que um objeto seja colocado, ele seja retirado logo depois de ser usado, evitando desperdício de memória e erros indesejáveis. Sua implementação deve conter um mapa que receberá sempre uma chave e um valor para guardar. Ela fornece os seguintes serviços:

- *public void put(Object key ,Object value):* Inclui no mapa um objeto chave, e um objeto que será o valor.
- *public Object get(Object key):* Dada uma chave, retorna o valor correspondente no mapa.

- *public void remove(Object key):* Dada uma chave, remove o objeto correspondente do mapa.

3.1.1.3) InputText

Esta interface representa uma caixa de entrada de texto que existe em uma dada página. É possível ler a entrada do usuário ou alterá-la. Fornece os seguintes serviços:

- *public String getValue():* Retorna o texto dado como entrada pelo usuário.
- *public void setValue(String value):* Altera o texto escrito nesta caixa de texto, o novo texto é recebido como parâmetro por este método.

3.1.1.4) Button

Esta interface representa um botão que existe em uma dada página. Fornece os seguintes serviços:

- *public String getText():* Retorna o nome deste botão.
- *public ActionCommander getAction():* Retorna qual a ação será disparada quando o botão for acionado pelo usuário.
- *public void setAction(final ActionCommander action):* Altera a ação que será disparada quando o botão for acionado pelo usuário. A nova ação é recebida como parâmetro por este método.

3.1.1.5) ActionCommander

Esta interface utiliza o padrão de projeto *Commander*, e é responsável por encapsular o método que será executado quando um dado botão for disparado. Seu único serviço é:

- *public void action():* Deve ser implementado pelo usuário e conter os comandos que se deseja executar quando um dado botão for acionado.

3.1.2) Controller

Na camada de controle, ou simplesmente Controller, o principal desafio foi em como desenvolver as interfaces as quais farão a conexão com o Banco de Dados, assim como a manipulação dos mesmos, de modo que estas fiquem desacopladas do framework de plataforma.

O Controller tem o objetivo de interligar as camadas de dados e controle. Ele é constituído de quatro interfaces e mais quatro classes que as implementam. As interfaces e suas respectivas classes são:

3.1.2.1) ConnectionDatabase

Esta interface é responsável por realizar a conexão do Projeto Implementado ao Banco de Dados correspondente. O Contexto será passado para essa interface e sua conexão ao Banco será realizada.

3.1.2.2) ContextDatabase

É aqui que está o contexto do Banco de Dados. Esta interface possui dois métodos que são os insert's SQL responsáveis por criar inicialmente o banco de dados.

- *public void insertSQL(ConnectionDataBase db, String scriptCreate):* Chama o método de atualização do SGBD[46] passando o banco criado.
- *public void insertSQL(ConnectionDataBase db, String[] scriptCreate):* Chama o método de atualização do SGBD[46] passando o banco criado utilizando um array de scripts.

3.1.2.3) PersistenseValues

Responsável por executar os métodos de *insert*, *delete*, *update* e controlar as transações.

- *public Long insert(String table,String nullColumnHack,PersistenceValues value):* É chamado sempre que é necessário fazer um inserção no banco de dados. No caso do Android, ele cria um *ContentValue* (classe utilizada pelo Android para armazenar um conjunto de dados a serem processados) com os dados que

serão inseridos e chama o método insert do *SQLiteDatabase* do Android que insere os dados na tabela indicada.

- *public int update(String table, PersistenceValues value,String whereClause, String[] whereArgs):* Chamado sempre que é necessário atualizar dados no banco. No Android, um *ContentValue*[45] será criado com os dados a serem atualizados e chama o método *update* do SQLiteDatabase.
- *public int delete(String table,String whereClause,String[] whereArgs):* Remove linhas de uma tabela utilizando o método delete do Banco de Dados utilizado.
- *public ResultDataBase query(String table,String[] columns,String selection,String[] selectionArgs,String groupBy, String having,String orderBy):* método responsável por executar uma determinada query no Banco de Dados. No Android, o método query do *SQLiteDatabase* será invocado e um Cursor será retornado como resultado. A interface *Cursor* torna possível ler e escrever nos resultados retornados pela query.
- *public void beginTransaction():* Determina o início de uma transação com o banco para que não ocorram duas alterações simultâneas. Com isso a consistência do banco é garantida.
- *public void setTransactionSuccessful():* Indica que uma transação terminou com sucesso.
- *public void endTransaction():* Termina a transação.

3.1.2.4) ResultDatabase

Esta classe é responsável por devolver o resultado da consulta realizada pelo usuário no Banco de Dados.

- *public HashMap<Object,HashMap<Object,Object>> getResultDataBase():* Retorna um *HashMap* com os resultados no Banco pesquisados. No caso do Android, utiliza-se o Cursor para preencher o *HashMap*.

3.2) Expansão da API

Estes são os serviços fornecidos atualmente pela API. Sugere-se que novos serviços sejam criados, para que outros elementos de tela, como *checkbox* ou *combobox*, por exemplo, possam ser lidos ou criados. Estes serviços não constam nesta versão da API, pois podem ser criados muito facilmente de maneira análoga aos casos do botão – Button (ver seção 3.1.1.4) e da caixa de entrada de texto – InputText (ver seção 3.1.1.3). Já no Controller da API pode-se simplesmente alterar a implementação das classes de conexão e contexto do Banco de Dados para que estas fiquem de acordo com o SGBD[46] desejado.

4) Estudo de caso

Neste capítulo será explicado o Estudo de Caso desenvolvido para demonstrar na prática como a API poderá ser utilizada em um sistema real. Para isso, um Projeto Piloto foi criado, a fim de exemplificar o uso da Interface de Programação de Aplicações, de forma que o acoplamento ao framework específico de plataforma seja mínimo no Projeto.

Será explicado também o porquê da escolha do Sistema Operacional Android junto com os prós e contras de outras plataformas também difundidas, como JavaFX[47] e Java Micro Edition (Java ME)[48].

A escolha de um Ambiente de Desenvolvimento Integrado, ou do inglês, IDE (Integrated Development Environment)[49] também foi de suma importância para que o Projeto fosse criado de forma simples e objetiva. O Eclipse[50] foi a IDE escolhida e para auxiliar o desenvolvimento foram utilizados os plug-ins do Android: ADT[51] e AVD[52].

4.1) Requisitos adotados

O Projeto tinha como ideia inicial empregar o framework MDArte (ver seção 2.2.2) para gerar sistemas para dispositivos móveis utilizando JavaFX, uma nova tecnologia que visa desenvolver Sistemas Desktop, Móveis, assim como Web, de forma a garantir desempenho e facilidade em seu uso no dispositivos que suportem esta tecnologia.

4.1.1) JavaFX

O JavaFX é uma plataforma, compatível com Java SE[53] e Java ME, as quais são plataformas bastante utilizadas nos dias de hoje. Esta nova tecnologia foi desenvolvida para criar e disponibilizar a RIA – Rich Internet Applications[54], inicialmente desenvolvida pela Sun[55]. A RIA é uma Aplicação Web que tem muitas das características de uma Aplicação Desktop e funciona normalmente através de um plug-in específico do Browser o qual tem uma Máquina Virtual independente.

Porém, ao pesquisarmos como seria feita a persistência de dados em um dispositivo móvel, que era o foco, não conseguimos uma solução prática, pois o JavaFX Mobile não suportava os plug-ins dos bancos pesquisados, como Oracle[56], MySQL[57] e HSQLDB[58]. Além do mais a Oracle, empresa responsável pela plataforma JavaFX, não tinha lançado novas versões para o Java FX Mobile. Assim, resolvemos mudar para outra tecnologia que está sendo amplamente utilizada: **o Android** (ver seção 2.7.1).

4.1.2) Eclipse

Para garantir ao desenvolvedor um ambiente simples e com muitos recursos, escolhemos um Ambiente de Desenvolvimento Integrado bastante utilizado mundialmente, o Eclipse. O fato dele ser usado por muitos desenvolvedores faz com que várias dúvidas e soluções sejam compartilhados por Fóruns[59] na Internet.

O Eclipse é uma IDE desenvolvida em Java para a criação de programas de computador. Ele começou a ser projetado inicialmente pela IBM, e depois tornou-se de domínio público mantendo-se assim, até hoje. A forte orientação ao desenvolvimento baseado em plug-ins e o amplo suporte ao usuário desenvolvedor faz com que esta IDE seja uma das mais utilizadas no mundo. Um plug-in empregado na criação do nosso Projeto Piloto é o ADT.

4.1.2.1) ADT

O Android Development Tools – ADT é um plug-in para o Eclipse que tem por objetivo dar o máximo de suporte ao desenvolvedor, criando Aplicações Android. Com ele é possível criar rapidamente um Sistema Android com sua Interface ao Usuário e também com componentes baseados no Framework do Android.

4.1.2.2) AVD

Para construir aplicações para o Android, será necessário ter instalado na máquina de desenvolvimento o Android SDK, o qual inclui um emulador, o que nos dá a possibilidade de testar o aplicativo criado sem a necessidade de um celular com o Sistema Operacional Android.

O Android Virtual Devices – AVD é um dispositivo virtual do Android SO, simulando um aparelho celular. Com ele é possível escolher um celular para ser emulado, configurando-o da melhor forma, como por exemplo, definindo se o dispositivo possui câmera ou não, qual a versão da plataforma Android será emulada, entre outras funcionalidades.

4.2) O Cartucho Android

Como os cartuchos implementados no MDArte não suportariam a implementação da ideia aqui escolhida, tivemos que modelar toda a estrutura de nosso cartucho, através da construção de Metafaçades e dos Templates para a transformação do modelo em código.

Primeiro foi desenvolvido um Projeto utilizando o plug-in ADT (ver seção 4.1.2.1) e também criando a estrutura a ser adotada (ver seção 4.3.3). Após isto, o Cartucho Android foi desenvolvido tendo como base este Projeto.

4.2.1) A Estrutura do Cartucho Android

O Projeto do Cartucho tem uma estrutura que é apresentada na *Figura 12*. Em seguida serão descritos seus principais elementos.

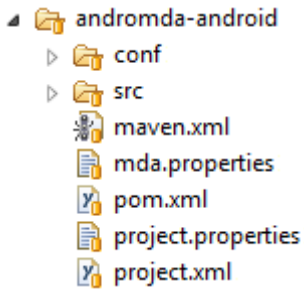

Figura 12: Cartucho Android

- **maven.xml**: contém os goals (objetivos) deste cartucho, os goals básicos são o install (compilação do cartucho) e o clean (limpeza do cartucho).
- **mda.properties**: contém as propriedades a serem utilizadas pela geração do modelo. Dois labels default estão declarados:

- **maven.build.properties**: indica o diretório onde ficarão os arquivos source do maven.
- **java.src.dir**: indica o diretório onde ficarão os arquivos source do Java.

- **pom.xml**: principal arquivo do cartucho. É no POM (Project Object Model) que estarão todas as configurações que aquele cartucho faz. A *Figura 13* demonstra um exemplo.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.andromda.cartridges</groupId>
        <artifactId>andromda-cartridges</artifactId>
        <version>3.1-SNAPSHOT</version>
    </parent>
    <artifactId>andromda-android-cartridge</artifactId>
    <packaging>jar</packaging>
    <name>AndroMDA Android Cartridge</name>
    <description>Produces Entity from a model.</description>
    <dependencies>
        <dependency>
            <groupId>org.andromda.cartridges</groupId>
            <artifactId>andromda-meta-cartridge</artifactId>
            <version>${version}</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
    <build>
        <filters>
            <filter>${basedir}/mda.properties</filter>
        </filters>
        <plugins>
            <plugin>
                <groupId>org.andromda.maven.plugins</groupId>
                <artifactId>andromda-maven-plugin</artifactId>
                <dependencies>
                    <dependency>
```

Figura 13: Trecho do pom.xml

Os principais elementos do pom.xml são explicados a seguir:

- **parent**: indica que o cartucho atual herda artefatos do cartucho pai. Dentro deste artefato existem três subartefatos:

        - **groupId**: pasta onde está o cartucho pai;
        - **artifactId**: nome do cartucho pai;
        - **version**: versão do cartucho pai.
    - **artifactId**: indica o nome do arquivo do cartucho que será gerado.
    - **packaging**: indica a extensão do arquivo do cartucho cujo nome é declarado no artefato anterior.
    - **name**: indica o nome do cartucho.
    - **description**: utilizado para dar uma breve descrição do que o cartucho faz.
    - **dependencies**: indica as dependências que este cartucho possui, ou seja, o cartucho atual possui propriedades que não estão contidas nele e sim em outros cartuchos.
    - **build**: indica configurações para a criação do cartucho.
- **project.properties**: indica labels que serão utilizados na geração dos arquivos. Podem conter labels dos Metafaçades ou mesmo dos testes que serão feitos na compilação.
- **project.xml**: mostra as informações do cartucho, tais como desenvolvedor, colaborador e também define as dependências de bibliotecas que o cartucho deverá utilizar.
- **conf**: define o caminho do arquivo andromda.xml para executar os testes desse cartucho. Nesse arquivo deverá constar também onde estará o xml.zip (ou xml) do modelo do cartucho.
- **src**: a pasta source é onde ficarão os Templates e os Metafaçades além do META-INF, que conterá os arquivos de configuração do cartucho.

4.2.2) O Modelo do Cartucho

Para que o cartucho possa ser compilado, antes é necessário modelar toda sua estrutura. Para isso dividimos o modelo em três diagramas de classe: MetaView, MetaController e MetaService.

O Modelo utiliza os Metafaçades existentes nos cartuchos do MDArte (ver seção 2.2.2), fazendo com que estes, uma vez modelados, possam ser utilizados pelo cartucho por nós desenvolvido.

Para seguir um padrão, foi adotado o critério de, na hora da modelagem, iniciar o nome dos Metafaçades por Android, indicando que o objetivo destes Metafaçades é atender o cartucho Android.

A *Figura 14* mostra a localização do modelo AndroidMetafacadeModel.xml.zip na estrutura do Cartucho Android:

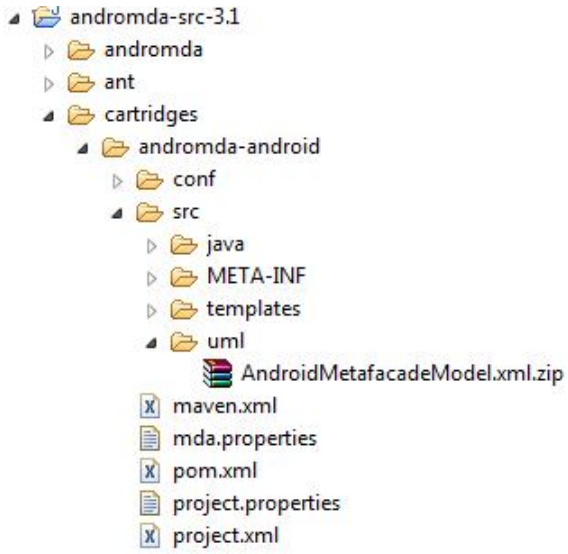

Figura 14: Local do modelo do Cartucho

4.2.2.1) MetaView

O modelo da Camada de Visão, denominado *MetaView* é mostrado na *Figura 15*.

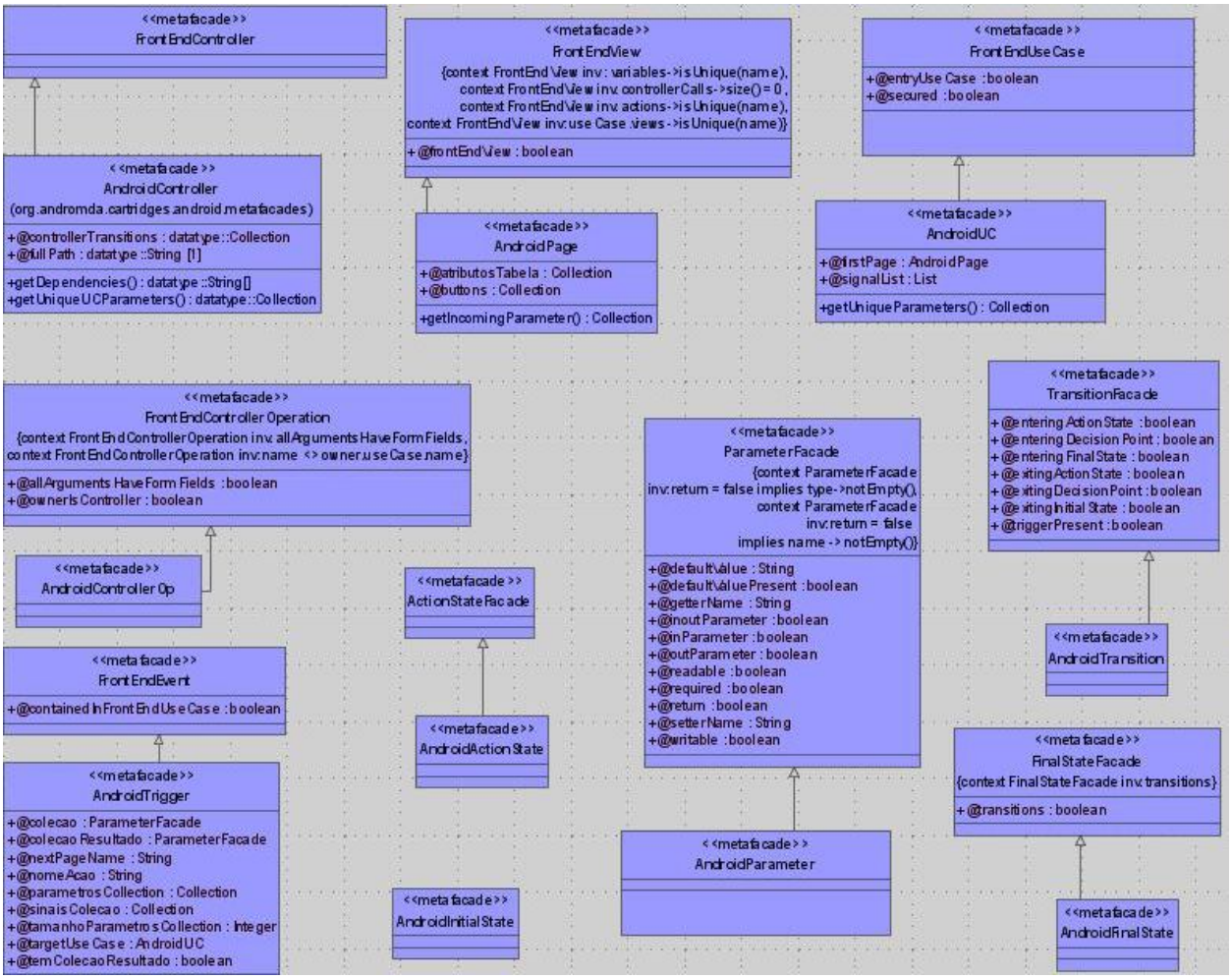

Figura 15: Parte do Diagrama de Classe da Camada de Visão

Nele foi criada uma série de Metafaçades listados abaixo. Estes representam os diferentes elementos que podem ser utilizados em um diagrama de classes UML. Alguns deles não são utilizados atualmente no cartucho desenvolvido, porém já se incluem no modelo com a finalidade de facilitar evoluções futuras.

- **AndroidUC**: Este Metafaçade representa os casos de uso do sistema, ele é um filho do Metafaçade *FrontEndUseCase*, que é um Metafaçade padrão do AndroMDA para representar casos de uso. Foram acrescidos nele, os seguintes atributos e métodos que se aplicam apenas ao caso de uso em questão:
    - *firstPage (tipo AndroidPage):* Representa a primeira página a ser exibida para o usuário,ou seja, a primeira ação com o estereótipo *FrontEndView*.
    - *signalList (tipo List):* É uma lista contendo todos os eventos do tipo sinal

    - *public Collection getUniqueParameters():* Retorna uma coleção sem repetições de todos os parâmetros utilizados.
- **AndroidController**: Este Metafaçade representa as classes de controle. Ele é um filho do Metafaçade *FrontEndController*, que é um Metafaçade padrão do AndroMDA para representar classes de controle. Foram acrescidos nele os seguintes atributos e métodos:
    - *fullPath (tipo String):* Indica o caminho onde esta classe de controle será gerada
    - *controllerTransitions (tipo Collection):* Coleção de classes de controle de casos de uso para os quais este caso de uso pode levar.
    - *public String[] getDependencies():* Retorna um array com todas as dependências desta classe de controle, é importante para fazer os *import's* das classes necessárias
    - *public Collection getUniqueUCParameters():* Similar ao método *getUniqueParameters* da classe AndroidUC.
- **AndroidControllerOP**: Este Metafaçade representa os métodos das classes de controle. É filho do Metafaçade *FrontEndControllerOperation*, que é o Metafaçade padrão do AndroMDA para representar métodos de classes de controle. Não possui nenhum atributo ou método específico.
- **AndroidInitialState**: Este Metafaçade representa o estado inicial dos casos de uso do sistema. É filho do Metafaçade *ModelElementFacade*, já que o AndroMDA não possui nenhum Metafaçade padrão para estados iniciais. Não possui nenhum atributo ou método específico
- **AndroidActionState**: Este Metafaçade representa os estados de ação dos casos de uso do sistema. É filho do Metafaçade *ActionStateFacade*, que é um Metafaçade padrão do AndroMDA para representar estados de ação. Não possui nenhum atributo ou método específico
- **AndroidPage**: Este Metafaçade representa as telas dos casos de uso do sistema, ou seja, ações com o estereótipo *FrontEndView*. É filho do Metafaçade *FrontEndView*, que é um Metafaçade padrão do AndroMDA para representar telas. Foram acrescidos nele os seguintes atributos e métodos:

- *atributosTabela (tipo Collection):* Representa os campos dos *Value Objects* (ver seção 2.6.5) recebidos como parâmetros que se deseja exibir em formato de tabela.
- *buttons (tipo Collection):* Representa os botões que esta tela deve conter.
- *public Collection getIncomingParameters():* Método que retorna uma coleção de parâmetros dados como entrada para esta tela

- **AndroidTransitions**: Este Metafaçade representa as transições entre estados dos casos de uso do sistema. É filho do Metafaçade *TransitionFacade*, que é um Metafaçade padrão do AndroMDA para representar transições de estados. Não possui nenhum atributo ou método específico
- **AndroidParameter**: Este Metafaçade representa os parâmetros utilizados nos casos de uso do sistema. É filho do Metafaçade *ParameterFacade*, que é um Metafaçade padrão do AndroMDA para representar parâmetros. Não possui nenhum atributo ou método específico.
- **AndroidFinalState**: Este Metafaçade representa os estados finais dos casos de uso do sistema. É filho do Metafaçade *FinalStateFacade*, que é um Metafaçade padrão do AndroMDA para representar estados finais. Não possui nenhum atributo ou método específico.
- **AndroidTrigger**: Este Metafaçade representa as transições do tipo “gatilho” dos casos de uso do sistema. É filho do Metafaçade *FrontEndEvent*, que é um Metafaçade padrão do AndroMDA para representar eventos. Foram acrescidos nele os seguintes atributos e métodos:
  - *nomeAcao (tipo String):* Representa o nome dado a esta ação
  - *nextPageName (tipo String):* Representa o nome da próxima página a ser exibida, após esta transição ser chamada
  - *temColecaoResultado (tipo boolean):* Indica se esta ação possui alguma coleção como resultado da ação alvo desta transição.
  - *colecaoResultado (tipo ParameterFacade):* Caso haja uma coleção como resultado da ação alvo desta transição, ela é armazenada neste atributo
  - *sinaisColecao (tipo Collection):* Caso haja uma coleção como resultado desta ação, podem ser modeladas transições que, na tela, se tornarão

botões existentes em cada linha. Estas transições são armazenadas nesta coleção.

- *targetUseCase (tipo AndroidUC):* Caso haja transição entre casos de uso, indica o caso de uso alvo desta transição.
- *parametrosCollection (tipo Collection):* Representa os campos dos *Value Objects* (ver seção 2.6.5) que se deseja apresentar em formato de tabela
- *tamanhoParametrosCollection (tipo Integer):* Armazena o tamanho da coleção *parametrosCollection*.
- *colecao (tipo ParameterFacade):* Similar ao *colecaoResultado*, porém armazena a coleção que é parâmetro desta transição.
- *atributosExibidos (tipo Collection)*: Utilizado para retornar os campos que devem ser exibidos em uma tabela.

- **AndroidControllerPackage**: Este Metafaçade representa o pacote das classes de controle de casos se uso, ele é um filho do Metafaçade *ClassifierFacade*, já que o AndroMDA não possui nenhum Metafaçade padrão para pacotes de classes de controle. Não possui nenhum atributo ou método específico.

4.2.2.2) MetaController

O modelo da Camada de Dados, denominado *MetaController*, é mostrado na *Figura 16*.

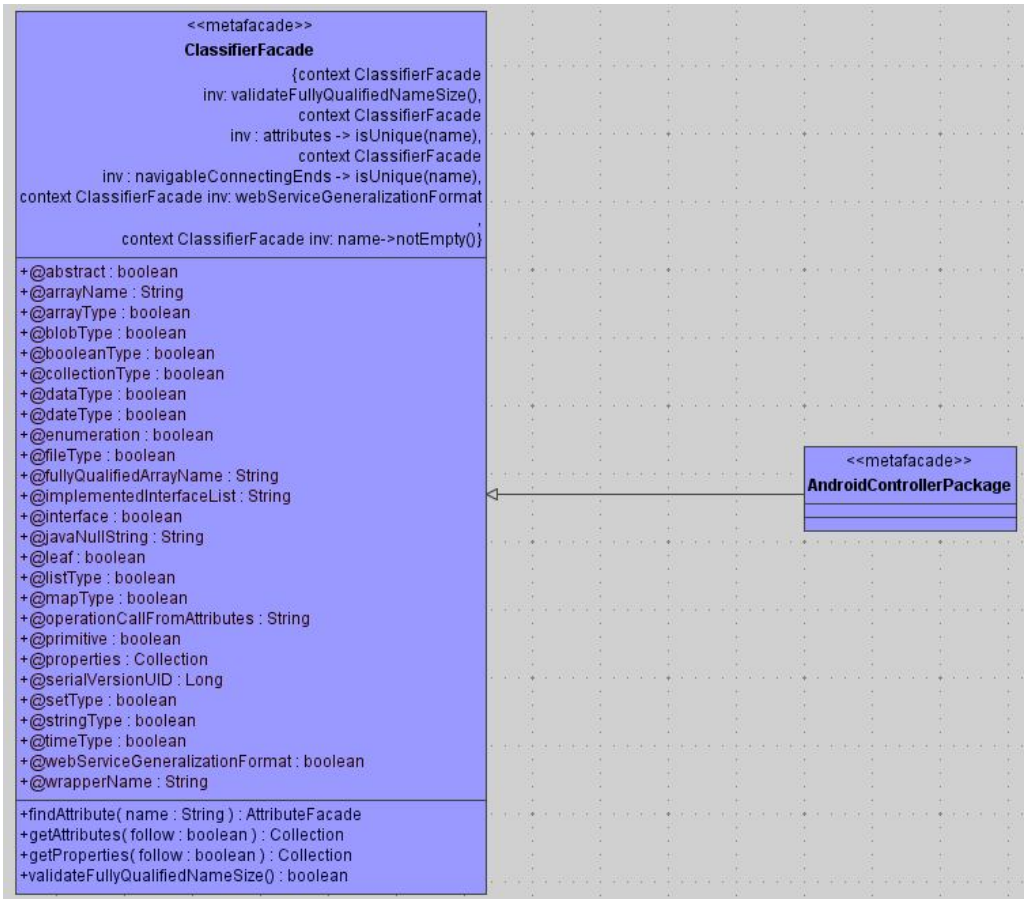

Figura 16: Diagrama de Classe da Camada de Dados

Nele foi criado o Metafaçade *AndroidControllerPackage*, o qual é uma especificação de outro Metafaçade, o *ClassifierFacade*. Este último, por sua vez, é responsável por fornecer métodos básicos que auxiliam na transformação dos modelos UML, como saber o nome da Entidade, o tipo de um atributo de uma Classe, o pacote de uma Classe entre outros métodos.

O *AndroidControllerPackage* é utilizado para transformar as Entidades modeladas na pasta “cd” em Classes Java.

4.2.2.3) MetaService

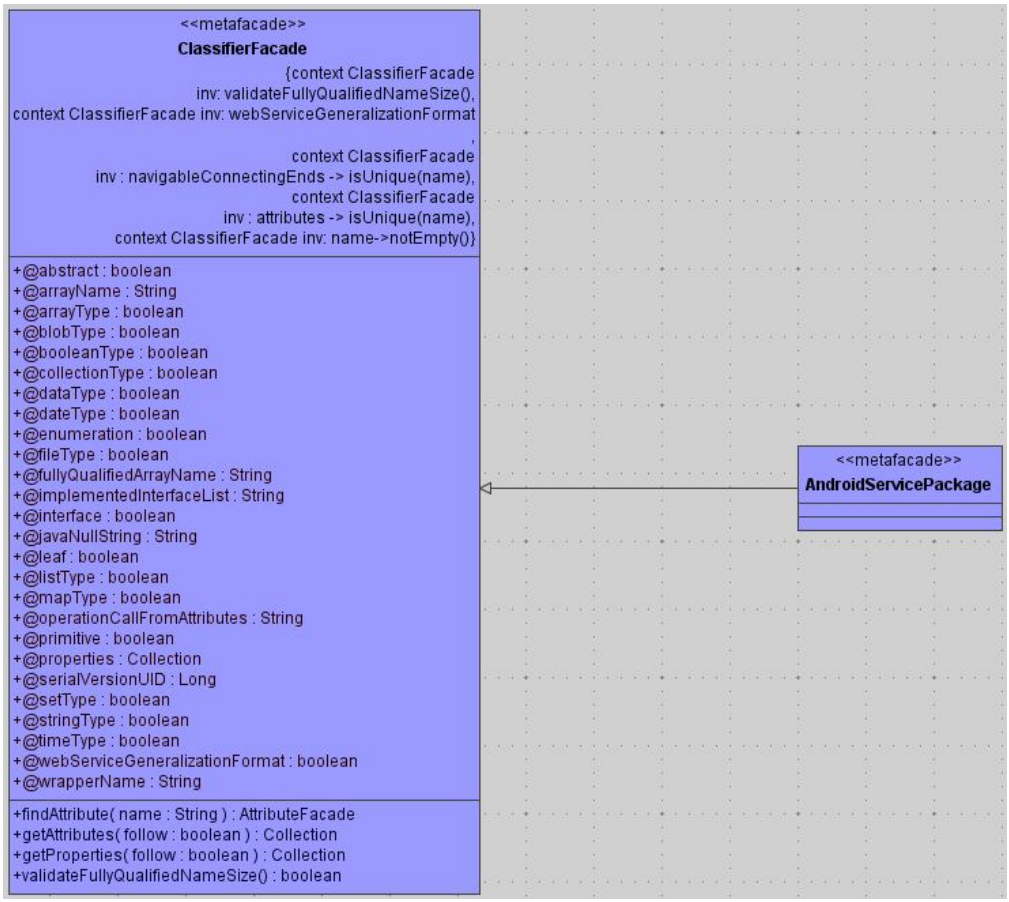

Figura 17: Diagrama de Classe da Camada de Serviço

Como mostrado na *Figura 17*, o modelo da Camada de Serviço é formado pelo Metafaçade *AndroidServicePackage*, que assim como o *AndroidControllerPackage*, é uma especificação do Metafaçade *ClassifierFacade*. No caso da Camada de Serviço, o *ClassifierFacade* é responsável por fornecer, além dos nomes da Classes e seus atributos, como na Camada de Controle, também as dependências contidas nas Classes com estereótipo <<*Service*>>.

4.2.3) Templates

Templates são arquivos moldes capazes de gerar outros arquivos texto de acordo com uma transformação pré-determinada.

No Cartucho Android foram desenvolvidos diversos Templates para que estes pudessem gerar os arquivos necessários para o funcionamento de um Projeto Android.

Para melhor organizar o entendimento dos Templates em nosso Cartucho, a estrutura do mesmo foi dividida em três partes: Api, Impl e o Projeto.

De acordo com a *Figura 18*, a localização dos Templates na estrutura do Cartucho Android é dada por “andromda-android/src/templates”:

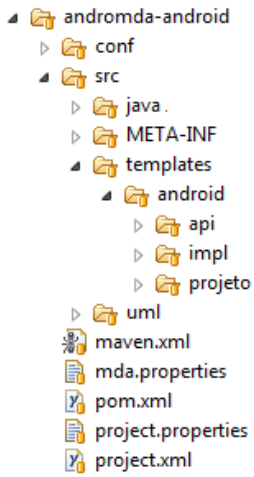

Figura 18: Estrutura dos Templates

Os Templates localizados na pasta “api” são Interfaces e serão implementados pelas classes em “impl”. Apenas os Templates que estão dentro de “impl” dependem de Classes específicas da plataforma, no caso o Android (ver seção 2.7.1).

Os que estão dentro de “projeto” são independentes de plataforma e definem os serviços e Classes básicas que serão criados de acordo com a modelagem do sistema.

4.2.3.1) Templates API

No diretório “api” estão os Templates responsáveis por gerar as Interfaces de nossa API. Tais interfaces serão os principais arquivos capazes de reduzir o acoplamento ao framework de plataforma, uma vez que estes farão a ligação do Projeto Piloto à plataforma desejada.

Como os Templates são escritos em Velocity, sua extensão é dada por vsl. Cada Template aqui presente tem seu papel fundamental na geração dos arquivos Java, conforme descrito na seção 3.1.

4.2.3.2) Templates Impl

Neste diretório estão os Templates que gerarão as Classes que implementam as Interfaces geradas pelos Templates “api”. Cada arquivo tem em seu nome o “Impl” acrescido, mostrando que poderá ser customizado após sua geração. Eles serão gerados na pasta Source do Projeto, uma vez que nesta ficarão os arquivos que poderão sofrer alguma alteração.

4.2.3.3) Templates Projeto

Para facilitar a organização dos Templates responsáveis por gerar os arquivos do Projeto Android, a estrutura deste diretório foi realizada da seguinte forma:

- **Controller**: onde ficará o Velocity responsável por gerar o arquivo da Camada de Controle.
- **Data**: onde ficarão os Velocities responsáveis por gerar os arquivos da Camada de Dados.
- **Service**: onde ficarão os Velocities responsáveis por gerar os arquivos da Camada de Serviço, onde contém as Classes referentes à lógica do projeto.
- **View**: onde ficarão os Velocities responsáveis por gerar os arquivos da Camada de Visão.

4.2.3.3.1) Template Controller

O Template aqui presente será o responsável por gerar a Classe DataBase. Esta, por sua vez, tem o objetivo de acolher o contexto do Banco de Dados e iniciar uma conexão com o mesmo. Ela será gerada no Target do Projeto (ver seção 4.3.3), indicando que este arquivo não poderá ser customizado.

Esta Classe fornece o seguinte serviço:

- *public void executeSQL(string sql):* responsável por executar uma query SQL passada como parâmetro.

A *Figura 19* demonstra um trecho do Velocity responsável por criar a Classe, este é o DataBase.vsl:

```
1#set ($generatedFile = "DataBase.java")
2#if($stringUtils.isNotEmpty($customTypesPackage))
3#set ($generatedFile = "${stringUtils.replace($customTypesPackage,'.','/')}/${controllerPackageName}/${generatedFile}")
4#end
5package ${customTypesPackage}.${controllerPackageName};
6
7/*
8 Attention: Generated code! Do not modify by hand!
9 Generated by: DataBase.vsl in andromda-android-cartridge.
10*/
11import br.ufrj.dcc.api.controller.ConnectionDataBase;
12import br.ufrj.dcc.api.controller.ContextDataBase;
13import br.ufrj.dcc.api.controller.PersistenceValues;
14import br.ufrj.dcc.api.view.PageFacade;
15import br.ufrj.dcc.impl.controller.ConnectionDataBaseImpl;
16import br.ufrj.dcc.impl.controller.ContextDataBaseImpl;
17import br.ufrj.dcc.impl.controller.PersistenceValuesImpl;
18import br.ufrj.dcc.impl.view.PageFacadeImpl;
19
20public class DataBase{
21
22        private static final String SCHEMA_DATABASE = "${schemaDataBase}";
23        private static final int DATABASE_VERSION = ${dataBaseVersion};
24        private ConnectionDataBase connection;
25        private ContextDataBase context;
26        protected static PersistenceValues db;
27
28        public DataBase (){
29
30                if (context == null){
31                        try {
32                                context = new ContextDataBaseImpl(SCHEMA_DATABASE, DATABASE_VERSION);
33                        } catch (Exception e) {
34                                PageFacade page = new PageFacadeImpl();
35                                page.showErrorMessage("Erro ao criar o Banco!");
36                        }
37                }
```

Figura 19: Trecho de código do Velocity DataBase.vsl

Na linha 1 do trecho do Template há a variável *generatedFile*. Esta variável é responsável por indicar qual o caminho do arquivo que será gerado, incluindo o nome do mesmo. Então, na linha 1 está sendo setado o nome do arquivo a ser criado, junto com a sua extensão, e nas linhas seguintes, o caminho está sendo mostrado.

As variáveis *customTypesPackage* e *controllerPackageName* são responsáveis por indicar o caminho do pacote do projeto e o pacote da classe DataBase, respectivamente.

Há também outras duas variáveis principais, indicadas nas linhas 22 e 23 do trecho de código: *schemaDataBase* responsável por informar qual é o *Schema* do Banco de Dados, e a *dataBaseVersion*, responsável por informar qual a versão atual do Banco de Dados.

4.2.3.3.2) Templates Data

Os Templates Data serão os responsáveis por gerar os arquivos da Camada de Dados, sejam eles *Data Access Objects*, *Data Transfer Objects* ou Entidades. A seguir será explicado cada Velocity deste diretório.

- **DAO.vsl**: este Template é responsável por gerar a Classe *DAO*, a qual faz a ligação entre a Classe DataBase (ver seção 4.2.3.3.1) e os *DAO's* de cada Entidade. Ele é gerado no Target do Projeto, indicando que não poderá ser customizado a posteriori.
- **EntityAbstract.vsl**: junto com o Metafaçade *AndroidControllerPackage* (ver seção 4.2.2), este Template gera a Classe Java de cada Entidade presente no Modelo UML do projeto. Esta Classe será abstrata, indicando que não poderá ser instanciada no desenvolvimento do Projeto. Ela conterá todos os atributos modelados, além do atributo Id, responsável por identificar um objeto. A *Figura 20* mostra um trecho de código deste Template:

```
13public abstract class ${entity.name}{
14
15    #foreach ( $attribute in $entity.attributes )
16        #set ($typeName = $attribute.type.fullyQualifiedName)
17        #if ($attribute.containsEmbeddedObject)
18            #set ($typeName = $attribute.type.fullyQualifiedEntityName)
19        #end
20    private $typeName $attribute.name;
21    #end
22
23    #foreach ( $attribute in $entity.attributes )
24        #set ($typeName = $attribute.type.fullyQualifiedName)
25        #if ($attribute.containsEmbeddedObject)
26            #set ($typeName = $attribute.type.fullyQualifiedEntityName)
27        #end
28
29        #if ($attribute.visibility != "private")
30
31    $attribute.visibility $typeName ${attribute.getterName}(){
32        return ${attribute.name};
33    }
34
35    $attribute.visibility void ${attribute.setterName}(${typeName} ${attribute.name}){
36        this.$attribute.name = ${attribute.name};
37    }
38
39        #end
40
41    #end
```

Figura 20: Trecho de código do Velocity EntityAbstract.vsl

Pode-se observar a partir da linha 31 do trecho que serão gerados todos os métodos getter's e setter's dos atributos. Estes métodos são responsáveis por atribuir e recuperar o conteúdo de um objeto.

- **EntityFactory.vsl**: este Template tem por objetivo gerar os *Data Access Objects* (ver seção 2.6.4) de cada Entidade modelada. Cada DAO conterá também o script de criação da tabela no Banco de Dados referente à sua Entidade. As Classes geradas serão abstratas, indicando que não poderão ser instanciadas. Os serviços oferecidos pelos DAO's são:
    - **Insert**: serviço responsável por fazer a persistência no Banco de Dados. O método responsável por este serviço tem como retorno um objeto do tipo *Long*, responsável por informar o número Identificador (Id) do dado que foi inserido.
    - **Update**: serviço responsável por fazer a atualização de um dado no Banco. Assim como o *Insert*, o método responsável pelo *Update* tem como retorno um objeto do tipo *Long*, capaz de informar o número Identificador (Id) do dado atualizado.
    - **Delete**: serviço responsável por remover um dado do Banco. Seu retorno informa o número Identificador do dado removido.
    - **Find**: serviço responsável por encontrar uma coleção de dados desejados. O método responsável pelo *Find* recebe como parâmetro um DTO (ver seção 2.6.5) com os dados a serem filtrados e retorna uma Lista com o resultado da Busca.

A *Figura 21* mostra um trecho deste Template:

```
26public abstract class ${entity.name}DAO extends DAO {
27
28    private static final String SCRIPT_DB_CREATE =
29           "create table if not exists ${stringUtils.upperCase(${entity.name})}(_id integer primary "+
30           "key autoincrement, " + getAttributes() #if($foreignKey) + getForeignKeys() #end + getEndSQL();
31
32    public ${entity.name}DAO() {
33        executeSQL(SCRIPT_DB_CREATE);
34    }
35
36    public abstract long insert(${entity.name} obj);
37
38    public abstract long update(${entity.name} obj);
39
40    public abstract int delete(long id);
41
42    public abstract List<${entity.name}> find${entity.name}(${entity.name}VO vo);
```

Figura 21: Trecho do Velocity EntityFactory.vsl

Os *DAO's* de cada Entidade serão gerados na pasta Target do Projeto (ver seção 4.3.3).

- **EntityFactoryImpl.vsl**: é responsável por gerar as Classes de Implementação dos *DAO's*. Estas Classes serão geradas na pasta Source do Projeto, uma vez que poderão ser customizadas após a geração. Elas estendem a Classe DAO para poder usufruir dos serviços oferecidos pela Classe DataBase (ver seção 4.2.3.3.1). Por serem Classes de Implementação, os nomes dos arquivos são acrescidos de "*Impl*" A *Figura 22* mostra um trecho deste Velocity:

```
12public class ${entity.name}DAOImpl extends ${entity.name}DAO {
13
14    public ${entity.name}DAOImpl() {
15    }
16
17    public long insert(${entity.name} $stringUtils.lowerCase(${entity.name})){
18       return db.insert(null, null, null);
19    }
20
21    public long update(${entity.name} $stringUtils.lowerCase(${entity.name})){
22       return db.update(null, null, null, null);
23   }
24
25   public int delete(long id){
26       return db.delete(null, null,null);
27   }
28
29   public List<${entity.name}> find${entity.name}(${entity.name}VO $stringUtils.lowerCase(${entity.name})VO){
30       return null;
31   }
32}
```

Figura 22: Trecho do Velocity EntityFactoryImpl.vsl

Como este arquivo é gerado de forma genérica, então os serviços providos pelos *DAO's* estão retornando uma instancia nula.

- **EntityImpl.vsl**: este Velocity é responsável por gerar a Classe da Entidade que poderá ser customizada. Ela estende a Classe abstrata da Entidade gerada pelo Template *EntityAbstract.vsl*, fazendo com que todos os serviços fornecidos pela Classe abstrata sejam disponibilizados nas classes customizadas. A *Figura 23* mostra este Template:

```
1#set ($generatedFile = "${entity.name}Impl.java")
2#if($stringUtils.isNotEmpty($customTypesPackage))
3#set ($generatedFile = "${stringUtils.replace($customTypesPackage,'.','/')}/${modelPackageName}/${generatedFile}")
4#end
5#if ($entity.packageName)
6package $entity.packageName;
7#end
8
9public class ${entity.name}Impl extends ${entity.name}{
10
11    public ${entity.name}Impl(){
12
13    }
14
15}
```

Figura 23: Trecho do Velocity EntityImpl.vsl

Por ser uma Classe de Implementação, os nomes destes arquivos são compostos pelo nome da Entidade fornecido no modelo UML junto com "Impl", como demonstrado na linha 1 do Template.

- **ValueObjectEntity.vsl**: toda Entidade modelada gera consigo também um *Data Transfer Object* – DTO (ver seção 2.6.5), ou também conhecido como *Value Object*. Este arquivo é gerado no Target do Projeto, logo não poderá ser customizado.
- **ValueObject.vsl**: este Template junto com o Metafaçade *ValueObject* do Cartucho AndroMDA, gera as Classes do modelo com o estereótipo <<*ValueObject*>>. O arquivo gerado ficará na pasta Target do Projeto.

4.2.3.3.3) Templates Service

Os Templates Service são responsáveis por fazer a conexão entre o Controle e os Dados. Portanto, a Visão chama o Controle, o qual chama um serviço quando é

necessária qualquer interação com o Banco de Dados, e o Serviço fará a persistência ou consulta desejada. A seguir será explicado cada Template desse diretório:

- **ServiceHandler.vsl**: Responsável por gerar os serviços criados no modelo com estereótipo <<*Service*>>. O nome do serviço será composto por (nome do serviço)Handler.java e ele será gerado na pasta “cs” do projeto e não será modificado pelo desenvolvedor. O arquivo criado terá os métodos *filter*, *insert*, *delete* e *update* para cada entidade associada a ele no modelo.

```
public abstract class ${service.name}Handler extends DataBase{

    #foreach ($dependencia in $service.sourceDependencies)

    protected ${dependencia.targetElement.name}DAO $stringUtils.lowerCase(${dependencia.targetElement.name})Dao = null;

    protected abstract Collection handleFilter(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO);
    protected abstract void handleInsert(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO);
    protected abstract void handleDelete(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO);
    protected abstract void handleUpdate(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO);

    public ${service.name}Handler(){
        if ($stringUtils.lowerCase(${dependencia.targetElement.name})Dao == null)
            $stringUtils.lowerCase(${dependencia.targetElement.name})Dao = new ${dependencia.targetElement.name}DAOImpl();
    }
```

Figura 24: Trecho do Velocity ServiceHandler.vsl

Como podemos ver na *Figura 24* o construtor do serviço irá instanciar um novo DAO para cada dependência, isto é, para cada classe associada ao *Handler*.

```
public Collection Filter(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO){
    List list${dependencia.targetElement.name} = (List) handleFilter($stringUtils.lowerCase(${dependencia.targetElement.name})VO);
    return list${dependencia.targetElement.name};
}

public void Insert(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO){
    db.beginTransaction();
    try {
        handleInsert($stringUtils.lowerCase(${dependencia.targetElement.name})VO);
        db.setTransactionSuccessful();
    } catch (Exception e) {
        System.out.println("Transaction Error: " + e.toString());
    } finally {
        db.endTransaction();
    }
}
public void Delete(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO){
    db.beginTransaction();
    try {
        handleDelete($stringUtils.lowerCase(${dependencia.targetElement.name})VO);
        db.setTransactionSuccessful();
    } catch (Exception e) {
        System.out.println("Transaction Error: " + e.toString());
    } finally {
        db.endTransaction();
    }
}
public void Update(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO){
    db.beginTransaction();
    try {
        handleUpdate($stringUtils.lowerCase(${dependencia.targetElement.name})VO);
        db.setTransactionSuccessful();
    } catch (Exception e) {
        System.out.println("Transaction Error: " + e.toString());
    } finally {
        db.endTransaction();
    }
}

#end
```

Figura 25: Trecho do Velocity ServiceHandler.vsl

A *Figura 25* mostra os métodos que serão criados pelo Velocity.

Os métodos *insert*, *delete* e *update* fazem o controle de transações, utilizando a classe *SQLiteDataBase* do Android, pois eles irão alterar dados no Banco.

Todos os métodos executam seus pares, respectivamente, que serão criados na classe *Impl* do serviço, passando como parâmetro o DTO da entidade associada.

- **ServiceHandlerImpl.vsl**: Gera o arquivo [nome do serviço]Impl.java. Esse arquivo será criado na pasta Source do projeto e poderá ser alterado pelo desenvolvedor.

```
import java.util.Collection;
#foreach ($dependencia in $service.sourceDependencies)
import ${customTypesPackage}.vo.${dependencia.targetElement.name}VO;
#end

public class ${service.name}HandlerImpl extends ${service.name}Handler{

    public ${service.name}HandlerImpl() {

    }
    #foreach ($dependencia in $service.sourceDependencies)
    @Override
    public Collection handleFilter(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO) {
        return null;
    }

    @Override
    public void handleInsert(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO) {

    }
    public void handleDelete(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO) {

    }

    public void handleUpdate(${dependencia.targetElement.name}VO $stringUtils.lowerCase(${dependencia.targetElement.name})VO) {

    }
    #end

}
```

Figura 26: Trecho do Velocity ServiceHandlerImpl.vsl

Essa classe estende da classe criada pelo Velocity *ServiceHandler.vsl* e implementa seus métodos (*filter*, *insert*, *delete* e *update*) para que eles possam ser customizados e o usuário possa criar as regras de negócio do projeto.

O desenvolvedor deve utilizar o *DAO* (ver seção 2.6.4) instanciado na classe abstrata do serviço para fazer as transações necessárias com o Banco de Dados.

4.2.3.3.4) Templates View

Os Templates View são os responsáveis por criar as telas que serão exibidas para o usuário, bem como as interfaces de acesso a esses elementos de tela. É importante ressaltar que, neste ponto do projeto, não há suporte a customização dos elementos de tela, eles estão

dispostos seguindo um layout que se assemelha aos layouts utilizados em páginas de formulário na maioria dos sistemas vigentes.

- **Controller.vsl** : Este Template gera as classes de controles de cada caso de uso. É ele o responsável por fazer o uso da API definida neste trabalho, para recuperar e apresentar dados na tela do usuário. Para cada parâmetro contido em um caso de uso, as classes geradas por este Template devem manter sincronia entre o que está sendo apresentado na tela neste momento e o conteúdo dos parâmetros do Form. Este controle é feito automaticamente, tornando esse trabalho transparente para o desenvolvedor. As classes geradas por este Template também são responsáveis pela navegação entre páginas e casos de usos, garantindo que a navegação represente tudo o que foi definido no modelo. Contém os seguintes métodos:
    - *public void iniciar():* Método responsável por fazer as inicializações necessárias da classe, bem como apresentar a página inicial deste caso de uso.
    - *protected void preInit():* Por padrão, este método não possui implementação. Ele existe para que o desenvolvedor possa fazer algum tipo de processamento nesta classe, antes da página ser apresentada ao usuário.
    - *protected void posInit():* De maneira semelhante ao método preInit, este método também não possui implementação, caso o desenvolvedor queira fazer algum processamento, após a inicialização da classe, porém antes do usuário poder executar algum comando, deve sobrescrever este método.
    - Métodos modelados na classe de controle, são criados aqui como abstratos, obrigando que o desenvolvedor implemente-os.
    - Métodos para preenchimento de tabelas também são criados aqui, tornando simples a tarefa de apresentar um número grande de dados referentes a um mesmo domínio.
- **ControllerImpl.vsl**: Este Template gera classes que são filhas das classes geradas pelo Template Controller.vsl. Aqui se encontram os chamados pontos de implementação, onde o desenvolvedor processará as requisições do usuário. Contém os métodos modelados na classe de controle.

4.2.3.3.4.1) Template Form

Este Template gera uma classe para cada caso de uso que representará os parâmetros de cada tela deste caso de uso. Uma escolha natural para este Template seria que o mesmo

fosse gerado para cada página do sistema, porém esta abordagem tende a gerar muitas classes, que inviabilizariam a maioria dos sistemas, além de todo o processamento que seria necessário para a passagem de parâmetros entre as telas. Numa outra abordagem, poderia ser gerado somente um Form para todo o sistema, o que melhoraria o desempenho, porém tornaria o desenvolvimento mais difícil, pois ficaria a cargo do desenvolvedor garantir que um parâmetro existente em um caso de uso, não estivesse em outro com o mesmo nome, ou que, caso estivesse, representassem a mesma coisa e também se preocupar em gerenciar os momentos em que se quer ou não manter um parâmetro no formulário. Acreditamos que a abordagem de um formulário para cada caso de uso, produz um sistema com performance satisfatória e de fácil implementação.

Os métodos gerados neste Template são métodos para ler ou alterar os parâmetros da tela. As classes de controle se ocupam de manter a sincronia entre as classes Form e o que é apresentado ao usuário.

4.2.3.3.4.2) Template Page

Em sistemas Android é sugerido que as telas sejam definidas em arquivos do tipo XML. É este Template que gera todas as telas do sistema (ações com o estereótipo FrontEndView).

4.2.4) Metafaçades

Metafaçades são Façades (ver seção 2.6.1) utilizados para fornecer acesso aos modelos carregados por um repositório. Eles são gerados pelo *andromda-meta-cartridge* (ver seção 2.2.1).

No Cartucho Android foram criados diversos Metafaçades a fim de gerar, junto com os Templates, arquivos para o Projeto Piloto, baseado em Android.

Os Metafaçades desenvolvidos a partir do Modelo têm acrescido "*LogicImpl*" em seu nome original, dizendo que estes arquivos Java podem ser customizados, acrescidos de novos métodos caso haja necessidade.

4.3) O Projeto Piloto

Para que possa se tornar mais clara a implementação da API, foi criado um Projeto Piloto. Este projeto exemplifica como um Modelo funcional em um Ambiente Desktop, também se torna funcional em um Ambiente Móvel.

Utilizamos o exemplo adotado no curso de Capacitação dos estagiários da Fundação Coppetec[60], o qual utiliza o MDArte como framework de desenvolvimento dos Sistemas Web lá existentes. Este exemplo é o projeto Suporte País, um Projeto modelado dentro dos padrões UML 2.0, desenvolvido no software Magic Draw[61].

4.3.1) O Suporte País

O Suporte País foi modelado de forma a obedecer a estrutura MVC (ver seção 2.3). A *Figura 27* mostra a estrutura do Modelo do Suporte País:

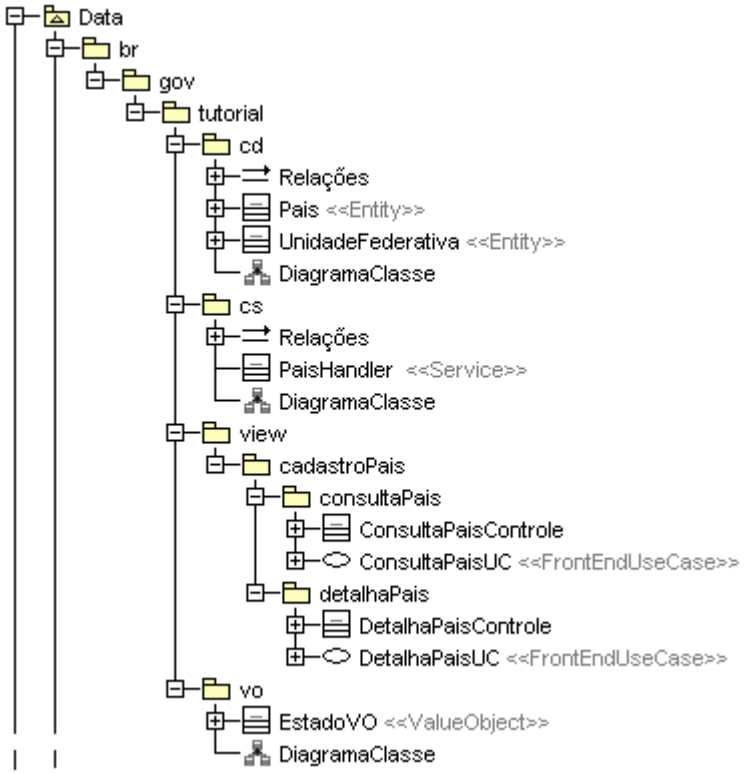

Figura 27: Modelo do Suporte País

O diretório br/gov/tutorial é a estrutura inicial do Projeto. Dentro da pasta “cd” estão todas as Entidades contidas no Modelo. As Classes são identificadas pelo estereótipo <<*Entity*>> apresentado ao lado de cada uma. Esta é a Camada de Dados. No Anexo A pode-se observar o Diagrama de Classe correspondente a esta camada.

Na Camada de Serviço, identificada pela pasta “cs”, estão as Classes modeladas como Serviço. Na *Figura 27* observa-se uma Classe com o estereótipo <<*Service*>>.

Na Camada de Visão, ou “view”, estão os Casos de Uso[62] do Projeto, assim como os Diagramas de Atividades[63] provenientes de cada um. É a partir deles, que estará todo o fluxo de funcionamento do Projeto. Cada Caso de Uso tem o estereótipo <<*FrontEndUseCase*>> indicando ao Cartucho que estes conterão as Classes de Controle, assim como as Telas de apresentação ao usuário, identificadas pelo estereótipo <<*FrontEndView*>>. As *Figuras 37*, *38* e *39 do Anexo A* mostram as Telas de Apresentação do Suporte País. São elas do Consultar País, Resultado da Consulta e Detalhar País respectivamente.

Por último, estão os *Data Transfer Objects* – DTO (ver seção 2.6.5) ou simplesmente *Value Objects* – VO. Essas Classes, cujo papel principal é transferir dados são identificadas pelo estereótipo <<*ValueObject*>>, conforme pode-se notar na *Figura 27*.

Conforme a *Figura 33* do Anexo A, um País pode ter uma ou mais Unidades Federativas relacionadas. Isso faz com que sua cardinalidade seja dada por 1..* (representação UML).

Já na *Figura 34* do Anexo A, o Diagrama de Classe mostra que o Serviço nomeado como *PaisHandler* tem uma dependência com a Entidade País, fazendo com que este Serviço possa seguir as regras de negócio atribuídas à Entidade País.

O Diagrama de Atividades informa o fluxo de dados de um determinado Caso de Uso, e na *Figura 35* do Anexo A pode-se observar o fluxo do Consultar País. Após o usuário preencher alguns campos de filtro na tela inicial, a busca é realizada e é exibida na tela de Resultado. Esta tem um botão Voltar para a tela anterior.

Finalmente, no Diagrama de Atividades do Detalhar País (ver *Figura 36* do Anexo A) é mostrado o fluxo de dados de um detalhamento. O identificador do País é passado como parâmetro para esse Caso de Uso, é realizada a busca no Banco de Dados, e na Lista “ufs” são passadas todas as Unidades Federativas relacionadas ao País. Estas Unidades Federativas, junto com o País, são exibidas na Tela final de Resultado, o qual tem também o botão de Voltar para a tela anterior.

Com este modelo é possível gerar um Sistema Web, utilizando o framework MDArte. No entanto, esse exemplo também pode ser usado para gerar um Sistema Android, utilizando a API por nós desenvolvida, como será demonstrado a seguir.

4.3.2) A Estrutura Inicial

Ao utilizar o plug-in Android Development Tool (ver seção 4.1.2.1) para criar um novo Projeto Android, uma estrutura inicial é gerada. Entretanto, antes de conhecer a estrutura formada, é necessário conhecer os quatro tipos principais de uma aplicação Android, são eles: Atividade, Serviços, Receptores e Provedor de Conteúdo.

4.3.2.1) Atividade

É a forma mais visível de uma aplicação Android. É a Atividade que apresenta a Tela de Visão, ou User Interface[64] – UI, à aplicação, juntamente com a Classe de Visualização (ver seção 4.3.2.5). Esta última é implementada com vários elementos da UI, como caixas de texto, botões, seletores, rótulos, entre outras.

Uma aplicação poderá conter somente uma atividade ou várias. Elas têm um relacionamento 1 para 1 com as Telas da Aplicação. E o Sistema Operacional tem o papel de determinar qual atividade é mais qualificada para atender o requisito, chamado de Intento, especificado.

4.3.2.2) Serviços

Um Serviço no Android é uma aplicação que não possui uma User Interface – UI. Aplicações cujas execuções estão em “background” e realizam diversas operações são consideradas Serviços pelo Android.

4.3.2.3) Receptor

Um Receptor é um componente da Aplicação responsável por receber pedidos dos Intentos. Este também não possui uma UI. No arquivo *AndroidManifest.xml* (ver seção 4.3.2.6.1) ficam os registros dos receptores.

4.3.2.4) Provedor de Conteúdo

O Provedor de Conteúdo, ou do inglês ContentProvider[65], é um mecanismo que funciona para abstrair o armazenamento de Dados do Android. Ele é utilizado para abstrair o acesso a um armazenamento de dados específico. Muitas vezes ele age como um servidor de Banco de Dados, pois para ler e gravar um determinado

conteúdo, este deve ser passado pelo ContentProvider apropriado ao invés de acessar um Banco de Dados diretamente.

4.3.2.5) Visualizações

Uma Atividade (ver seção 4.3.1.1) no Android pode empregar Visualizações para que os elementos da UI possam ser exibidos. Tais visualizações podem seguir um dos designs de layout:

- **LinearVertical**: cada elemento subsequente segue o anterior ficando abaixo dele em uma única coluna.
- **LinearHorizontal**: cada elemento subsequente segue o anterior ficando ao lado dele em uma única linha
- **Relative**: cada elemento subsequente pode ser posicionado a partir da posição do elemento anterior.
- **Table**: é uma série de linhas e colunas conforme a disposição das tabelas HTML, onde cada célula pode conter um elemento de Visualização.

O Layout utilizado no Suporte País será o primeiro explicado aqui, ou seja, o *LinearVertical*, uma vez que este já é utilizado pelo Framework MDArte para aplicações de Java para Web (JEE).

Ao selecionar um layout específico, as visualizações serão utilizadas para apresentar a UI. Alguns elementos de Visualização serão listados a seguir:

- Button
- ImageButton
- EditText
- TextView (semelhante ao Label)
- CheckBox
- Radio Button
- Gallery e ImageSwitcher (para exibição de imagens)
- List
- Grid

- DatePicker
- TimePicker
- Spinner (semelhante a uma caixa de combinação)
- AutoComplete (EditText com recurso de conclusão de texto)

4.3.2.6) A Estrutura

A *Figura 28* mostra a estrutura inicial gerada pelo plug-in ADT (ver seção 4.1.2.1):

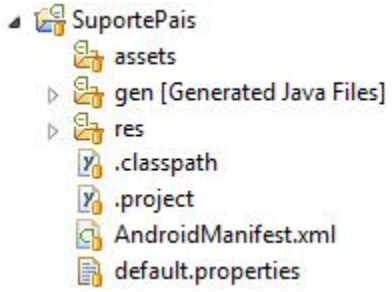

Figura 28: Estrutura Inicial do Suporte País para Android

No diretório “gen”, ficam os arquivos Java gerados pelo Wizard do plug-in do Android. Um desses arquivos é o *R.java*.

A *Figura 29* mostra o arquivo *R.java* do Suporte País. Nela pode-se observar que este arquivo é de suma importância para o funcionamento da aplicação, uma vez que é responsável por conectar os recursos visuais ao código fonte Java. Ele possui subclasses anônimas, sendo que elas contêm identificadores para cada elemento da UI. Pode-se observar que todas essas subclasses são estáticas.

O *R.java* é gerado e gerenciado pelo plug-in ADT (ver seção 4.1.2.1), portanto, não cabe ao gerador de código aqui proposto sobrescrevê-lo. É possível, no momento de se criar um UI para o Android, especificar o uso de identificadores gerenciados pelo ADT ou não. Optamos por utilizá-los com o objetivo de tornar o desenvolvimento próximo ao que os desenvolvedores Android empregam atualmente, facilitando, assim o uso da ferramenta.

```java
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package br.gov.tutorial;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int Tabela01=0x7f06000d;
        public static final int TableLayout01=0x7f060001;
        public static final int TableRow01=0x7f060002;
        public static final int TableRow02=0x7f060004;
        public static final int TableRow03=0x7f060006;
        public static final int TableRow04=0x7f060008;
        public static final int TableRow05=0x7f06000b;
        public static final int codigo=0x7f060003;
        public static final int codigoAuxiliar=0x7f060007;
        public static final int consulta=0x7f06000c;
        public static final int descricao=0x7f060009;
        public static final int main=0x7f060000;
        public static final int novaconsulta=0x7f06000a;
        public static final int valor=0x7f060005;
    }
    public static final class layout {
        public static final int detalhamentodopais_detalhapais=0x7f030000;
        public static final int preenchaosdadosdaconsultadepais_consultarpais=0x7f030001;
        public static final int resultadodaconsultadepais_consultarpais=0x7f030002;
    }
    public static final class string {
        public static final int Nova_Consulta=0x7f050009;
        public static final int app_name=0x7f050001;
        public static final int codigo=0x7f050002;
        public static final int codigoAuxiliar=0x7f050003;
        public static final int consulta=0x7f050006;
        public static final int descricao=0x7f050005;
        public static final int hello=0x7f050000;
        public static final int titulo_preencherosdadosdaconsulta=0x7f050007;
        public static final int titulo_resultadodaconsultadepais=0x7f050008;
        public static final int valor=0x7f050004;
    }
    public static final class style {
        public static final int BotaoConsulta=0x7f040000;
```

Figura 29: Arquivo R.java do Suporte País

No diretório “res” estão os recursos da Aplicação, ou do inglês application resources. É neste diretório que estarão as imagens, na pasta “drawable”; os layouts das telas e dos formulários, na pasta “layout”; e os arquivos de variáveis, na pasta “values”.

Cada layout é representado por um arquivo no formato XML. Este arquivo declara os componentes e suas propriedades para uso da Aplicação. A *Figura 30* mostra um trecho da tela de Consulta do Suporte País como exemplo:

```
1<?xml version="1.0" encoding="utf-8"?>
2<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3    android:id="@+id/main"
4    android:orientation="vertical"
5    android:layout_width="fill_parent"
6    android:layout_height="fill_parent"
7    >
8    <HorizontalScrollView
9        android:layout_width="fill_parent"
10       android:layout_height="fill_parent">
11       <ScrollView
12           android:layout_width="fill_parent"
13           android:layout_height="fill_parent">
14               <TableLayout
15                   android:id="@+id/TableLayout01"
16                   android:layout_width="fill_parent"
17                   android:layout_height="wrap_content">
18                   <TextView
19                       android:layout_width="fill_parent"
20                       android:layout_height="wrap_content"
21                       android:text="@string/titulo.preencherosdadosdaconsulta"
22                   />
23                   <TableRow
24                       android:id="@+id/TableRow01"
25                       android:layout_width="fill_parent"
26                       android:layout_height="wrap_content">
27                       <TextView
28                           android:layout_width="wrap_content"
29                           android:layout_height="wrap_content"
30                           android:text="@string/codigo"
```

Figura 30: Trecho do arquivo de Layout do Consulta País

Esse layout é do tipo *LinearVertical* (ver seção 4.3.2.5), conforme pode-se comprovar na linha 4 do trecho apresentado. Isto significa que todos os elementos estarão em uma única coluna.

Os componentes serão manipulados pelas Classes Java, então será necessário associá-los a variáveis na *Activity* (ver seção 4.3.2.1) para que se possa, por exemplo, adicionar ações aos botões, adquirir os valores preenchidos nas Caixas de Texto, atribuir valores aos campos etc.

4.3.2.6.1) AndroidManifest.xml

Este arquivo é o descritor de implementação de aplicação para as Aplicações feitas para o Android. É ele que lista todas as atividades, serviços, provedor de

conteúdo ou receptores contidos na Aplicação. A *Figura 31* mostra o AndroidManifest.xml do Suporte País, como exemplo:

```
1<?xml version="1.0" encoding="utf-8"?>
2<manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="br.gov.tutorial"
4      android:versionCode="1"
5      android:versionName="1.0">
6    <application android:icon="@drawable/icon" android:label="@string/app_name">
7        <activity android:name=".Main"
8                  android:label="@string/app_name">
9            <intent-filter>
10               <action android:name="android.intent.action.MAIN" />
11               <category android:name="android.intent.category.LAUNCHER" />
12           </intent-filter>
13       </activity>
14
15   </application>
16
17
18</manifest>
```

Figura 31: AndroidManifest.xml do Suporte País

Alguns artefatos importantes serão explicados a seguir:

- O nome do pacote do arquivo de origem é informado neste arquivo. Na tag <*manifest*> são declaradas as principais informações do Projeto, como o pacote a que ele pertence, versão e configurações da Aplicação.
- A tag <*application*> possui atributos que fazem referência a outros atributos da Aplicação. O símbolo "@" antes de um identificador significa que o arquivo deverá consultar o diretório informado para que possa reconhecer o elemento a ser acessado. Um exemplo seria o elemento ícone que está dentro da pasta drawable, então será representado por *@drawable/icon*, conforme mostrado na linha 6 da *Figura 31*. Nesta tag também pode ser informada o nome da aplicação através do atributo *android:label*. Pode-se observar na linha 6 que este nome é identificado através da cadeia de caracteres representado por *app_name*, localizado dentro da pasta "string".
- A tag <*activity*> informa os principais artefatos sobre uma Atividade, tais como:
    - A Classe Java a qual será representada.

- A tag <*intent-filter*> representa o IntentFilter[66]. Esse filtro informa que está implementando-se a ação principal, localizada no "ativador" do Sistema Operacional Android, ou seja, ela pode ser iniciada como uma aplicação a partir da lista Principal de Aplicações do Android.

4.3.3) A Estrutura adotada

Para organizar melhor o desenvolvimento do Projeto Suporte País para Android, uma estrutura semelhante à adotada pelo MDArte (ver seção 2.2.2) também foi seguida. Esta estrutura constitui-se basicamente de dois diretórios principais: o Target e o Source.

O Target é onde ficarão os arquivos gerados pelo framework e que não poderão sofrer customizações após sua criação. As Classes abstratas geradas a partir de Entidades modeladas são exemplos de arquivos que ficarão no Target. Outro exemplo, é a Classe responsável por fazer conexão com o Banco de Dados, a *DataBase* (ver seção 4.2.3.3.1) , uma vez que a forma de pegar o contexto e o conectar ao Banco não muda.

Já os arquivos que poderão ser customizados, após serem criados pelo MDArte ficarão no Source, representado pela pasta "src". Exemplos de arquivos são as Classes de Controle, as quais terão as chamadas dos *DAO's* e também a ligação com as View's, as Classes de DAOImpl's (ver *EntityFactoryImpl.vsl* na seção 4.2.3.3.2), onde estarão os métodos de persistências e busca declarados, entre outros.

Assim, o Projeto foi gerado conforme o modelo UML apresentado na seção 4.3.1, sendo que a sua estrutura é repetida tanto na pasta Target, quanto na Source. Dessa forma, a consistência do projeto é garantida, uma vez que o local de acesso de uma determinada Classe seguirá sempre um padrão.

4.3.4) Utilizando a API

Após a geração parcial de código do Projeto, o desenvolvedor necessitará implementar alguns códigos. Esta implementação se dá a partir da utilização da API desenvolvida, uma vez que o objetivo é justamente reduzir o acoplamento de

frameworks específicos de plataforma, produzidos quando a geração parcial de código é realizada. Assim, a *Figura 32* exemplifica a utilização a API com o desenvolvimento de código da Classe *DataBase*.

```
1  package br.gov.tutorial.controller;
2
3  import br.ufrj.dcc.api.controller.ConnectionDataBase;
4  import br.ufrj.dcc.api.controller.ContextDataBase;
5  import br.ufrj.dcc.api.view.PageFacade;
6  import br.ufrj.dcc.impl.controller.ConnectionDataBaseImpl;
7  import br.ufrj.dcc.impl.controller.ContextDataBaseImpl;
8  import br.ufrj.dcc.impl.controller.PersistenceValuesImpl;
9  import br.ufrj.dcc.impl.view.PageFacadeImpl;
10
11 public class DataBase{
12
13     private static final String NOME_BANCO = "SuportePaisDB";
14     private static final int VERSAO_BANCO = 1;
15     private ConnectionDataBase connection;
16     private ContextDataBase context;
17     protected static PersistenceValuesImpl db;
18
19     public DataBase (){
20
21         if (context == null){
22             try {
23                 context = new ContextDataBaseImpl(NOME_BANCO, VERSAO_BANCO);
24             } catch (Exception e) {
25                 PageFacade page = new PageFacadeImpl();
26                 page.showErrorMessage("Erro ao criar o Banco!");
27             }
28         }
29         if (connection == null){
30             try {
31                 connection = new ConnectionDataBaseImpl(((ContextDataBaseImpl)context));
32             } catch (Exception e) {
33                 PageFacade page = new PageFacadeImpl();
34                 page.showErrorMessage("Erro ao conectar ao Banco!");
35             }
36         }
37         if (db == null){
38             db = new PersistenceValuesImpl(connection);
39         }
40     }
```

Figura 32: Trecho do código da Classe DataBase

Como se pode notar, entre as linhas 3 e 9 da *Figura 32*, há os *import's* das Classes contidas na API desenvolvida. Isto significa que a Classe DataBase só utilizará instâncias das Classes da API, fazendo com que a Conexão e o Contexto do Banco de Dados, que é o objetivo desta Classe, seja programado de uma forma padrão, e que a parte específica de conexão e contexto fique nas Classes "Impl" da API. Da maneira

que foi desenvolvida a Classe *DataBase* não precisará sofrer alterações caso o SGBD venha a mudar. A mudança será necessária apenas nas Classes de Implementação da API.

A *Figura 40 do Anexo A* mostra um exemplo de implementação de código utilizando a API e não utilizando a API. Neste caso existem dois exemplos: um implementado utilizando o Banco de Dados SQLite e outro utilizando o Banco de Dados Oracle.

Assim, foi desenvolvido todo o Suporte País, utilizando Classes da API, tornando o acoplamento mínimo, já que a importação de algumas Classes específicas não é mais necessária.

5) Conclusão

No desenvolvimento de sistemas em geral é muito comum a necessidade de alterar a sua plataforma inicial. Isso se deve ao fato de que muitas tecnologias começam a ficar ultrapassadas e a mudança pode trazer muitos ganhos para o usuário. No MDA, como grande parte do sistema é desenvolvido via modelos independentes de plataforma, esse trabalho é minimizado.

A solução apresentada neste projeto propõe que o código utilizado também seja o mais independente possível. Ao longo de seu desenvolvimento foi implementada uma solução para que seja facilitado o desacoplamento de sistemas construídos, utilizando o framework MDArte, isto é, para que seja mais fácil a portabilidade deste sistema entre várias plataformas. Essa solução foi a criação de uma API que permite uma menor dependência entre o sistema e a linguagem adotada. Com ela o desenvolvedor poderá desenvolver o código implementando interfaces, as quais estão divididas em dois grupos: View e Controller. As interfaces View são responsáveis pela Camada de Visão do Projeto. Já as interfaces Controller são as responsáveis por fazer a conexão com o Banco assim como persistências e consultas. Isto facilita, pois em caso de portabilidade de plataforma, o código implementado no Projeto não sofrerá alteração, e sim as implementações da API.

Assim conclui-se que é possível diminuir muito a dependência entre os sistemas e suas plataformas, utilizando APIs para cada linguagem. Quando o MDA chegar a geração total de código, todos os sistemas serão livres de plataforma, porém é muito difícil chegar a este objetivo.

Como futuros trabalhos podem-se citar a extensão da API aqui proposta, bem como a implementação da mesma para as plataformas mais utilizadas no mercado e, melhorá-las cada vez mais, de modo que fosse facilitada a migração de um sistema para outra linguagem.

6) Referências

[1] Disponível em http://www.omg.org/mda/, *acessado em 04/01/2011*
[2] Disponível em http://www.omg.org/, *acessado em 04/01/2011*
[3] Disponível em http://www.uml.org/, *acessado em 04/01/2011*
[4] Disponível em http://www.omg.org/mof/, *acessado em 04/01/2011*
[5] Disponível em http://en.wikipedia.org/wiki/Platform-independent_model, *acessado em 04/01/2011*
[6] Disponível em http://en.wikipedia.org/wiki/Platform-specific_model, *acessado em 04/01/2011*
[7] Disponível em http://en.wikipedia.org/wiki/Application_programming_interface, *acessado em 04/01/2011*
[8] Disponível em http://www.softwarepublico.gov.br/ver-comunidade?community_id=9022831, *acessado em 04/01/2011*
[9] Disponível em http://pt.wikipedia.org/wiki/Android, *acessado em 04/01/2011*
[10] Disponível em http://www.andromda.org, *acessado em 04/01/2011*
[11] Disponível em http://en.wikipedia.org/wiki/Object_Constraint_Language, *acessado em 07/01/2011*
[12] Disponível em http://en.wikipedia.org/wiki/XMI, *acessado em 07/01/2011*
[13] Disponível em http://pt.wikipedia.org/wiki/XML, *acessado em 07/01/2011*
[14] Disponível em http://www.andromda.org/docs/andromda-templateengines/index.html, *acessado em 07/01/2011*
[15] Disponível em http://www.andromda.org/docs/andromda-metafacades/index.html, *acessado em 07/01/2011*
[16] Disponível em http://velocity.apache.org, *acessado em 07/01/2011*
[17] Disponível em http://www.apache.org, *acessado em 07/01/2011*
[18] Disponível em http://en.wikipedia.org/wiki/JAR_file, *acessado em 07/01/2011*
[19] Disponível em http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html, *acessado em 07/01/2011*
[20] Disponível em http://en.wikipedia.org/wiki/EAR_(file_format), *acessado em 07/01/2011*
[21] Disponível em http://www.andromda.org/docs/andromda-cartridges/andromda-hibernate-cartridge/index.html, *acessado em 07/01/2011*
[22] Disponível em http://www.andromda.org/docs/andromda-cartridges/andromda-ejb-cartridge/index.html, *acessado em 07/01/2011*
[23] Disponível em http://www.andromda.org/docs/andromda-cartridges/andromda-spring-cartridge/index.html, *acessado em 07/01/2011*
[24] Disponível em http://www.andromda.org/docs/andromda-cartridges/andromda-jsf-cartridge/index.html, *acessado em 07/01/2011*
[25] Disponível em http://pt.wikipedia.org/wiki/MVC, *acessado em 07/01/2011*
[26] Disponível em http://www.fragmental.com.br/wiki/index.php?title=MVC_e_Camadas, *acessado em 07/01/2011*
[27] Disponível em http://www.java.com/pt_BR/, *acessado em 07/01/2011*

[28] Disponível em http://xdoclet.sourceforge.net/xdoclet/index.html, *acessado em 07/01/2011*
[29] Disponível em http://boss.bekk.no/boss/middlegen/ant/index.html, *acessado em 07/01/2011*
[30] Disponível em http://www.jetbrains.com/idea/, *acessado em 07/01/2011*
[31] Disponível em http://en.wikipedia.org/wiki/Symbian, *acessado em 09/01/2011*
[32] Disponível em http://www.openhandsetalliance.com/, *acessado em 09/01/2011*
[33] Disponível em http://pt.wikipedia.org/wiki/Google, *acessado em 09/01/2011*
[34] Disponível em http://www.eclipse.org, *acessado em 09/01/2011*
[35] Disponível em http://pt.wikipedia.org/wiki/Dalvik_virtual_machine, *acessado em 09/01/2011*
[36] Disponível em http://en.wikipedia.org/wiki/Dalvik_%28software%29, *acessado em 09/01/2011*
[37] Disponível em http://www.nokia.com.br, *acessado em 09/01/2011*
[38] Disponível em http://www.canalys.com/, *acessado em 09/01/2011*
[39] Disponível em http://www.motorola.com.br, *acessado em 09/01/2011*
[40] Disponível em http://www.samsung.com.br, *acessado em 09/01/2011*
[41] Disponível em http://www.htc.com, *acessado em 09/01/2011*
[42] Disponível em http://veja.abril.com.br/noticia/vida-digital/android-supera-symbian-em-celulares, *acessado em 09/01/2011*
[43] Disponível em http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html, *acessado em 09/01/2011*
[44] Disponível em http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html, *acessado em 09/01/2011*
[45] Disponível em http://developer.android.com/reference/android/content/ContentValues.html, *acessado em 09/01/2011*
[46] Disponível em http://pt.wikipedia.org/wiki/Sistema_de_gerenciamento_de_banco_de_dados, *acessado em 09/01/2011*
[47] Disponível em http://javafx.com/, acessado em 09/01/2011
[48] Disponível em http://www.oracle.com/technetwork/java/javame/index.html, *acessado em 09/01/2011*
[49] Disponível em http://en.wikipedia.org/wiki/Integrated_development_environment, *acessado em 09/01/2011*
[50] Disponível em http://www.eclipse.org/, acessado em 09/01/2011
[51] Disponível em http://developer.android.com/sdk/eclipse-adt.html, *acessado em 10/01/2011*
[52] Disponível em http://developer.android.com/guide/developing/tools/avd.html, *acessado em 10/01/2011*
[53] Disponível em http://www.oracle.com/technetwork/java/javase/overview/index.html, *acessado em 10/01/2011*
[54] Disponível em http://en.wikipedia.org/wiki/Rich_Internet_application, *acessado em 10/01/2011*

[55] Disponível em http://www.oracle.com/us/sun/index.html, *acessado em 10/01/2011*
[56] Disponível em http://www.oracle.com/br/index.html, *acessado em 13/01/2011*
[57] Disponível em http://www.mysql.com/, *acessado em 13/01/2011*
[58] Disponível em http://hsqldb.org/, *acessado em 13/01/2011*
[59] Disponível em http://www.eclipse.org/forums/, *acessado em 10/01/2011*
[60] Disponível em http://www.coppetec.coppe.ufrj.br, *acessado em 10/01/2011*
[61] Disponível em http://www.magicdraw.com/, *acessado em 10/01/2011*
[62] Disponível em http://pt.wikipedia.org/wiki/Caso_de_uso, *acessado em 14/01/2011*
[63] Disponível em http://pt.wikipedia.org/wiki/Diagrama_de_atividade, *acessado em 15/01/2011*
[64] Disponível em http://en.wikipedia.org/wiki/User_interface, *acessado em 15/01/2011*
[65] Disponível em http://developer.android.com/reference/android/content/ContentProvider.html, *acessado em 15/01/2011*
[66] Disponível em http://developer.android.com/reference/android/content/IntentFilter.html, *acessado em 15/01/2011*
[67] ANNEKE KLEPPE & JOS WARMER & WIM BAST. MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison-Wesley, 2003.
[68] STEPHEN J. MELLOR & KENDALL SCOTT & AXEL UHL & DIRK WEISE. MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, 2004
[69] ERIC FREEMAN & ELISABETH FREEMAN & KATHY SIERRA & BERT BATES. Use a cabeça! Padrões de Projeto, Alta Books, 2005
[70] Disponível em http://www.ibm.com/developerworks/br/library/os-eclipse-android/, *acessado em 13/01/2011*
[71] Figura disponível em http://www.modeliosoft.com/technologies/technologies-mda.html, *acessado em 13/01/2011*
[72] Figura desenvolvida no Software MagicDraw[61] , *acessado em 13/01/2011*
[73] Figura desenvolvida no Software MagicDraw[61] , *acessado em 13/01/2011*
[74] Figura disponível em http://agileea.wikidot.com/workproduct-list, *acessado em 13/01/2011*
[75] Figura disponível em [10], *acessado em 06/01/2011*
[76] Figura disponível em http://www.alexandre-julien.com/php/joomla/joomla-api-framework-mvc-1-5-creation-dune-application-tuto-crud/, *acessado em 12/01/2011*
[77] Disponível em http://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software, *acessado em 12/01/2011*

Anexo A

Neste Anexo será apresentado os Diagramas de Classe do Projeto Suporte País, assim como os Diagramas de Atividade.

As Figuras 33 e 34 mostram o Diagrama de Classe da Camada de Dados e da Camada de Serviço, respectivamente.

A Cardinalidade da relação entre as Entidades País e Unidade Federativa é 1 para N, ou seja, um País pode se relacionar com uma ou mais Unidades Federativas.

Nas Figuras 35 e 36 são mostrados os Diagramas de Atividade dos Casos de Uso Consulta País e Detalha País, desde o início do Fluxo até o seu fim, quando se pode voltar para a Tela anterior.

As Figura 37, 38 e 39 mostram as Telas de Apresentação ao Usuário do Suporte País. São elas: Consultar País, Resultado da Consulta e Detalhar País, respectivamente.

A Figura 40 mostra como seria feita uma implementação no código parcialmente gerado utilizando a API desenvolvida e não utilizando a API. Neste caso, dois exemplos são mostrados: implementando com o Banco Android SQLite e com o Banco Oracle. Pode-se notar que utilizando uma plataforma específica, o Sistema desenvolvido fica acoplado à plataforma, ou seja, caso no futuro se queira realizar uma portabilidade, o código deverá ser refeito. Já no código utilizando a API, basta mudar a implementação da mesma, pois o código implementado no Sistema não sofrerá alterações.

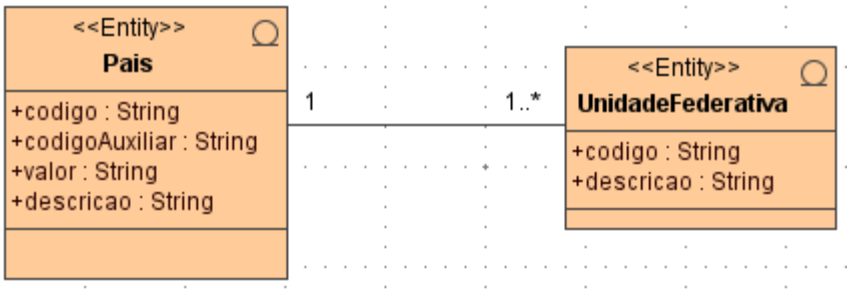

Figura 33: Diagrama de Classe da Camada de Dados

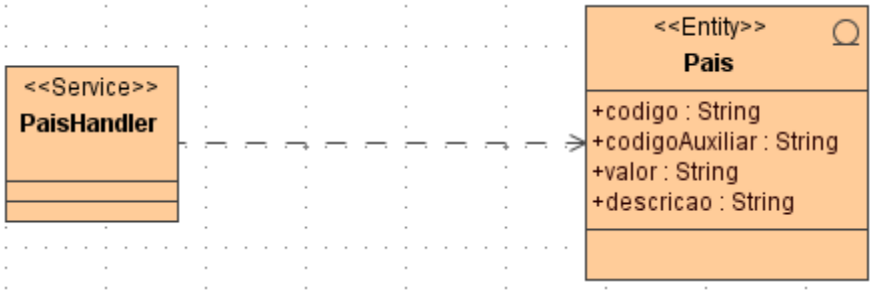

Figura 34: Diagrama de Classe da Camada de Serviço

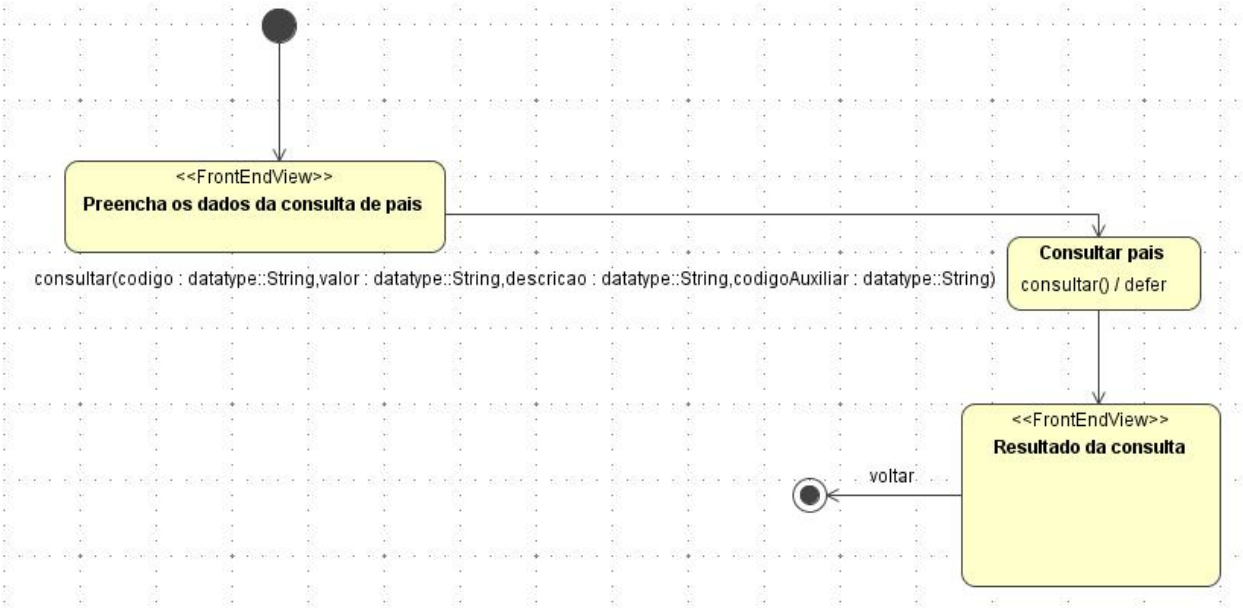

Figura 35: Diagrama de Atividades do Caso de Uso Consulta País

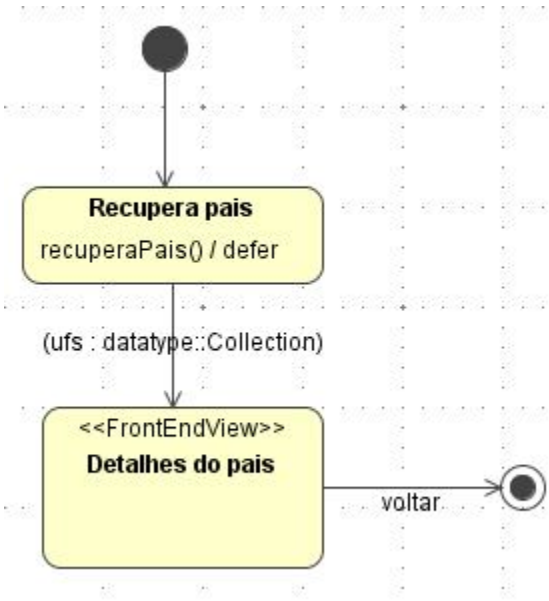

Figura 36: Diagrama de Atividades do Caso de Uso Detalha País

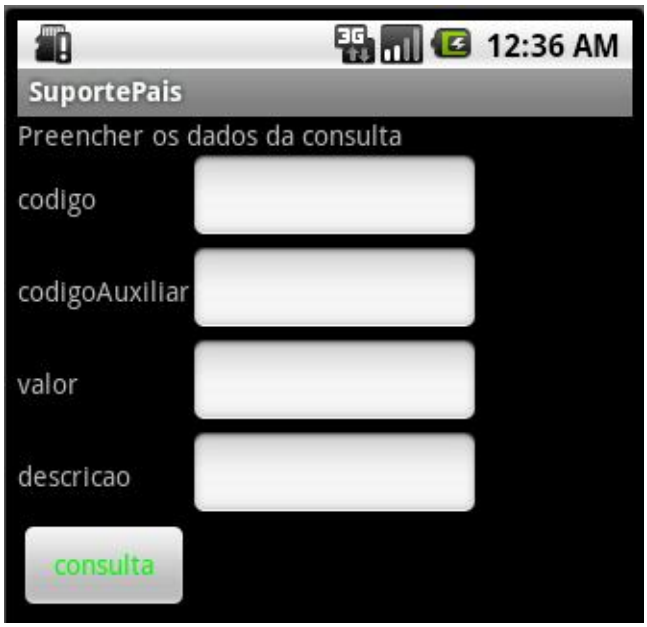

Figura 37: Tela de Apresentação do Consultar País

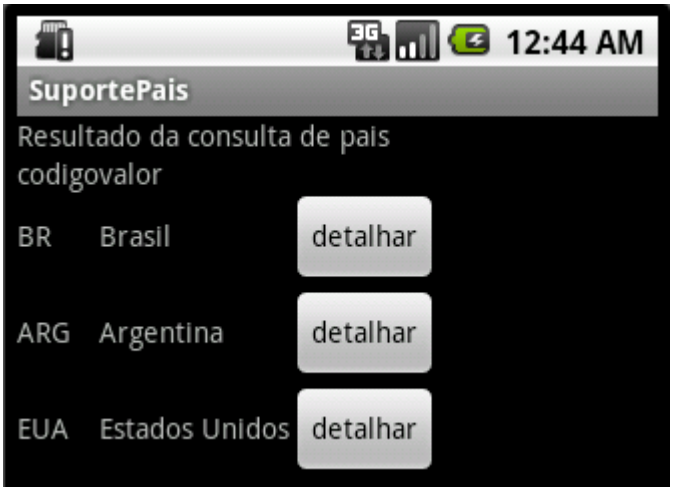

Figura 38: Tela de Apresentação do Resultado da Consulta

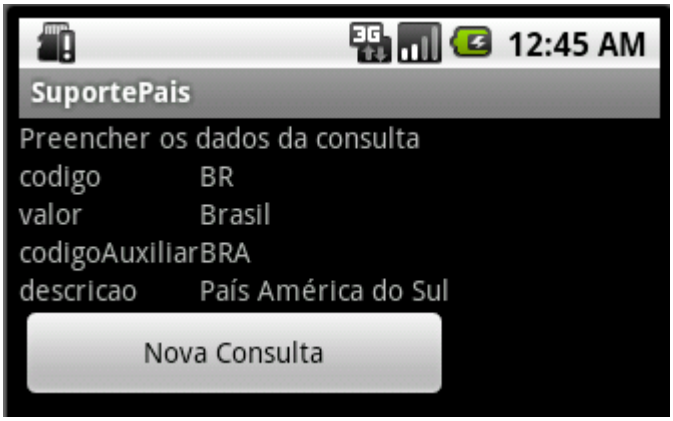

Figura 39: Tela de Apresentação do Detalhar País

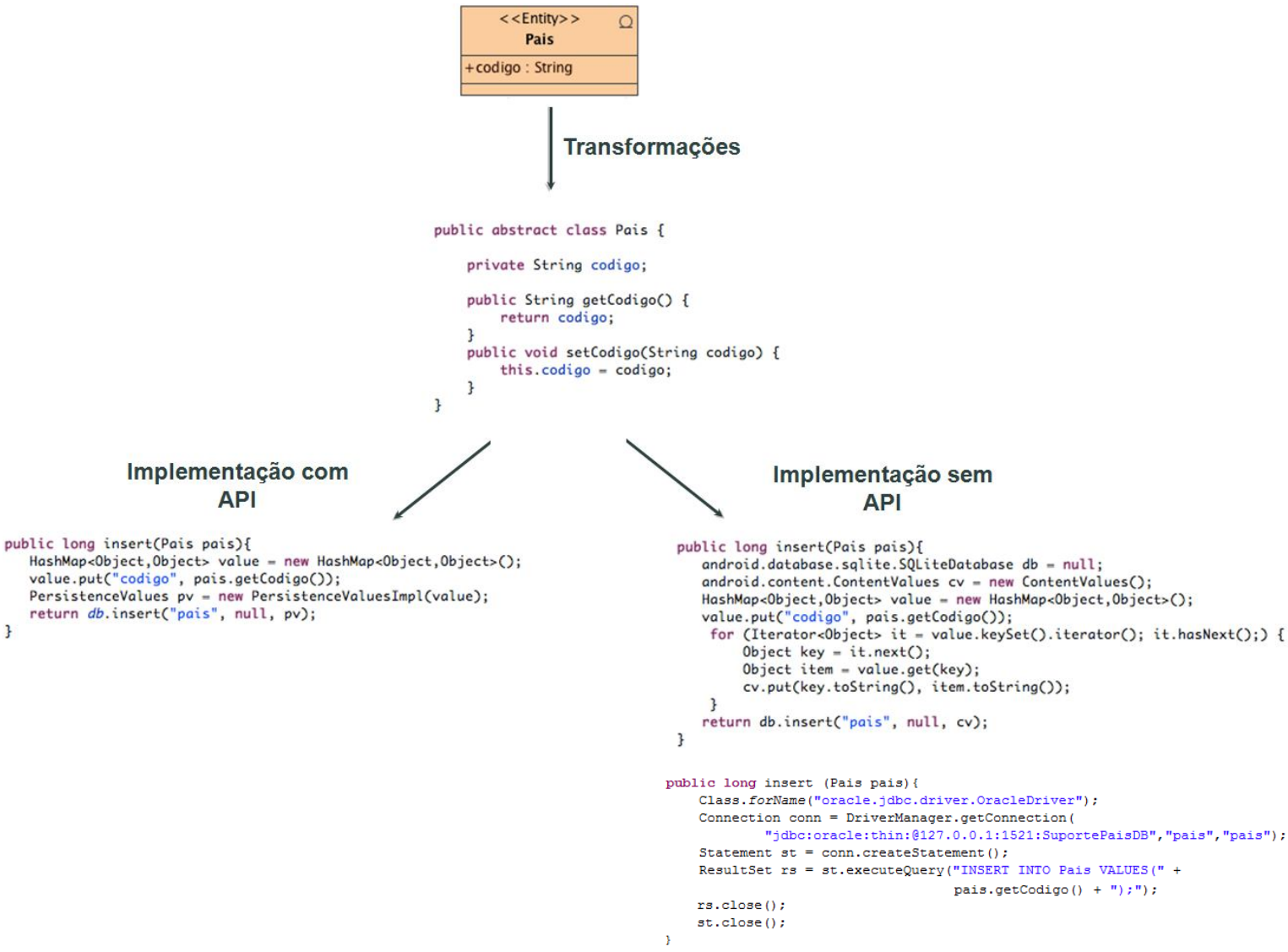

Figura 40: Implementação de código com e sem API utilizando Sqlite e Oracle